Zero-Shot Learning for Fast Optimization of Computation Graphs

Aditya Paliwal[*]
Google Research
adipal@google.com

Felix Gimeno
DeepMind
fgimeno@google.com

Vinod Nair
DeepMind
vinair@google.com

Yujia Li
DeepMind
yujiali@google.com

Miles Lubin
Google Research
mlubin@google.com

Pushmeet Kohli
DeepMind
pushmeet@google.com

Oriol Vinyals
DeepMind
vinyals@google.com

Abstract

We present a deep reinforcement learning approach to minimizing the execution cost of neural network computation graphs in an optimizing compiler. Unlike earlier learning-based works that require expensive online training steps, we propose a "zero-shot learning" approach that trains an optimizer offline that successfully generalizes to previously unseen graphs. This allows our approach to produce high-quality execution decisions on real-world TensorFlow graphs in seconds instead of hours. We consider two optimization tasks for computation graphs: minimizing running time and peak memory usage. In comparison to an extensive set of baselines, our approach achieves significant improvements over classical and other learning-based methods on these two tasks.

1 Introduction

Recent years have seen the emergence of optimizing compilers for Deep Learning (DL)—such as Glow [24], MLIR [3], TVM [8], and XLA [27]—that aim to efficiently map neural network computation graphs expressed in high-level frameworks like TensorFlow [5] onto hardware. The complex discrete decisions made by these compilers are themselves a natural target for studying how machine learning can assist in solving combinatorial optimization problems [7].

In this work, we look at applying learning to jointly optimize what are currently two passes in the XLA compiler: operation-to-device assignment and static scheduling of operations within a device, where a device may be a GPU or tensor processing unit (TPU). These optimization problems have been studied classically as *task scheduling* and are known to be NP-hard in typical settings [26, 18].

The setting of an optimizing compiler imposes several challenging constraints on the learning methodology that can be employed. The solution must 1) be fast enough to avoid unacceptably large compilation delays, 2) work across diverse families of computation graphs, and 3) scale to large graphs and (hence) decision spaces. The runtime constraints led us to adopt a *zero-shot learning* approach where only a single inference pass is needed when a new task (i.e., optimization problem) is given. These constraints also preclude the use of expensive simulators or hardware evaluations;

[*]Google AI Resident

Preprint. Under review.

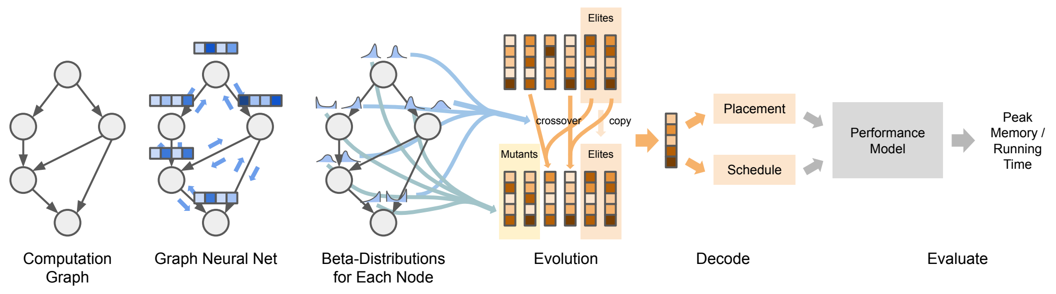

Fig. 1: Given a computation graph, we run a GNN to predict beta distributions for each node. These distributions are used by BRKGA to generate mutants in the evolution process. The evolution process decodes chromosomes into placements (device assignments) and schedules for the computation graph and evaluates their fitness using a cost model.

instead we follow the typical approach in optimizing compilers of constructing a coarse cost model to evaluate execution decisions and optimizing with respect to it, as done in [1, 17]. To satisfy the second and third requirements, the learned optimizer must be able to generalize to previously unseen computation graphs.

Existing work does not adequately address this setting. [22, 21] apply RL to the op-to-device assignment problem, although their solution does not generalize across computation graphs and hence requires hours for each instance. Their results were recently improved by [2] who demonstrate limited generalization within families of neural network architectures. AutoTVM [8] applies learning to optimize low-level implementations for tensor ops but does not consider op-to-device assignment or scheduling of entire programs. In a related domain, [20] learn scheduling algorithms for data processing clusters.

The key idea of our approach (see Figure 1) is to augment an existing proven classical heuristic with learning; we combine a Graph Neural Network (GNN) [10] with a heuristic based on Biased Random-Key Genetic Algorithms (BRKGA) [14]. The GNN takes a computation graph as input and returns parameters for BRKGA's mutant-sampling distributions. BRKGA is run using these input-dependent distributions, searching for good solutions via the cost model. Since explicit supervision is not available, the GNN is trained using REINFORCE [28] using BRKGA's final objective value as the reward. Our approach, "REINFORCE-based Genetic Algorithm Learning" (REGAL), significantly improves solution quality of the heuristic and allows zero-shot generalization to new graphs. The results are promising, yet they remain to be validated by integration back into a compiler like XLA.

This work makes the following contributions: 1) We are the first to demonstrate zero-shot learning with broad generalization for optimizing real-world TensorFlow graphs on two separate tasks of minimizing runtime and peak memory usage. 2) We demonstrate that BRKGA can be significantly improved using GNNs. 3) Our approach is made possible by a dataset of 372 topologically distinct real-world TensorFlow graphs, which is an order of magnitude larger than the datasets used in previous work [22, 21, 9, 1]. 4) We present extensive comparisons with heuristic and exact (i.e., enumerative) baselines.

2 Approach

Figure 1 shows an overview of the pipeline for our approach. The next three subsections explain its key components in more detail.

2.1 The problem setup

We consider a computation graphs as a collection of ops, each of which may produce or consume a number of tensors, free of control flow. These are like *Blocks* in MLIR [4], but without execution order specified. More concretely, for learning, we represent a computation graph as an attributed multigraph $G = (V, E)$, where, the nodes V correspond 1:1 to the ops, and the edges $e \in E$ exist from u to v for each tensor that op v requires that is produced by op u. As a tensor can be required by multiple ops, the correspondence from edges to tensors may be many to one. Each node $v \in V$ and edge $e \in E$ has an attribute vector $\boldsymbol{x}_v$ and $\boldsymbol{x}_e$. The attributes contain respective features, e.g., sizes of the tensors and runtime estimates of the ops.

We assume that we are given d homogeneous devices (like GPUs or TPUs) on which to execute the computation graph. The set of decisions to be made are 1) which op to run on which device, and 2) the order in which to run operations. In XLA, decisions 1) and 2) are currently made in separate stages, yet here we consider them jointly. The two objectives considered are minimizing the execution time of the graph and minimizing the peak memory use (the amount of memory needed to run the graph). The objectives are evaluated based on a cost model; see the appendix for a description.

2.2 The BRKGA-based heuristic

We build our work on top of the BRKGA-based heuristic that is implemented in the XLA compiler to compute within-device schedules, generalizing it to also compute op-to-device assignments. BRKGA is a general meta-heuristic framework that has obtained state of the art results for a broad range of combinatorial optimization problems [14, 13], making it a promising target to improve by learning.

Here we present only the aspects of the BRKGA-based heuristic relevant to discussing how we extend it with learning, with some parts simplified; see the appendix for more details. In the heuristic, solutions are encoded as vectors in $[0, 1]^{|V|*(d+1)}$ called *chromosomes*. One entry per node corresponds to the node's scheduling priority, and the remaining d entries correspond to its affinity to be placed on each of the d devices, values that are used in decoding the chromosome into schedules and op-to-device assignments.

As a genetic algorithm, BRKGA generates random chromosomes called *mutants* within its search procedure. Traditionally, each entry of these chromosomes is sampled uniformly at random from $[0, 1]$. To provide knobs for learning, we generalize this so that each entry in the chromosome is sampled from a beta distribution with its own parameters. We write the heuristic as a function $H(G, D)$ that returns the objective value of the best solution found for an input computation graph $G = (V, E)$ given a list of $(d + 1) * |V|$ beta distributions $D = D_1, D_2, \dots, D_{|V|*(d+1)}$ used for mutant sampling. Adjusting these distributions allows learning to guide the search.

2.3 Integrating learning with BRKGA

We use a model based on Graph Neural Networks (GNNs) [25, 19, 12, 10] to propose the *instance-dependent* mutant-sampling distributions D to BRKGA. The learning model can be understood as a contextual bandit that takes the computation graph as input and computes categorical distributions over the choice of the beta distributions.

Each beta distribution D_i has two parameters $\alpha_i, \beta_i \in \mathbb{R}_{\geq 0}$. We discretize the possible values for these parameters. The bandit must produce $k = 2 \times (d+1) \times |V|$ actions for each input G. The GNN computes an embedding vector $\boldsymbol{h}_v$ for each node $v \in V$. These node-level embedding vectors are then fed into $2 \times (d+1)$ multi-layer perceptrons (MLP) to compute the logits for k categorical distributions. After sampling α and β from these distributions, the corresponding beta distributions D are passed to BRKGA. Thus, REGAL can be written as a function $M(G) = H(G, D \sim B(G))$, where B is the contextual bandit from which we sample the list of beta distributions D. In Reinforcement Learning terminology, the sampled beta distribution parameters are the model's actions. Observe that the action space is different from graph to graph; The GNN-based bandit can adapt to graphs of varying sizes, although it assumes that the value of d is fixed. Different models could be trained for different values of d, although we did not try this.

BRKGA is run with the sampled beta distributions D for a fixed iteration limit, and the final objective value is used to compute the reward. To make the reward values comparable across different graphs, we divide the objective value $M(G) = H(G, D \sim B(G))$ achieved on a graph G with action D, by $H(G, U)$, where U is the uniform random distribution, i.e., BRKGA without learning. Since we want to minimize the objective, we define the reward as $r(D, G) = -M(G)/H(G, U)$. So a reward > -1 corresponds to an action that achieves a better objective value with learning.

We maximize the expected reward $L = \mathbb{E}_G [p(D|G) r(D, G)]$, where $\mathbb{E}_G$ is an expectation over graphs in our training set. Learning is done by REINFORCE [28]. We add a scalar baseline $b(G)$ for reducing the variance of the gradient estimates.

Table 1: % Gap from best known solution for various methods on the datasets. Results are averages over test set graphs. Lower is better. Note: CP, an enumerative algorithm, is run for up to 24 hours only to establish provably global optima (if possible) for evaluation purposes.

Algorithm	TF Runtime test set	TF Peak Memory test set	Synthetic Runtime test set
CP 24hr	1.00%	8.06%	0.00%
GP + DFS	66.98%	14.77%	93.66%
Local Search	22.63%	7.24%	24.60%
BRKGA 5k	20.19%	7.98%	24.63%
Tuned BRKGA	16.40%	7.11%	20.76%
GAS	15.24%	7.67%	23.48%
IDRS	28.60%	12.39%	39.72%
REGAL	**11.04%**	**4.44%**	**18.57%**

3 Experimental results

In this section we describe the datasets, the baseline algorithms we compare to, and quantitative evaluation of REGAL. BRKGA is run for 5k evaluations during training and test time. We optimize for $d = 2$ devices.

3.1 Tasks and datasets

We consider two tasks, minimizing peak memory and minimizing running time, both on two homogeneous devices with 16 GiB of memory each and synchronous tensor transfers with zero cost (zero latency and infinite bandwidth). We train a separate neural network for each task-dataset pair. Recall that we are evaluating objectives on a cost model and not actual hardware.

We have collected a dataset of 372 topologically-unique real-world TensorFlow graphs by mining machine learning jobs on Google's internal compute cluster. These graphs are augmented by adding multiplicative noise. The final *TF runtime dataset* has 16329 training, 5470 validation, and 5266 test graphs. The *TF peak memory dataset* has 22400 training, 7400 validation, and 7400 test graphs. For reproducibility, we have generated released a synthetic dataset of computation graphs with 10000 training, 1000 validation, and 1000 test cases at https://github.com/deepmind/deepmind-research/tree/master/regal.

3.2 Baselines

We consider the following baselines: (1) a *Graph Partitioning + Depth First Search (GP+DFS)* heuristic similar to the current approach in XLA that separates device assignments and scheduling into two passes [6] (the GP algorithm is analogous to the Scotch [23] baseline used in [1]), (2) a *Local Search* heuristic that starts with a random solution and greedily improves, (3) *Graph-As-Sequence Model (GAS)*: Like [22, 21], the graph is converted a sequence using a topological sort and then an RNN predicts node-level distributions used by BRKGA, (4) BRKGA run for X thousand iterations (*BRKGA XK*) with uniform sampling distributions, (5) BRKGA with hyperparameters tuned by grid search (*Tuned BRKGA*), (6) *Instance-dependent Random Search (IDRS)* is the same as REGAL, but BRKGA is replaced with only random sampling, (7) *Constraint Programming (CP)* [15], an enumerative approach to establish provably global optima run for up to 24 hours. We fix the number of cost model evaluations to 5,000 for algorithms (2)-(6).

3.3 Comparison to baseline algorithms

We compare algorithms by measuring the *average percent gap from the best known solution*. If CP finishes within the time limit, it will always find the globally optimal solution. If not, one of the other methods may result in the best known. We report geometric averages over test set graphs.

Table 1 compares REGAL to other algorithms on all three datasets. REGAL outperforms all the baselines on all three tasks. It gives $1.38\times$ and $1.6\times$ larger improvements than the next best algorithm on runtime and peak memory minimization tasks, respectively. REGAL reduces the gap from the best known solution by about $1.8\times$ with respect to BRKGA 5K on both TensorFlow test sets, and

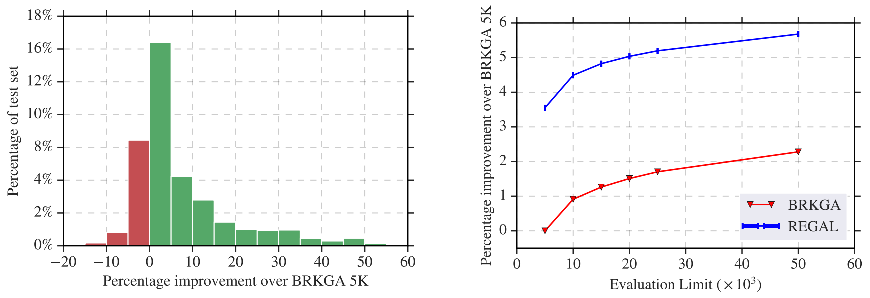

Fig. 2: (Left) Histogram of % improvements on the TensorFlow peak memory dataset for test graphs on which REGAL is better (green) and worse (red) than BRKGA. Ties are omitted from the figure for clarity but are included in the histogram percentage calculation. (Right) Average % improvement over BRKGA 5K given by REGAL and BRKGA on the TensorFlow peak memory dataset as the evaluation limit is increased.

by about 6× and 3.3× with respect to GP + DFS on the TensorFlow Runtime and Peak Memory test sets, respectively. The synthetic test set shows similar results. The learned policy successfully generalizes to previously unseen graphs, to the extent that a large fraction of the estimated room for improvement over BRKGA 5K is captured by REGAL using the *same* evaluation limit.

Comparing REGAL directly with BRKGA, Figure 2 (left) shows histograms of % improvements in peak memory achieved by REGAL over BRKGA 5K on the test sets. REGAL improves as much as 54.3% over BRKGA, while the worst regression is 17.9%. To assess whether the improvements provided by REGAL's policy generalize to evaluation limits other than the one for which it was trained (5K), we varied the evaluation limit used by both BRKGA and REGAL at test time. The results are shown in Figure 2 (right). REGAL's performance improves with more evaluations, confirming that the policy generalizes to higher evaluation limits. Interestingly, even with 50,000 evaluations, BRKGA is not able to match REGAL's performance with just 5,000 evaluations!

BRKGA 5K's average runtime on the test set is 0.87 seconds, while REGAL's, with the same evaluation limit, is 1.01 seconds. Surprisingly the difference is due to the additional cost of sampling from beta distributions (which we did not optimize), not from the cost of evaluating the GNN.

4 Conclusions and future work

By training a GNN policy to predict input-dependent mutant sampling distributions for BRKGA, REGAL successfully generalizes to new graphs, significantly outperforms all baselines in solution quality, and computes solutions in about one second on average per TensorFlow test set graph. REGAL's speed and generalization ability show that it is possible to satisfy the challenging constraints of designing learning techniques for optimizing compilers. Validation of REGAL with end-to-end testing and generalizations for improving other compiler passes are natural next steps.

References

[1] Ravichandra Addanki, Shaileshh Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2018. Placeto: Efficient Progressive Device Placement Optimization. In *Workshop on ML for Systems at NeurIPS 2018*.

[2] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2019. Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning. *CoRR* abs/1906.08879 (2019). arXiv:1906.08879 `http://arxiv.org/abs/1906.08879`

[3] The MLIR Authors. 2018. Multi-Level Intermediate Representation. `https://github.com/tensorflow/mlir`. (2018). Accessed: 2019-05-22.

[4] The MLIR Authors. 2019. MLIR Specification. `https://github.com/tensorflow/mlir/blob/4cc85f3cc5bba7603dfa784400e6cd4e1d0045f0/g3doc/LangRef.md`. (2019). Accessed: 2019-09-17.

[5] The TensorFlow Authors. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). arXiv:1603.04467 http://arxiv.org/abs/1603.04467

[6] The TensorFlow authors. 2019. hlo_memory_scheduler. https://github.com/tensorflow/tensorflow/blob/4bfa2359152e9d106c2c20e9fff67643c8578c81/tensorflow/compiler/xla/service/hlo_memory_scheduler.h#L53. (2019). Accessed: 2019-01-25.

[7] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. 2018. Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon. *CoRR* abs/1811.06128 (2018). arXiv:1811.06128 http://arxiv.org/abs/1811.06128

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI 2018*. https://arxiv.org/abs/1802.04799

[9] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Neural Information Processing Systems 2018*. https://arxiv.org/abs/1805.08166

[10] Peter W. Battaglia et al. 2018. Relational inductive biases, deep learning, and graph networks. *CoRR* (2018). arXiv:1806.01261 http://arxiv.org/abs/1806.01261

[11] Lionel Eyraud-Dubois, Loris Marchal, Oliver Sinnen, and Frédéric Vivien. 2015. Parallel Scheduling of Task Trees with Limited Memory. *ACM Trans. Parallel Comput.* 2, 2, Article 13 (June 2015), 37 pages. https://doi.org/10.1145/2779052

[12] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212* (2017).

[13] José Fernando Gonçalves and Mauricio G. C. Resende. 2018. *Random-Key Genetic Algorithms*. Springer International Publishing, Cham, 703–715. https://doi.org/10.1007/978-3-319-07124-4_30

[14] José Fernando Gonçalves and Mauricio G. Resende. 2011. Biased Random-key Genetic Algorithms for Combinatorial Optimization. *Journal of Heuristics* 17, 5 (Oct. 2011), 487–525. https://doi.org/10.1007/s10732-010-9143-1

[15] Google. 2019. CP-SAT Solver. https://developers.google.com/optimization/cp/cp_solver. (2019). [Online; accessed 21-January-2019].

[16] Audrūnas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. 2016. Memory-efficient Backpropagation Through Time. In *Proceedings of the 30th International Conference on Neural Information Processing Systems (NIPS'16)*. Curran Associates Inc., USA, 4132–4140. http://dl.acm.org/citation.cfm?id=3157382.3157559

[17] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond Data and Model Parallelism for Deep Neural Networks. *CoRR* (2018). http://arxiv.org/abs/1807.05358

[18] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.* 31, 4 (Dec. 1999), 406–471.

[19] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[20] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. ACM, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[21] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. 2018. A Hierarchical Model for Device Placement. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Hkc-TeZ0W

[22] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. 2430–2439. `http://proceedings.mlr.press/v70/mirhoseini17a.html`

[23] Francois Pellegrini. 2009. Distillating knowledge about SCOTCH. In *Combinatorial Scientific Computing (Dagstuhl Seminar Proceedings)*, Uwe Naumann, Olaf Schenk, Horst D. Simon, and Sivan Toledo (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany. `http://drops.dagstuhl.de/opus/volltexte/2009/2091`

[24] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* (2018). arXiv:1805.00907 `http://arxiv.org/abs/1805.00907`

[25] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80.

[26] O. Sinnen. 2007. *Task Scheduling for Parallel Systems*. Wiley.

[27] The XLA team. 2017. XLA - TensorFlow compiled. Post in the Google Developers Blog. `http://web.archive.org/web/20170308172654/https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html`. (2017).

[28] Ronald J. Williams. 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Mach. Learn.* 8, 3-4 (1992), 229–256.

A Appendix

A.1 Cost model

We provide a brief description of the cost model, its assumptions, and its limitations.

Given an assignment of ops to devices and a schedule (ordering) of the ops, our cost model simulates the execution of the computation graph and computes the total runtime and the peak memory use. It assumes runtime estimates for each op (e.g., given by a simulator [17]) and memory use of each tensor (an assumption that holds in static compilers like XLA). The simulator tracks the lifetime of the tensors on each device, freeing them from memory after all local consumers have run. Tensors can be transferred across devices by synchronous (blocking) transfers.

The cost model does not permit rematerialization of tensors (as in [16]), nor does it account for fragmentation when computing memory use. Additionally, we do not model asynchronous transfers, which are an option on many systems. Despite these simplifications, the model yields slight variants of problems that are known to be NP-hard [11].

A.2 BRKGA

In BRKGA, *chromosomes* in a population are encoded as n-dimensional vectors with entries in $[0, 1]$ for some fixed n. The goal of BRKGA is to find the chromosome that maximizes a given fitness function $f : [0, 1]^n \to \mathbb{R}$. BRKGA evolves its populations by first sorting the chromosomes by fitness into *elites* and *non-elites*. The next generation of the population is composed of (1) the elites of the previous generation, (2) children of the elites and non-elites generated by a cross-over procedure, and (3) *mutants* sampled from a distribution D. In the standard application of BRKGA, D is a uniform distribution over $[0, 1]^n$. In our work, we learn D. Our use of BRKGA is standard [14] except for the mutant sampling distribution.

Chromosomes are decoded into solutions as follows. Let d be the number of devices, o the number of ops, and t the number of tensors. The chromosome encoding a scheduling solution has three distinct parts: (1) $o \times d$ entries specifying op-to-device affinities; (2) o entries specifying scheduling priorities for each op; (3) $t \times d$ entries specifying tensor-to-device priorities for transfers that may be needed. Given a chromosome, op-to-device assignments are picked by maximum affinity. Transfer ops are created as implied by the device assignments. We then obtain a schedule by performing a topological sort over the ops given their tensor dependencies, breaking ties by using the corresponding node priorities. When enforcing a memory constraint (e.g., 16 GiB per device), the fitness of a schedule is encoded such that all memory-feasible schedules have better fitness than infeasible schedules. Part (3) of the chromosome was omitted in the simplified discussion; we learn sampling distributions only for parts (1) and (2).

A.3 Graph Neural Network Architecture

We reuse the notation introduced for the input computation graph G as defined in section 2.1. Formally, given a graph G, the GNN computes representation vectors $\boldsymbol{h}_v^T$ for each node v through T rounds of iterative message passing. These $\boldsymbol{h}_v^T$'s are then used to compute logits of the softmax distributions, from which we sample the beta parameters:

$$\begin{aligned}
\boldsymbol{h}_v^{(0)} &= \mathrm{MLP}_n(\boldsymbol{x}_v), & \boldsymbol{h}_e &= \mathrm{MLP}_e(\boldsymbol{x}_e) \\
\boldsymbol{m}_e^{(t)} &= \mathrm{MLP}_{\mathrm{msg}}([\boldsymbol{h}_{e_s}^{(t)}, \boldsymbol{h}_{e_t}^{(t)}, \boldsymbol{h}_e]), & \boldsymbol{m}_e^{(t)'} &= \mathrm{MLP}'_{\mathrm{msg}}([\boldsymbol{h}_{e_s}^{(t)}, \boldsymbol{h}_{e_t}^{(t)}, \boldsymbol{h}_e]) \\
\boldsymbol{h}_v^{(t+1)} &= \mathrm{MLP}_{\mathrm{node}}\left(\left[\boldsymbol{h}_v^{(t)}, \textstyle\sum_{e:e_t=v} \boldsymbol{m}_e^{(t)} + \sum_{e:e_s=v} \boldsymbol{m}_e^{(t)'}\right]\right), \\
\alpha_i &\sim \text{Softmax}\left(\mathrm{MLP}_{a,i\%(d+1)}\left(\boldsymbol{h}^T_{\left\lfloor \frac{i+|V|-1}{|V|} \right\rfloor}\right)\right) \\
\beta_i &\sim \text{Softmax}\left(\mathrm{MLP}_{b,i\%(d+1)}\left(\boldsymbol{h}^T_{\left\lfloor \frac{i+|V|-1}{|V|} \right\rfloor}\right)\right) \\
D &= \left\{\text{beta}(\alpha_1, \beta_1), \text{beta}(\alpha_2, \beta_2), \ldots, \text{beta}(\alpha_{(d+1)\times|V|}, \beta_{(d+1)\times|V|})\right\}
\end{aligned} \tag{1}$$

Here, e_s is the source node of edge e and e_t is the target node. MLP_n and MLP_e are multilayer perceptrons (MLPs) that encode node and edge attributes, $\mathrm{MLP}_{\mathrm{msg}}$ and $\mathrm{MLP}'_{\mathrm{msg}}$ compute messages along the edges in the edge direction ($\boldsymbol{m}_e^{(t)}$) and the opposite direction ($\boldsymbol{m}_e^{(t)'}$), $\mathrm{MLP}_{\mathrm{node}}$ updates node representations and $[.]$ represents flat vector concatenation. After T rounds of message passing, the representation for each node $\boldsymbol{h}_v = \boldsymbol{h}_v^{(T)}$ will contain information from the T-hop neighborhood around v in the graph. $\mathrm{MLP}_{a,i}$'s and $\mathrm{MLP}_{b,i}$'s are shared across all nodes for computing the logits of the beta parameter categorical distributions.

The scalar baseline $b(G)$ is computed using a separate GNN, where after we obtained the node representations $\boldsymbol{h}_v$, we aggregate across nodes and compute $b(G)$ as $b(G) = \mathrm{MLP}_b\left(\frac{1}{|V|}\sum_v \mathrm{MLP}_g(\boldsymbol{h}_v)\right)$.