Accelerating Exact Combinatorial Optimization via RL-based Initialization – A Case Study in Scheduling

Jiaqi Yin
University of Utah
Salt Lake City, US
jiaqi.yin@utah.edu

Cunxi Yu
University of Utah
Salt Lake City, US
cunxi.yu@utah.edu

***Abstract*—Scheduling on dataflow graphs (also known as computation graphs) is an NP-hard problem. The traditional exact methods are limited by runtime complexity, while reinforcement learning (RL) and heuristic-based approaches struggle with determinism and solution quality. This research aims to develop an innovative approach that employs machine learning (ML) for addressing combinatorial optimization problems, using scheduling as a case study. The goal is to provide guarantees in optimality and determinism while maintaining the runtime cost of heuristic methods. Specifically, we introduce a novel two-phase RL-to-ILP scheduling framework, which includes three steps: 1) RL solver acts as coarse-grain scheduler, 2) solution relaxation and 3) exact solving via ILP. Our framework demonstrates the same scheduling performance compared with using exact scheduling methods while achieving up to 128 $\times$ speed improvements. This was conducted on actual Edge TPU platforms, utilizing ImageNet DNN computation graphs as input. Additionally, the framework offers improved on-chip inference runtime and acceleration compared to the commercially available Edge TPU compiler.**

I. Introduction

Combinatorial optimization (CO) is an optimization problem that finds the optimal solution from a vast search space, where graph-based scheduling is one of the classic CO problems: allocate the operators in computational graph into the given number of stages and minimize the on-cache memory, and communication cost simultaneously. Computational graph scheduling is fundamental to a wide range of hardware deployment domains such as FPGAs, CPU/GPU, and domain-specific accelerators. For example, the DNN model execution is computationally intensive, which requires many computational resources including on-cache memory, I/O-bandwidth, etc [1]. This problem becomes more critical for edge devices considering the small buffer size and fewer processing elements. More importantly, in the increasing uses of cloud-based domain-specific accelerators, enabling a fast runtime compilation (scheduling) is critical to the global management of cloud computing resources in efficiency and cloud utilization [2]. All together make scheduling an important task in pursuing both optimal quality of results and fast runtime.

Challenges of graph-level scheduling: To clearly motivate this work, we provide a simple case study by evaluating conventional scheduling methods, including exact methods [3], [4], heuristic methods (commercial Edge TPU compiler), and stand-alone RL/ML based scheduling methods. Specifically, we pick Google Edge TPU as the case study platform, with ImageNet model ResNet152 [5]. The properties of the three scheduling methods are presented in Table I. The key conclusions are summarized as follows: **1)** The advantage of exact methods [3], [4], [6] (e.g., ILP or SAT) is the optimality of scheduling results. ILP constructs the formulations and constraints to solve the scheduling problem, which guarantees solution optimality. However, due to the nature of the complexity, exact methods suffer from scalability issues with the highest solving runtime; **2)**. Heuristic methods [7], [8] have less solving runtime overhead but the scheduling results are sub-optimal. Similar to the Edge TPU compiler, many vendor-specific libraries such as Nvidia cuBLAS, TVM, TensorFlow-Lite, and Edge TPU compiler [9]–[11] are built upon the heuristic approach. **3)** Machine learning (ML) techniques have been applied to many combinatorial problems such as scheduling and other EDA problems [12]–[17], which try to solve the given problem at inference runtime to accelerate the solving process, mostly aiming at generating near-optimal scheduling with inference runtime [18], [19], [19]–[21]. However, there are well known limitations in lacking determinism and guarantees for the performance and solving procedure.

TABLE I: Comparison of existing scheduling methodologies with motivating example of ResNet152 computation graph.

	Runtime (s)	QoR(MB)	Determinism
Exact Methods	Highest (268×)	Optimal	✓
Heuristic	High (66×)	Sub-optimal	✓
Ours (RL only)	Lowest (1×)	Sub-optimal	✗
Ours (RL+ILP)	Low (5.5×)	Optimal	✓

Our solutions: In this work, we use DNNs computation graph scheduling as a case study to illustrate the possibility of accelerating exact CO problem solving using a combination of reinforcement learning (RL) and exact solving techniques. Specifically, we propose Inc-ILP, a novel graph scheduling framework that combines both of merits of RL-based schedul-

ing and the exact method, where RL agent performs coarse-grain solving, and the exact method refines the coarse-grained initial solving and produces the optimal result. Thus, as shown in Table I, Inc-ILP generates deterministic and optimal scheduling results with significantly reduced solving runtime cost. The main contribution of this work can be summarized as follow: **(1)** we adopt an RL-based framework to imitate the behavior of existing exact methods, which provides initialized scheduling results as output. **(2)** To refine the RL solution for optimality and determinism, we perform local relaxation w.r.t RL scheduling solution and use exact method to obtain optimal scheduling results. **(3)** We select Google Edge TPU [22] as the backend and integrate Inc-ILP in an end-to-end compile-to-deploy framework to evaluate real-world performance. **(4)** Finally, we perform a comprehensive evaluation of Inc-ILP in optimality and solving runtime compared to commercial Edge TPU compiler (heuristic) and pure exact method [4]. The results confirm the capability of Inc-ILP in generating optimal and deterministic results with significant speedups. As a result, we believe this work can offer some new intuitions for broadly improving determinism and (semi)optimality for ML in EDA and general combinatorial optimization.

II. Preliminary

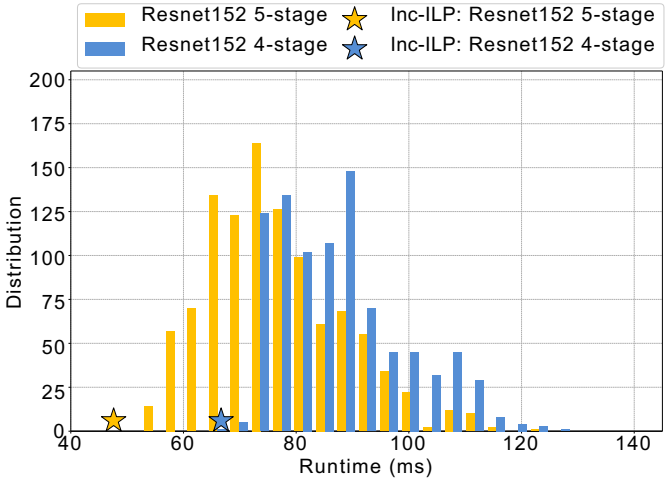


Fig. 1: Measurement system illustration of multi-stage pipelined Edge TPUs (physical system setup included in Figure 6). The performance of Inc-ILP scheduling results is located in the bottom-left corner with star markers.

A. Computational graph scheduling

DNN models can be represented as computation graph $G(V, E)$ with nodes V and edges E. The nodes represent the operators in DNN models and the edges describe the dataflow dependency that connects all operators. Specifically, the problem of multi-stage DNN model scheduling can be formulated as follow: given the input of a computation graph and a set of scheduling constraints including the number of pipeline stages and resource constraint (e.g., on-cache memory resource), DNN model scheduling aims to generate the optimal scheduling solution $S = s_0, s_1, ... s_n$, where $n \leq |V|$ and s_n represents the stage that node n scheduled. The operators in computation graphs will be deployed to given pipeline stages with the minimum (or maximum) optimization objectives like on-cache memory, communication cost, etc. Figure 2 displays a simple 3-stage pipelined Edge TPU scheduling outcome. Considering the left graph, the nodes for start, Input1 and Input2 are designated to stage-0. Stage-1 is composed of BatchNorm, ReLU, Conv1, and Conv2, while stage-2 incorporates Conv3, Zeropad, Avgpad, and end nodes.

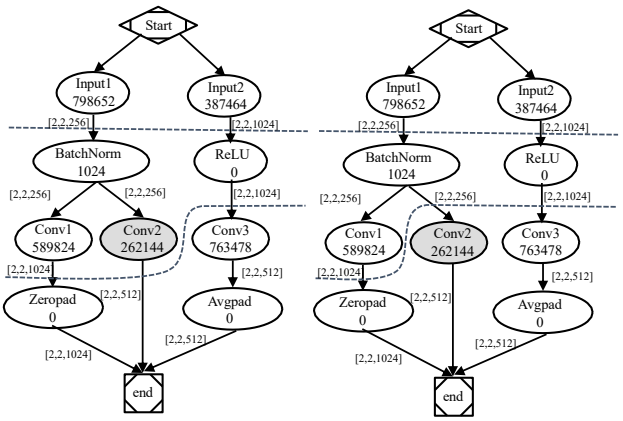


Fig. 2: Two distinct scheduling examples for a straightforward DNN computation graph differ solely in the allocation of the Conv2 node - the left scheduling results in reduced off-chip memory overhead, while the right scheduling minimizes device-to-device communication expenses.

While hardware device pipelining can improve DNN execution performance by decreasing the size of off-cache memory, the runtime performance is highly sensitive to scheduling solutions, and finding the optimal one is critical due to limited computation resources including memory, and communication bandwidth. To demonstrate the impacts of scheduling solutions on the pipelining system, we select Google Edge TPU as backends and perform a comprehensive runtime performance analysis using 4-stage, 5-stage ImageNet ResNet152 [5] models pipelining, which is shown in Figure 1. We can observe that (1) scheduling solutions make significant differences in pipelined Edge TPU systems, which could result in $2\times$ runtime difference; (2) many scheduling solutions might perform similarly but finding the best performance scheduling is statistically challenging. Hence, there is a great need of developing an near-optimal scheduling approach at low solving complexity. Note that for all two benchmarks, the performance of our proposed Inc-ILP scheduler is located in the left bottom corner which provides near-optimal runtime performance.

The optimal scheduling solution should aim to minimize costs, such as peak memory usage and communication expenses. Figure 2 displays two distinct scheduling approaches for a synthetic DNN computation graph, with the number of parameters represented in the vertices. Although both schedules are valid, they differ in the assignment of the N_{Conv2} node.

In the context of a three-stage pipelining system, the total execution time (T) of the computational graph is the sum of on-device execution runtime (T_d) and communication runtime (T_c), such that $T = T_d + T_c$. Here, T_c is primarily influenced by off-chip memory usage, while T_d is affected by the communication expenses, i.e. size of tensors passed between stages. The scheduling solution with lower off-chip memory usage and communication overhead will likely have a faster execution time. It is important to note that in a pipeline scenario, T_d is dominated by the stage with the longest runtime. Among the two scheduling solutions, the left scheduling in Figure 2 has a lower memory upper bound (76347 in stage 3), resulting in an improved T_c. On the other hand, the right scheduling ([2,2,1024]+[2,2,256]+[2,2,1024]) has a smaller device-to-device communication overhead between stages 2 and 3 compared to the left scheduling solution ([2,2,1024]+[2,2,512]+[2,2,1024]). As a result, the left scheduling has better memory usage and lower communication costs, leading to an optimized T_d. In this work, we concentrate on Edge TPU backends where T_c has a greater impact than T_d. Our aim is to first optimize the memory upper bound and then minimize device-to-device communication expenses.

B. Combinatorial problem solving methods

Graph scheduling is a NP-hard combinatorial optimization problem and there is no optimal solution in polynomial solving runtime. There existing some traditional approaches including reinforce learning, heuristic, and exact approach, etc.

Reinforcement learning [23], [24] has been applied to many combinatorial optimization problem [16], [25], [26]. In [24], Bello et al. present an innovative approach to addressing combinatorial optimization challenges through the application of neural networks and reinforcement learning. The authors specifically target the traveling salesman problem (TSP) and develop a recurrent neural network capable of predicting a probability distribution of various city arrangements, given a collection of city coordinates. The method generates high-quality solutions for various combinatorial optimization problems, often surpassing the performance of traditional heuristics and meta-heuristics. However, the solution based on reinforcement learning is non-deterministic and lacking optimal performance guarantees.

Several renowned heuristic algorithms exist, such as list scheduling and the force-directed algorithm. The force-directed algorithm incorporates a "force" on each operation and aims to balance computation across all stages by minimizing this "force". List scheduling, in contrast, ranks operations based on specific metrics, scheduling the operation with the highest priority when available. Despite their capabilities, both heuristic algorithms share common limitations: they lack determinism and cannot guarantee the optimal quality of results.

On the other hand, exact approaches [27]–[30] include integer linear programming (ILP), satisfiability modulo theories (SMT), and integer difference constraints (SDC). Integer Linear Programming (ILP) and Satisfiability Modulo Theories (SMT) guarantee finding optimal solutions and serve as potent methods for tackling intricate optimization problems. However, they share the same scalability issues as the size and complexity of the problems they address increase. For ILP, the combinatorial nature of integer variables leads to a rapid growth in the number of potential solutions, making it difficult for ILP solvers to find optimal solutions efficiently. The scalability of ILP solvers is further affected by the presence of multiple objectives or constraints, which can significantly increase the search space. On the other hand, SMT solvers deal with the satisfiability of logical formulas with respect to various background theories. As the size and complexity of these formulas grow, the search space for satisfying assignments becomes increasingly large and challenging to navigate. Additionally, SMT solvers may struggle with certain types of problems, such as those involving non-linear arithmetic, which can exacerbate the scalability issue.

III. Approach

A. Overview

The overall workflow of Inc-ILP including input, output is demonstrated in Figure 3. The Inc-ILP scheduling framework includes four phases: 1) RL pre-scheduling agent generates the coarse-grained scheduling results, 2) search space relaxation based on graph topological order, 3) Inc-ILP produces partially optimal results using exact solving, 4) scheduling results deployment on actual hardware devices.

In phase 1, we adopt the RL model as pre-scheduling agent architecture. We take the benefits of the RL model which can mimic the behavior of exact methods with short inference runtime. Phase 2 performs search space relaxation based on the order of computational graph topological sorting. Then we exactly solve the refined search space and generate the partial optimal solution in phase 3. Specifically, we build ILP constraint formulation to solve the refined search space, which includes dependency constraints, parameter constraints, etc. Finally, in phase 4, we select Google Edge TPU as the backend to deploy scheduling results. The multi-stage pipelined Edge TPU system is shown in Figure 6.

B. Phase 1: RL based pre-scheduling

As we mentioned before, RL pre-scheduling generates coarse-grained results, and the quality of results determines if we can achieve global optimality by search space relaxation. Figure 4 is the expanded illustration of the RL pre-scheduling agent specifications, which are composed of 4 steps (Phase 1.1 - Phase 1.4): embedding, encoder, decoder, and RL model training.

Computation graph embedding (Phase 1.1) - The embedding of the computational graph extracts the key information and passes it to the RL model. The graph embedding consists of several components: (1) the absolute and relative coordinates of each node: for node N_i, its absolute coordinate is the topological level of N_i, and the relative coordinate will be the topological level of parent node N_i. Note that we perform depth-first-search (DFS) algorithm to generate topological level, where each node is ordered to the closest position from the source node. For example, the topological order of node

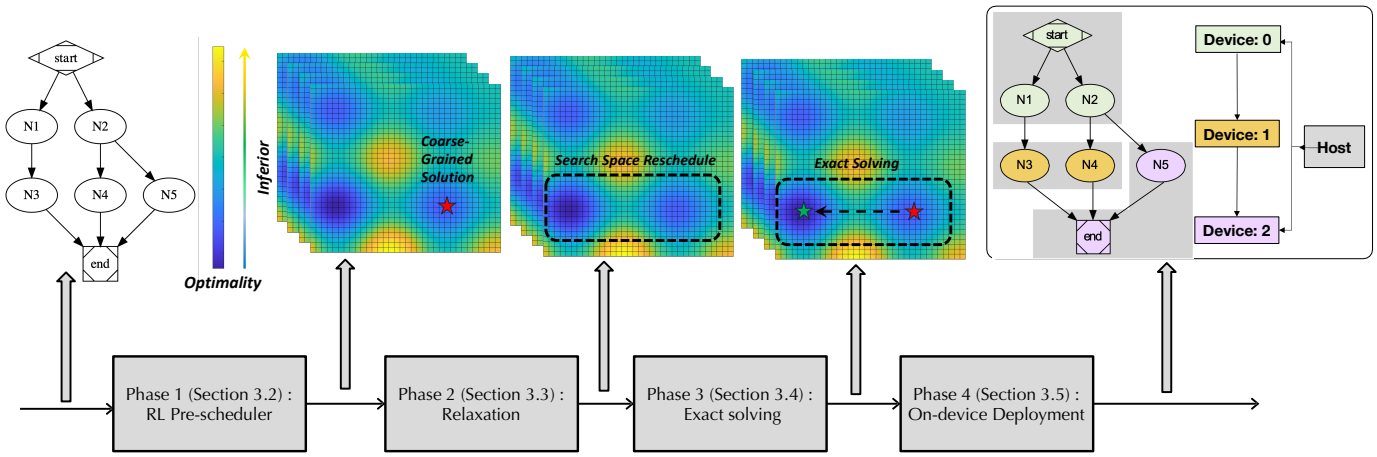


Fig. 3: Inc-ILP overview including four-phases: RL pre-scheduling, search space relaxation, exact solving, and device deployment.

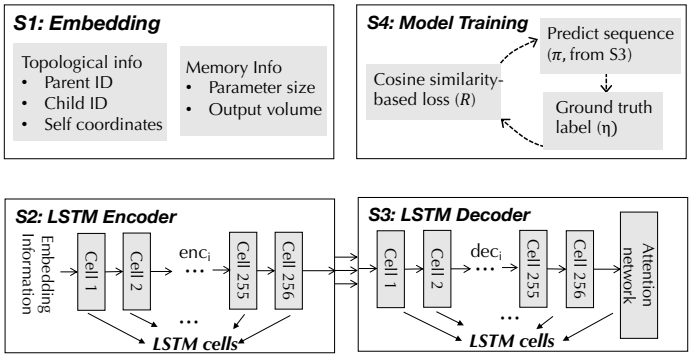


Fig. 4: Four-steps Reinforcement Learning (RL) Agent: The encoder processes embedded data, converting it into contextual information, which is then relayed to the decoder. The decoder generates a predicted sequence, and a cosine similarity-based loss is computed to train the RL agent.

$Concat$ in Figure 5 is 2. (2) node IDs: it is generated by hashing all operator names. For node N_i, its node ID is N_i. (3) memory consumption of each node and its output volume size: scheduling objectives of DNN model pipelining include minimizing peak-memory parameter caching and device-to-device communication. So, memory consumption and output volume size are the most critical attributes of each operator, which can be obtained from model attributes.

RL agent architecture (Phase 1.2, 1.3) - As shown in Figure 4. The RL agent architecture is composed of an encoder and a decoder. Both the encoder and decoder are built with 256 Long Short-Term Memory (LSTM) cells [31].

Encoder - The encoder digests the graph embedding and transforms it into context information. For each LSTM cell in the encoder, it produces a sequence of latent (enc_i) memory states which record the encoding information and propagate along the LSTM dataflow. The last latent memory state will be passed to the decoder as input.

Decoder - With the computation graph embedding and the latent memory state from the encoder, decoder produces a selection probability distribution over candidate computation nodes using the pointing mechanism. Firstly, similar to the encoder, decoder also generates another latent memory state (dec_i) at each LSTM cell. Then the latent memory will be passed to an attention network [32] to produce candidate node probability distribution. We adopt the pointer network (PtrNet) [24] in the attention network, which has been applied to many graph combinatorial optimization problems. Finally, the candidate node probability distribution and latent memory state will be passed to the next LSTM cell. Note that the first decoding latent memory state is a trainable parameter.

RL model training (Phase 1.4) - In this step, we iteratively optimize the parameter in the RL agent. We choose the cosine similarity as the basis for constructing the reward function. For each node i in computation graph G(V, E) ($i \leq |V|$), we use $\eta(i)$ and $\pi(i)$ to demonstrate the ground truth label and RL agent output sequence respectively. The cosine similarity can be shown in Equation 1. Here we introduce a constant η to ensure the validity of the equation.

$$R = \frac{\sum(\pi(i) \cdot \eta(i))}{\max(\sqrt{\sum \pi(i)^2} \cdot \sqrt{\sum \eta(i)^2}, \epsilon)} \tag{1}$$

The reward function for model parameter θ optimization policy is the maximization of the *cosine similarity*.

$$J(\theta|G) = \mathbb{E}_{\pi \sim p_\theta(\cdot|G)}(1 - R(\pi|G)) \tag{2}$$

For ground truth label $\eta(i)$, we use exact methods to generate the global optimal scheduling results. RL pre-scheduling agent aims to imitate the behavior of the ground truth label.

Training dataset - Collecting sufficient data for Reinforcement Learning (RL) model training is one of the most significant challenges. The DNN computational graphs is limited, which restricts the scalability of RL agents. This challenge is exacerbated by the need to balance dataset coverage and graph size. To overcome this issue, we employ randomly generated synthetic datasets to train our RL model, which

offers better data coverage over graph complexity and size while avoiding the need for extensive data collection. To ensure the generated DAGs align with real-world distributions, we gather statistical data from real-world DNN models and analyze the key information including graph size and complexities (maximum in-degree for nodes in the computational graph), etc. Building on this insight, we introduce a DAG generator tailored for RL training. This generator consistently outputs computational graphs with a size of $|V| = 30$ and various graph complexities ranging from $deg(V) \in 2, 3, 4, 5, 6$, mirroring the distributions found in real-world scenarios. Our DAG generator provides a total of **1 million** computational graphs, with 200,000 graphs for each complexity $deg(V)$ in $2, 3, 4, 5, 6$.

The synthetic dataset has superior generalizability with test datasets for two main reasons. Firstly, it mimics the graph structure and provides a similar degree of the test dataset, ensuring that the RL model is trained on synthetic data that closely resembles the test data. Secondly, the graph sampler generates a vast dataset of over 1 million synthetic graphs, providing extensive coverage of different types of test graphs. This large and diverse dataset improves the robustness and performance in real-world scenarios.

C. Phase 2: Search Space Relaxation

The main drawback of coarse-grained scheduling results from RL pre-scheduling is lacking determinism and optimality. In this section, we perform relaxation to broaden the search space and obtain the partial optimum. Space relaxation enables the application of exact methods to produce partial optimums without considering about scalability issues.

Figure 5 illustrates the search space relaxation of 2-stage pipelining with the relaxation level $\gamma = 1$. Firstly, RL pre-scheduling results pipeline the input model into 2 stages, where all boundary edges cut through stage-0 to stage-1. Second, we create an ASAP (As Soon As Possible) topological sort for the computation graph, positioning each node as near to the source node as feasible. The ASAP topological sorting can be seen on the left side of Figure 5. Thirdly, we find the earliest stage s_i of the source node and the latest stage s_j of the sink node in the boundary edges. Finally, we refine all nodes whose topological sorting order is located between s_i to s_j. Using the computational graph in Figure 5 as an example, the boundary edges include {Relu_1, Conv_1}, {Conv_2, Add}, and {Concat, Add}. We use the star marker to identify the boundary edges. The earliest topological sorting level for source nodes is 2 (Relu_1, Concat). Similarly, the latest sorting level for sink nodes is 4 (Add). Because we assume $\gamma = 1$, all nodes whose sorting stage is located between 1 to 5 will be refined in the next step using exact methods. **While the exact methods based scheduling problem is an NP-hard problem, the relaxed problem on top of RL-based scheduling is significantly simplified due to drastic search space reduction.**

D. Phase 3: Exact Solving via Multi-Obj ILP

Phase 3 utilizes SDC+ILP-based scheduling [4] to execute the final refinement solving, aiming for optimal solutions. According to the systematic design of our multi-stage Edge TPU pipelining system, we adopt the following three optimization objectives when constructing ILP formulations: (1) peak memory usage for parameter caching; (2) total off-cache memory across all stages; (3) maximum communication cost in the multi-stage pipelining system. This section discusses the development of the ILP formulation.

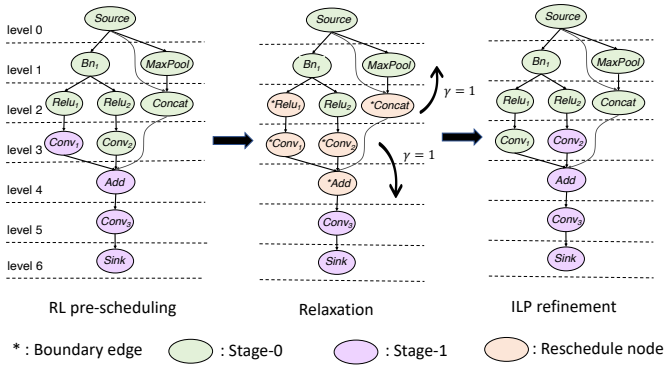


Fig. 5: Search space relaxation based graph topological sorting.

ILP formulation generation – In order to reschedule the refined search space, we construct an Integer Linear Programming (ILP) formulation comprised of various constraints. This formulation includes data dependence constraints, parameter caching constraints, and communication cost constraints, among others. In this section, we provide a concise overview of these three representative constraints.

(1) Dependence constraint – Dependence constraint formulates the dependence correctness of operators in the computation graph. Specifically, for edge (N_i, N_j), the dependence constraint requires the N_i to be executed before N_j. We use s_i and s_j to represent the stage of node N_i, N_j. Then the dependence constraint [4] can be formulated as:

$$s_{\mathrm{i}} - s_{\mathrm{j}} \leq 0 \tag{3}$$

(2) Parameter caching constraint – The most important optimization objective is the peak memory footprint m_{peak}. We use m_k to denote the memory usage in stage k, which can be estimated by the sum of parameter sizes of operators that are scheduled to stage k. We use the following expression to formulate the parameter caching constraint:

$$m_{\mathrm{k}} = \sum_{\forall v \in V} p_v \cdot s_v^k \tag{4}$$

where variable p_v is the estimation of memory consumption for operator v based on parameter size, and s_v^k is binary variable to denote if operator v is scheduled to stage k.

m_{peak} is the highest memory footprint among all stages:

$$\begin{cases} \textbf{min} \quad m_{\mathrm{peak}} \\ \wedge \quad m_{\mathrm{k}} \leq m_{\mathrm{peak}} \\ \wedge \quad \text{Equations (3)} \end{cases} \tag{5}$$

(3) Communication cost optimization – Communication cost constraints build ILP formulations to minimize the communication cost (i.e., tensor size) $com_{k,k+1}$ transferred between

stages k and stage $k+1$. For edge the e from node v_i to node v_j, we build the following ILP formulations:

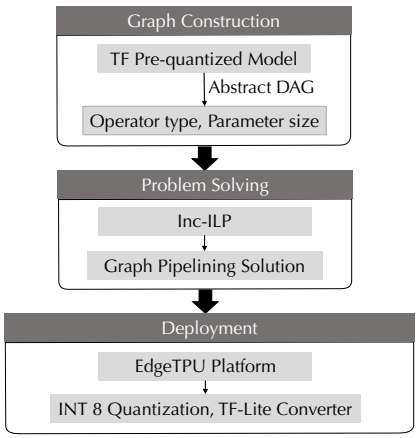
$$com_{k,k+1} = \sum_{e} t_{\mathrm{e}} \cdot \alpha_{\mathrm{e}} \tag{6}$$

where t_{e} represents the tensor size between nodes v_i and v_j, and α_{e} is a binary variable to denote if $(s_i == k) \wedge (s_i < s_j)$.

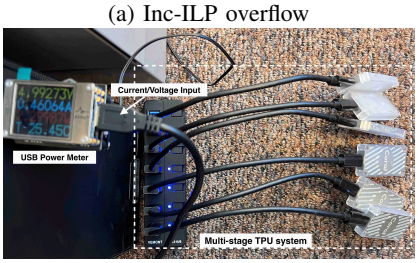
In the previous formulation, we employ logical operators to represent the variable α_{e}. Consequently, we require an additional set of constraints to represent these logical operators. For instance, consider the logical AND operation. Given two binary variables x and y, the logical AND $\mathrm{ILP}_{\wedge}(x, y)$ can be represented as follows:

$$\begin{cases} \mathrm{ILP}_{\wedge}(x,y) \geq x + y - 1 \\ \mathrm{ILP}_{\wedge}(x,y) \leq x \\ \mathrm{ILP}_{\wedge}(x,y) \leq y \end{cases} \tag{7}$$

E. Phase 4: Deployment and system integration



(a) Inc-ILP overflow



(b) Multi-stage pipeline Edge TPUs system

Fig. 6: Inc-ILP end-to-end overflow and multi-stage pipeline Edge TPUs and energy measurement system.

We build an on-device framework that integrates the Inc-ILP with multi-stage pipelined Edge TPU systems and the illustration of the integration framework is shown in Figure 6. The workflow includes the following several steps:

- **Graph construction -** The Inc-ILP approach takes a graph, representing the dataflow of deep neural networks (DNNs), as its initial input. Our framework initially extracts the computation graph from TensorFlow Lite (TFLite), which includes details such as operator type, parameter dimensions, and output volume size, among others.
- **Scheduling via Inc-ILP -** With the computational graph, Inc-ILP solves the formulations generated in the prior step and extracts solutions corresponding to the original model(s) TensorFlow frozen graph. Subsequently, our framework employs the TFLite TOCO converter to generate model subgraphs, which will be assigned to particular Edge TPU devices based on the obtained solution.
- **Deployment -** In the final step, our framework utilizes the Edge TPU Compiler to transform the subgraph operations into Edge TPU-specific operations and deploy the subgraphs onto Edge TPUs without any additional optimizations. As all Edge TPU models undergo INT8 quantization prior to deployment, our scheduling framework factors in this quantization when generating communication and memory caching constraints.

TABLE II: Statistics of DNN computational graph benchmarks

	Xception [33]	ResNet50 [5]	ResNet101 [5]	ResNet152 [5]
$\lvert\mathbf{V}\rvert$	134	177	347	517
$\mathbf{deg}(\mathbf{V})$	2	2	2	2
Depth	125	168	338	508
	DenseNet121 [34]	ResNet101v2 [35]	ResNet152v2 [35]	DenseNet169 [34]
$\lvert\mathbf{V}\rvert$	429	379	566	597
$\mathbf{deg}(\mathbf{V})$	2	2	2	2
Depth	428	371	558	596
	DenseNet201 [34]	InceptionResNetv2 [36]		
$\lvert\mathbf{V}\rvert$	709	782		
$\mathbf{deg}(\mathbf{V})$	2	4		
Depth	708	571		

IV. Experiment

In this section, we aim to experimentally demonstrate the results and effectiveness of Inc-ILP results. The comparison baselines include exact method (ILP) and the heuristic method (commercial EdgeTPU compiler). We demonstrate the performance of our work from three aspects: quality of results w.r.t parameter caching, solving runtime speedup, and on-device runtime performance.

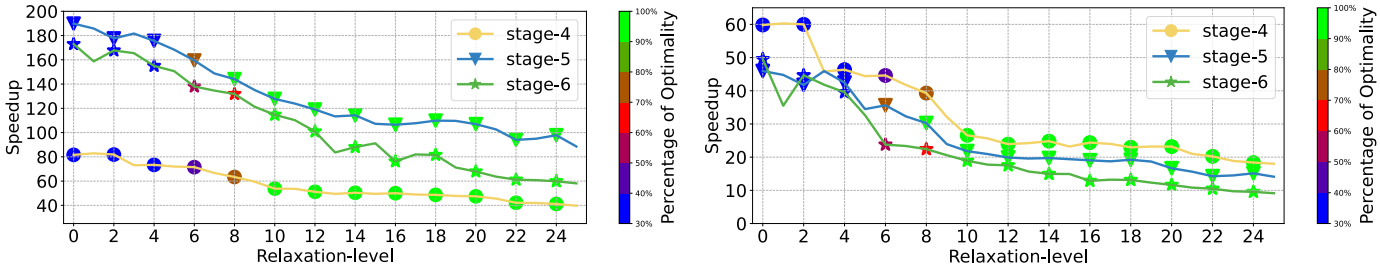
A. Experiment Setup

We select Google Edge TPU as Inc-ILP backends and choose 10 ImageNet classification models as benchmarks to verify the effectiveness of Inc-ILP which is shown in Table II. During the RL model training process, we use Nvidia 2080 Ti with Intel Xeon Gold 6230 x20 CPU. For RL training setup, we select 10^{-4} as the learning rate with 300 epochs and batch size 128 using Adam optimizer. As we mentioned before, the RL encoder and decoder are built with 256 LSTM blocks. Besides, we adopt the IBM ILOG CPLEX [1] as the incremental ILP solver. Finally, we select the Edge TPU compiler as the baseline which is a heuristic-based DNN scheduling commercial tool for Edge TPU devices. We build a multi-stage Edge TPU system for on-device evaluation which is shown in Figure 6b.

B. Quality of results and solving runtime analysis

The main goal of Inc-ILP is to achieve optimal solutions at significantly reduced runtime. Thus, we thoroughly examine its capacity to achieve optimal solutions and analyze the runtime expenses involved in the process.

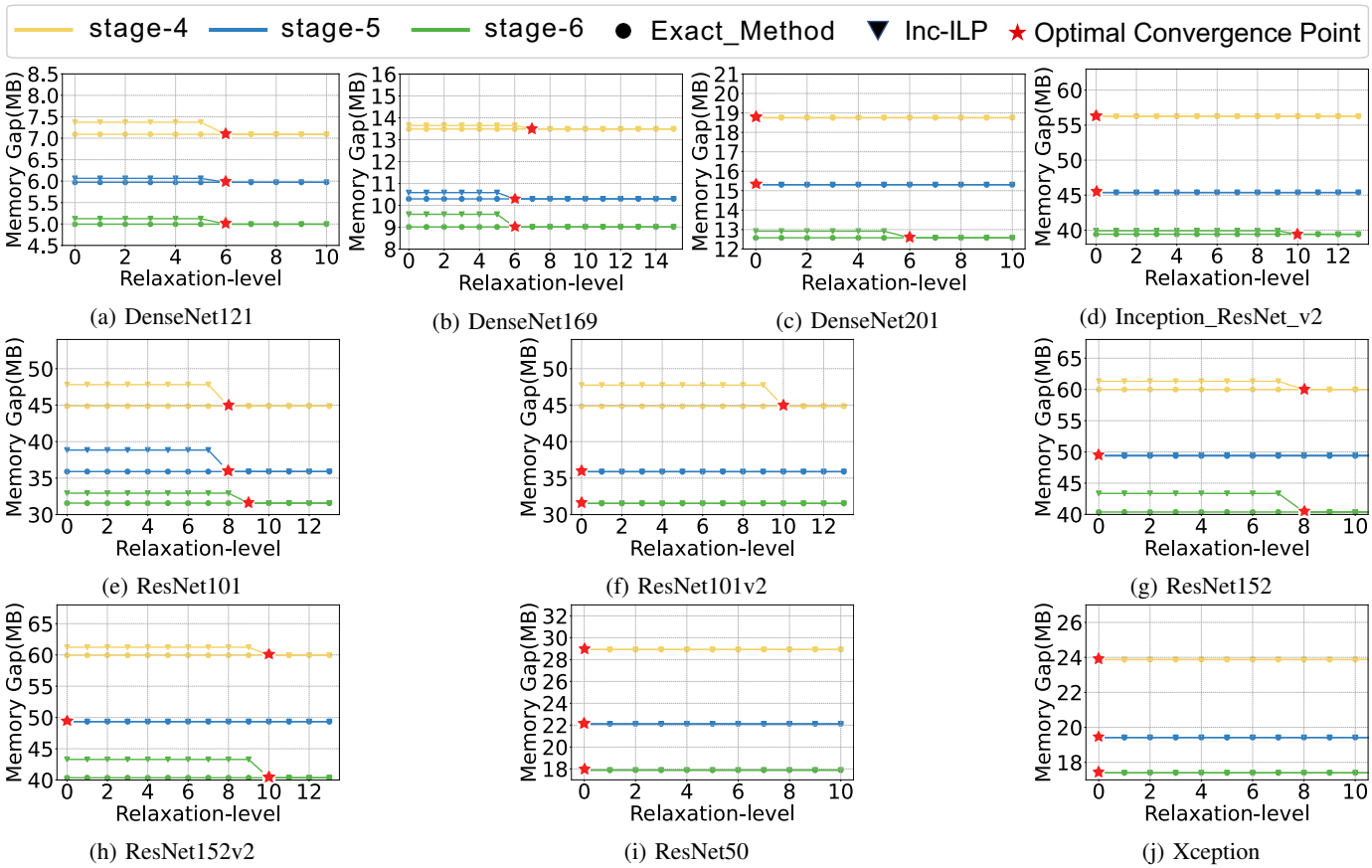
[1] https://www.ibm.com/analytics/cplex-optimizer



(a) Speedup over exact methods

(b) Speedup over Edge TPU compiler

Fig. 7: Inc-ILP solving runtime speedup over two baselines. According to the colorbar, the percentage of benchmarks reaching optimum is shown in marker color. Inc-ILP provides global optimal scheduling results with significant solving runtime speedup.



(a) DenseNet121 (b) DenseNet169 (c) DenseNet201 (d) Inception_ResNet_v2

(e) ResNet101 (f) ResNet101v2 (g) ResNet152

(h) ResNet152v2 (i) ResNet50 (j) Xception

Fig. 8: Convergence analysis w.r.t relaxation depth using parameter caching metric as example. All three objectives reach optimal after the convergence point.

First, we compare the solving runtime speedups of Inc-ILP with two baselines: ILP-based exact method and commercial Edge TPU compiler, while the quality-of-result metrics if the percentage of DNNs graphs are optimally scheduled (Figure 7). The vertical axis is the Inc-ILP average solving runtime speedup over all ten benchmarks in Table II, where the percentage of benchmarks that achieve global optimum are presented w.r.t the color-bar. As shown in Figure 7, Inc-ILP offers significant solving runtime speedup over exact method with identical optimal solutions, where full optimality (100%) is reached at relaxation level $\gamma = 10$. Specifically, Inc-ILP provides $54\times$, $128\times$, and $115\times$ speedups for 4-stage, 5-stage, and 6-stage scheduling, respectively ($\gamma = 10$), with full optimality achieved. This confirms that RL-based initialization successfully narrows the search space, which alleviates the scalability issue of exact methods and makes finding the optimal solution within short solving runtime possible. Secondly, Inc-ILP offers $27\times$, $22\times$, and $19\times$ speedups ($\gamma = 10$) over EdgeTPU compiler (heuristic), while offering guarantees in optimality. In conclusion, Inc-ILP not only provides the solving runtime improvement but also generates the optimal scheduling solution. We can also see that relaxation level γ is an important factor that drives the

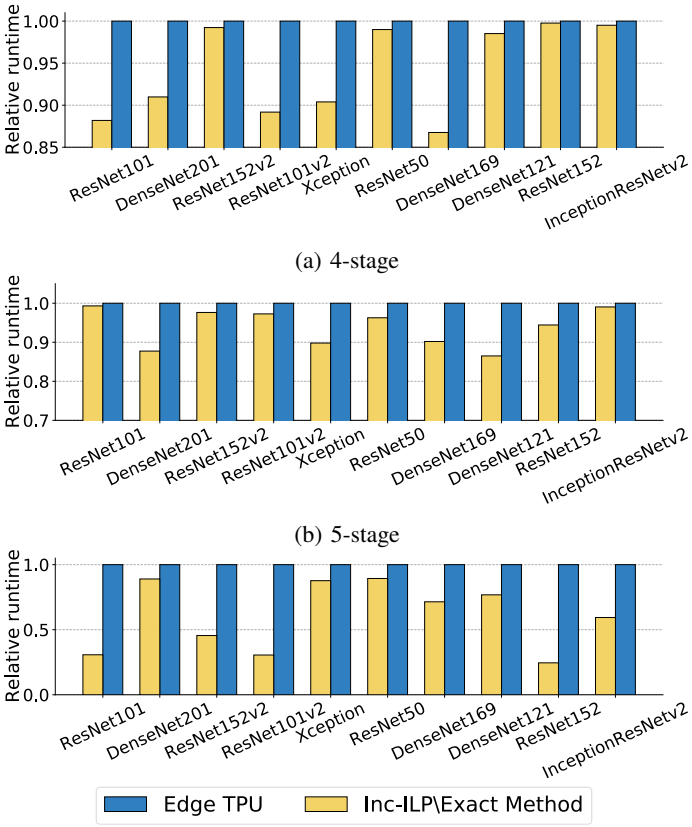
trade-off between optimality and runtime.

To understand the impacts of the relaxation level γ, we perform a case-by-case analysis for each benchmarks, shown in Figure 8. While we use the peak parameter caching as the objective metric for this evaluation, the three metrics all reach optimal points after careful search space relaxation. In this figure, we compare the size of peak memory usage between Inc-ILP and exact methods where the x-axis and y-axis represent the relaxation level γ and the parameter caching results of Inc-ILP and the optimum. Three important conclusions can be derived from Figure 8. First, RL-based pre-scheduling provides a great initialization result for later exact ILP solving. For example, three benchmarks achieve the global optimum without ILP refinement using RL-based scheduling ($\gamma = 0$). As for all benchmarks, the scheduling results from the RL pre-scheduling agent have peak parameter caching gaps of 1.88%, 1.12%, and 4.50% to optimum, in 4-stage, 5-stage, and 6-stage pipelining, respectively. The quality of the initialization confirms the great confidence in including the global optimal solution in the relaxed search space. Second, most benchmarks achieve optimal scheduling results with small relaxation levels. For example, ResNet152 depth is 508, while the optimal convergence point is reached by Inc-ILP with only $\gamma = 8$. We observe that for all graphs tested, the 10-level relaxation is sufficient to generate global optimal results for all cases. Increasing the relaxation level beyond 10 brings marginal benefits but could reduce the advantages of Inc-ILP in solving runtime speedups. Finally, although the graph size of DNN computation graph benchmarks is vary from 134 to 782, the relaxation level to achieve optimal peak-memory results remains stable for different benchmarks, which proves the scalability and generalizability of Inc-ILP. For example, both DenseNet121 and DenseNet201 achieve optimal results when $\gamma = 6$, but they differ in graph size (V=429 for DenseNet121 and V=709 for DenseNet201).

C. On-device inference runtime analysis

Finally, we compare the runtime performance between Inc-ILP, exact methods, and commercial Edge TPU compiler. The test system is shown in Figure 6 and the pipelined Edge TPU devices are connected to stable 5V supplying voltage for all tests. The on-chip inference runtime results are shown in Figure 9. The vertical axis represents the relative runtime normalized to the Edge TPU compiler. Note that as we mentioned before, Inc-ILP has identical scheduling results with exact methods. Thus we merge the scheduling results of Inc-ILP and exact methods into one column.

As shown in Figure 9, Inc-ILP has significant runtime performance improvement over the Edge TPU compiler. Inc-ILP benefits from the ILP refinement and guarantees partial optimality. Overall, Inc-ILP provides 1.065×, 1.068×, and 2.056× execution runtime performance speedup in 4-stage, 5-stage, and 6-stage pipelining. For example, Inc-ILP provides 4.07× runtime speedup for ResNet152 in 6-stage pipelining. Besides, we observe more substantial performance improvement in 6-stage pipelining. The main reason is the scheduling results



(a) 4-stage

(b) 5-stage

(c) 6-stage

Fig. 9: On-device runtime comparison (Inc-ILP meet same performance of exact method). Inc-ILP performs better than Edge TPU compiler for all benchmarks.

quality degradation of heuristic methods when scheduling complexity increases for large number stage pipelining.

V. Conclusion

This work proposes a novel approach in enhancing ML-based optimization approaches in determinism and optimality. The key concept leverages ML-based solving at earlier stage that is coarse but low runtime cost, and deploys exact solving to refine the coarse solution to reach optimality from significantly reduced search space. Specifically, we propose a novel framework for scheduling, namely Inc-ILP, a DNN model scheduling framework that combines both exact methods and RL-based scheduling approaches. RL pre-scheduling agent narrows the searching space and Inc-ILP strategically refines RL initialization results with the exact method. Inc-ILP benefits from the scalability of the RL-based scheduling approach as well as the scheduling optimality of exact methods. The experimental results confirm the proposed concept, where Inc-ILP obtains the exact optimal scheduling with significant speedups over exact methods, and heuristic-based commercial EdgeTPU compiler (non-optimal scheduling).

Acknowledgement This work is funded by National Science Foundation under awards NSF-2019336, NSF-2008144, and NSF-2047176.

REFERENCES

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017, pp. 1–12.

[2] J. W. Rittinghouse and J. F. Ransome, *Cloud computing: implementation, management, and security*. CRC press, 2017.

[3] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

[4] Z. Zhang and B. Liu, "SDC-Based Modulo Scheduling for Pipeline Synthesis," *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2013.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[6] J. Yin, Z. Zhang, and C. Yu, "Exact memory-and communication-aware scheduling of dnns on pipelined edge tpus," in *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*. IEEE, 2022, pp. 203–215.

[7] T. Yang and A. Gerasoulis, "List scheduling with and without communication delays," *Parallel Computing*, 1993.

[8] B. H. Ahn, J. Lee, J. M. Lin, H.-P. Cheng, J. Hou, and H. Esmaeilzadeh, "Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices," *arXiv preprint arXiv:2003.02369*, 2020.

[9] T. Chen *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *OSDI*, 2018, pp. 578–594.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[11] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[12] H. Ren and J. Hu, *Machine Learning Applications in Electronic Design Automation*. Springer Nature, 2023.

[13] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen, J. Wu, Y. Xu, H. Zhang, K. Zhong *et al.*, "Machine learning for electronic design automation: A survey," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 5, pp. 1–46, 2021.

[14] C. Yu and Z. Zhang, "Painting on placement: Forecasting routing congestion using conditional generative adversarial nets," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[15] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[16] N. Wu, Y. Li, C. Hao, S. Dai, C. Yu, and Y. Xie, "Gamora: Graph learning based symbolic reasoning for large-scale boolean networks," *Design Automation Conference (DAC'23)*, 2023.

[17] C. Yu, "Flowtune: Practical multi-armed bandits in boolean optimization," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[18] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 270–288.

[19] H. Chen and M. Shen, "A deep-reinforcement-learning-based scheduler for fpga hls," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[20] S. Sheng, P. Chen, Z. Chen, L. Wu, and Y. Yao, "Deep reinforcement learning-based task scheduling in iot edge computing," *Sensors*, vol. 21, no. 5, p. 1666, 2021.

[21] J. Yin, Y. Li, D. Robinson, and C. Yu, "Respect: Reinforcement learning based edge scheduling on pipelined coral edge tpus," *arXiv preprint arXiv:2304.04716*, 2023.

[22] A. Boroumand *et al.*, "Mitigating edge machine learning inference bottlenecks: An empirical study on accelerating google edge models," *arXiv preprint arXiv:2103.00768*, 2021.

[23] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3, pp. 229–256, 1992.

[24] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

[25] C. Yu and W. Zhou, "Decision making in synthesis cross technologies using lstms and transfer learning," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, 2020, pp. 55–60.

[26] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a sat solver from single-bit supervision," *arXiv preprint arXiv:1802.03685*, 2018.

[27] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[28] K. Fan, M. Kudlur, H. Park, and S. Mahlke, "Cost sensitive modulo scheduling in a loop accelerator synthesis system," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE, 2005, pp. 12–pp.

[29] G. Ramalingam, J. Song, L. Joskowicz, and R. E. Miller, "Solving Systems of Difference Constraints Incrementally," *Algorithmica*, 1999.

[30] S. Dai, G. Liu, and Z. Zhang, "A scalable approach to exact resource-constrained scheduling based on a joint sdc and sat formulation," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 137–146.

[31] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[33] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.

[34] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European conference on computer vision*. Springer, 2016, pp. 630–645.

[36] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-first AAAI conference on artificial intelligence*, 2017.