Model Parallelization in Tensorflow

Jonas Klotz
jonas.klotz@campus.tu-berlin.de

***Abstract*—In the past few years, the size of the training dataset and the number of model parameters of neural networks experienced growth, resulting in improved classification accuracy. To guarantee efficient training it is now common to parallelize deep neural network training with data parallelism, model parallelism or a combination of those. However, these strategies often lead to inadequate parallelization performance. In this case study, we used a *tf.distribute.server architecture* to parallelize a convolutional neural network with increasing parameter sizes on up to 4 concurrent processes. The results indicate that model parallelization of a large neural network leads to decreased training time, as it alleviates the effect of parameter synchronization and encounters memory limitations. Furthermore, it was shown that model-parallelism is necessary for efficient training when the model is too large for the memory of a worker. To verify the results and test for significance the experiments should be repeated with a higher amount of parallelization and larger models.**

I. INTRODUCTION

It is indisputable that machine learning has a great impact on various domains such as speech recognition [1], [2], visual object recognition [3] or text processing [4]. This increasing influence is due to various factors like the availability of large data sets, more sophisticated machine learning models and the falling cost of computational power [5]. One example that confirms the value of end-to-end deep learning methods for speech recognition is the Deep Speech 2 network. It is a high-performing speech recognizer that recognizes the two very different languages English and Mandarin without any given expert knowledge [6]. An example of face recognition is DeepID3, which is a very deep neural network that stacks multiple convolution/inception layers before each pooling layer. This results in 96.0% closed-set and 81.4% open-set face identification accuracies [7]. In the context of deep learning, early work has focused on training relatively small models on a single machine [4]. Nowadays, there is a tendency towards bigger models [8], as current research showed that deep learning scales well with a bigger training dataset and a larger number of model parameters [4]. The combination of those can immensely improve classification accuracy [4], [9], [10]. To guarantee efficient training it is now common to use distributed heterogeneous clusters and parallelize Deep Neural Network (DNN) training amongst them [4], [5].

The main strategy for distributed DNN training is Batch-splitting (data-parallelism), due to its capability of Single-Program-Multiple-Data (SPMD) programming and universal usability [11]. In data-parallel algorithms each worker receives a subset of the training data and a local copy of the model parameters. On this basis the algorithm parallelly computes partial updates for all model parameters in each worker and then combines these updates for a global estimation of the new model parameters [12], [13]. However, this approach only works well with small models. If there is a larger number of examples and parameters, the size of the GPU memory (usually less than 6 GB) and the time necessary to synchronize the parameters limits the training speed-up drastically [4], [11].

A different method for parallelization is model-parallelism [4], [14], where large models can be partitioned across multiple machines [4] so that the computation on different nodes is assigned to different machines. The idea is to parallelize computation between all of the available cores, and handle ”communication, synchronization and data transfer between machines during both training and inference” [14]. A model parallel algorithm attempts to update a part of the parameters on each worker simultaneously using the full data set or splitting it into subsets [12]. This approach eliminates parameter synchronization between devices but requires data transfers between operators [14]. Such a scheme directly alleviates memory bottlenecks caused by massive parameter sizes in big models. But even for small or mid-sized models, an effective model-parallelism scheme is still highly valuable since it can speed up an algorithm by updating multiple parameters concurrently, using multiple machines [12]. Besides that, models with a large number of parameters or high computational demands typically benefit from access to more CPU’s and memory, up to the point where communication costs dominate [4].

Nevertheless, model-parallelism can also restrain the architecture possibilities of the model. For instance in the Google Neural Machine Translation [15] System, an end-to-end learning approach for translation, it was not possible to have only bi-directional LSTM encoder layers, because each layer needed to finish its computations in order for the following layer to proceed. This would result in reduced parallelism among subsequent layers [11] and restrict the computations’ parallelization to two GPU’s only [15].

This paper will give a brief overview of different deep learning libraries and model parallelization in Tensorflow and Mesh Tensorflow. Also, it will present a case study showing that parallelizing models with a large number of parameters leads to a decreased training time as it alleviates the effect of parameter synchronization and encounters memory limitations. Finally, it will shortly introduce different automatic frameworks to find an efficient parallelization strategy.

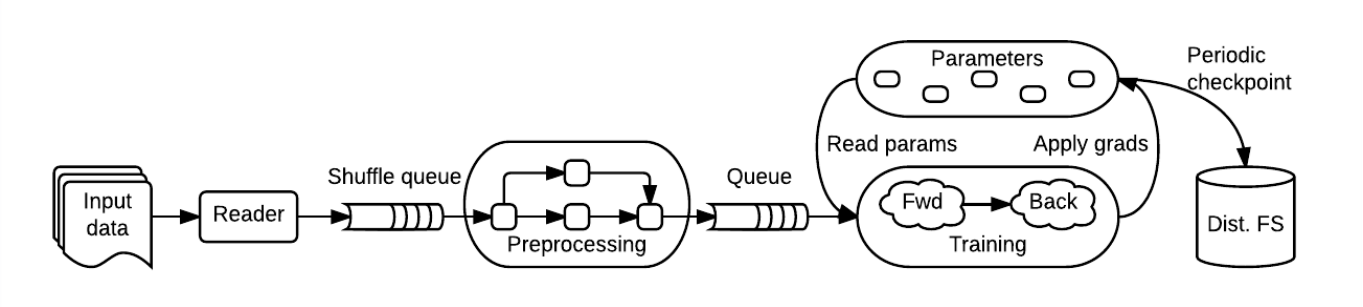

Fig. 1. Example of a data flow graph of Tensorflow [5]

II. DEEP LEARNING LIBRARIES

Deep learning has emerged to one of the most promising approaches to solve classification problems. During recent years we have witnessed a trend towards larger models to improve classification accuracy. For efficient DNN training deep learning libraries need to be able to distribute computations on a large scale onto multiple devices. This paper briefly introduces MXNet and Pytorch and compares them to Tensorflow on which the main focus lies. [5].

A. Pytorch and MXNet

Pytorch [16] is a machine learning library that focuses on usability and speed. Its backend is written in C++ while its style is very pythonic. Pytorch performs immediate execution of dynamic tensor computations. To achieve that, it uses a dynamic computation graph. In more recent updates, the possibility to build a static computation graph was added. PyTorch strictly separates its control flow (i.e. loops) and data flow. It is designed to execute operators asynchronously. For multiprocessing, it extends the python multiprocessing module to automatically move the tensor data sent to other processes into the shared memory, instead of sending it via a communication channel [17]. Pytorch also provides automatic differentiation and computes the gradients through reverse-mode automatic differentiation.

MXNet [18] is a collaborative opensource deep learning framework. The core of MXNet is written in C++. It uses a dataflow system supporting cyclic computation graphs with mutable state and a parameter server. It allows both data and model-parallelism to be implemented. Also, MXNet allows for synchronous and asynchronous training. To parallelize the computation, the dependencies in the computation processes are analysed by a runtime dependency engine. On top of that engine is a middle layer for memory and graph optimization [19]. MXNet also provides basic fault tolerance via checkpointing. Furthermore, it offers automatic differentiation to update the gradients [18].

Recent comparisons of Tensorflow, MXNet and Pytorch have shown that on a larger scale Tensorflow performs best [20]. The experiments were conducted on a ResNet-50 model and measured the images per second throughput. They showed that Tensorflow running on 265 nodes is 2.13× faster than Pytorch. On a single node, Tensorflow is 8% better than MXNet but as it scales more efficient, it outperforms MXNet and is 1.7× faster on 256 nodes. In the experiments, Pytorch performed worst, as on 256 nodes MXNet is 1.25× better than Pytorch [20]. Different smaller-scaled experiments showed that Pytorch may outperform Tensorflow. On 4 GPUs for a ResNet-152 implementation, Pytorch is 1.12 × faster [21]. Another comparison showed that Tensorflow performs better using CPUs but on a small number of GPUs Pytorch exceeds [19]. There are even faster deep learning frameworks because Tensorflow, MXNet and Pytorch use high-level abstractions which cause overhead. But these abstractions are necessary to enable the systems to be implemented on multiple platforms like GPUs and TPUs [19]. This paper is focused on Tensorflow as scaling is an important factor for distributing deep learning models with a large number of parameters.

B. Tensorflow

The key concept of TensorFlow is that it combines the input preprocessing, the mathematical operations, the model parameters and their corresponding update rules in a single data flow graph [Fig. 1]. This simplifies the parallelization of computations, as it visualizes the necessary communication between the sub computations [5].Figure 1 shows an example of a data flow graph of Tensorflow. The vertices of the graph represent abstractions of local computation operations. Data is flowing along the edges, modeled as tensors (n-dimensional arrays of mostly primitive data types) [5]. After the input data is read and preprocessed it is used for the training and the updating of the parameters. Additionally, the graph contains stateful operations that are called variables and queues e.g. in preprocessing pipelines

The data flow graph consists of subgraphs for the computation of gradients to automatically differentiate the loss function [5] and evenly split the computation between multiple devices. Then the gradients are applied to the parameters [Fig. 1]. A neural network build with Tensorflow automatically derives the backpropagation only given the composition of layers and a loss function [5], while also taking care of possible GPU limitations, as differentiating might consume a lot of memory [5]. Even when largely distributed, the training of a model

takes a lot of time. Therefore, Tensorflow requires a scheme for fault tolerance to counter possible failures [5]. User-level periodic check-pointing [Fig. 1] offers a clean solution for that. Because one can easily recompute the updates given the input data, it also provides customizable checkpoints to fine-tune the model parameters regarding some machine learning metric [5]. In regular feed-forward networks, the forwarding graph is acyclic. Tensorflow also supports Recurrent Neural Networks [5], which are often used in sequence problems like handwriting recognition [22] or speech recognition [23]. It parallelizes conditional branches and iterations over multiple devices and processes [5].

Nowadays it is usually faster to run the training synchronously on many GPU machines. But TensorFlow also offers an asynchronous replica coordination, where in each step every worker reads the current parameters and applies its gradient at the end [5], guaranteeing high utilization traded for more inefficient training steps [5]. To manage the communication on multiple devices, TensorFlow either uses a synchronous version utilizing queues or one using queues and backup workers. The queues supply equal parameters for the workers and collect their gradients to apply them atomically [5]. The version using backup workers is actively searching for slow workers to relieve them. [5].

The data flow graph allows a clear analysis of which parts have the potential for parallelization. In the parallelization process, the client firstly produces a subgraph that consists of at least one input and output edge of the data flow graph. Then, the subgraph is pruned regarding the necessary set of operations for the computation [5]. Each repetition of this is called a step and TensorFlow can parallelize the computation of multiple steps [5]. The different steps train the model with different input-batches, implementing data-parallel training. Furthermore, separate subgraphs read the data and take care of checkpointing for fault tolerance. Usually these parallel steps run asynchronously [5].

As already mentioned, neural networks, especially the ones used in language processing, can be huge, consisting of millions of parameters. These models do not fit on every worker, and it is not efficient to copy the parameters in each step and distribute them. To solve this problem model-parallelism is necessary as it divides the parameters onto different workers [11].

III. MODEL PARALLELIZATION

A. Model Parallelization in TensorFlow

The current standard in distributed DNN training is batch-splitting (data-parallelism) [11]. However, it suffers from major disadvantages when training very big models. For instance, the time necessary to synchronize the parameters on all devices or the memory size required to store the parameters [11]. These issues can lead to very inefficient training when only data-parallelism is used [11]. Choosing model-parallelism as

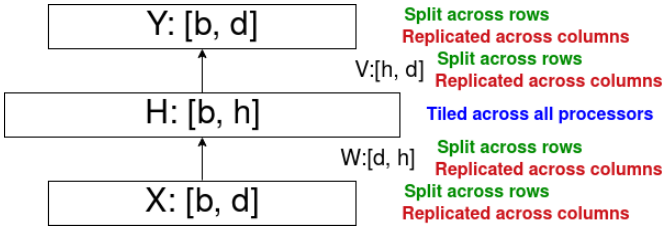

Fig. 2. Split across ”batch” and ”hidden” dimension [Noam Shazeer, Mesh Tensorflow: Model Parallelism for Supercomputers, TF Def Summit ’19]

distribution strategy can solve these issues, but designing an efficient algorithm can be very challenging [11].

For deploying Tensorflow on multiple machines each operation is allocated on a particular device in a particular task [5]. The device manages a kernel to execute all operations registered on that device. Furthermore, Tensorflow provides the possibility to have multiple kernels working on the same computation [5]. The placement of the operation is calculated by a placement algorithm. This algorithm partitions each operation on a device, considering operations that must be co-located and settle them in co-location groups [5]. Besides, it allows users to ”optimize performance by manually placing operations” while maintaining good performance if they choose not to [5]. If the operations are assigned to a device and the stepwise subgraph is calculated, the operations are partitioned into per-device-subgraphs containing all operations for the respective device. As it works on multiple devices the edges are now substituted by *Send* and *Receive* operations [5].

Tensorflow is designed for repeating large subgraphs with low latency, therefore it caches the subgraphs on its device. This simplifies the initiating of a large graph, managed by a session client as it maps the step into the cached subgraph [5]. Whenever very large models are used, the design of the distribution algorithm can become very complicated. This leads to an even larger program which may be laborious to compile [11]. Also if it is aimed to optimize the performance, the operations must be manually placed onto a chosen device. For large and complex models this induces challenging design decisions, as gaining maximum efficiency is difficult [11].

Particularly for more complex models model parallelization can put numerous restrictions on the model architecture [15]. For example, in the Google Neural Machine Translation System complex design would have led to reduced parallelism [11]. These restrictions can be responsible for less accuracy and therefore are one of the main disadvantages of model-parallelism with Tensor Flow.

B. Model Parallelization with Mesh-Tensorflow

A different view on data-parallelism is to see it as a subdivision regarding the dimension of all tensors and operations [11]. Every tensor and operation containing a ”batch” dimension is distributed across all devices, while if it doesn’t contain a ”batch” dimension it is replicated on every device. When an operation out reduces the ”batch” dimension, there is an additional MPI-allreduce necessary. Mesh-Tensorflow

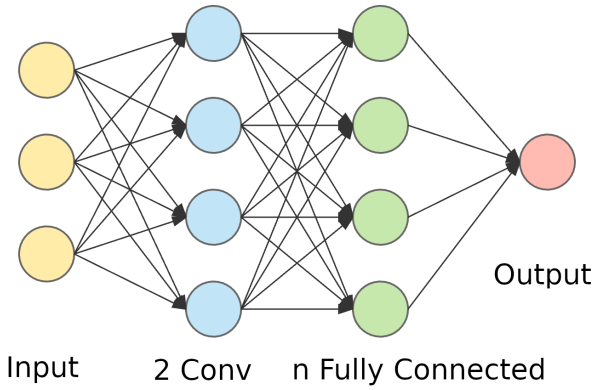

Fig. 3. Neural Network used in the Case Study with n FC Layers

derives from this approach to split the computations across all dimensions. [11] A similarity to regular data-parallelism is that a subset of each tensor in computation and each operation is distributed to each processor. Each device only calculates a part of the final solution and only a few operations need collective communication like MPI-allreduce [11]. Thus the program can be implemented as SPMD [11].

Mesh-Tensorflow is designed to work on an n-dimensional cluster of homogeneous processors, logically structured as a mesh. To identify the dimensions of the mesh unique names are assigned to them. A new feature is that the realization of new logical dimensions (like the "batch") is possible as the tensors now also have unique named dimensions [11]. The main feature of Mesh is that the computational layout allows a partial mapping of the tensor-dimension to the mesh-dimension. This mapping defines which tensor-dimension is split across which mesh-dimension [11]. For example in Figure 2 with a two-dimensional mesh and the computing layout[("batch", "rows"), ("hidden", "cols")] all tensors with a "batch" dimension are split across the rows and all tensors with a "hidden" dimension are split across the columns of the mesh. Vice-versa the "batch" tensors are replicated across the columns and the "hidden" tensors are replicated across the rows, thus resulting in a very efficient layout [11].

Since Mesh-Tensorflow provides an easy generalization of the distribution of computation in Tensorflow, it simplifies both data and model-parallelism and also the combination of both [11]. This solves one of the main issues of model-parallelism, as it handles the difficult design even of complex placement algorithms. Also, it is placed on top of the Tensorflow library, so the language is nearly identical to Tensorflow. As the Mesh-Tensorflow graph is used to generate a regular Tensorflow data flow graph [11], it provides the same syntax and advantages like Tensorflow does, (e.g. variables, devices and automatic gradient computation).

Nevertheless, there are still some disadvantages. As Mesh-Tensorflow does not offer an automated search for the optimal computation layout, finding one turns out to be still quite difficult [11]. Furthermore, it is specifically designed for homogeneous clusters, which restricts people without access to one of those. Also if one is using a heterogeneous cluster and the computation power is not evenly split across all nodes, it can be more efficient to manually place the operations on the particular devices. This is because one can consider the differences between the nodes and locate the more demanding operations on the nodes with more computational power [11].

IV. IMPLEMENTATION

The current trend in machine learning leads to models with a larger number of parameters to improve classification accuracy [4]. Consequently countering complications regarding parameter synchronization and memory limitations has become more relevant. Model Parallelism is a distribution strategy that deals with these problems [11].

In our case study, we want to examine if parallelizing models with a large number of parameters results in a decreased training time. To confirm this assumption we measured the training time for a fixed batch size regarding multiple parallelization degrees of different numbers of parameters. We continuously enlarged the model until the limitations of the worker's memory strongly increased the training time. Then we split up the model to investigate the relation between parallelization and a resulting decrease in training time.

For the experiments, we used a common image recognition approach to classify the MNIST dataset which contains handwritten digits. The model consisted of two convolutional layers with max-pooling and n fully connected layers [Fig. 3]. To enlarge the model n was set to 20/40/60 resulting in parameter numbers:

$$n_{20} = 20.026.752$$
$$n_{40} = 41.018.752$$
$$n_{60} = 62.010.752$$

The increased parameter size led to a decreasing accuracy as the training was very short, but the focus of the study was on oversizing the model not maximizing the classification accuracy. The parallelization was implemented with *tf.distribute.server* architecture. It consisted of one master node(chief) and 1-3 worker nodes. The computations were equally distributed amongst the chief and the workers through manually placing operations on each device i via *with tf.device(devices[i])*. Each worker ran on a different process communicating via the server. For the experiments, we used a GPU Machine with 96 GB RAM and 2 graphic cards (RTX2080TI 11 GB RAM GDDR6) from the CIT department of the TU Berlin.

There were a few challenges on the way mostly concerning the design of the model. At first, we wanted to make a more complex model but in the end, we decided for a very straight forward approach as the complex model appeared more difficult to parallelize. Also, the access to the cluster was limited therefore there wasn't much time for experimenting.

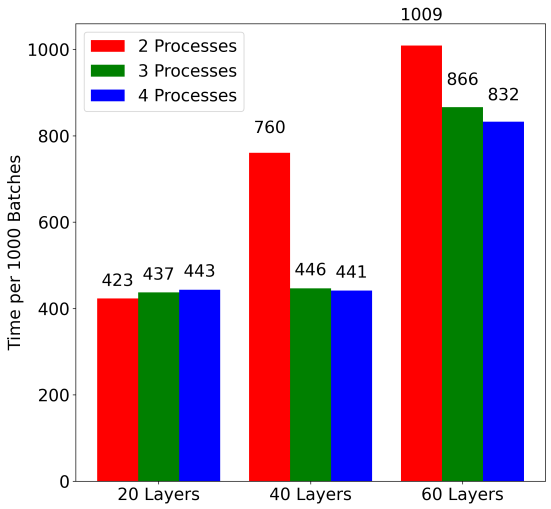

Fig. 4. Comparison - Time, Number of parameters

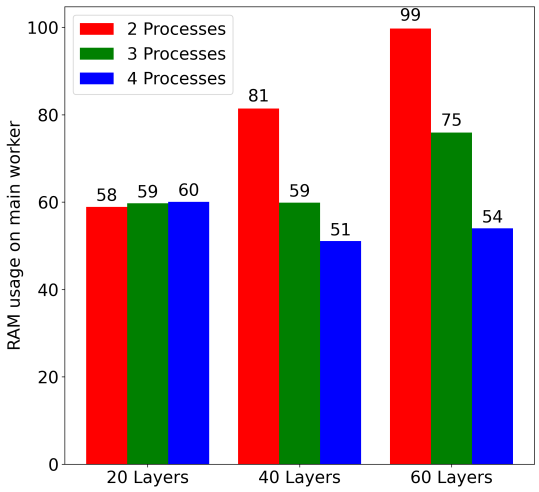

Fig. 5. Comparison - RAM Usage, Number of parameters

V. RESULTS AND DISCUSSION

As the size of a deep learning network heavily scales with classification accuracy, models have grown in recent years. To efficiently train large-scaled networks distribution strategies are necessary as large models don't fit into the memory of a single machine. Model-parallelism can solve these issues as it distributes the parameters onto different devices. To examine the impact of parallelization on large models we measured the training time regarding different parallelization degrees of models with a varying number of parameters.

The x-axis of Figure 4 and Figure 5 describes the number of layers n, the experiments were conducted with n = 20/40/60 and the corresponding parameters $n_{20} \approx 20$ million, $n_{40} \approx 41$ million, $n_{60} \approx 62$ million. The three bars visualize the number of processes used for parallelization. The y-axis of Figure 4 shows the training time in seconds for a fixed batch size of 1000, while the y-axis of Figure 5 shows the average RAM usage in percent on the master chief during the training.

The comparison of Fig. 4 and Fig. 5 shows that more parallelization leads to decreased training time and less RAM usage. For $\sim$20 million parameters (20 layers) all 2, 3 and 4 processes perform nearly equally. The parallelization with 4 processes is the slowest, probably due to communication overhead. For $\sim$41 million parameters (40 layers) 3 and 4 processes have almost the same performance whereas 4 processes perform 1.72 $\times$ better than 2 processes, while also consuming 30% less RAM. The differences remain significant for $\sim$62 million parameters (60 layers), although they decrease.

The main focus lies on the parallelization of 2 processes because there the critical RAM overload was reached. The RAM usage for $\sim$62 million parameters (4 Layers) was close to 100%. Also, for $\sim$41 million total parameters (3 Layers) the RAM usage was 81%. This led to a highly increased training time as the memory limited the training speed-up, resulting in a correlation between training time and RAM usage. There are indicators that this correlation also exists for 3 and 4 parallelized processes although the critical RAM overload was not reached. For the RAM overload of 2 processes, $\sim$30 million parameters per worker were necessary. The maximum workload for 3 processes was $\sim$20 million parameters and for 4 processes $\sim$15 million parameters per worker. This led to fewer differences between the two as no RAM overload was achieved. Also, the increased RAM usage of 21% between 3 and 4 processes handling $\sim$62 million parameters only made a slight time difference of 4% (34 seconds). This raises the question if a RAM overload is necessary to achieve a significant difference in training speed-up when comparing model-parallelism to single machine models. Besides that, it is an open question whether the RAM was also used by other processes resulting in a different workload on the device and how this would interfere with the measurements.

VI. CONCLUSION

As larger machine learning models result in better classification accuracy more and more domains use increasingly complex and computational intensive DNNs. To train them in an acceptable time frame it is necessary to distribute the training on multiple devices and most work regarding parallelization is focused on data-parallelism. In this paper, we evaluate the impact of model-parallelization on larger DNNs. Based on a performance characterization of 2 – 4 parallel processes we showed that model-parallelism can lead up to 1.72$\times$ lower training time as RAM overload drastically limits the training speed-up. This shows that model-parallelism is necessary for efficient training when the model is too big for the memory of a worker. However, the speed-up increase only appeared when the RAM overload was achieved. Therefore,

it is difficult to conclude how big the impact of the actual parallelization was in the experiments. To ascertain that and verify the results, there should be more experiments conducted with a higher amount of parallelization and more parameters

VII. Related Work

Data and model-parallelism are the most common strategies to achieve distributed deep learning [11]. However, both have their disadvantages. Data-parallelism is inefficient for models with a large number of parameters and can become a bottleneck on a larger scale [14], [24]. Model-parallelism can lead to limited parallelism, even if it reduces the communication cost for synchronizing network parameters [14], [24]. There are also automated frameworks to find the most efficient parallelization strategy in a limited search space [14].

A. Layer-wise parallelism - OptCNN

Through the different characteristics of the layers in a CNN, the optimal parallelization strategy can differ in each layer [24]. Data and model-parallelism do not consider this, as they only apply a single parallelization strategy for the whole network [24]. OptCNN proposes layer-wise parallelism, which allows each layer to use an individual parallelism strategy [24]. To find the best possible parallelization strategy they define the search space with parallelization configurations and a cost model that evaluates their performance [24]. Then they use a dynamic programming approach to find a globally optimal strategy minimizing the cost model.

Their approach improves the state-of-the-art as it can increase training throughput by 1.4-2.2× and reduce communication costs by 1.3-23.0× [24]. However, this approach does not consider parallelism across different operators and is limited to linear DNN's. Therefore many problems using large models like language modeling, machine translation and recommendations do not profit from OptCNN as they tend to use RNN's and other non-linear networks [14].

B. Device Placement Optimization with Reinforcement Learning - ColocRL

The placement of operations on the devices mostly relies on decisions of human experts and heuristics and can be challenging if the network is more complex [8]. The ColocRL algorithm tries to find the optimal device placement in model-parallelism via reinforcement learning. The main goal is to find which part of the model should be located on which device and to optimize the order of the computations to minimize communication costs [8]. This is achieved through a sequence-to-sequence model that gets information about the operations and their dependencies as input and produces an optimal placement strategy for them [8]. In the process, they use the colocation feature of Tensorflow to find operations that need to be colocated on the same device [8].

Their implementation surpasses highly optimized algorithmic solvers and human experts on different tasks like image classification, language modeling, and machine translation [8]. Still, it only explores parallelism in the operator dimension [14].

C. Flexflow and SOAP search space

One of the more recent publications introduces the SOAP (Sample-Operator-Attribute-Parameter) search space which considers more dimensions than the previous approaches [14]. The operator dimension indicates how the different operators are parallelized. For each operator, the sample and parameter dimension define the distribution of the training samples and model parameters [14]. The attribute dimension describes how the attributes of a sample are distributed(e.g. channel or height of an image) [14]. Also, they present FlexFlow which is an algorithm to explore the SOAP space [14]. FlexFlow uses an execution simulator to evaluate parallelization strategies and a Markov Chain Monte Carlo (MCMC) search algorithm to explore the search space rapidly [14].

In comparison to ColocRL, Flexflow achieved a 3.4 - 3.8× speedup and even against OptCNN it achieved a 1.2-1.6× speedup as it exploited parallelism across different operators [14]. However, they can still achieve a sub-optimal mapping in terms of communication as their parallelizable dimensions are defined as the set of all divisible dimensions in the output tensor [11].

References

[1] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Transactions on audio, speech, and language processing*, vol. 20, no. 1, pp. 30–42, 2011.

[2] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 8599–8603.

[3] J. Ba, V. Mnih, and K. Kavukcuoglu, "Multiple object recognition with visual attention," *arXiv preprint arXiv:1412.7755*, 2014.

[4] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[6] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, J. Chen, J. Chen, Z. Chen, M. Chrzanowski, A. Coates, G. Diamos, K. Ding, N. Du, E. Elsen, J. Engel, W. Fang, L. Fan, C. Fougner, L. Gao, C. Gong, A. Hannun, T. Han, L. Johannes, B. Jiang, C. Ju, B. Jun, P. LeGresley, L. Lin, J. Liu, Y. Liu, W. Li, X. Li, D. Ma, S. Narang, A. Ng, S. Ozair, Y. Peng, R. Prenger, S. Qian, Z. Quan, J. Raiman, V. Rao, S. Satheesh, D. Seetapun, S. Sengupta, K. Srinet, A. Sriram, H. Tang, L. Tang, C. Wang, J. Wang, K. Wang, Y. Wang, Z. Wang, Z. Wang, S. Wu, L. Wei, B. Xiao, W. Xie, Y. Xie, D. Yogatama, B. Yuan, J. Zhan, and Z. Zhu, "Deep speech 2 : End-to-end speech recognition in english and mandarin," ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 173–182. [Online]. Available: http://proceedings.mlr.press/v48/amodei16.html

[7] Y. Sun, D. Liang, X. Wang, and X. Tang, "Deepid3: Face recognition with very deep neural networks," *arXiv preprint arXiv:1502.00873*, 2015.

[8] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," *arXiv preprint arXiv:1706.04972*, 2017.

[9] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 215–223.

[10] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition," *Neural computation*, vol. 22, no. 12, pp. 3207–3220, 2010.

[11] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young *et al.*, "Mesh-tensorflow: Deep learning for supercomputers," in *Advances in Neural Information Processing Systems*, 2018, pp. 10 414–10 423.

[12] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," in *Advances in neural information processing systems*, 2014, pp. 2834–2842.

[13] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Advances in neural information processing systems*, 2013, pp. 1223–1231.

[14] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *arXiv preprint arXiv:1807.05358*, 2018.

[15] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[16] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[17] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: experiences on accelerating data parallel training," *arXiv preprint arXiv:2006.15704*, 2020.

[18] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[19] K. Zhang, S. Alqahtani, and M. Demirbas, "A comparison of distributed machine learning platforms," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2017, pp. 1–9.

[20] A. Jain, A. A. Awan, H. Subramoni, and D. K. Panda, "Scaling tensorflow, pytorch, and mxnet using mvapich2 for high-performance deep learning on frontera," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*. IEEE, 2019, pp. 76–83.

[21] A. Jain, A. A. Awan, Q. Anthony, H. Subramoni, and D. K. D. Panda, "Performance characterization of dnn training using tensorflow and pytorch on modern clusters," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–11.

[22] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber, "A novel connectionist system for unconstrained handwriting recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 31, no. 5, pp. 855–868, 2008.

[23] X. Li and X. Wu, "Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 4520–4524.

[24] Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks," *arXiv preprint arXiv:1802.04924*, 2018.