Optimizing Multi-GPU Parallelization Strategies for Deep Learning Training

Saptadeep Pal
University of California

Eiman Ebrahimi
NVIDIA

Arslan Zulfiqar
NVIDIA

Yaosheng Fu
NVIDIA

Victor Zhang
NVIDIA

Szymon Migacz
NVIDIA

David Nellans
NVIDIA

Puneet Gupta
University of California

***Abstract*—Deploying deep learning (DL) models across multiple compute devices to train large and complex models continues to grow in importance because of the demand for faster and more frequent training. Data parallelism (DP) is the most widely used parallelization strategy, but as the number of devices in data parallel training grows, so does the communication overhead between devices. Additionally, a larger aggregate batch size per step leads to statistical efficiency loss, i.e., a larger number of epochs are required to converge to a desired accuracy. These factors affect overall training time and beyond a certain number of devices, the speedup from DP scales poorly. This work explores hybrid parallelization, where each data parallel worker comprises more than one device to accelerate each training step by exploiting model parallelism. We show that at scale, hybrid training will be more effective at minimizing end-to-end training time than exploiting DP alone. We project that, for Inception-V3, GNMT, and BigLSTM, the hybrid strategy provides an end-to-end training speedup of at least 26.5%, 8%, and 22%, respectively, compared to what DP alone can achieve at scale.**

Digital Object Identifier 10.1109/MM.2019.2935967
Date of publication 19 August 2019; date of current version 10 September 2019.

Deep learning (DL) models continue to grow and the data sets used to train them are increasing in size, leading to longer training times. Therefore, practitioners accelerate training by using multiple devices (e.g., GPUs/TPUs) in parallel. Data parallelism (DP) is the simplest parallelization strategy where replicas of a model are trained on independent devices using independent subsets of data, referred to as minibatches. However, as the number of devices used to exploit DP increases, the total batch size per step also typically increases. This poses a fundamental problem for data parallel scalability because there exists a global batch size beyond which converging to the desired accuracy requires a significantly larger number of iterations. This is primarily due to reduced statistical efficiency in training.[1] Additionally, as the number of devices employed increases, the synchronization/communication overhead of sharing gradients across devices increases, further limiting overall training speedup.

Model parallelism (MP) is a complementary technique in which the model dataflow graph (DFG) is split across multiple devices while working on the same minibatch. MP has been traditionally used to split large models (which cannot fit in a single device's memory), but employing MP can also help speed up each training step by placing and running concurrent operations on separate devices. Unfortunately the amount of parallelism that exists in today's DL models is often limited, either by the algorithm or by its implementation. Therefore, using MP alone to obtain performance through parallelization typically does not easily scale to a large number of devices. Additionally, maximizing the speedup from MP is often nontrivial and requires careful partitioning with knowledge of the model DFG and underlying system hardware.

This article studies which parallelization strategies to adopt to minimize end-to-end training time for a given DL model on an available hardware. We ask the question: how can we improve DP scaling by combining MP and DP to achieve the best possible end-to-end training time to a given accuracy? The novel insight of this article is that when the number of devices (and hence global batch size) grows to a point where scaling from DP slows significantly, MP should then be used in conjunction with DP to continue improving training times. The speedup obtainable via MP is critical to this tipping point. We show that MP's speedup can help overcome DP's scaling and statistical efficiency degradation at a unique scale for each network. We make the following contributions:

This article studies which parallelization strategies to adopt to minimize end-to-end training time for a given DL model on an available hardware. We ask the question: how can we improve DP scaling by combining MP and DP to achieve the best possible end-to-end training time to a given accuracy?

- We show that when DP's inefficiencies become large, a hybrid parallelization strategy where each parallel worker is model parallelized across multiple devices will further scale multidevice training.
- We develop an analytical framework to systematically find this cross-over point, in terms of devices, to determine the most efficient parallelization strategy on a given system.
- We show that hybrid parallelization outperforms DP alone at different scales for different DL networks. We implement 2-way model parallel versions of Inception-V3, GNMT, and BigLSTM, and project that using them, hybrid training provides a speedup of at least 26.5%, 8%, and 22%, respectively, above DP-only training at scale.
- We propose *DLPlacer*, an integer linear programming based tool to find optimal operation-to-device placement that maximizes MP speedup. We demonstrate DLPlacer's effectiveness by deriving an optimal placement for Inception-V3,[2] showing the obtained 1.32× model parallel speedup with two GPUs is within 6% of the optimal predicted by DLPlacer.

RELATED WORK

In this article, we identify scaling and statistical efficiency losses as the greatest challenges to scalable data parallel training. Many other

articles also focus on improving the scalability of both data and model parallel training. We summarize the most significant related advancements here.

Hybrid Parallelization

Previous work[3–7] has also used hybrid parallelization for scaling DL training. To the best of our knowledge, none of these articles provide a systematic analysis to find what parallelization strategy minimizes end-to-end training time when a set of N compute devices are available for training a given DL model. For example, Yadan *et al.*[4] showed that a hybrid (2-way DP, 2-MP) approach performs better than both MP-only and DP-only when training AlexNet on a 4-GPU system, but do not discuss the cause of the results or evaluate this effect across different GPU counts. Dean *et al.*[7] used hybrid parallelism to train models that would not fit in a single GPU's memory. Therefore, in each data parallel worker, the model replica is model parallelized across multiple devices. However, using model parallelism for models that do not fit in a single GPU's memory is largely orthogonal to the issue we address in this article.

Orthogonal Parallelization Strategies

Exploiting model parallelism is just one way to achieve per step speedup without increasing global batch size. Other strategies exist that can be combined with, or used in place of, model parallelism to augment data parallel scaling under our proposed model. Jia *et al.*[8] propose layer-wise parallelism for CNNs where each network layer can use an individual parallelization strategy. A combination of the four-dimensional (4-D) tensor dimensions can be used to parallelize a given layer, and exploring multiple dimensions may provide larger runtime benefits than MP. However, such a technique is not yet supported by most frameworks and is evaluated using a custom framework. GPipe[9] and PipeDream[10] proposed partitioning a DL model's DFG into multilayer stages and applying pipeline parallelism. It is likely that one or a combination of the layer-wise, pipeline, and model parallelism techniques can be combined with the DP training to maximize end-to-end training performance and efficiency.

Reinforcement-Learning-Based Device Placement

Prior work has shown that by using reinforcement learning (RL)-based placement of operations onto devices, MP can achieve training speedup and that the RL generated placement is nontrivial.[11] However, RL-based approaches can be long-running and compute-intensive with no notion of optimality. On the other hand, DLPlacer can provide optimal device placement solutions, though can still be compute intensive for complex DL networks and when the system contains a large number of devices. On the other hand, it is worth noting that for the models with simple DFGs, tools for finding device placement may be unnecessary and simple heuristics may achieve near-optimal placement results.

DECOMPOSING END-TO-END TRAINING TIME

End-to-end training time for a DL model depends on three factors: the average time per step (T), the number of steps per epoch (S) and the number of epochs (E) required to converge to a desired accuracy. Therefore, the total training time, i.e., time to converge (C) can be expressed as

$$C = T \times S \times E. \tag{1}$$

T is determined primarily by compute efficiency, i.e., given the same training setup, algorithm, and minibatch size, T depends solely on the compute capability of a device; better performing hardware provides smaller T values. The number of steps per epoch (S), is equal to the total number of items in the training data set, divided by the global batch size or the number of inputs per step. The number of epochs to converge (E) depends on the global batch size and other training hyperparameters.

Quantifying Data Parallel Training Time

In data parallel training, the network parameters (weights) are replicated across multiple worker devices and each worker performs a forward and a backward pass individually on a distinct batch of inputs [shown in Figure 1(a)]. In this article, we focus on synchronous stochastic gradient decent ($sync - SGD$) for weight updates. In

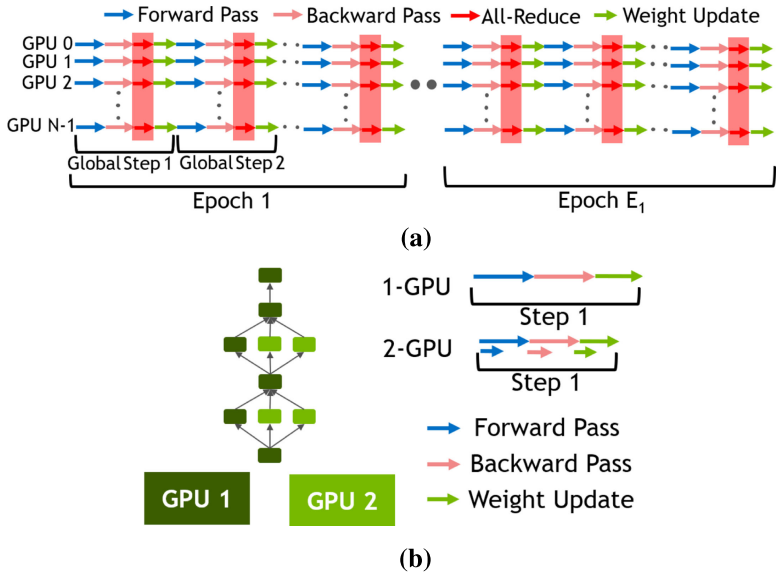

Figure 1. Different training parallelization strategies. (a) Data parallel training. (b) Model parallel training.

$sync-SGD$ workers are synchronized, i.e., the gradients from workers are shared and the network parameters are updated such that all workers have the same parameters after each step. An alternative approach uses asynchronous updates, usually with a parameter server. When scaling to a large number of devices, this approach performs poorly.[12] Therefore, we use a ring-based all-reduce mechanism for data parallel training that provides superior performance and scalability and primarily supports $sync-SGD$. We call the batch of inputs per worker a *minibatch* and the collection of all the minibatches in a training step a *global batch*. When using DP alone to accelerate training, the speedup from employing *N*-way DP (SU_N) compared to training on a single device can be expressed as

$$SU_N = \frac{T_1}{T_N} \times \frac{S_1}{S_N} \times \frac{E_1}{E_N}. \quad (2)$$

T_1 is the average training time per step when only one device is used for training, whereas T_N is the time per step when N data parallel devices (with a constant minibatch size per device) are used. T_N is always greater than T_1 due to the additional time required to communicate gradients among devices at the end of each global step [see Figure 1(a)], straggler effects due to slow workers, and I/O bottlenecks. As such, T_1/T_N is typically less than one and we call this ratio the scaling efficiency (SE_N) of N-way DP.

S_1 and S_N are the total number of steps per epoch with one and N devices, respectively. With a single device, the global batch size is equal to the minibatch size. In N-way data parallelism, we assume the global batch size is N-times the minibatch size per device. Thus, S_1/S_N is equal to N. (Differing from CPU implementations,[3] reducing minibatch size to maintain a constant global batch size can lead to under-utilization of GPU compute resources.)

E_1 and E_N refer to the number of epochs required to converge when one device or N devices are used. At larger global batch sizes (higher N), the gradients from a larger number of training samples are averaged, which results in a tendency to get attracted to local minima. This eventually leads to poor generalization[13] and typically more epochs are required to converge. As such, E_1/E_N is usually less than one. Equation (2) can, thus, be simplified as

$$SU_N = SE_N \times N \times \frac{E_1}{E_N}. \quad (3)$$

When training at larger device counts (N) both SE_N and E_1/E_N decrease. We describe how we calculate the values for SE_N, E_1, and E_N in detail in the "Methodology" section.

Quantifying Model Parallel Training Time

In model parallel training, the model is split by placing different operations of its DFG onto different devices, as shown in Figure 1(b). This enables more than one device to work on the same minibatch and execute independent operations concurrently. While MP has been traditionally used for large models whose parameters do not fit on a single device,[14] we focus on MP's ability to improve per-step training time; improving term T in (1). We call the speedup from M-way MP SU^M, and it includes the communication cost of data movement between dependent operations placed across multiple devices.

The inherent benefit of using MP to improve per-step training is that it does not increase global batch size. As such, the number of epochs required to converge do not change. Hence, improving SU^M reduces convergence time by solely reducing term T in (1). We find that typically the inherent parallelism of a given model or its implementation limits the achievable SU^M. Therefore, MP alone has not been considered a broadly applicable scalable parallelization strategy. However, we show that MP can be combined with DP to extend training scalability beyond today's limits.

Hybrid Data and Model Parallel Training

Let's assume that we have scaled our training system up to N devices using N-way DP. If additional devices (say $M \times N$ devices, where M is an integer) were to become available for training, how should we best use these devices for distributed training? Our goal is to identify when to use DP alone, and when to combine DP with MP to obtain the highest possible training speedup. Using DP alone, the speedup from $M \times N$ devices compared to one device is [substituting $M \times N$ for N in (3)]

$$SU_{M\times N} = SE_{M\times N} \times M \times N \times \frac{E_1}{E_{M\times N}}. \quad (4)$$

A few observations are important when comparing the speedup from $M \times N$-way DP (4) and speedup from N-way DP (3): First, since global batch size is larger at $M \times N$ devices (using a constant minibatch size per device), the number of steps per epoch is smaller by a factor M compared to N-way DP. Second, $SE_{M\times N}$ is smaller than SE_N because now all-reduce communication happens between a larger number of devices. Depending on the values of N, M, and system configuration, all-reduce communication potentially crosses slower internode links that increases all-reduce times and reduces $SE_{M\times N}$. Third, the number of epochs required $E_{M\times N}$, is greater than or equal to E_N.

When using $M \times N$ devices in a hybrid parallelization strategy with N DP workers and each worker being split using M-way MP, overall training speedup can be expressed as

$$SU_N^M = SU^M \times SE_N \times N \times \frac{E_1}{E_N}. \quad (5)$$

In the hybrid configuration, every M devices are grouped into a single data-parallel worker, and therefore, the global batch size, and as a result the number of epochs to convergence remains the same as that of N-way DP. Additionally, the number of steps per epoch remains the same. As such, the per-step speedup achieved through MP increases the overall training speedup by a factor of SU^M, when comparing (3) and (5).

Choosing the Best Parallelization Strategy

By substituting (4) and (5) into

$$\begin{aligned} SU_N^M &> SU_{M\times N} \\ SU^M \times SE_N \times N \times \frac{E_1}{E_N} &> SE_{M\times N} \times M \times N \times \frac{E_1}{E_{M\times N}} \\ SU^M &> M \times \frac{SE_{M\times N}}{SE_N} \times \frac{E_N}{E_{M\times N}} \end{aligned} \quad (6)$$

we determine the conditions under which using hybrid parallelization will be better than DP alone for $M \times N$ devices. Equation (6) shows that if the speedup obtained from MP (for a given model parallel implementation) is large enough to overcome the scaling and statistical efficiency losses that come from increased synchronization overhead and global batch size, employing a hybrid MP and DP strategy will improve training time.

For a fixed device count P, the hybrid approach would use M-way MP and P/M-way DP. Depending on the speedup obtained from M-way MP and P-way DP's efficiency losses, the choice of parallelization strategy would depend on the factors in (6). Therefore, depending on these relative improvements at any device count, the choice of parallelization strategy is critical to the training speedup obtained while scaling to yet larger number of devices. This choice depends on the DL network's properties and system configuration parameters, so there is no one size fits all solution to efficient scale-out multidevice training.

METHODOLOGY

We use the following DL models in our evaluations with their default hyperparameters, unless specified.

- *Inception-V3*[2] is an image recognition network composed of multiple blocks, each with several branches of convolution and pooling

operations. We use the implementation provided with the public NVIDIA Tensorflow container v18.07 and train the network using the ImageNet data set. We scale the initial learning rate linearly with the increase in global batch size as originally proposed by Goyal *et al.*[15] For measuring epoch counts, we train the model until a training loss of 6.1 is achieved.

- *GNMT*[14] is a language translation network with attention mechanism. We use four LSTM layers of size 1024 in the encoder and decoder. The learning rate schedule is well optimized for the global batch sizes of 512–2048. We use exponential learning rate warm-up for 200 training steps. The learning rate decay is started after 6000 steps and decays for a total of four times after every 500 iterations with a decay factor of 0.5. Such a technique has been shown to scale well when global batch size is scaled. We train the network using the WMT'16 German–English data set until a BLEU score of 21.8 is achieved.
- *BigLSTM*[16] is a large scale language modeling network. It consists of an input embedding layer of size 1024, two LSTM layers with hidden state size of 8192, and a Softmax projection layer of size 1024. We implemented the network in the public NVIDIA PyTorch container v19.06, used a learning rate of 0.1, and trained using the 1 billion word language modeling data set to a perplexity of 67.

System Configuration and Evaluation Points

Our experiments mostly use an NVIDIA DGX-1 with 4 T V100 GPUs connected via NVLink with 16 GB of memory capacity. For the BigLSTM experiments, we use GV100 cards with 32 GB of memory, because this network requires more capacity to execute on a single GPU. We use NCCL2.0 based all-reduce communication for gradient sharing.

To project when hybrid training will perform better than DP alone, we measure the epoch counts to convergence for DP at different GPU counts. We also measure the speedup achieved via MP when M GPUs are used for a model-parallel worker in a hybrid strategy. Without loss of generality, we fix $M = 2$ to make a case for future hybrid parallelization strategies. In practice, the value chosen for M (for a DL model) will depend on the speedup obtained from M-way MP and the efficiency losses of DP alone at scale.

Measuring Epoch Counts to Convergence

Typically, epoch counts to convergence for DP on N compute nodes are obtained by running the training on N nodes. We select minibatch sizes to saturate single GPU throughput or lower if the desired minibatch size is limited by GPU memory capacity. We gather epochs to convergence on a 4-GPU system, so the maximum global batch size possible to measure is $4 \times B$, where the minibatch size is B. To emulate larger global batch sizes (corresponding to more than four GPUs), we use the delayed gradient update approach where multiple minibatches are processed per GPU before the gradients are shared for weight update. It is worth noting that even though we complete training of a DL model once to find E_N, in practice, many DL models are often retrained many times during development or as new data becomes available. Our proposed systematic modeling approach helps find the best parallelization strategy for optimizing the turnaround time of such subsequent training runs.

Learning rate schedules are sometimes optimized to keep epoch counts to convergence low at large global batch sizes. For example, the learning rate schedules we use for GNMT and Inception V3 were tuned for this purpose. However, in general, hyperparameter tuning is time consuming and requires many training runs. Similar to prior work,[15] we find that even with such tuning, beyond a certain global batch size, the number of epochs required to converge increases rapidly. As such, the proposals of this work are orthogonal to such efforts.

Estimating Scaling Efficiency

When using just four GPUs, we cannot obtain the scaling efficiency (SE_N) of data parallel training corresponding to larger number of GPUs. Thus, we conservatively assume a scaling efficiency (SE_N) of 1. This assumption means the time overhead of communication and synchronization after each step is negligibly small compared to the time taken for the forward and backward passes. This optimistic assumption for DP-only training, minimizes the impact of hybrid parallelization, but reflects the reality that

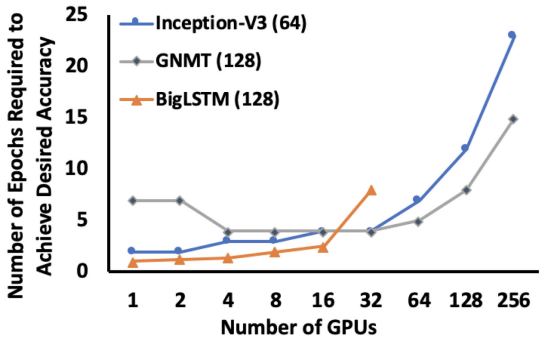

Figure 2. Number of epochs required for the networks to converge versus increasing global batch size with increase in the number of GPUs. Minibatch sizes are shown alongside the model names.

framework developers and system architects are constantly working to improve overheads that hinder DP scaling efficiency.

Model Parallel Splitting

Inception-V3's implementation allows a traditional model parallel mapping of independent operations to different GPUs. We split Inception's DFG across two GPUs using DLPlacer, described in the "Maximizing MP Performance" section. For GNMT and BigLSTM, we split their DFGs using pipeline parallelism.[9] Pipeline parallelism is appropriate for implementing MP on these networks due to the use of optimized libraries and fused RNN kernels in their implementations. Pipelining could similarly be useful for models that do not have parallel branches and are sequential in nature (e.g., ResNet, AmoebaNet).

It is worth noting that the original GNMT implementation[14] uses 8-way MP. However, since we use a system with V100 GPUs that have 14× more FLOPs compared to the K80 GPUs used in that prior work, the ratio of communication overhead to computation is larger in our configuration. We use up-to-date CuDNN libraries with fused RNN kernels and observe that splitting the model beyond 2-way provides marginal per-step speedup because of kernel overheads and pipeline imbalance. In general, deeper pipeline parallel MP implementations can be nontrivial, as pipeline imbalance becomes more prevalent. We similarly observe that splitting Inception-V3 beyond 2-way MP, results in marginal speedup given the efficiency of DLPlacer for this model (see the "Inception-V3 Case Study" section).

EVALUATION

Figure 2 shows the number of epochs required to hit the desired accuracy versus the number of GPUs (workers) used in data parallel training. For Inception-V3, the number of epochs increases sharply from four to seven, as the global batch size increases beyond 2048 (i.e., 32 GPUs) and grows to 23 epochs at a global batch size of 16 384 (i.e., 256 GPUs). For GNMT, the epoch count decreases slightly when going from two to four GPUs because the hyperparameters used are tuned for large global batch sizes. Even with these tuned hyperparameters, as the GPU count increases beyond 64, the number of epochs required grows rapidly. For BigLSTM, beyond 16 GPUs (i.e., global batch size of 2048), the number of epochs increases rapidly and in fact, almost 3.2 times the number of epochs is required for 32-way DP compared to 16-way DP. Beyond 32-way DP, training did not converge in meaningful time. Overall, as we increase the number of GPUs used in DP training, E_1/E_N becomes smaller, which ultimately hinders the overall speedup achievable through data parallel training alone.

Splitting each network across two GPUs using model parallelism results in per-step speedup when done successfully. Table 1 shows the measured MP speedups on our test system for our evaluated networks. Using the number of epochs required and per step speedup from MP, together with the conservative estimates of scaling efficiency, we calculate the minimum projected speedup (over DP alone) of a hybrid parallelization strategy over DP alone for different GPU counts.

Inception-V3: As shown in Figure 3(a), beyond 32 GPUs, a hybrid parallelization strategy performs better than DP-only. This is because of the sharp increase in the number of epochs required when the global batch size grows beyond 2048,

Table 1. MP splitting strategy and the speedup obtained when split across two GPUs.

Network	MP splitting strategy	Speedup
Inception-V3	Partitioned w/ DLPlacer	1.32×
GNMT	Pipeline parallelism	1.15×
BigLSTM	Pipeline parallelism	1.22×

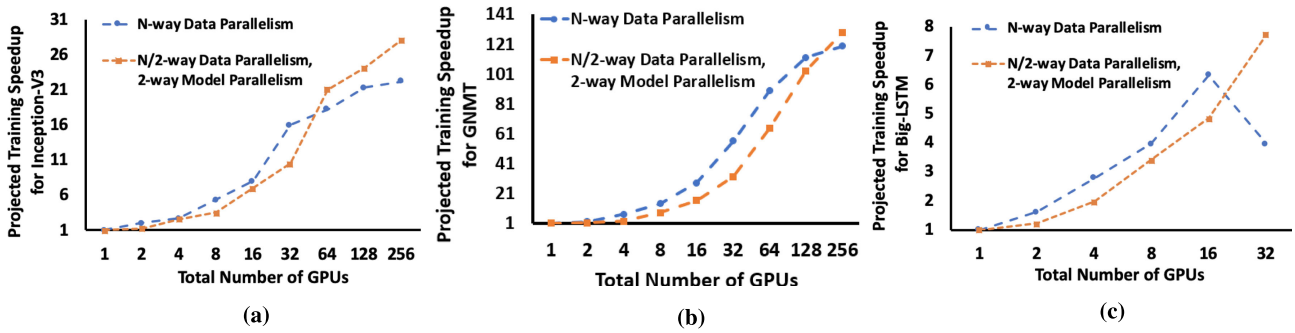

Figure 3. Projected speedup of hybrid MP-DP parallelization versus DP-only parallelization. (a) Inception-V3. (b) GNMT. (c) BigLSTM.

which saturates the speedup obtainable from DP-only parallelization. When moving from 32 GPUs to 64 GPUs, it is better to use the additional 32 GPUs to do 2-way MP, and our conservative estimates show that the hybrid-strategy will outperform DP alone by at least 15.5%. As the numbers of GPUs grow further, only marginal speedup can be obtained from DP-only parallelization and at 256 GPUs, the hybrid-strategy will be atleast 26.5% better than the DP-only strategy.

GNMT: As shown in Figure 3(b), GNMT scales very well to a large number of GPUs using DP alone. However, even with tuned hyper-parameters for larger batch sizes, DP-only speedup starts to slow down beyond 64 GPUs and dramatically slows down when moving from 128 to 256 GPUs. The hybrid parallelization strategy with 2-way MP and 128-way DP outperforms 256-way DP by 8%.

BigLSTM: As shown in Figure 3(c), beyond 16 GPUs, BigLSTM does not scale well with an increasing number of GPUs using DP-only. This is because the statistical efficiency of training decreases rapidly with increasing global batch size, and therefore, the significantly larger number of required epochs offsets the throughput increase of multiple GPUs. At 32-GPUs, the large loss in statistical efficiency impacts the overall training speedup of DP-only strategy and the speedup drops significantly. As a result, the hybrid policy provides a 1.22× speedup over the best performing scale of DP-only, which happens at 16-GPUs, as shown in Figure 3(c).

In summary, these results show that when statistical efficiency loss reduces the effectiveness of DP-only parallelization, hybrid parallelization (combining DP with MP) will enable higher performance than employing DP alone. Notably, using real scaling efficiency loss values (we conservatively assumed $SE_N = 1$), the improvements from hybrid parallelization would be more pronounced since SE_{2N}/SE_N is often smaller than 0.9. Based on (6), the smaller the ratio, the higher the speedup from hybrid parallelism (SU_N^M) compared to DP alone ($SU_{M\times N}$).

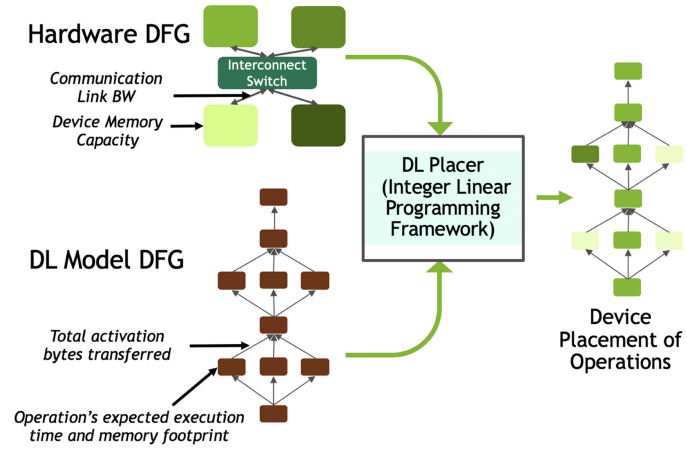

Figure 4. DLPLacer flow diagram.

MAXIMIZING MP PERFORMANCE

Maximizing the speedup obtained from MP for a given model improves the scalability of hybrid parallelism. For some networks, optimal placements are easy to find by examining a network's DFG. For others, finding the optimal operation-to-device placement is nontrivial. Therefore, we developed an integer-linear programming (ILP) based device placement tool called DLPlacer. DLPlacer extracts parallelism between operations in a model and finds the placement to achieve minimum per step execution time.

Figure 4 shows DLPlacer's tool flow. We express DL models as DFGs, with nodes corresponding to compute operations and unidirectional edges showing operation dependencies. Each node contains the operation's expected execution time and memory footprint. An edge weight corresponds to the number of bytes exchanged between the operations it connects. The node and edge weights can be obtained by profiling a model on a compute device (e.g., GPU) or can be analytically calculated, with the former approach being more robust and the latter more flexible. Using similar notation, we express a system as a hardware graph where the nodes are compute devices (e.g., GPUs) or network switches that have bandwidth constraints (e.g., NVSwitch), and edges are the physical links between these nodes.

DLPlacer's ILP solver minimizes per step training time by providing an assignment of compute DFG operations to the hardware graph (placement), a schedule, and a communication routing of activations, weights, and gradients. The constraints DLPlacer satisfies are as follows.

- Each operation must be mapped to only one device.
- Dependencies between operations must be satisfied.
- Multiple operations can be mapped to a device, but colocated operations must not overlap in execution.
- The total amount of memory allocated (inputs, weights, activations) cannot exceed the device memory capacity.

In satisfying the above, DLPlacer assumes the following.

1. Operations colocated on a device are executed back-to-back, without any delay in between the end of one operation and the beginning of the other.
2. Communicating a data chunk of size S on a link with bandwidth B and latency L takes $(S/B + L)$ time.
3. Communication of tensors between devices can be overlapped with computation.

Based on these assumptions DLPlacer predicts the training speedup for a given MP solution. Note

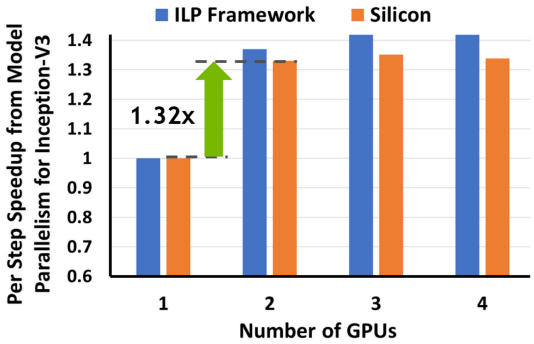

Figure 5. Normalized per-step speedup from model parallelism as estimated by DLPlacer and obtained from silicon experiments for the Inception-V3 network.

that we considered operations at the granularity of Tensorflow operations (e.g., conv2D, conv3D), however, DLPlacer can be used to even find placements when the operations are partitioned into finer granularity operations (e.g., partitioned by channels, filters, etc.). But, such fine grained operation splitting requires framework support for correct back-propagation, and therefore, was not a focus of this article. Also, note that we do not model framework-induced overheads or operating system and runtime effects, which are challenging to model and can lead to prediction inaccuracies. Despite these challenges, we believe ILP based MP optimization is worthwhile to pursue based on the observed improvements over manual optimization.

Inception-V3 Case Study: To evaluate the usefulness of DLPlacer, we use Inception-V3 as a case study. In Figure 5, the blue bars show the normalized per-step speedup estimated by DLPlacer for the optimal placement solution it finds. DLPlacer's runtime on an 18-core Xeon-E5 system to find Inception-V3's placement solution is ~11–18 min depending on the number of device nodes in the hardware graph. The orange bars show speedup as measured on real silicon with DLPlacer's placement applied to the Tensorflow implementation. The speedups predicted by DLPlacer are within 6% of the actual speedup obtained from silicon runs. It is interesting to note that the 1.32× speedup obtained with the 2-GPU placement is almost the same as what is optimally obtainable with three or four GPUs. This is due to the limited parallelism available in the network, which DLPlacer almost completely exploits with a 2-GPU placement. Identifying a 2-GPU placement that gives this

performance by simple observation of the network and without using a tool like DLPlacer is nontrivial. DLPlacer essentially finds the placement with the shortest possible critical path among many feasible placement solutions and places the operations on the critical path in one GPU, so as to avoid communication overhead. This shows the importance of such a tool for maximizing performance obtainable from MP while using minimum number of GPUs. Further details about DLPlacer's formulation can be found in the article by Pal *et al.*[17]

We analyze the end-to-end training time of DP to understand how scaling and statistical efficiency loss impacts training scalability, and show that the MP speedup achieved for a given DL model is critical to the overall scalability of a hybrid parallelization strategy.

CONCLUSION

This article demonstrates the benefits of combining MP with DP to overcome the inherent scaling and statistical efficiency losses that data parallel training has at scale. We demonstrate that when the global batch size in DP grows to a point where DP-only training speedup drops off significantly, MP can be used in conjunction with DP to continue improving training times beyond what DP can achieve alone. We analyze the end-to-end training time of DP to understand how scaling and statistical efficiency loss impacts training scalability, and show that the MP speedup achieved for a given DL model is critical to the overall scalability of a hybrid parallelization strategy. We evaluate the performance benefits of such a hybrid strategy and project that for Inception-V3, GNMT, and BigLSTM, the hybrid strategy provides an end-to-end training speedup of at least 26.5%, 8%, and 22%, respectively, compared to what DP alone can achieve at scale.

REFERENCES

1. E. Hoffer, I. Hubara, and D. Soudry, "Train longer, generalize better: Closing the generalization gap in large batch training of neural networks," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1731–1741.
2. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015, *arXiv:1512.00567*.
3. D. Das *et al.*, "Distributed deep learning using synchronous stochastic gradient descent," 2016, *arXiv:1602.06709*.
4. O. Yadan, K. Adams, Y. Taigman, and M. Ranzato, "Multi-GPU training of convnets," 2013, *arXiv:1312.5853*.
5. S. Sridharan *et al.*, "On scale-out deep learning training for cloud and HPC," 2018, *arXiv:1801.08030*.
6. A. Gholami, A. Azad, P. Jin, K. Keutzer, and A. Buluç, "Integrated model, batch, and domain parallelism in training neural networks," in *Proc. 30th Symp. Parallelism Algorithms Architectures*, 2018, pp. 77–86, doi: 10.1145/3210377.3210394.
7. J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1223–1231.
8. Z. Jia, S. Lin, C. R. Qi, and A. Aiken, "Exploring hidden dimensions in parallelizing convolutional neural networks," *CoRR*, vol. abs/1802.04924, 2018.
9. Y. Huang *et al.*, "GPipe: Efficient training of giant neural networks using pipeline parallelism," 2018, *arXiv:1811.06965*.
10. A. Harlap *et al.*, "PipeDream: Fast and efficient pipeline parallel DNN training," 2018, *arXiv:1806.03377*.
11. A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *Proc. Int. Conf. Learn. Representations*, 2018.
12. J. Chen, R. Monga, S. Bengio, and R. Józefowicz, "Revisiting distributed synchronous SGD," *CoRR*, vol. abs/1604.00981, 2016.
13. N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang, "On large-batch training for deep learning: Generalization gap and sharp minima," *CoRR*, vol. abs/1609.04836, 2016.
14. Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.
15. P. Goyal *et al.*, "Accurate, large minibatch SGD: Training imageNet in 1 hour," 2017, *arXiv:1706.02677*.
16. R. Józefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu, "Exploring the limits of language modeling," 2016, *arXiv:1602.02410*.
17. S. Pal *et al.*, "Optimizing multi-GPU parallelization strategies for deep learning training," 2019, *arXiv:1907.13257*.

Saptadeep Pal is currently working toward a PhD in the Department of Electrical and Computer Engineering at the University of California, Los Angeles. His research interests include scale-out system architectures and design of waferscale processors. Contact him at: saptadeep@ucla.edu.

Eiman Ebrahimi is a senior research scientist at NVIDIA, focusing on efficient scale-out performance of multi-GPU systems for deep learning. Before joining Research, he worked on both discrete and embedded GPU memory systems in product teams at NVIDIA. He has a PhD in computer engineering from the University of Texas at Austin, where his research focused on fair and high-performance memory system architectures for chip multiprocessors. Contact him at: eebrahimi@nvidia.com.

Arslan Zulfiqar joined NVIDIA in 2014 and is currently a senior GPU architect. He has authored/coauthored numerous publications and patents spanning ' improving multi-GPU scaling performance of deep learning models. He has a BS in electrical engineering from the University of Illinois (Urbana-Champaign) and a PhD in electrical engineering from the University of Wisconsin (Madison). Contact him at: azulfiqar@nvidia.com.

Yaosheng Fu is a research scientist with the architecture research team at NVIDIA. His research interests include computer architecture, memory systems, and deep learning acceleration. He received the B.S. degree in electronic engineering from the Tsinghua University and the Ph.D. degree in electrical engineering from the Princeton University. Contact him at: yfu@nvidia.com.

Victor Zhang has been a senior deep learning architect with NVIDIA since April 2018. His research interests include sparsity and precision studies of deep learning models. He has a PhD in theoretical chemistry from the Fudan University. Contact him at: viczhang@nvidia.com.

Szymon Migacz is a senior CUDA deep learning algorithms software engineer at NVIDIA. He joined NVIDIA in 2015, initially worked on CUDA Math Libraries and then shifted his focus to deep learning (DL). His research interests include DL inference in reduced precision, efficient parallel implementations of basic DL building blocks, and scaling computations to many devices, currently focusing on NLP. Contact him at: smigacz@nvidia.com.

David Nellans manages the system architecture research group at NVIDIA. His team works to codesign scalable hardware and software architectures that enable efficient strong and weak scaling for future accelerated computing platforms. He has a PhD in computer science from the University of Utah. Contact him at: dnellans@nvidia.com.

Puneet Gupta is a professor with the Department of Electrical and Computer Engineering, University of California at Los Angeles. His research interests include optimizing across hardware-software and integrated circuit design manufacturing interfaces. He has a PhD from the University of California, San Diego. He cofounded Blaze DFM Inc., (acquired by Tela Inc.) in 2004 and was its product architect till 2007. Contact him at: puneetg@ucla.edu.