A Hierarchical Model for Device Placement

Azalia Mirhoseini*, Anna Goldie*, Hieu Pham, Benoit Steiner, Quoc V. Le and Jeff Dean
*Equal Contribution
Google Brain
{azalia,agoldie,hyhieu,bsteiner,qvl,jeff}@google.com

1 INTRODUCTION

We propose a hierarchical model for efficient placement of computational graphs onto hardware devices, especially in heterogeneous environments with a mixture of CPUs, GPUs, and other computational devices.

Device placement can be framed as learning to partition a graph across available devices, making traditional graph partitioning methods [6, 12, 15] a natural baseline. A well-established prior work is Scotch [15], an open source library for graph partitioning, which includes optimizations such as k-way Fiduccia-Mattheyses [6], Multilevel methods [3, 8, 11], Band Method [5], the Diffusion Method [14], and Dual Recursive Bipartitioning Mapping [17].

Using deep networks and reinforcement learning for combinatorial optimization has already been proposed [4, 13, 20]. ColocRL [13] uses a recurrent neural network (RNN) policy network to predict the placement of operations in a computational graph. While this approach outperforms traditional graph partitioning heuristics and human expert placements, it is limited to small graphs (with fewer than 1000 nodes) and requires human experts to manually partition the graph into collocation groups as a pre-processing step.

In this paper, we introduce a more flexible, end-to-end approach which learns to optimize device placement for neural networks that have tens of thousands of operations. Unlike previous methods which require human experts to feed in properties of the hardware or manually cluster operations, our method is automated and scales to much larger computational graphs and novel hardware devices. Our approach finds non-trivial placements over multiple devices for models such as Inception-V3 [19], ResNet [7], Language Modeling [10], and Neural Machine Translation [22]. The placements found by our model outperform TensorFlow's default placements [1], the Scotch algorithm's placements, and human expert placements, achieving runtime reductions of up to 60.6% per training step.

2 METHOD

We train a hierarchical policy network that generates optimized placements. Our policy consists of two sub-networks: a Grouper that assigns operations in an input TensorFlow graph to groups and a Placer that assigns groups to target devices. We use a policy gradient approach to jointly train the two sub-networks, incorporating the runtime of the predicted placements as our reward, Figure 1.

The objective of the proposed approach, which we refer to as the Hierarchical Planner, is to minimize the runtime for one forward pass, one back-propagation pass, and one parameter update of the target neural network. To measure runtime, predicted placements are run on actual hardware.

The Grouper is a feed forward model and the Placer is a sequence-to-sequence model [18] with Long Short-Term Memory [9] and a content-based attention mechanism [2]. To represent operations as inputs to the Grouper, we encode information about the operation, including type (e.g., MatMul, Conv2d, Sum, etc.), size and number of outputs, as well as connections to other operations. We create group embeddings by combining the embeddings of the member operations. More precisely, each group embedding is the concatenation of three components: the average of the member operation type embeddings, the average of the member operation sizes and number of outputs, and intra-group and inter-group connectivity information encoded as an adjacency matrix.

The Placer's RNN encoder reads the group embeddings one at a time and produces M hidden states. We treat M, which is equal to the number of groups, as a hyperparameter. The Placer's decoder RNN predicts one device per time step. The devices are returned in the same order as the input group embeddings, i.e., the operations in the first group will be placed on the device returned by the first decoder step, and so on. Each device has its own trainable embedding, which is then fed as input to the next decoder time step.

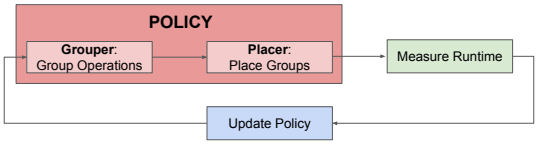

Figure 1: Hierarchical model for device placement.

The planner optimizes the training time for a target model (e.g., a TensorFlow graph) given the decisions made by the Grouper and Placer. Let r_d be the runtime per training step for a predicted device placement d. We define the reward for placement d as $R_d = -sqrt(r)$. The planner should try to maximize the expectation of R_d given its decisions. As such, the cost function we are optimizing for is:

Tasks	CPU Only	GPU Only	#GPUs	Human Expert	Scotch	MinCut	Hierarchical Planner	Runtime Reduction
Inception-V3	0.61	0.15	2	0.15	0.93	0.82	**0.13**	16.3%
ResNet	-	1.18	2	1.18	6.27	2.92	**1.18**	0%
RNNLM	6.89	1.57	2	1.57	5.62	5.21	**1.57**	0%
NMT (2-layer)	6.46	OOM	2	2.13	3.21	5.34	**0.84**	60.6%
NMT (4-layer)	10.68	OOM	4	3.64	11.18	11.63	**1.69**	53.7%
NMT (8-layer)	11.52	OOM	8	**3.88**	17.85	19.01	4.07	-4.9%

Table 1: Model Runtimes (seconds) for different placements (lower is better). OOM: Out Of Memory.

$$J(\theta_g, \theta_d) = \mathbf{E}_{\mathbf{P(d;\theta_g,\theta_d)}}[R_d] = \sum_{g\sim\pi_g} \sum_{d\sim\pi_d} p(g;\theta_g)p(d|g;\theta_d)R_d$$

Let θ_g and θ_d be parameters of the Grouper and Placer, respectively. Here, $p(g;\theta_g)$ is the probability of a sample group assignment g drawn from the Grouper softmax distribution $\sim \pi_g$ and $p(d;\theta_d)$ is the probability of a sample device placement d drawn from the Placer softmax distribution $\sim \pi_d$. We use the REINFORCE rule [21] to optimize the cost function.

Our policy is trained in a distributed manner with a parameter server that is shared among several controllers. The controllers update the policy asynchronously. We use 4 controllers and 16 workers (4 per controller). Each worker executes the placement given by its controller and reports the runtime. Each controller is hosted on a single GPU. The workers run the placements in parallel. Once all workers have finished running the placements, the controller computes the gradients using the measured runtimes.

3 EXPERIMENTS

We evaluate our method on four widely used neural network models: Inception-V3 (batch size=32) [19] with 24,713 operations, ResNet (batch size=128) [7] with 20,586 operations, RNNLM (batch size=64) [10] with 9,021 operations and NMT (batch size=64) [2, 23] with 2, 4, and 8 layers of encoder-decoder with 28,044, 46,600, and 83,712 operations respectively.

We compare our results against the following methods: CPU and GPU only baselines where the entire model is placed on a single CPU or GPU respectively. Scotch static mapper [16] which takes as input the graph, the computational cost of each operation and the compute and communication capacities of the pertinent devices. The Mincut baseline is similar to Scotch but we only consider GPUs as our devices. We use hand-crafted placements from previous publications. For Inception-V3 and Resnet human experts place the graph on a single GPU. For RNNLM and NMT, existing work [18, 23] places each LSTM layer on a separate GPU.

Our experiments are run on machines with 1 Intel Haswell 2300 CPU and up to 8 Nvidia Tesla K40 GPUs. We use TensorFlow r1.3 to run our evaluations.

Results: In Table 1, we report the performance of the Hierarchical Planner. The only information available to our method is the TensorFlow graph and a list of devices. The reduction percentages are computed by taking the difference between the runtime achieved by the Hierarchical Planner and that of the best prior placement, and then dividing it by that best prior runtime. For each of the models, we train a new policy which learns to optimize placements for that particular model. All results are after 1000 iterations of updating the policy. In practice, this takes at most three hours for our largest benchmark. The policy itself is a lightweight network that is trained on a single GPU.

For ResNet and RNNLM, our model learns that it is more efficient to use a single GPU, as this minimizes communication cost. For Inception-V3, the Hierarchical Planner learns to distribute the model across 2 GPUs, achieving a 16.3% reduction in runtime over placing the model on a single GPU. For NMT with 2, 4, and 8 layers, we ran experiments with 2, 4, and 8 GPUs, respectively. We outperform the best prior results by 60.6% for NMT (2-layer) and 53.7% for NMT (4-layer). For NMT (8-layer), the Hierarchical Planner finds a placement that is 4.9% slower than that of human experts. Even in this one case where the method slightly underperforms, it is still useful to have an automated method of finding placements that are comparable to those of human experts.

Results associated with both Scotch and MinCut were significantly worse than human expert baselines, which is consistent with results reported in [13].

Given that we train target neural networks for hundreds of thousands of steps, the overhead of policy training is justified. For example, to train WMT'14 En->Fr dataset which has more than 36 million examples for one epoch (with batch-size=64), we need to run the NMT model for approximately 562500 steps. Since we reduce the runtime per step from 3.64 to 1.69 seconds, this saves us 304 GPU-hours in each epoch, which is significant even if we consider the roughly 102 GPU-hours we spent on training the policy.

4 CONCLUSION

We present a hierarchical method for efficiently placing the operations of a computational graph onto devices. Our method is entirely end-to-end and scales to computational graphs containing over 80,000 operations. Our approach finds highly granular parallelism within the graph, enabling us to outperform prior methods by up to 60.6%.

REFERENCES

[1] MartÃŋn Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *OSDI*.

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *ICLR*.

[3] S. T. Barnard and H. D. Simon. 1994. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: practice and Experience, 6(2):101âĂŞ117* (1994).

[4] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. 2016. Neural Combinatorial Optimization with Reinforcement Learning. *arXiv preprint arXiv:1611.09940* (2016).

[5] C. Chevalier and F. Pellegrini. 2006. Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multilevel framework. *EuroPar, Dresden, LNCS 4128*, 243âĂŞ–252.

[6] Charles M Fiduccia and Robert M Mattheyses. 1988. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*. ACM, 241–247.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR*.

[8] B. Hendrickson and R. Leland. 1993. A multilevel algorithm for partitioning graphs. *Technical Report SAND93âĂŞ1301, Sandia National Laboratories* (June 1993).

[9] Sepp Hochreiter and Juergen Schmidhuber. 1997. Long short-term memory. *Neural Computation* (1997).

[10] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410* (2016).

[11] G. Karypis and V. Kumar. 1995. A fast and high quality multilevel scheme for partitioning irregular graphs. *Technical Report 95-035, University of Minnesota* (June 1995).

[12] George Karypis and Vipin Kumar. 1995. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. (1995).

[13] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Mohammad Norouzi, Naveen Kumar, Rasmus Munk Larsen, Yuefeng Zhou, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*.

[14] F. Pellegrini. 2007. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. *EuroPar, Rennes, LNCS 4641*, 191âĂŞ–200.

[15] François Pellegrini. 2009. Distillating knowledge about Scotch. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[16] F. Pellegrini. 2009. Distillating knowledge about SCOTCH. (2009).

[17] F. Pellegrini and J. Roman. 1996. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. *Research Report, LaBRI, Universite Bordeaux I* (August 1996).

[18] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *NIPS*.

[19] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *CVPR*.

[20] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *NIPS*.

[21] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*.

[22] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, et al. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144* (2016).

[23] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, ÅĄukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144* (2016).