A Single-Shot Generalized Device Placement for Large Dataflow Graphs

Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Lin-Kit Wong, Peter Ma, Qiumin Xu, Azalia Mirhoseini, and James Laudon
Google

***Abstract*—With increasingly complex neural network architectures and heterogeneous device characteristics, finding a reasonable graph partitioning and device placement strategy is challenging. There have been prior attempts at learned approaches for solving device placement, these approaches are computationally expensive, unable to handle large graphs consisting over 50 000 nodes, and do not generalize well to unseen graphs. To address all these limitations, we propose an efficient single-shot, generalized deep RL method (SGDP) based on a scalable sequential attention mechanism over a graph neural network that is transferable to new graphs. On a diverse set of representative deep learning models, our method on average achieves 20% improvement over human placement and 18% improvement over the prior art with 15× faster convergence. We are the first to demonstrate super human performance on 8-layer recurrent neural network language model and 8-layer GNMT consisting of over 50 000 nodes, on 8-GPUs. We provide rationales and sensitivity study on model architecture selections.**

Neural networks have demonstrated remarkable scalability–improved performance can usually be achieved by training a larger model on a larger dataset.[6,7] Training such large models efficiently while meeting device constraints, like memory limitations, necessitate partitioning of the underlying dataflow graphs for the models across multiple devices. However, devising a good partitioning and placement of the dataflow graphs requires deep understanding of the model architecture, optimizations performed by

Digital Object Identifier 10.1109/MM.2020.3015188
Date of publication 7 August 2020; date of current version 1 September 2020.

Published by the IEEE Computer Society

domain-specific compilers, as well as the device characteristics, and is therefore extremely hard even for experts.

Graph partitioning and device placement can be specified through a programming interface or through compiler optimizations. ML practitioners often rely on their domain knowledge to determine a reasonable partitioning and device mapping for computational graphs. For example, programmers can manually assign devices to operations through a programming interface such as Tensorflow and Mesh-Tensorflow. However, relying solely on the model architecture while ignoring the effect of the partitioning on subsequent compiler optimizations like operation scheduling can lead to suboptimal placements and consequently under-utilization of available devices. Alternatively, a compiler can apply heuristics to annotate graphs and assign devices to Tensors or operations. The heuristics not only lead to suboptimal configurations but also need to be constantly modified to accommodate new cases arising from previously unseen model architectures.

The goal of an automatic device placement is to find the optimal assignment of operations to devices such that the end-to-end execution time for a single step is minimized and all device constraints like memory limitations are satisfied. Since this objective function is nondifferentiable, prior approaches[1,3,11] have explored solutions based on reinforcement learning (RL). However, these RL policies are impractical to be used in a real production compiler for several reasons. First, they are designed for small to medium sized computation graphs and do not demonstrate strong performance on large graphs where device placement is truly needed. Second, these RL policies are usually not transferable and require training a new policy from scratch for each individual graph. This makes such approaches impractical due to the significant amount of compute required for the policy search itself, at times offsetting gains made by the reduced step time.

In this article, we propose an end-to-end, single-shot deep RL method for device placement (SGDP) where the learned policy is generalizable to new graphs. For the speed of training, we propose a single-shot placement using a re-engineered Transformer-XL network. Instead of generating placement decisions one node a time,[1,3,16] the policy network generates decisions for the entire graph in a single shot fashion. In order to handle large graphs consisting of over 50 000 nodes, we use a Transformer-XL based on segmented recurrent attention[10,18] that partitions the input sequences and generates placement decisions one sequence each of the time while using caching to track intersequence dependencies. The segmented Transformer-XL removes any hard constraints such as hierarchical grouping of operations[3] or colocation heuristics to reduce the placement complexity.[1] For generalization, we apply a graph neural network (GNN) to encode operation features and dependencies into a trainable graph representation, and learn the graph representation end-to-end with the placement policy decisions.

Both our graph-embedding network and placement network can be jointly trained in an end-to-end fashion using a supervised reward, without the need to manipulate the loss functions at multiple levels. We empirically show that the network learns flexible placement policies at a per-node granularity and can scale to problems over 50 000 nodes. By transferring the learned graph embeddings and placement policies, we are able to achieve faster convergence and thus use less resources to obtain high-quality placements.

We empirically show that the network learns flexible placement policies at a per-node granularity and can scale to problems over 50 000 nodes. By transferring the learned graph embeddings and placement policies, we are able to achieve faster convergence and thus use less resources to obtain high-quality placements.

Our contributions can be summarized as follows.

1) An end-to-end deep RL framework to automatically learn graph partitioning and device placement in a single-shot fashion. Our method is demonstrated 15× faster than the prior SoTA based on a hierarchical LSTM model.[1,3]

2) A scalable placement network with an efficient recurrent attention mechanism, which eliminates the need for an explicit grouping stage before placement. Our method handles large graphs consisting over 50 000 nodes and is the first to demonstrate superhuman placement performance on large problems such as 8-layer GMNT and 8-layer recurrent neural network language model (RNNLM).
3) An end-to-end device placement network that can generalize to arbitrary and held-out graphs. This is enabled by jointly learning a transferable GNN along with the placement network.
4) Superior empirical performance over a wide set of important workloads in computer vision, speech, and NLP (InceptionV3, AmoebaNet, RNNs, GNMT, Transformer-XL,[10] WaveNet).
5) Detailed rationales and sensitivity studies on model architecture selections for the policy network. Compared against LSTMs, MLPs, and graph attention networks (GANs).[20]

RELATED WORK

Model-Level Parallelism

Model-level parallelism partitions a neural network model among multiple devices and each device is responsible for the weights updates of the assigned operations or layers. Model-level parallelism enables training large models exceeding the size constraint of the device memory.

There are different forms of model-level parallelism and many of them are supported at the programming language level. Mesh-TensorFlow[13] is a language that built on top of Tensorflow that provides a general class of distributed tensor computations. While data-parallelism can be viewed as splitting tensors and operations along the "batch" dimension, in Mesh-TensorFlow the user can specify any tensor-dimensions to be split across any dimensions of a multidimensional mesh of processors. FlexFlow[2] introduces SOAP, a more comprehensive search space of parallelization strategies for DNNs which allows parallelization of a DNN in the Sample, Operator, Attribute, and Parameter dimensions. It uses guided randomized search to find a parallelization strategy.

GPipe[12] proposed pipeline parallelism, by automatically splitting a minibatch of training examples into smaller micro-batches. By pipelining the execution across micro-batches, accelerators can operate in parallel. PipeDream[19] introduces pipeline parallelism with more flexibility, allowing gradient updates of multiple mini-batches to happen in parallel. However, this introduces the staleness and consistency issue for weigh updates. In addition, both GPipe and PipeDream partition a model at the granularity of layers instead of operations. Instead of proposing different parallelism strategies or programming primitives, our work focuses on a general deep RL algorithmic solution for automating device placement at operation granularity.

Automatic Device Placement

RL has been used for device placement of a given dataflow graph[1] and demonstrated runtime reduction over human crafted placement and conventional heuristics. For improved scalability, a hierarchical device placement (HDP) strategy[3] has been proposed that clusters operations into groups before placing the operation groups onto devices. Spotlight[11] applies proximal policy optimization (PPO) and cross-entropy minimization to lower training overhead. Both HDP and Spotlight rely on LSTM controllers that are difficult to train and struggle to capture very long-term dependencies over large graphs. In addition, both methods are restricted to process only a single graph at a time, and cannot generalize to arbitrary and held-out graphs. Placeto[16] represents the first attempt to generalize device placement using a graph embedding network. But like HDP, Placeto also relies on hierarchical grouping and only generates placement for one node at each time step. Our approach leverages a recurrent attention mechanism and generates the whole graph placement at once. This significantly reduces the training time for the controller. We also demonstrate the generalization ability of SGDP over a wider set of important workloads.

Compiler Optimization

REGAL[8,9] uses deep RL to optimize the execution cost of computation graphs in a static compiler. The method leverages the policy's ability to transfer to new graphs to improve the quality of the genetic algorithm for the same objective budget. However, REGAL does not show strong

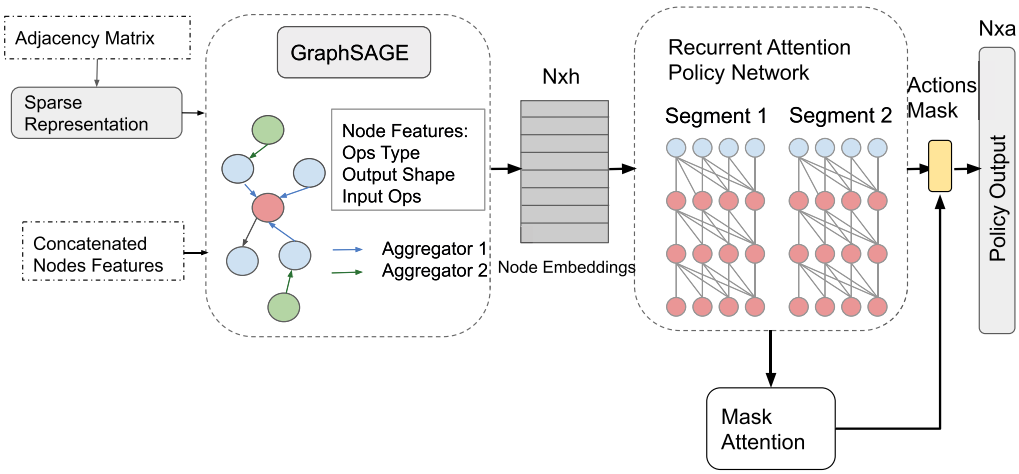

Figure 1. Overview of SGDP: An end-to-end placement network that combines graph embedding and sequential attention. N: Number of nodes. h: Hidden size. d: Number of devices.

empirical results on large graphs, especially graphs consisting of over 50 000 nodes such as 8-layer GNMT and 8-layer RNNLM. In addition, REGAL relies on a performance model to approximate the rewards, while we demonstrate superior performance using real machine measurements in an online learning fashion.

END-TO-END PLACEMENT POLICY

Given a dataflow graph $G(V, E)$ where V represents atomic computational operations (ops) and E represents the data dependence, our goal is to learn a policy $\pi : \mathcal{G} \mapsto \mathcal{D}$ that assigns a placement $D \in \mathcal{D}$ for all the ops in the given graph $G \in \mathcal{G}$, to maximize the reward $r_{G,D}$ defined based on the runtime. $\mathcal{D}$ is the allocated devices that can be a mixture of CPUs and GPUs. In this article, we represent policy π_θ as a neural network parameterized by θ.

Unlike prior works that focus on a single graph only, the RL objective in GDP is defined to simultaneously reduce the expected runtime of the placements over a set of N dataflow graphs

$$J(\theta) = \mathbb{E}_{G\sim\mathcal{G}, D\sim\pi_\theta(G)}[r_{G,D}] \approx \frac{1}{N}\sum_{G} \mathbb{E}_{D\sim\pi_\theta(G)}[r_{G,D}]. \quad (1)$$

In the following, we refer to the case when $N = 1$ as individual training and the case when $N > 1$ as *batch training*. We optimize the objective above using PPO[15] for improved sample efficiency.

Figure 1 shows an overview of the proposed end-to-end device placement network. Our proposed policy network π_θ consists a graph embedding network that learns the graphical representation of any dataflow graph, and a placement network that learns a placement strategy over the given graph embeddings. The two components are jointly trained in an end-to-end fashion. The policy $p(a|G)$ is applied to make a set of decisions at each node. These decisions, denoted as a_v for each $v \in V$ across all nodes, form one action $a = \{a_{v\in V}\}$. One decision corresponds to playing one arm of a multibandit problem, and specifying the entire a corresponds to playing several arms together in a single shot. Note the architecture is designed to be invariant over the underlying graph topology, enabling us to apply the same learned policy to a wide set of input graphs with different structures.

Graph Embedding Network

We leverage GNNs[4,5] to capture the topological information encoded in the dataflow graph. Most graph embedding frameworks are inherently transductive and can only generate embeddings for a given fixed graph. These transductive methods do not efficiently extrapolate to handle unseen nodes (e.g., in evolving graphs), and cannot learn to generalize to unseen graphs. GraphSAGE[4] is an inductive framework that leverages node attribute information to efficiently generate representations on previously unseen data. While our proposed framework is generic,

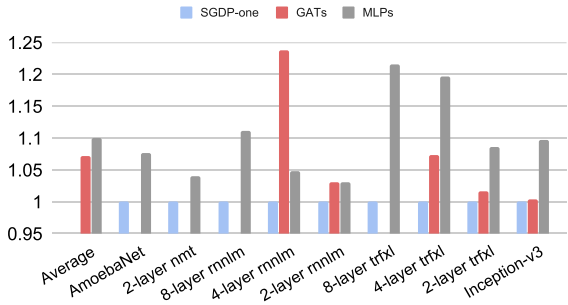

Figure 2. Normalized runtime to SGDP-one (the lower the better). Comparison of different policy network architectures: MLPs consists of graph embedding network with MLPs. GATs incorporates an attention of neighbor nodes into a GNN. SGDP-one is our method. Results for GATs on all 8-layer input graphs, GNMT, and AmoebaNet are missing as GATs fails to generate valid placements.

we adopt the feature aggregation scheme proposed in GraphSAGE to model the dependencies between the operations and build a general, end-to-end device placement method for a wide set of dataflow graphs.

In SGDP, nodes and edges in the dataflow graph are represented as the concatenation of their meta features (e.g., operation type, output shape, adjacent node ids) and are further encoded by the graph embedding network into a trainable representation. The graph embedding process consists of multiple iterations, and the computation procedure for the lth iteration can be outlined as follows.

First, each node $v \in V$ aggregates the feature representations of its neighbors, $\{h_u^{(l)}, \forall u \in \mathcal{N}(v)\}$, into a single vector $h_{\mathcal{N}(v)}^{(l)}$. This aggregation outcome is a function of all previously generated representations, including the initial representations defined based on the input node features. In this article, we use the following aggregation function with max pooling:

$$h_{\mathcal{N}(v)}^{(l)} = \max(\sigma(W^{(l)} h_u^{(l)} + b^{(l)}), \forall u \in \mathcal{N}(v)) \tag{2}$$

where $(W^{(l)}, b^{(l)})$ define an affine transform and σ stands for the sigmoid activation function. We then concatenate the node's current representation, $h_v^{(l)}$, with the aggregated neighborhood vector, $h_{\mathcal{N}(v)}^{(l)}$, and feed this concatenated vector through a fully connected layer $f^{(l+1)}$

$$h_v^{(l+1)} = f^{(l+1)}(\text{concat}(h_v^{(l)}, h_{\mathcal{N}(v)}^{(l)})). \tag{3}$$

Different from GraphSAGE, parameters in our graph embedding network are trained jointly with a placement network via stochastic gradient descent with PPO, in a *supervised* fashion, as described in the "End-to-End Placement Policy" section. That is, we replace the unsupervised loss with our task-specific objective.

Placement Network

The GNN works as a feature aggregation network that learns a trainable feature representation for the computational graph, we still need a policy network that produces actions on a per node basis. Given h_v's, the policy network produces a_v's through conditionally independent predictions, where the prediction for one node v does not depend on the prediction of other nodes

$$p(a|G) = \prod_v p(a_v|G) = \prod_v p(a_v|f(h_v)). \tag{4}$$

Next, we discuss the selection of a proper neural network model to create a policy network.

Multilayer Perceptron (MLPs) While f can be represented using MLPs, where the MLPs is shared across all nodes for prediction the placement output distributions. However, MPLs lack a dependence tracking mechanism across nodes. In practise, the placement of one node can be determined by the placement of another node, where the placed node may consume a large size of data produced by the other node.

LSTM The conventional LSTM models proposed for language tasks usually target a shorter sequence length. For example, in a language task, a typical sequence length is between a few hundred to a thousand. However, in the device placement problem, models truly needs device placement can consists of over 50 000 of nodes. HDP[3] has been proposed to address this issue, however, the proposed grouper network comes with limited flexibility and generality. For example, the grouper network leverages an aggregated feature representation by averaging feature vectors for nodes within the same group.

The nondifferentiable grouping procedure prevents training the graph-embedding and placement networks end-to-end.

Graph Attention An attention network can learn internode dependence and the relative importance of dependencies across an entire graph. An intuitive way is to incorporate an attention mechanism into the GNN to enable nodes to attend over their neighborhood's features, following a self-attention strategy. GATs[20] has several strengths: first, it is efficient and is parallelizable across node-neighbor pairs; second, it can specify arbitrary weights to the neighbors; third, the model can generalize to completely unseen graphs. However, GATs only supports local attention, as contrary to the long-term global attention using a conventional Transformer network. Moreover, it has limited scalability that cannot handle large graphs consisting of over 10 k's of nodes.

Our Method As we increase the input graph size, the complexities of a GNN and an attention network can scale up. However, due to the reduction used in the aggregation layers in GraphSAGE, the complexity usually scales linearly with the number of nodes. But for an attention network, the complexity is $\mathcal{O}(N^2)$. Therefore, designing a more scalable attention network is critical to large input graphs.

We propose to use a transformer-based attentive network to generate operation placements in an end-to-end fashion. As the graph embedding already contains spatial (topological) information for each node, we remove the positional embedding in the original transformer to prevent the model from overfitting node identifications. To capture long-term dependencies efficiently among a large set of nodes, we adopt segment-level recurrence introduced in Transformer-XL,[10,17] where hidden states computed for the previous set of nodes are cached (with gradient flows disabled) and reused as an extended context during the training of the next segment. This reduces the complexity to $\mathcal{O}(N^2/k)$, where k is the number of segments.

Besides achieving extra long context than a GAN, we empirically find the segment-level recurrent attention much faster than a conventional LSTM-based GNMT model. In our experimental evaluation, we compare both the performance and speedup of our placement network with that of the LSTM-based HDP.

To improve policy optimization for the big search space $[\mathcal{O}(d^N)]$, where d is the number of devices and N is the number of nodes, we apply an additional mask attention to the last layer of feature map generated by the recurrent attention policy network. The generated actions mask is position-wise multiplied with the actions to selectively choose nodes to place. Intuitively, this enables selecting important ops in the graph to change placements while minimizing the cuts for the entire graph.

EXPERIMENT

Experimental Setup

Workloads: We evaluate SGDP using the computational graphs of six diverse architectures from different domains. To create a larger set of workloads, we vary architectural parameters like the number of layers for each of these workloads. All our workloads are implemented in TensorFlow. Further details about the graphs is in Appendix A.

Runtime Measurement: For placement task, where TensorFlow provides an API for device assignment, our experiments are evaluated on actual hardware with configuration of one Intel Broadwell CPU and up to eight Nvidia P100 GPUs.

Baselines: We choose three different baselines against which we compare the performance of SGDP along various metrics in the "Performance on Individual Graphs" section. They are the default heuristics-based optimizations used in TensorFlow (METIS), a human-expert solution, and finally solutions found using a learning based strategy like HDP.[3] For sensitivity study in Section refsubsec:sensitivity, we compare against MPLs and GATs.[20]

Performance on Individual Graphs

We evaluate SGDP by training the model separately on six important deep learning computation graphs, including RNN Language Modeling, GNMT, Transformer-XL, Inception, AmoebaNet, and WaveNet. We name this approach *SGDP-one*. Since TensorFlow provides an API for assigning operation placement, all reported measurements for placement task are on real hardware. As

Table 1. Runtime comparison between SGDP-one, human expert, TensorFlow METIS, and HDP on six graphs (RNNLM, GNMT, Transformer-XL, Inception, AmoebaNet, and WaveNet). Search speedup is the policy network training time speedup compared to HDP (reported values are averages of six runs).

Model (#devices)	**SGDP-one** (s)	HP (s)	METIS (s)	HDP (s)	Runtime speedup over HP / HDP	Search speedup over HDP
2-layer RNNLM (2)	0.173	0.192	0.355	0.191	9.9% / 9.4%	2.95x
4-layer RNNLM (4)	0.210	0.239	0.503	0.251	13.8% / 16.3%	1.76x
8-layer RNNLM (8)	0.320	0.332	OOM	0.764	3.8% / 58.1%	27.8x
2-layer GNMT (2)	0.301	0.384	0.344	0.327	27.6% / 14.3%	30x
4-layer GNMT (4)	0.350	0.469	0.466	0.432	34% / 23.4%	58.8x
8-layer GNMT (8)	0.440	0.562	OOM	0.693	21.7% / 36.5%	7.35x
2-layer Transformer-XL (2)	0.223	0.268	0.37	0.262	20.1% / 17.4%	40x
4-layer Transformer-XL (4)	0.230	0.27	OOM	0.259	17.4% / 12.6%	26.7x
8-layer Transformer-XL (8)	0.350	0.46	OOM	0.425	23.9% / 16.7%	16.7x
Inception (2) b32	0.229	0.312	OOM	0.301	26.6% / 23.9%	13.5x
Inception (2) b64	0.423	0.731	OOM	0.498	42.1% / 29.3%	21.0x
AmoebaNet (4)	0.394	0.44	0.426	0.418	26.1% / 6.1%	58.8x
2-stack 18-layer WaveNet (2)	0.317	0.376	OOM	0.354	18.6% / 11.7%	6.67x
4-stack 36-layer WaveNet (4)	0.659	0.988	OOM	0.721	50% / 9.4%	20x
GEOMEAN	-	-	-	-	**20.5% / 18.2%**	**15x**

shown in Table 1, SGDP-one consistently outperforms human expert placement (HP), TensorFlow METIS placement, and HDP. Overall, SGDP-one achieves on average 20.5% and 18.2% run time reduction across the evaluated 14 graphs, compared to HP and HDP, respectively. Importantly, with the efficient end-to-end single-shot placement, SGDP-one has a 15× speedup in convergence time of the placement.

Scalability Analysis: SGDP is designed in a way to scale up to extremely large graphs, consisting of over 80 000 nodes (8-layer GNMT). Therefore, unlike any of the prior works including HDP,[3] REGAL,[9] and Placeto,[16] we can demonstrate super human performance on large graphs such as 8-layer GNMT (21.7%/36.5% better than HP/HDP) and 8-layer RNNLM (3.8%/58.1% better than HP/HDP). For all of the related SoTA work, Placeto[16] and REGAL[9] do not provide any results on 8-layer RNNLM or 8-layer GNMT (more than 50 000 nodes). HDP[3] reports inferior performance on 8-layer RNNLM and 8-layer GNMT than human placement.

Sensitivity Study on Model Architectures

We compare our method with two alternative architectures (MLPs and GATs), as explained in "Placement Network" section. An LSTM-based HDP has been compared in Table 1 and we will leave it out in this section. SGDP-one consistently outperforms both MLPs and GATs by an average of 10% and 7%. (We only include valid placements for GATs). GATs fails to generate valid placements for large graphs consisting of over 10 000 nodes, such as 8-layer RNNLM, 2-layer GNMT, 8-layer Transformer-XL, and AmoebaNet. The results imply that SGDP yields better performance with an attention network, as compared to MLPs, and provides better scalability with a decoupled segmented attention network, as compared to GATs.

Generalization

SGDP enables the training of multiple heterogeneous graphs in a single batch (*GDP-batch*). We empirically show that GDP-batch generates better placements for many workloads such as Transformer-XL (7.6%), WaveNet (15%), and 8-layer GNMT (8%). Table 2 compares the run time of 11 tasks using SGDP-batch. SGDP-batch yields slightly better runtime compared to SGDP-one in majority of the tasks, while being only slightly worse on AmoebaNet. Compared to training graphs separately, SGDP-batch reduces network parameters and enables transfer learning among different graphs.

We test on unseen graphs from different workloads by pretraining the SGDP on different subsets of five workloads, excluding the entire workload of the unseen graph. For example, for an RNNLM input graph, all RNNLM models are excluded from the pretraining dataset. During pretraining, we perturb the number of layers, hidden size, and batch size of the input graphs to augment the data and add more randomness to the input data. *SGDP-zeroshot* directly runs inference on the pre-trained SGDP model to generate placements for the target graph. *SGDP-finetune* further trains the pre-trained SGDP model for an additional 50 training steps and generates placements using the fine-tuned SGDP model. We find that SGDP-finetune almost matches the performance of SGDP-one, degrading the placement runtime on average by only 1.2% compared to SGDP-one and outperforms both human placement and HDP significantly. SGDP-zeroshot completely eliminates the training for the target unseen graphs, while being only 3.7% worse on average than SGDP-one and being over 10% better than human placement. This indicates that both graph embedding and the learned policies transfer and generalize to the unseen data.

CONCLUSION

We propose an efficient single-shot, generalized deep RL method (SGDP) and demonstrate superior performance on a wide set of representative deep learning models, including Inception-v3, AmoebaNet, RNNLM, GNMT, Transformer-XL, and WaveNet. Our method on average achieves 20% improvement over human experts and 18% improvement over the prior art with 15× faster convergence, being the first to demonstrate super human performance on large graphs consisting of over 50 000 nodes.

Table 2. Runtime comparison on SGDP-batch versus SGDP-one.

Model	Speedup	Model	Speedup
2-layer RNNLM	0	Inception	0
4-layer RNNLM	5%	AmoebaNet	−5%
2-layer GNMT	0	4-stack 36-layer WaveNet	3.3 %
4-layer GNMT	0	2-stack 18-layer WaveNet	15%
2-layer Transformer-XL	7.6%	8-layer Transformer-XL	1.5%
4-layer Transformer-XL	3%		

APPENDIX A
INPUT GRAPHS

We used a variety of widely used workloads from computer vision, speech, and NLP. In this section, we give a detailed explanation on the selected models and hyperparameters.

Inception-V3

Inception-V3 is a multibranch convolutional network used for a variety of computer vision tasks, including classification, recognition, or generation. The network consists of blocks made of multiple branches of convolutional and pooling operations. Within a block, the branches of ops can be executed in parallel. However, the model is mostly sequential as the outputs of each block are concatenated together to form the input to the next block. We use a batch size of 64. The TensorFlow graph of this model contains 24 713 operations.

AmoebaNet

AmoebaNet is an automatically designed neural network that yields SOTA performance on ImageNet. Similar to Inception-V3, it contains Inception-like blocks called cells, which receives a direct input from the previous cell and a skip input from the cell before it. We use a batch size of 64. The TensorFlow graphs contains 9430 operations.

Recurrent Neural Network Language Model

RNNLM is made of many LSTM cells organized in a grid structure. The processing of each LSTM

Table 3. Hyperparameters for policy network. gs_layers**: GraphSAGE layers,** gs_knn**: GraphSAGE maximum neighbors,** trf_d_model**: Dimension of the Transformer-XL model,** trf_n_head**: Number of attention heads,** trf_layers**: Number of Transformer-XL layers,** trf_d_heads**: Dimension of each attention head,** trf_d_inner**: Dimension of inner hidden size in positionwise feed-forward.**

Parameters	Value	Parameters	Value
gs_layers	4	gs_dim	128
gs_knn	5	trf_layers	2
trf_d_model	128	trf_n_head	5
trf_d_head	25	trf_d_inner	256

Table 4. Hyperparameters for PPO.

Parameters	Value	Parameters	Value
learing rate	0.5	*num of rollouts*	800
minibatches	40	*epochs*	20
epsilon	0.2	*entropy*	0.5
optimizer	Adam		

cell only depends on the results of 2 other cells (from the previous layer, and from the previous time step), which make the concurrent execution of many LSTM cells possible given enough hardware resources. We use batch size 64 and a hidden size of 2048. The corresponding TensorFlow graph contains 9021 operations for a 2-layer model. The number of ops grow roughly proportional with the number of layers.

GNMT

Neural machine translation with attention mechanism has an architecture similar to that of RNNLM, but its many hidden states make it far more computationally expensive than RNNLM. We use batch size 64. The original 2-layer encoder-decoder consisting of 28 044 operations. An extended 4-layer version consisting of 46 600 operations, An even larger 8-layer version consisting of 83 712 operations.

Transformer-XL

Transformer-XL[10] is an modified version of Transformer[18] that supports segment-level recurrence and a novel positional encoding scheme. This innovation enables learning dependence that is 80% longer than RNNs, and 450% longer than vanilla Transformers. We use a Transformer-XL with batch size of 64, sequence length of 256, segment length of 64, model hidden dimension of 500 and feed forward hidden dimension of 1000, 10 heads, and head dimension of 50. The 2-layer Transformer-XL contains 2618 operations. The number of ops grow roughly proportional with the number of layers.

WaveNet

WaveNet is a generative model for speech synthesis. The model is fully probabilistic and autoregressive, with the predictive distribution for each audio sample conditioned on all previous ones. We use a WaveNet model with batch size 64 and a receptive field size of 2048 (9-layers per stack). An 5-stack WaveNet contains 4374 operations and a 10-stack WaveNet contains 8516 operations.

Appendix B
Hyperparameters

We list out all the selected hyperparameters in our experiments for reproducibility in Tables 3 and 4.

ACKNOWLEDGMENTS

This work was done during internship at Google.

REFERENCES

1. A. Mirhoseini *et al.*, "Device placement optimization with reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2017.
2. Z. Jia *et al.*, "Beyond data and model parallelism for deep neural networks," in *Proc. 35th Int. Conf. Mach. Learn.*, 2018.
3. A. Mirhoseini *et al.*, "A hierarchical model for device placement," in *Proc. Int. Conf. Learn. Representations*, 2018.
4. W. L. Hamilton *et al.*, "Inductive representation learning on large graphs," in *Proc. Conf. Neural Inf. Process. Syst.*, 2017.
5. K. Xu *et al.*, "How powerful are graph neural networks?," in *Proc. Int. Conf. Learn. Representations*, 2019.
6. H. Joel *et al.*, "Deep learning scaling is predictable, empirically," 2017, *arXiv:1712.00409*.
7. S. Noam *et al.*, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," in *Proc. Int. Conf. Learn. Representations*, 2017.

8. A. Paliwal *et al.*, "REGAL: Transfer learning for fast optimization of computation graphs," *Knowl. Discovery Database*, 2019.
9. A. Paliwal *et al.*, "Reinforced genetic algorithm learning for optimizing computation graphs," in *Proc. Int. Conf. Learn. Representations*, 2020.
10. Z. Dai *et al.*, "Transformer-XL: Attentive language models beyond a fixed-length context," in *Proc. 57th Annu. Meeting Assoc. Comput. Linguistics*, 2019, pp. 2978–2988.
11. I. Sutskever *et al.*, "Spotlight: Optimizing device placement for training deep neural networks," in *Proc. Conf. Neural Inf. Process. Syst.*, 2018.
12. Y. Huang *et al.*, "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. NeurIPS*, 2019.
13. N. Shazeer *et al.*, "Mesh-tensorflow: Deep learning for supercomputers," in *Proc. NeurIPS*, 2018.
14. B. Cheung *et al.*, "Superposition of many models into one," in *Proc. NeurIPS*, 2019.
15. J. Schulman *et al.*, "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.
16. R. Addanki *et al.*, "Placeto: Learning generalizable device placement algorithms for distributed machine learning," in *Proc. NuerIPS*, 2019.
17. Z. Dai, "Improving deep generative modeling with applications," 2019.
18. V. Ashish *et al.*, "Attention is all you need," in *Proc. Conf. Neural Inf. Process. Syst.*, 2017.
19. D. Narayanan *et al.*, "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. Symp. Oper. Syst. Principles*, 2019.
20. P. Veličković *et al.*, "Graph attention networks," in *Proc. Int. Conf. Learn. Representations*, 2018.

Yanqi Zhou is currently a Research Scientist with Google Brain. She works on generalizing machine learning to optimize systems problems, including compiler graph optimizations and hardware accelerator design. In addition, she builds large-scale deep learning models for speech and language tasks. Zhou received her Ph.D. degree from Princeton University, working on configurable computer architectures and resource provisioning for clouds. Contact her at yanqiz@google.com.

Sudip Roy is currently a Senior Research Scientist with Google AI. He is interested in design and development of systems for machine learning and also in applying machine learning to solve optimization problems that arise in systems. He has worked on a variety of problems in this space including infrastructure for large-scale distributed machine learning, data management solutions for managing data lakes, using reinforcement learning to solve optimization problems in machine learning compilers, and geo-replicated transaction processing systems. Roy received a Ph.D. degree in computer science from Cornell University. Contact him at sudipr@google.com.

Amirali Abdolrashidi is currently working toward the Ph.D. degree in computer science and engineering from the University of California, Riverside, where he works on speeding up data-dependent workloads on GPU architectures. Abdolrashidi received the M.S. degree in electrical engineering from New York University, in 2014, and during his Software Engineering Internship with Google, he worked to improve the performance of deep learning applications through prioritized fusion of operations. Contact him at abdolrashidi@gmail.com.

Daniel Lin-Kit Wong is currently working toward the Ph.D. degree with the Computer Science Department, Carnegie Mellon University. He is a systems builder and hacker with a focus on systems design and distributed systems. His research focus has been machine learning for caching. Contact him at wonglkd@gmail.com.

Peter Ma is currently a Software Engineer with Google, and he works on machine learning accelerator architecture and machine learning platforms performance. Ma received the Ph.D. degree in mechanical engineering with a Ph.D. Minor in computational and mathematical engineering from Stanford University. Contact him at pcma@google.com.

Qiumin Xu is currently a Senior Software Engineer with Google Brain, working on the performance of machine learning accelerators. Xu received the Ph.D. degree in electrical engineering from the University of Southern California. Contact her at qiuminxu@google.com.

Azalia Mirhoseini is currently a Senior Research Scientist with Google Brain, where she works on deep reinforcement learning based approaches to solve problems in computer systems. Mirhoseini received the Ph.D. degree in electrical and computer engineering from Rice University. She was the recipient of a number of awards, including the *MIT Technology Review* 35 under 35 award, the Best Ph.D. Thesis Award at Rice, and a Gold Medal in the National Math Olympiad in Iran. Her work has been covered in various media outlets including *MIT Technology Review* and *IEEE Spectrum*. Contact her at azalia@google.com.

James Laudon is currently an Engineering Director with Google Brain. His research interests focus on hardware and software co-design for high-performance systems and he is currently working on domain-specific computer architectures for machine learning. Before joining Google Brain, he was Founder and Site Director for the Google Madison office. He has contributed to the architecture and implementation of multiple computer systems including the Stanford DASH, SGI Origin 2000, and Sun UltraSPARC T1. Laudon received the B.S. degree in electrical engineering from the University of Wisconsin—Madison and the M.S. and Ph.D. degrees in electrical engineering from Stanford University. Contact him at jlaudon@google.com.