Robust Scheduling with GFlowNets

David W. Zhang[†]
University of Amsterdam

Corrado Rainone **Markus Peschl** **Roberto Bondesan**
Qualcomm AI Research

Abstract

Finding the best way to schedule operations in a computation graph is a classical NP-hard problem which is central to compiler optimization. However, evaluating the goodness of a schedule on the target hardware can be very time-consuming. Traditional approaches as well as previous machine learning ones typically optimize proxy metrics, which are fast to evaluate but can lead to bad schedules when tested on the target hardware. In this work, we propose a new approach to scheduling by sampling proportionally to the proxy metric using a novel GFlowNet method. We introduce a technique to control the trade-off between diversity and goodness of the proposed schedules at inference time and demonstrate empirically that the pure optimization baselines can lead to subpar performance with respect to our approach when tested on a target model. Furthermore, we show that conditioning the GFlowNet on the computation graph enables generalization to unseen scheduling problems for both synthetic and real-world compiler datasets.

1 Introduction

Scheduling is the action of assigning operations to the available compute resources, such as threads, cores, or nodes in a cluster [22, 14, 30]. Unfortunately, finding the schedule with the shortest possible *makespan* (start-to-end runtime) is in general NP-hard [28]. As a result, domain experts have come up with heuristics that are tailored to specific problem instances [19]. Machine learning approaches promise the possibility to automate this process allowing for fast adaptation to new graph distributions [35, 6]. In this work, we consider the problem of scheduling a set of operations with precedence constraints on a fixed number of homogeneous devices, i.e., any operation can run on any device and the runtime is the same on all devices.

Evaluating the makespan of a schedule involves running all operations in the computation graph on some target hardware. This can be very resource intensive, especially when the computation graph includes lengthy operations, the evaluated schedule is inefficient, or the intended target hardware is a cluster with many nodes. Heuristic optimizers, like genetic algorithms [18], or machine learning [24] approaches further exacerbate this problem because they require many evaluations to converge [8]. Proxies are a much faster alternative that estimate the makespan using a simplified model of the hardware. However, this comes at the cost of discrepancies between the proxy makespan and the one observed on the hardware; as a result, performant solutions on the proxy might ultimately be unsatisfactory once tested on the target. Nonetheless, proxies remain a good indicator for most schedules and are essential due to their efficiency. We aim to learn a scheduler that can be trained using the proxy, whilst being robust to its inaccuracies.

The common approach to scheduling problems (and combinatorial optimization problems in general) is to look for the single best schedule. We propose a different philosophy: generate a set of candidate schedules that have a low makespan according to the proxy and are diverse. By having multiple good schedules that are significantly different, we can reduce the impact of systematic errors in the proxy, and hope for robust performance on the target.

[†]Work completed during internship at Qualcomm Technologies Netherlands B.V. Qualcomm AI Research is an initiative of Qualcomm Technologies, Inc.

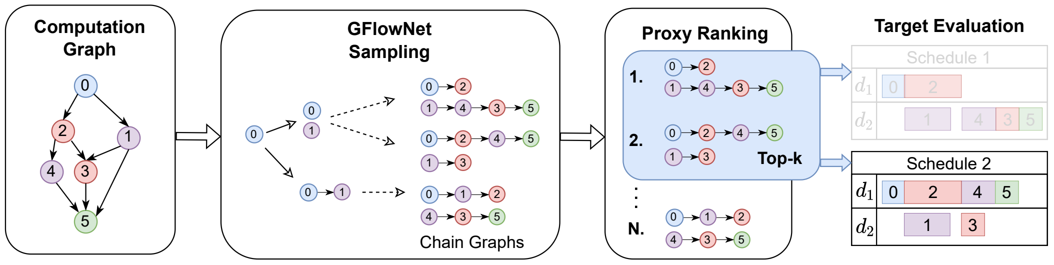

Figure 1: Full pipeline of our generative scheduling approach. Conditioned on the computation graph we generate multiple candidate schedules using GFlowNet, filter for the best k with the proxy and pick the best performing one out of the k that we check on the target. Here we illustrate the pipeline for $k = 2$ and two devices, d_1, d_2.

Our goal is to learn a generative model that assigns higher probability to low-makespan schedules, and importantly can also discover the different modes associated with local optima of the makespan cost. Generative flow networks (GFlowNets) have recently been introduced as a method for learning a stochastic policy that can piece-by-piece construct discrete and composite objects, proportional to a given reward [5]. By computing the reward from the proxy-makespan we can use GFlowNets to sample a diverse set of candidate schedules.

Our main contributions are the following: **1.** We introduce an alternative to the pure proxy optimization viewpoint of scheduling that achieves better robustness to proxy errors, by generating multiple candidate schedules to evaluate directly on the target hardware. **2.** We extend GFlowNets to generate schedules conditioned on a computation graph. Additionally, we introduce a method to control diversity and goodness at inference time, without the need for retraining. These contributions may be of general interest, beyond the scheduling problem. **3.** We empirically demonstrate the robustness of our method to proxy errors and verify the generalization ability on a diverse set of synthetic and real-world computation graphs.

2 Robust scheduling

Problem definition. In scheduling, we are given a computation graph $G_C = (O, P)$ that is a direct acyclic graph (DAG) consisting of operations (nodes) $o \in O$ and precedence constraints (edges) $p \in P$ that encode a partial order in which the operations need to be executed. In particular, the edge p_{ij} encodes that operation o_i needs to finish before o_j can start, for example, because o_j requires the output of o_i as input. Our task is to run all operations on a set of *devices* $\mathcal{D} = \{d_1, \ldots, d_m\}$, without violating the precedence constraints. In addition to the precedence constraints, the devices can only run one operation at a time. We can then view scheduling as performing two distinct tasks: assign a device to each operation, and determine a (complete) order among all operations on the same device that is compatible with the precedence constraints encoded in G_C. We can model the schedule as a chain of operations for each device, where the chain denotes the order in which the operations run on that device. See Figure 1 for a visual example of the chain graphs. Our aim is to find the schedule with the lowest makespan for some target hardware.

Target model vs. proxies. A *proxy* is any tool that allows one to estimate the makespan of a given schedule, without having to run the schedule on the target hardware. Proxies come with significant speed advantages, at the cost of possible mistakes in the estimation of the makespan and relative comparison of schedules. Mistakes can occur for example when the task durations are not accurately profiled, memory movements are too complex to fully model, or additional hardware-specific features are changed. Ideally, we would like to rely on a proxy for the majority of the schedule evaluations, and only evaluate a small fraction of promising schedules on the target hardware. This approach differs from previous works, that either evaluate every schedule on the target [21], leading to very long optimization times, or evaluate everything on the proxy [27], which is susceptible to modeling failures.

Let us denote with C_d, $d \in \mathcal{D}$ the set of edges of the chain graph that specifies the operation order on device d and with $D := \bigcup_{k=1}^{m} C_{d_k}$ the set of all device constraints. The operations correspond to graph nodes and are labeled in the same way as in G_C. No other operation can run on the same device during the *runtime* or *duration* ρ_i of operation o_i. In practice, ρ_i is estimated directly on the hardware in a profiling stage that precedes scheduling. We denote the start time of o_i as τ_i and can thus express the precedence constraints as $\tau_j \geq \tau_i + \rho_i, \quad \forall (i,j) \in P \cup D$. An operation cannot start unless all of those that produce its inputs and all of those that precede it on its assigned device have finished first. To ensure that these constraints are satisfied the proxy assigns each operation o_i the start time

$$\tau_i = \max_k \{\tau_k + \rho_k | (k,i) \in P \cup D\} \tag{1}$$

If a node has no parents in $P \cup D$ the proxy assigns the start time $\tau_i = 0$. The start times of all $o_i \in O$ can be computed by assigning a start time to a node whenever it has no parents or all its parents have an assigned start time. If the graph $(O, P \cup D)$ is a DAG, then this algorithm is guaranteed to assign start times that satisfy Equation 1. The proxy then estimates the *makespan* T of the schedule x as $T(x) \coloneqq \max_i(\tau_i + \rho_i) - \min_i(\tau_i)$.

Generative scheduling. If we had a ranking over all the schedules according to the proxy, we could just go through the list top-to-bottom, and add a schedule to the batch whenever it is significantly different from the previous ones. A full ranking like this is infeasible to construct, but we can instead learn a generative model that samples higher ranked schedules with higher probability. To avoid generating invalid schedules that violate the constraints in Section 2, we construct a schedule in a step-by-step process: start with an empty schedule at the initial state s_0, and at each step add an operation to the partial schedule until the schedule contains all operations ending at the terminal state s_n. At each intermediate state s_t, an action a_t consists in picking an operation and assigning it to one of the devices, leading to a new state s_{t+1}. Picking an operation o_t is a valid action if and only if $\forall k : (k,t) \in P$, o_k is already in the partial schedule at state s_t. This is a sufficient condition for the final "schedule" graph $(O, P \cup D)$ to be a DAG, implying that the constructed schedule is feasible. The final states represent full schedules x, for which we can compute the makespan $T(x)$ with the proxy, given the runtimes $\{\rho_i\}_{i=1}^n$. We compute the relative *speedup* compared to the makespan on a single device as $U(x){=}\sum_i \rho_i / T(x)$, from which we compute the reward as we present in the next section.

3 Generative Flow Networks for scheduling

GFlowNets [4, 5] are methods for training a stochastic policy to sample discrete and composite objects x proportionally to a given reward, through a sequence of constructive actions. In the previous section, we discussed how to limit the action space to guarantee that we sample valid schedules. After a brief introduction to GFlowNets, the following sections will present our proposed extensions that include a new training objective that is suitable for learning conditional GFlowNets and a method for controlling the selectivity of samples at inference time.

Background. We denote by $\boldsymbol{s}{=}(s_0, s_1, \dots, s_n)$ a *trajectory* that consists of a sequence of states s_t. We denote by $\mathcal{T}$ the set of all such trajectories and by $\mathcal{T}_x$ the set of trajectories that end at $s_n{=}x$. Based on this, we define a flow function $F : \mathcal{T} \to \mathbb{R}^+$ and its associated normalized probability distribution $P(\boldsymbol{s}) = F(\boldsymbol{s})/Z$, with $Z = \sum_{\boldsymbol{s} \in \mathcal{T}} F(\boldsymbol{s})$. A flow function that fulfills the condition: $R(x) = \sum_{\boldsymbol{s} \in \mathcal{T}_x} F(\boldsymbol{s})$ (every terminal state has a total flow matching its reward), results in a probability over schedules $P(x) = \sum_{\boldsymbol{s} \in \mathcal{T}_x} F(\boldsymbol{s})/Z$ that is proportional to the reward $P(x) \propto R(x)$, and further entails that $Z = \sum_x R(x)$ [4, 5].

For any Markovian flow, we can decompose the probability of a trajectory in terms of the forward probability: $P(\boldsymbol{s}) = \prod_{t=1}^n P_F(s_t|s_{t-1})$. This way, we can generate trajectories $\boldsymbol{s}$ by sampling a sequence of actions starting from s_0. In Section 2 we described how to limit the action space appropriately to guarantee that every sampled schedule is valid. Similarly, we can define a backward probability P_B that factorizes the trajectory probability conditioned on a terminal state: $P(\boldsymbol{s}|s_n = x) = \prod_{t=1}^n P_B(s_{t-1}|s_t)$.

The training objectives in previous works aim for consistency between the flow in the forward and backward directions [5, 23]. Malkin et al. [23] formulate the *trajectory balance constraint* as:

$$Z \prod_{t=1}^{n} P_F(s_t|s_{t-1}) = R(x) \prod_{t=1}^{n} P_B(s_{t-1}|s_t) \tag{2}$$

Based on Equation 2, they propose to estimate Z, P_F, and P_B by optimizing the trajectory balance loss which is the squared difference between the logarithms of the l.h.s. and the r.h.s. of Equation 2.

3.1 Log-partition variance loss

In order to apply the trajectory balance loss in the conditional case, we would need to learn an additional regression model that estimates the log-partition function $\log Z$ conditioned on G_C. Training such a network accurately is difficult and in practice, we found this to lead to non-convergent training behaviors. Instead, we can rewrite Equation 2 to implicitly estimate $\log Z$ based on the forward and backward flows of a single trajectory $\boldsymbol{s}$, where P_F and P_B are neural networks with parameters $\boldsymbol{\theta}$:

$$\zeta(\boldsymbol{s};\boldsymbol{\theta}) = \log R(x) + \sum_{t=1}^{n} \log P_B(s_{t-1}|s_t;\boldsymbol{\theta}) - \sum_{t=1}^{n} \log P_F(s_t|s_{t-1};\boldsymbol{\theta}) \tag{3}$$

In the optimal case, $\zeta(\boldsymbol{s};\boldsymbol{\theta})$ is equal to the true $\log Z$ which is the same for all trajectories corresponding to the same computation graph G_C. Thus, our optimization goal turns into minimizing the variance of $\zeta(\boldsymbol{s};\boldsymbol{\theta})$ over different trajectories $\boldsymbol{s}$ with the loss:

$$\mathcal{L}_{\mathrm{V}}(\boldsymbol{s};\boldsymbol{\theta}) = \left(\zeta(\boldsymbol{s};\boldsymbol{\theta}) - \mathbb{E}_{\boldsymbol{s}}\left[\zeta(\boldsymbol{s};\boldsymbol{\theta})\right]\right)^2 \tag{4}$$

In the experiments, we use the training distribution to estimate $\mathbb{E}_{\boldsymbol{s}}\left[\zeta(\boldsymbol{s})\right]$ with a mini-batch of sampled trajectories.

3.2 Temperature-conditioned Topoformer

Reward temperature. We compute the log-reward as $\log R(x; m, \sigma) = (U(x) - m)/\sigma$ where $U(x)$ is the speedup of the schedule x, m is the number of devices, and $\sigma \in \mathbb{R}^+$ plays the role of a temperature. A low temperature concentrates the distribution around the modes and increases the selectivity of the generator. This is useful since there can be many more schedules with low speedup when compared to good ones. For example, when simply setting the reward equal to the speedup, we observed that finding schedules with high speedup requires a prohibitively large amount of samples. We expect this temperature term to allow trade-offs between diversity and shifting the mean of the sampling distribution towards better schedules.

Training with a constant temperature can lead to low performance (when set too high), and low diversity or unstable training (when set too low). Furthermore, different computation graphs can have different ideal temperature values, making this approach less suitable when learning conditional GFlowNets. Instead, we propose to learn a single model for multiple different reward functions, by conditioning the policy networks (P_F and P_B) on the temperature σ. Approximating the temperature-conditioned policy with a neural network is feasible because flows for a given temperature can be continuously morphed into flows for any other temperature when $R(x; m, \sigma)$ is continuous with respect to σ. We provide a proof for the following theorem in Appendix A.

Theorem 1 (Flow Continuity)**.** *Let $\{R_i\}_{i=1}^{\infty}$ be a sequence of positive reward functions such that for all terminal states x, $(R_i(x)/R(x)) \to 1$ as $i \to \infty$. Then, for any flow F^R with reward R, there exists a sequence of flow functions $\{F^{R_i}\}_{i=1}^{\infty}$ with $F^{R_i}(\boldsymbol{s}) \to F^R(\boldsymbol{s})$ for all $\boldsymbol{s} \in \mathcal{T}$.*

Topoformer architecture. For the neural network architecture of our policy, we use the Topoformer [12], which has been recently introduced for learning topological orderings of computation graphs. It builds on the Transformer encoder architecture [34] and additionally masks the multi-head attention depending on the topology of the computation graph. Both forward and backward policies use separate MLP heads on top of a shared Topoformer encoder. Taking inspiration from the successful use of time conditioning in diffusion models [32, 15], we add temperature conditioning by first embedding the temperature using an MLP to produce e_σ, and then reuse the embedding in every first linear layer block of the Topoformer:

$$\texttt{lin}(h, e_\sigma) = \texttt{lin}_{\text{scale}}(e_\sigma) \odot \texttt{lin}(h) + \texttt{lin}_{\text{shift}}(e_\sigma) \tag{5}$$

Here $\texttt{lin}_{\text{scale}}$ and $\texttt{lin}_{\text{shift}}$ are linear layers and $\odot$ is the elementwise multiplication [29]. In contrast to diffusion models, we observe better performance on large temperature ranges with the ReLU [26] activation function. We hypothesize that this is connected to the monotonicity of the underlying policy function with respect to decreasing temperatures (see the appendix, Corollary 1) and the propensity for linear extrapolation of ReLU MLPs [37].

Table 1: Robustness results on a single computation graph. Higher diversity correlates with better robustness against a mismatch of the proxy and the target, with GFlowNet achieving the best diversity and the best target performance on average.

	Speedup				Diversity		
	Proxy	Noisy Runtimes	Bandwidth Limited	Latency Limited	GED	d_{inv}	d_{sen}
List scheduling	3.23±0.00	2.75±0.00	1.02±0.00	1.74±0.00	0	0	0
BRKGA	3.22±0.00	2.86±0.15	1.29±0.45	1.80±0.34	55.92±2.56	22.83±2.39	56.21±1.50
PPO	3.28±0.07	3.07±0.09	1.38±0.49	1.87±0.38	85.08±3.54	31.71±0.05	105.64±0.08
GFlowNet	3.21±0.02	3.05±0.04	1.78±0.03	2.11±0.03	94.79±0.15	42.08±0.33	115.98±0.09

4 Experiments

In this section, we evaluate different aspects of our generative scheduling approach. First, we restrict training and evaluation to a single computation graph, which corresponds to the same unconditional setting considered by previous works on GFlowNets [4, 10, 20]. Next, we train with multiple computation graphs and evaluate on unseen ones. To the best of our knowledge, this is the first time that generalization of conditional GFlowNets is tested empirically. Finally, we verify the generalization ability on real-world computation graphs of neural networks.

Candidate sampler. We consider two heuristic and two neural methods for generating candidate schedules. The first is our GFlowNet approach described in Section 3 from which we generate 1000 samples at temperature $\sigma = 0.005$ and take the top 100 following the proxy. Second is the critical path-based list scheduling, a heuristic algorithm for scheduling on homogeneous devices [25]. List scheduling first forms a topological order of the operations and then assigns them in that order one by one to a free device. Third is the Biased Random Key Genetic Algorithm (BRKGA) [13], a genetic algorithm that has previously shown good performance on scheduling tasks [27]. We use the top 100 schedules from the final population as the candidate schedules. Fourth is Proximal Policy Optimization (PPO) [31], a deep reinforcement learning method that has been successfully applied to scheduling problems [40]. PPO also trains a stochastic policy, which makes it a natural choice for comparison with GFlowNets [4]. We employ the same definitions of states, actions, and reward function (with temperature $\sigma = 0.25$; lower was not beneficial) as the GFlowNet approach. Same as for GFlowNets, we sample 1000 schedules and pick the top 100 as the candidate schedules.

Experimental setup and metrics. In all experiments, we only use the node time duration as a feature of the computation graph. For simplicity and ease of reproducibility, we avoid any complicated heuristics to add extra features. All our experiments are based on four homogenous devices. We measure the performance in terms of the speedup $U(x)$. For the diversity, we report three different measures: graph-edit distance (GED), the L2 distance between the proxy start-times (d_{inv}), and the L2 distance between the proxy start-times concatenated with the device placement (d_{sen}). See Appendix C.2 for more details on diversity measures.

4.1 Proxy errors: diversity for robust scheduling

We examine how differences between the proxy and the target performance model can affect the final runtime. To do so, we first focus on a single computation graph that is used both for training and testing to avoid possible confounding factors that may happen in the generalization setting. We design three different target models that each reflect a different failure mode of the proxy, as discussed in Section 2. In the first setting node durations are incorrectly profiled (Noisy Runtimes). In the second and third settings, the target models the memory movement across devices with a linear model [33], which can be either bottlenecked by limited bandwidth (Bandwidth Limited), or by high latency (Latency Limited). The linear model has been shown to be a good makespan estimator for certain devices [16, 9]. In Figure 2, we show the correlation between the proxy and the different target environments. For all three targets, the proxy is highly correlated but can have target speedups that differ by a factor of up to $\times 2$ for the schedules with high proxy speedups.

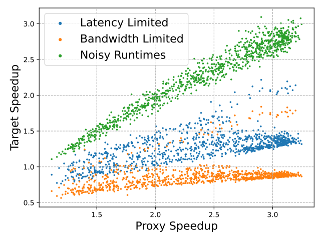

Figure 2: Proxy and target speedup correlations.

	Speedup		Diversity		
	Proxy 1	Proxy 100	GED	d_{inv}	d_{sen}
List scheduling	3.44	3.44	0	0	0
BRKGA	3.46	3.45	46.59	12.75	40.11
PPO	3.48	3.46	69.54	13.45	80.84
GFlowNet	3.46	3.41	92.02	24.27	90.17

Table 2: Generalization to unseen synthetic computation graphs. The Speedup Proxy 100 column reports the average proxy speedup over the top 100 schedules.

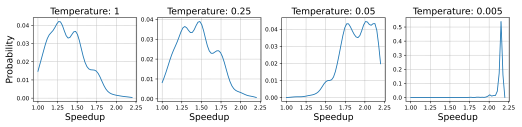

Figure 3: Sample reward distribution for different inference temperatures in the conditional GFlowNet. Lower temperatures allocate more probability mass to better schedules.

We report the speedups and diversity measures in Table 1. The results highlight that any method that can generate multiple good candidate schedules achieves higher speedups than list scheduling, which only produces a single unique schedule. Furthermore, if the candidate schedules are more diverse — as is the case for GFlowNet — the target performance is also better on average. The results confirm our hypothesis that a diverse set of candidate schedules can improve robustness towards a misspecified proxy.

4.2 Generalizing to unseen synthetic computation graphs

Next, we evaluate how well our conditional GFlowNet can generalize to unseen computation graphs. The computation graphs in the training dataset have 50 nodes each and are sampled from two different random graph distributions, whereas the test dataset additionally samples from three other random graph distributions. For details on the generative process of the computation graphs, we refer to Appendix C.4.

Results in Figure 2 demonstrate that the conditional GFlowNet can generalize to previously unseen computation graphs, regardless of whether they originate from the same random graph distribution. Next, we ablate our proposed temperature conditioning method by generating 1000 samples at different temperature values given a computation graph of size 25. In Figure 3, we observe that decreasing the temperature does indeed shift the sample distribution to the right and also sharpens it when the temperature approaches zero. Notably, the temperature $\sigma = 0.005$ is not in the training distribution, which demonstrates that the model can extrapolate to temperature values outside of the training range. Surprisingly, we observe that training with a variable temperature can improve the performance further than is possible with a fixed temperature, which we demonstrate in Appendix D.

4.3 Real world computation graphs

Finally, we verify the generalization ability on a small set of real-world computation graphs used for the commercial development of our artificial intelligence hardware and software products (see Appendix C.5 for details). We report the speedup on the same target models used in Section 4.1 to assess robustness on unseen real-world computation graphs. The results are in Table 3.

Table 3: Generalization on real-world graphs. GFlowNet retains a high diversity and exhibits consistently better performances than the baselines on the target models. PPO uses the same hyperparameters as in the previous experiments but does not manage to converge on this dataset.

	Speedup					Diversity		
	Proxy 1	Proxy 100	Noisy Runtimes	Bandwidth Limited	Latency Limited	GED	d_{inv}	d_{sen}
List scheduling	2.74±0.00	2.74±0.00	2.51±0.00	0.89±0.00	1.43±0.00	0	0	0
BRKGA	2.59±0.18	2.58±0.18	2.46±0.16	1.55±0.17	1.80±0.18	52.32±21.59	17.14±8.17	42.64±12.23
PPO	2.41±0.20	2.23±0.27	2.28±0.26	0.91±0.20	1.43±0.10	53.05±7.27	42.70±3.44	64.92±4.08
GFlowNet	2.71±0.03	2.66±0.01	2.71±0.01	1.73±0.01	1.95±0.03	87.95±0.13	26.56±0.56	91.33±0.15

5 Conclusion

We have empirically demonstrated how the conventional optimization approach to scheduling, which optimizes a proxy of the real makespan, is brittle to modeling failures in the proxy itself. Our proposed approach evaluates multiple schedules on the target and thereby achieves more robustness to discrepancies between the proxy and the target. We demonstrated that GFlownets can sample a diverse set of candidate schedules that achieve better target performance than alternative methods which achieve lower diversity. Further, we showed that conditioning on temperature allows a trade-off between diversity and proxy performance, and that conditional GFlowNets can generalize to unseen computation graphs. Interesting future directions include scaling up our method to larger graphs and integrating scheduling heuristics to speed up training.

References

[1] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. “Placeto: Learning Generalizable Device Placement Algorithms for Distributed Machine Learning”. In: *Advances in Neural Information Processing Systems 32 (NIPS 2019)* (2019). URL: `https://par.nsf.gov/biblio/10169218` (cit. on p. 10).

[2] Zafarali Ahmed, Nicolas Le Roux, Mohammad Norouzi, and Dale Schuurmans. “Understanding the impact of entropy on policy optimization”. In: *International conference on machine learning*. PMLR. 2019, pp. 151–160 (cit. on p. 11).

[3] Réka Albert and Albert-László Barabási. “Statistical mechanics of complex networks”. In: *Reviews of modern physics* 74.1 (2002), p. 47 (cit. on p. 11).

[4] Emmanuel Bengio, Moksh Jain, Maksym Korablyov, Doina Precup, and Yoshua Bengio. “Flow network based generative models for non-iterative diverse candidate generation”. In: *Advances in Neural Information Processing Systems*. Vol. 34. 2021, pp. 27381–27394 (cit. on pp. 3, 5, 10, 11).

[5] Yoshua Bengio, Tristan Deleu, Edward J Hu, Salem Lahlou, Mo Tiwari, and Emmanuel Bengio. “Gflownet foundations”. In: *arXiv preprint arXiv:2111.09266* (2021) (cit. on pp. 2, 3, 10).

[6] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. “Machine learning for combinatorial optimization: a methodological tour d’horizon”. In: *European Journal of Operational Research* 290.2 (2021), pp. 405–421 (cit. on p. 1).

[7] J. Blank and K. Deb. “pymoo: Multi-Objective Optimization in Python”. In: *IEEE Access* 8 (2020), pp. 89497–89509 (cit. on p. 11).

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. “{TVM}: An automated {End-to-End} optimizing compiler for deep learning”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594 (cit. on p. 1).

[9] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. “LogP: Towards a realistic model of parallel computation”. In: *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 1993, pp. 1–12 (cit. on p. 5).

[10] Tristan Deleu, António Góis, Chris Emezue, Mansi Rankawat, Simon Lacoste-Julien, Stefan Bauer, and Yoshua Bengio. “Bayesian Structure Learning with Generative Flow Networks”. In: *arXiv preprint arXiv:2202.13903* (2022) (cit. on pp. 5, 10, 11).

[11] Paul Erdős, Alfréd Rényi, et al. “On the evolution of random graphs”. In: *Publ. Math. Inst. Hung. Acad. Sci* 5.1 (1960), pp. 17–60 (cit. on p. 11).

[12] Mukul Gagrani, Corrado Rainone, Yang Yang, Harris Teague, Wonseok Jeon, Herke Van Hoof, Weiliang Will Zeng, Piero Zappi, Christopher Lott, and Roberto Bondesan. “Neural Topological Ordering for Computation Graphs”. In: *arXiv preprint arXiv:2207.05899* (2022) (cit. on pp. 4, 11).

[13] José Fernando Gonçalves and Mauricio GC Resende. “Biased random-key genetic algorithms for combinatorial optimization”. In: *Journal of Heuristics* 17.5 (2011), pp. 487–525 (cit. on p. 5).

[14] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011 (cit. on p. 1).

[15] Jonathan Ho, Ajay Jain, and Pieter Abbeel. “Denoising diffusion probabilistic models”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 6840–6851 (cit. on p. 4).

[16] Roger W Hockney. “The communication challenge for MPP: Intel Paragon and Meiko CS-2”. In: *Parallel computing* 20.3 (1994), pp. 389–398 (cit. on p. 5).

[17] Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. “Stochastic blockmodels: First steps”. In: *Social networks* 5.2 (1983), pp. 109–137 (cit. on p. 11).

[18] Edwin SH Hou, Nirwan Ansari, and Hong Ren. “A genetic algorithm for multiprocessor scheduling”. In: *IEEE Transactions on Parallel and Distributed systems* 5.2 (1994), pp. 113–120 (cit. on p. 1).

[19] Oscar H Ibarra and Chul E Kim. “Heuristic algorithms for scheduling independent tasks on nonidentical processors”. In: *Journal of the ACM (JACM)* 24.2 (1977), pp. 280–289 (cit. on p. 1).

[20] Moksh Jain, Emmanuel Bengio, Alex Hernandez-Garcia, Jarrid Rector-Brooks, Bonaventure FP Dossou, Chanakya Ajit Ekbote, Jie Fu, Tianyu Zhang, Michael Kilgour, Dinghuai Zhang, et al. "Biological Sequence Design with GFlowNets". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 9786–9801 (cit. on pp. 5, 10, 11).

[21] Shauharda Khadka, Estelle Aflalo, Mattias Mardar, Avrech Ben-David, Santiago Miret, Shie Mannor, Tamir Hazan, Hanlin Tang, and Somdeb Majumdar. "Optimizing Memory Placement using Evolutionary Graph Reinforcement Learning". In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=-6vS_4Kfz0 (cit. on pp. 2, 10).

[22] Yu-Kwong Kwok and Ishfaq Ahmad. "Static scheduling algorithms for allocating directed task graphs to multiprocessors". In: *ACM Computing Surveys (CSUR)* 31.4 (1999), pp. 406–471 (cit. on p. 1).

[23] Nikolay Malkin, Moksh Jain, Emmanuel Bengio, Chen Sun, and Yoshua Bengio. "Trajectory Balance: Improved Credit Assignment in GFlowNets". In: *arXiv preprint arXiv:2201.13259* (2022) (cit. on pp. 3, 10).

[24] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. "Learning scheduling algorithms for data processing clusters". In: *Proceedings of the ACM special interest group on data communication*. ACM, 2019, pp. 270–288 (cit. on p. 1).

[25] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994 (cit. on p. 5).

[26] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Icml*. 2010 (cit. on p. 4).

[27] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. "Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs". In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=rkxDoJBYPB (cit. on pp. 2, 5, 10).

[28] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998 (cit. on p. 1).

[29] Ethan Perez, Florian Strub, Harm De Vries, Vincent Dumoulin, and Aaron Courville. "Film: Visual reasoning with a general conditioning layer". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. 2018 (cit. on p. 4).

[30] Michael L Pinedo. *Scheduling*. Vol. 29. Springer, 2012 (cit. on p. 1).

[31] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017) (cit. on pp. 5, 11).

[32] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. "Score-based generative modeling through stochastic differential equations". In: *arXiv preprint arXiv:2011.13456* (2020) (cit. on p. 4).

[33] Leslie G Valiant. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (1990), pp. 103–111 (cit. on pp. 5, 11).

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 4).

[35] Zheng Wang and Michael O'Boyle. "Machine learning in compiler optimization". In: *Proceedings of the IEEE* 106.11 (2018), pp. 1879–1901 (cit. on p. 1).

[36] Duncan J Watts and Steven H Strogatz. "Collective dynamics of 'small-world'networks". In: *nature* 393.6684 (1998), pp. 440–442 (cit. on p. 11).

[37] Keyulu Xu, Mozhi Zhang, Jingling Li, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. "How neural networks extrapolate: From feedforward to graph neural networks". In: *arXiv preprint arXiv:2009.11848* (2020) (cit. on p. 4).

[38] Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Xu Chi. "Learning to dispatch for job shop scheduling via deep reinforcement learning". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1621–1632 (cit. on p. 10).

[39] Dinghuai Zhang, Nikolay Malkin, Zhen Liu, Alexandra Volokhova, Aaron Courville, and Yoshua Bengio. “Generative Flow Networks for Discrete Probabilistic Modeling”. In: *arXiv preprint arXiv:2202.01361* (2022) (cit. on p. 10).

[40] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Phitchaya Phothilimtha, Shen Wang, Anna Goldie, et al. “Transferable graph optimizers for ml compilers”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 13844–13855 (cit. on pp. 5, 10).

A Proof for Theorem 1

In the following, we will denote F^R to be the flow corresponding to a flow function $F : \mathcal{T} \to \mathbb{R}_{\geq 0}$ with corresponding reward R. That is, $F^R : \mathcal{T} \to \mathbb{R}_{\geq 0}$ is a flow function such that for any terminal state x we have $R(x) = \sum_{\boldsymbol{s} \in \mathcal{T}_x} F^R(\boldsymbol{s})$, where $\sum_{\boldsymbol{s} \in \mathcal{T}_x} F^R(\boldsymbol{s}) =: F^R(x)$ is the total flow at the terminal state x.

Theorem (Flow Continuity). *Let $\{R_i\}_{i=1}^{\infty}$ be a sequence of positive reward functions such that for all terminal states x, $(R_i(x)/R(x)) \to 1$ as $i \to \infty$. Then, for any flow F^R with reward R, there exists a sequence of flow functions $\{F^{R_i}\}_{i=1}^{\infty}$ with $F^{R_i}(\boldsymbol{s}) \to F^R(\boldsymbol{s})$ for all $\boldsymbol{s} \in \mathcal{T}$.*

Proof. Let $\{x_i\}_{i=1}^{M}$ be the set of terminal states and define

$$F^{R_i}(\boldsymbol{s} = (s_0, \dots, x)) := F^R(\boldsymbol{s}) \frac{R_i(x)}{R(x)} \tag{6}$$

To see that F^{R_i} is a valid flow for R_i, we note that $F^{R_i} \geq 0$ and

$$\begin{aligned} F^{R_i}(x) &= \sum_{\boldsymbol{s} \in \mathcal{T}_x} F^{R_i}(\boldsymbol{s}) = \frac{R_i(x)}{R(x)} \sum_{\boldsymbol{s} \in \mathcal{T}_x} F^R(\boldsymbol{s}) \\ &= \frac{R_i(x)}{R(x)} R(x) = R_i(x) \end{aligned} \tag{7}$$

for any terminal state x. Finally, for any $\boldsymbol{s} \in \mathcal{T}$ with terminal state x we have

$$F^{R_i}(\boldsymbol{s}) = \frac{R_i(x)}{R(x)} F^R(\boldsymbol{s}) \to F^R(\boldsymbol{s}) \tag{8}$$

■

Corollary 1. *Let $R_{\sigma_i} := \log R(x; m, \sigma_i) = (U(x) - m)/\sigma_i$ be a sequence of temperature conditioned reward functions with $\sigma_i \searrow \sigma_0$. Then, for any $\epsilon > 0$ and flow $F^{R_{\sigma_0}}$ there exists a neighborhood $(\sigma_0, \sigma_0 + \delta)$ containing flow functions F^{R_σ} with $F^{R_\sigma}(\boldsymbol{s}) - F^{R_{\sigma_0}}(\boldsymbol{s}) < \epsilon$ for all $\boldsymbol{s} \in \mathcal{T}$. Furthermore, we can approximate $F^{R_{\sigma_0}}$ with monotonically changing flow functions $F^{R_{\sigma_i}}$.*

B Related work

Reinforcement learning for scheduling. Reinforcement learning has been the predominant machine learning approach to optimize the makespan for computation graph schedules [1, 27, 38]. The rewards used include simple analytical proxies of the makespan [27, 40], but also more refined proxies which incorporate modeling of memory movements [1]. Khadka et al. [21] directly train on the target hardware, but consider only a few computation graphs, and do not show generalization to unseen ones. Addanki et al. [1] use a sophisticated simulator of the makespan which is customized to the target hardware. Similar to our work, Zhang et al. [38] also construct the schedule piece-by-piece. Instead of finding a single (local) mode of the proxy, our work proposes to learn the full distribution over the proxy to improve the robustness against inaccuracies in the proxy.

Generative Flow Networks. GFlowNets have been applied to generating small molecules [4], Bayesian networks [10], discrete images [39], and biological sequences [20]. We extend its application to scheduling, a classical combinatorial optimization problem. Conditional GFlowNets have previously only been theoretically discussed by Bengio et al. [5]. We enable training conditional GFlowNets with our proposed log-partition variance loss and empirically demonstrate generalization to unseen computation graphs. The log-partition variance loss only needs to parametrize the forward and backward probabilities P_F and P_B. This is similar to the non-forward trajectory loss mentioned in the appendix by Malkin et al. [23], which also does not involve learning any state flows, including the initial flow Z. However, our loss does not mix forward and backward steps from different trajectories and directly optimizes the consistency of the total flow Z for each trajectory associated with a given computation graph G_C. To control the selectiveness of the generator, previous works augment the reward with a fixed temperature [4, 10, 20]. Instead, we condition the policy neural network on the temperature term which allows us to tune the selectiveness of the generator at inference time.

C Experiment details

C.1 Candidate samplers

We use the popular open-source library pymoo [7] to implement the BRKGA candidate sampler. Our PPO implementation is based on algorithm 1 at `https://spinningup.openai.com/en/latest/algorithms/ppo.html`, and we follow [31] to implement the entropy regularisation by adding the entropy term directly to the PPO-clip loss. We decay this entropy term during training similar to Ahmed et al. [2]. We use the same learning rate for both the actor and the critic, and we decay it with an exponential schedule.

We train GFlowNets conditioned on a temperature randomly sampled between 0.01 and 1. At inference, we use 0.005 for the temperature in all experiments.

C.2 Metrics

The graph-edit distance (GED) compares two schedules in their chain-graphs form. In particular, we can model a schedule for a computation graph, by constructing a chain graph for each device that specifies the additional precedence constraints we introduce to complete the order in which the operations are run on each device. The GED is then computed simply by taking the difference between the adjacency matrices and normalizing it by the total number of edges.

The L2 distance between the start (d_{inv}) times simply takes the start times as assigned by the proxy model and computes the L2 norm of the difference.

The L2 distance including the device assignment (d_{sen}) additionally concatenates the device placement to the times.

C.3 Proxy errors: Diversity for robust scheduling

In this experiment, we consider a single real-world graph that has 78 nodes.

The linear memory model [33] computes the delay as a linear function $f(m) = am + b$ of the memory m with a modeling the amount of data that can be transferred per time and b modeling the startup delay. In the Bandwidth Limited setting the a term dominates the delay, while in the Latency Limited setting b has a greater effect.

C.4 Generalization to unseen computation graphs

We generate the synthetic graph dataset from random graph distributions over undirected graphs. To get DAGs from these graphs, we randomly choose a direction for every edge in a way that produces no cycles.

We sample the runtimes for each node from the uniform distribution $\mathcal{U}(0, 1)$.

For training, we use 1000 different computation graphs, with equally many sampled from the two random graph distributions: Erdős–Rényi [11], and Layered Graphs [12]. We report test performances on 50 different computation graphs with equally many sampled from the five different random graph distributions: Erdős–Rényi [11], Layered Graphs [12], stochastic block model [17], Watts-Strogatz [36], and Barabási–Albert [3].

C.5 Real World Computation Graphs

The computation graphs in this dataset originate from a diverse set of neural network architectures with different applications, including for example classification and denoising. We train on 8 real-world computation graphs of sizes below 100 nodes and evaluate on 4 different computation graphs of sizes between a dozen and 128 nodes.

Sub-graph training. Training with a full computation graph might not always be necessary and we hypothesize that learning on sub-graphs can lead to policies that generalize to the full computation graph. This can be seen as a form of data augmentation and increases the amount of training data, while simultaneously improving the training time. To speed up training, we apply the graph subsampling strategy to randomly pick between 25 to 75 nodes at every training step.

In Table 3, we observe that the conditional GFlowNet retains the benefits of high diversity and robustness to misspecifications in the proxy even when applied to graphs not seen during training and of larger sizes. PPO shows unstable training behavior and the reward training curve does not converge, despite using the same hyperparameters that worked for the previous two experiments. We conjecture that this is due to the inhomogeneous maximum possible speedup of the training graphs that lead to different reward scales per training graph. In comparison, GFlowNet still converges as before without any changes to the hyperparameters.

D Temperature conditioning ablation

In order to increase the likelihood of sampling good schedules, one could introduce a fixed temperature throughout training and inference. However, we have observed that this procedure is unreliable for small temperatures. Figure 4 shows generalization performance during training on the synthetic graph experiment of Section 4.2. As can be seen, choosing a temperature of 0.01 results in a smaller maximum reward as opposed to training on a higher temperature of 0.03. On the other hand, sampling a range of temperatures between 0.01 and 1 and evaluating on 0.01 samples the best performing schedules on unseen computation graphs.

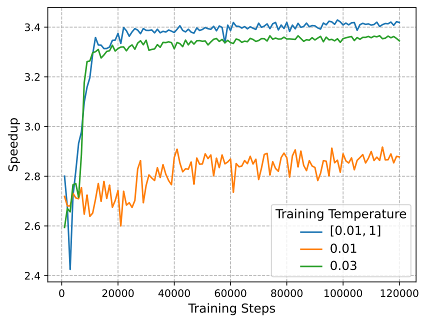

Figure 4: The impact of different temperature regimes on top-1 generalization performance. Training on single temperatures prevents learning when set too low (orange). On the other hand, training on a range of different temperatures (blue) results in better performance when performing inference with the minimum training temperature.