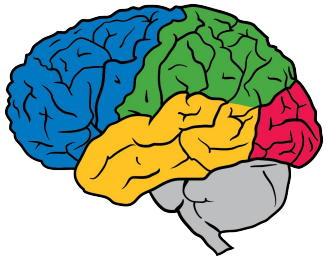


Google

ML for ML Compilers

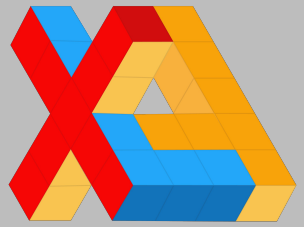
Yanqi Zhou, Google Brain

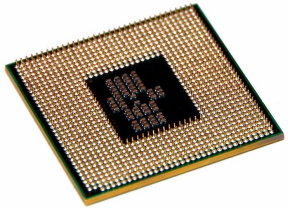
Background

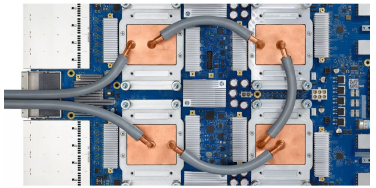
A compiler transforms a program into machine executable code.

PyTorch

...







Background

- General in face of explosion of models and frameworks.
- Targeting/Retargeting various hardware backends.

Figure credit: https://tvm.apache.org/2019/03/18/tvm-apache-announcement

Background

Search-based Compilers

optimization scope

graph	TASO PET	DeepCuts
subgraph		TVM Halide TensorComp... FlexTensor Ansor AdaTune Chameleon

Background

Search-based Compilers

optimization scope

graph	TASO PET	DeepCuts
subgraph		TVM Halide TensorComp... FlexTensor Ansor AdaTune Chameleon

Background

Search at sub-graph level can be sub-optimal!

A common strategy partitions a graph into subgraphs according to the neural net layers, ignoring cross-layer optimization opportunities.

Empirical result: a **regression** of **up to 2.6x** and **32% on average** across 150 ML models by limiting fusions in XLA to be within layers.

Background

Long Compilation Time

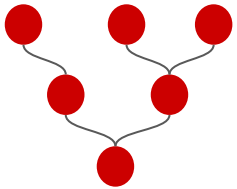
optimization scope

graph	XLA	TASO PET	DeepCuts
subgraph			TVM Halide TensorComp... FlexTensor Ansor AdaTune Chameleon
	seconds	***minutes***	***hours***

compile time (for ResNet like inference)

Background

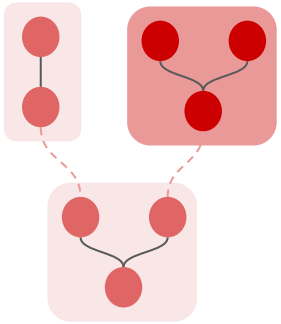
A XLA TPU Compiler



tensor computation graph

Graph-Level Optimizations

Algebraic simplification,
Dot conversion,
Layout assignment,
Cross-replicas sharding,
Operator fusion,
Rematerialization,
Operator scheduling



kernels / subgraphs

Kernel-Level Optimizations

Loop tiling,
Loop ordering/unrolling,
Heuristics parameters (flags),
Overlapping data-transfer & compute,
2D register mapping

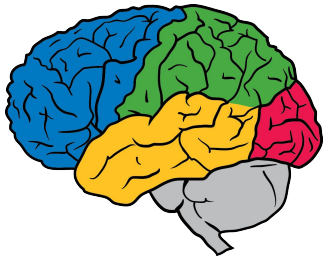
Today's agenda

- **Supervised Learning**
  - A Learnt Performance Model for TPUs, MLSys 2021
- **Reinforcement Learning**
  - GO: Transferable Graph Optimizers for ML Compilers, NeurIPS 2020
- **Production**
  - Partitioning ML Models on Multi-Chip Modules, MLSys 2022

Today's agenda

- **Supervised Learning**
  - A Learnt Performance Model for TPUs, MLSys 2021
- **Reinforcement Learning**
  - GO: Transferable Graph Optimizers for ML Compilers, NeurIPS 2020
- **Production**
  - Partitioning ML Models on Multi-Chip Modules, MLSys 2022

Google

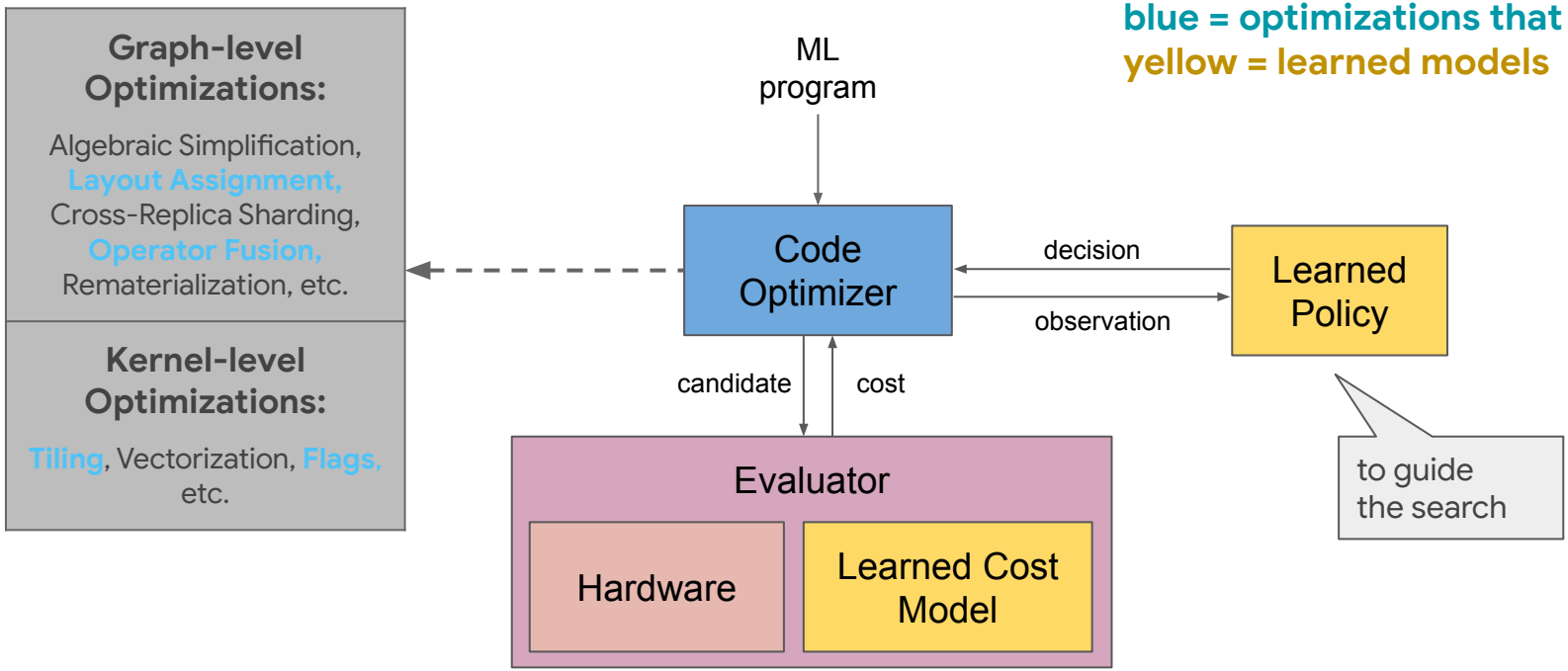


A Learnt Performance Model for TPUs

Samuel Kaufman*, Mangpo Phothilimthana, Yanqi Zhou,
Charith Mendis, Amit Sabne, Mike Burrows

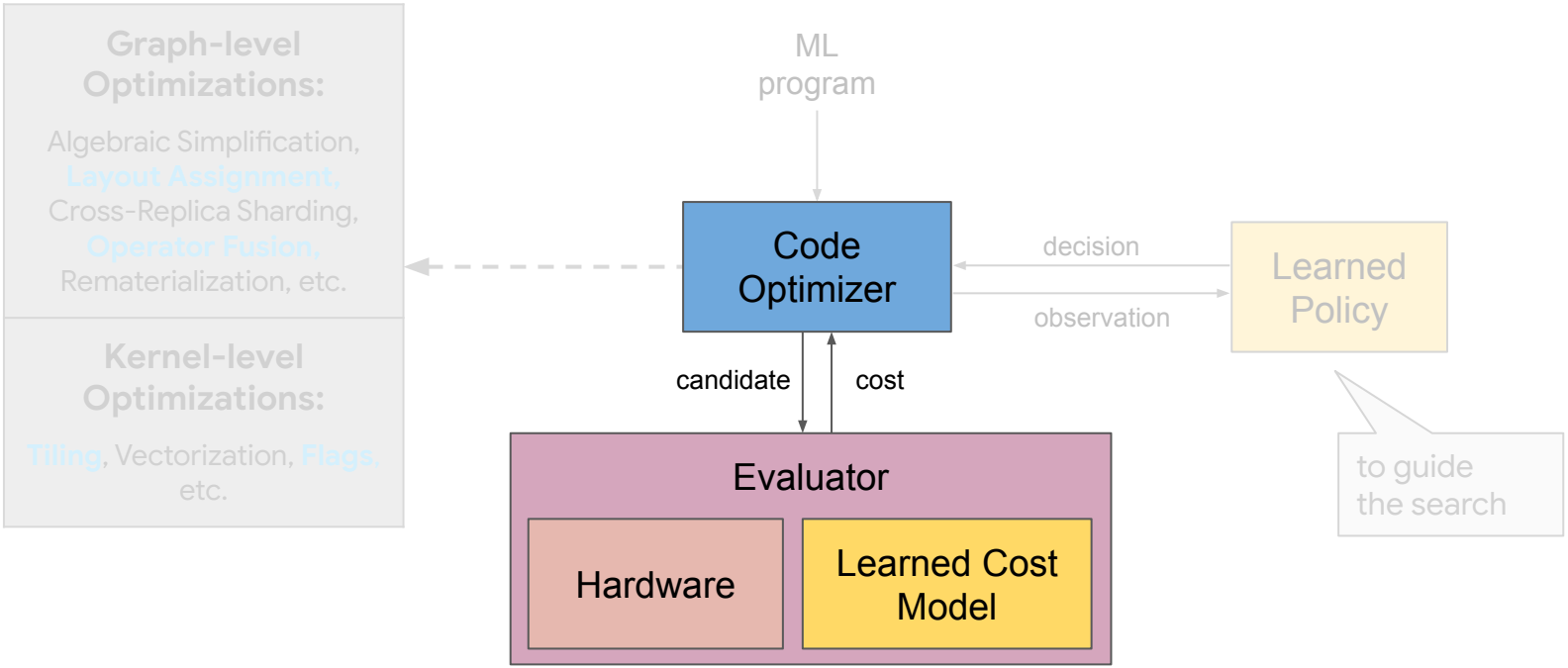
*Internship work at Google

XTAT: XLA TPU Autotuner



Phothilimthana et al.,"A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers", PACT 2021.

Learned Cost Model



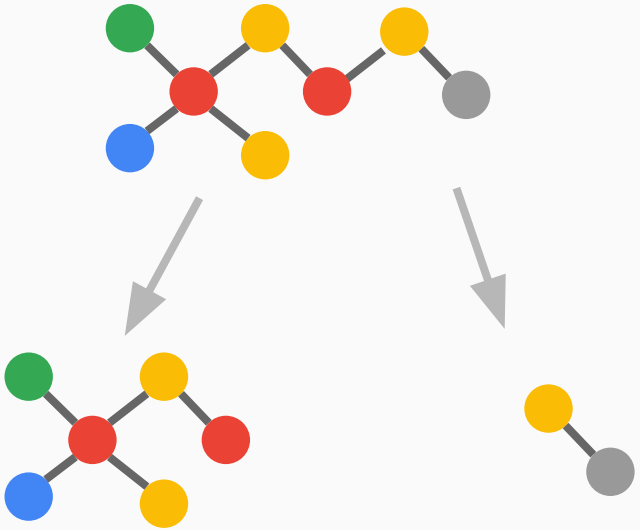
Kaufman and Phothilimthana et al., A Learned Performance Model for Tensor Processing Units, MLSys 2021.

Learned Cost Model: Objective

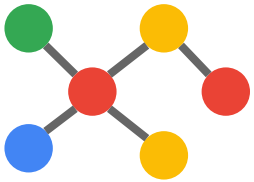
- General enough to handle non-trivial constructs
  - e.g. multi-level loop nests
- Generalize across programs of different application domains
- Should not rely on well-crafted features
- Retargetable to different optimization tasks

Overview of Cost Model

1. Decompose Into Kernels



2. Regression Per Kernel

$f($  $)$

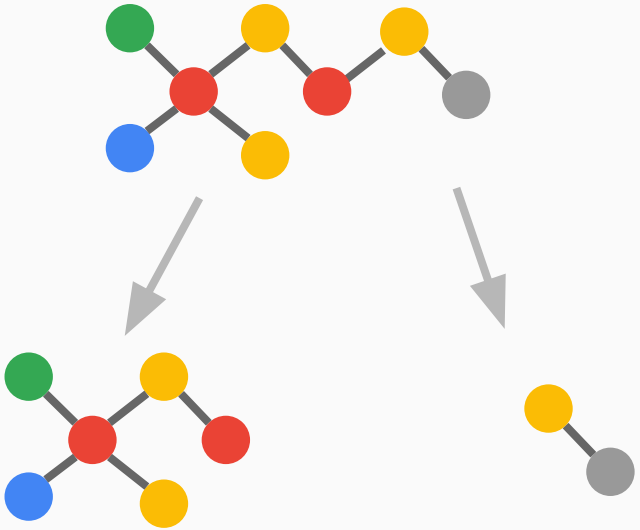
KERNEL

$\cong 5.2s$

RUNTIME

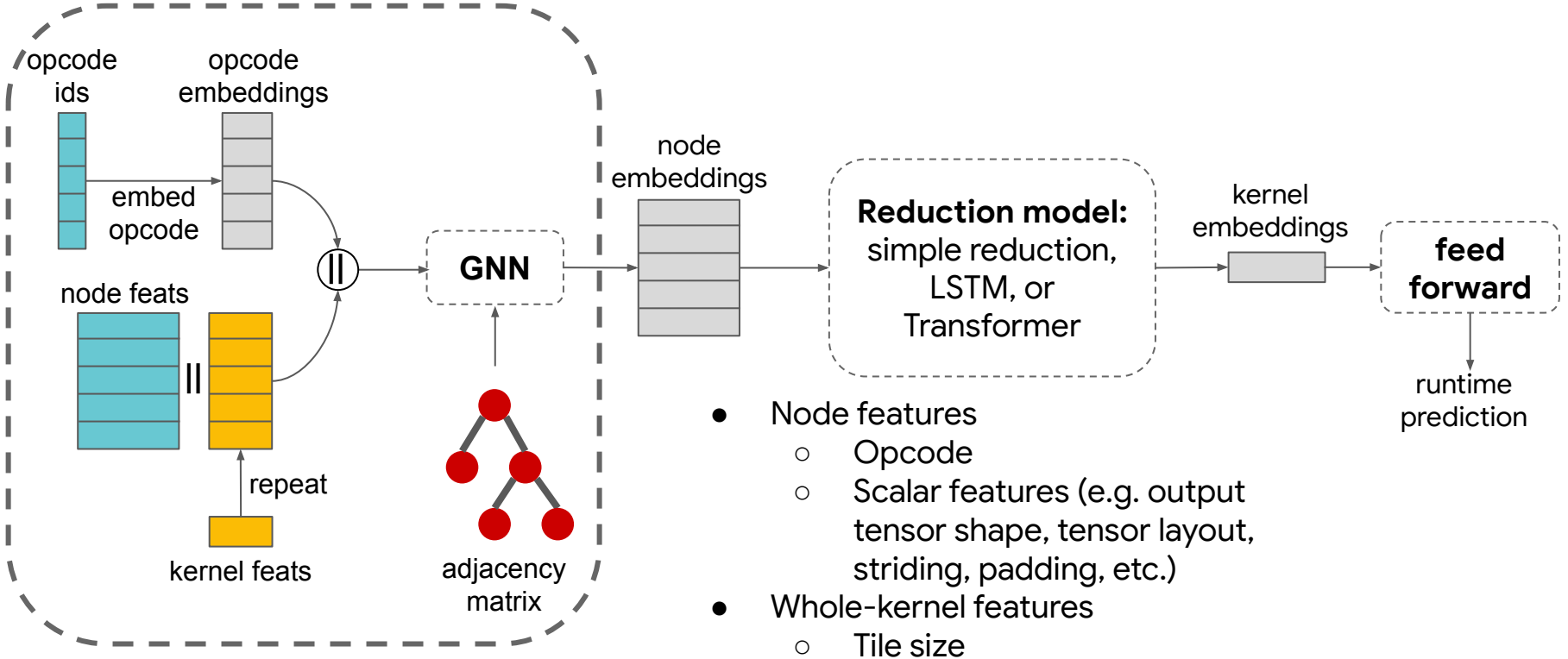
Benefits of Kernel-level Regression

1. Decompose Into Kernels



- General to various tasks
- More accurate at low-level representation

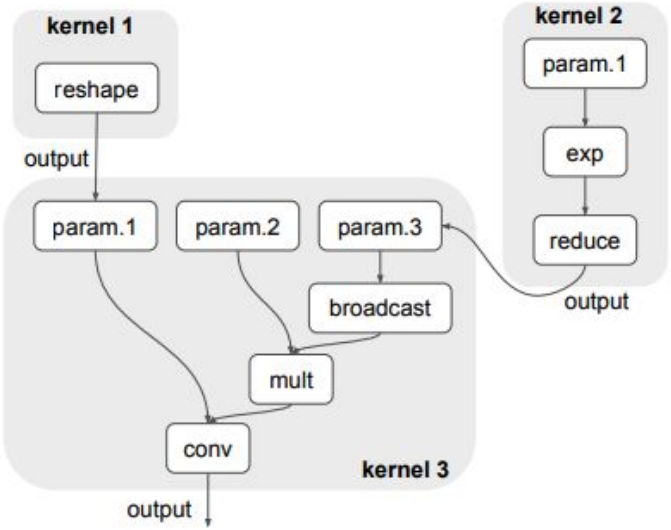
Model Design: Model Inputs



- Node features
  - Opcode
  - Scalar features (e.g. output tensor shape, tensor layout, striding, padding, etc.)
- Whole-kernel features
  - Tile size
  - Static performance info
- Adjacency matrix

Model Design: Adjacency Matrix

- An optimized tensor computation graph consists of multiple kernels.
- Each kernel contains a graph of nodes of primitive operations.



Model Design: Node Embedding

GNN

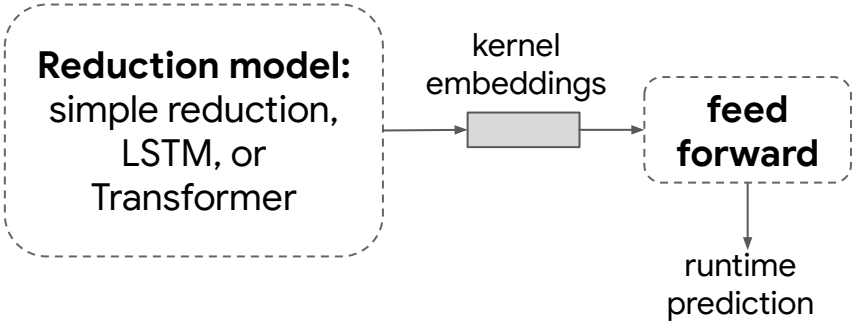
- Tensor compute kernel is represented as a graph.
- Learn node features conditioned on its neighbors.
- Isomorphism for generalization
- Choose GraphSAGE

$$\varepsilon_i^k = l_2\left(f_3^k\left(concat\left(\varepsilon_i^{k-1}, \sum_{j \in neighbors(i)} f_2^k(\varepsilon_j^{k-1})\right)\right)\right)$$

Hamilton et al., GraphSAGE, 2017.

Model Design: Node Embedding

- Reduce mean, reduce sum
- The final state of an LSTM
- Transformer Encoder



Losses

Mean Squared Error
for absolute runtime prediction.
Targets are log-transformed.

$$L = \sum_{i=1}^{n} (y'_i - y_i)^2$$

Pairwise Rank Loss
for relative runtime prediction.

$$L = \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{\phi(y'_i - y'_j) \cdot pos(y_i - y_j)}{n \cdot (n-1)/2}$$

$$\phi(z) = \begin{cases} (1-z)_+ & \text{hinge function } \textbf{or} \\ log(1+e^{-z}) & \text{logistic function} \end{cases}$$

$$pos(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

Accuracy Evaluation and Baseline

- **Accuracy evaluation tasks**
  - Tile size selection (relative runtimes)
  - Fusion (absolute runtimes)
- **Baseline**: XLA's hand-written, analytical performance model
  - XLA argmins all tile sizes using this performance model
  - Fusion does not use this model. It uses other heuristics.

Accuracy: Tile Size Selection

Compare **true** runtimes between best predicted and actual best tile size. **APE:**

$$100 \times \frac{\sum_{k \in K} |t^k_{c'_k} - \min_{c \in C_k} t^k_c|}{\sum_{k \in K} \min_{c \in C_k} t^k_c}$$

In **random** split, learned model **~halves** APE.

	Learned	Analytical
ConvDRAW	9.7	3.9
WaveRNN	1.5	2.8
NMT Model	3.1	13.1
SSD	3.9	7.3
RNN	8.0	10.2
ResNet v1	2.8	4.6
ResNet v2	2.7	5.4
Translate	3.4	7.1
Median	3.3	6.2
Mean	3.7	6.1

Accuracy: Fusion

Compare **Mean Absolute Percentage Error** of kernel runtime predictions.

Random split: learned model improves MAPE by **~85%**.

	Learned	Analytical
ConvDRAW	17.5	21.6
WaveRNN	2.9	322.9
NMT Model	9.8	26.3
SSD	11.4	55.9
RNN	1.9	20.5
ResNet v1	3.1	11.5
ResNet v2	2.4	13.3
Translate	2.1	27.2
Median	3.0	24.0
Mean	4.5	31.1

Ablations: takeaways

- Using a **rank loss** for the tile-size task reduced APE by 10 pts. on average.
- **GraphSAGE** outperformed using a **sequence model** or **Graph Attention Networks** and was less sensitive to hyperparameter selection.
- Replacing the **LSTM/Transformer reduction** with a **non-learned reduction** works almost as well (and improves inference time).

Training for All Optimization Tasks

- **Generate training data from 150 ML models** using random layout, fusion, tile size, and flag configurations.

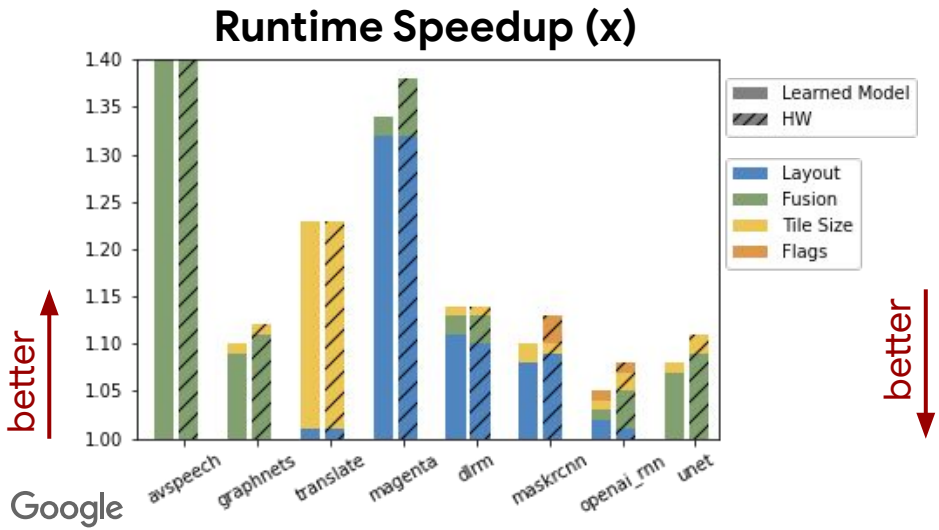
- Train:
  - one model for all **graph-level** optimizations to predict **absolute runtime**
  - one model for **tile-size** to predict **relative runtime**
  - one model for **flags** to predict **relative runtime**

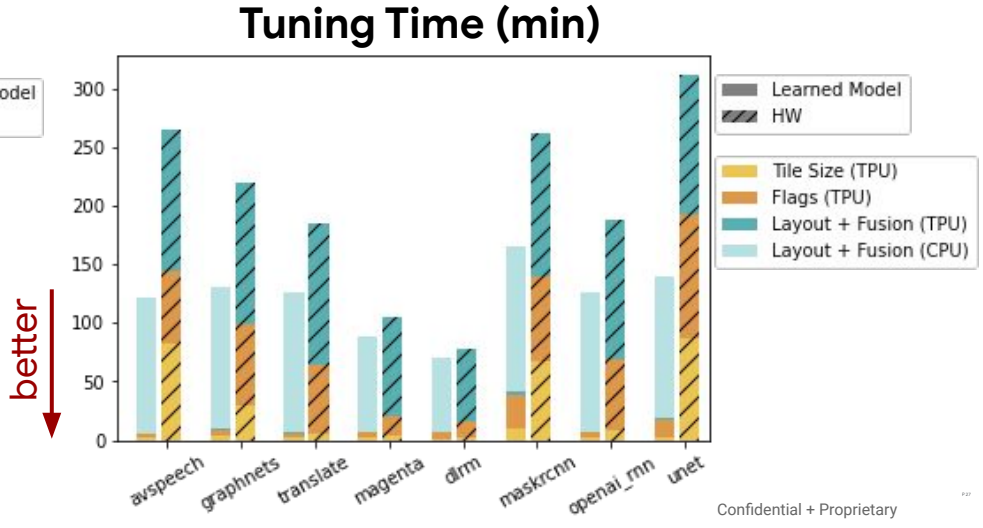
- The graph embedding network is shared between tile-size and flags models.

Tuning with Learned Cost Model

Execute the top k configurations from each worker according to the model on real hardware and pick the best.

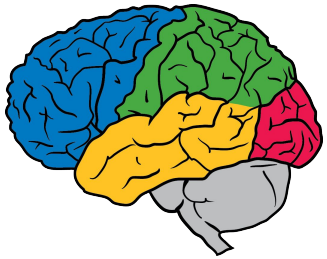
- k = 10 for graph-level optimizations
- k = 5 for kernel-level optimizations





Today's agenda

- Supervised Learning
  - A Learnt Performance Model for TPUs, MLSys 2021
- **Reinforcement Learning**
  - GO: Transferable Graph Optimizers for ML Compilers, NeurIPS 2020
- Production
  - Partitioning ML Models on Multi-Chip Modules, MLSys 2022



Google

Transferable Graph Optimizers for ML Compilers

Yanqi Zhou, Sudip Roy, Amirali AbdolRashidi*, Daniel Wong*, Peter Ma, Qiumin Xu, Hanxiao Liu, Mangpo Phothilimtha, Shen Wang, Anna Goldie, Azalia Mirhoseini, and James Laudon

*Internship work at Google

What are we trying to address?

- Heuristics based compiler optimizations are suboptimal, treating each optimization task in isolation.

Figure credit: https://tvm.apache.org/2019/03/18/tvm-apache-announcement

What are we trying to address?

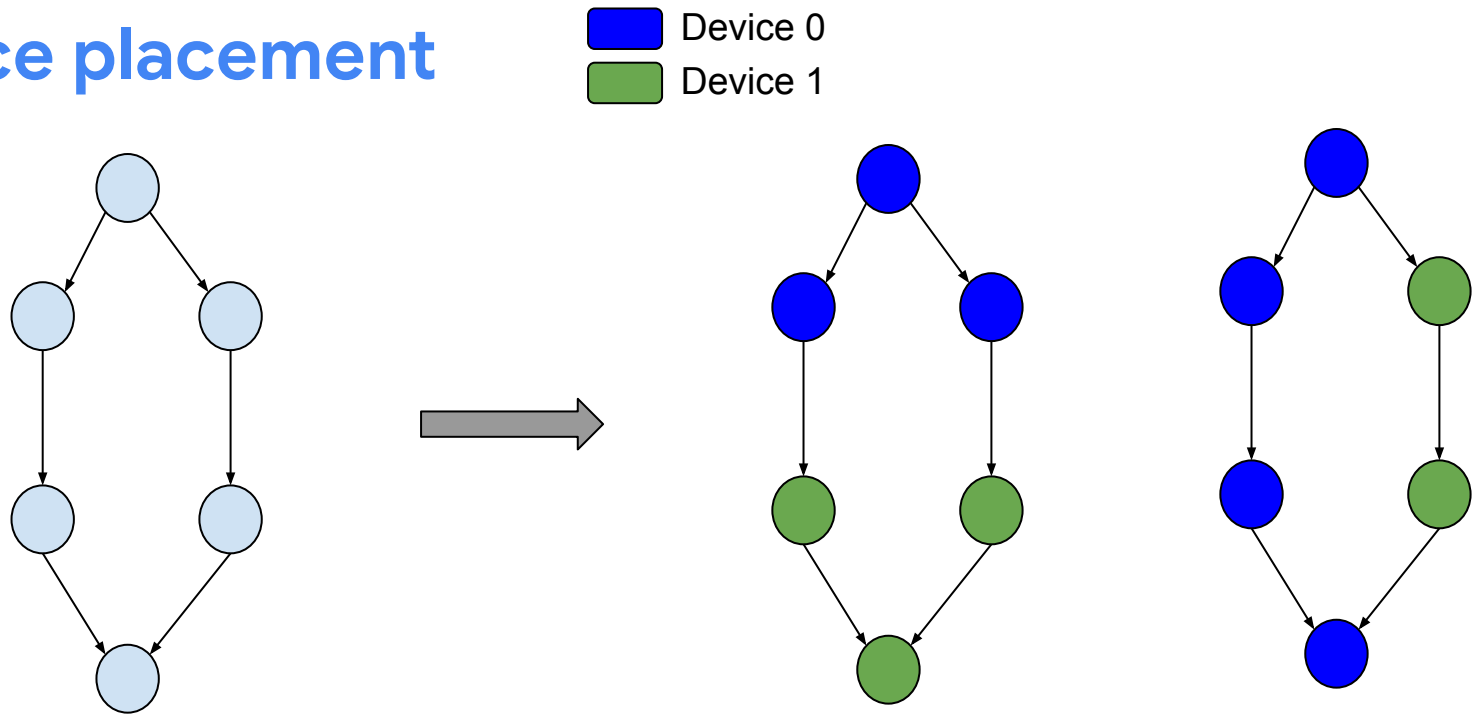
- Existing learning based methods are sample inefficient, tackle a single optimization problem, or do not generalize to unseen graphs.
  - Hierarchical device placement
  - Placeto
  - NeuRewriter
  - REGAL
- Generalization is important for deployment

Azalia Mirhoseini and others. A hierarchical model for device placement. ICLR, 2018.
Ravichandra Addanki and others. Placeto: Learning generalizable device placement algorithms for distributed machine learning. NeurIPS, 2019.
Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. Neurips, 2019.
Aditya Paliwal and others. REGAL: transfer learning for fast optimization of computation graphs. ICLR, 2020.

GO: Transferable graph optimizers

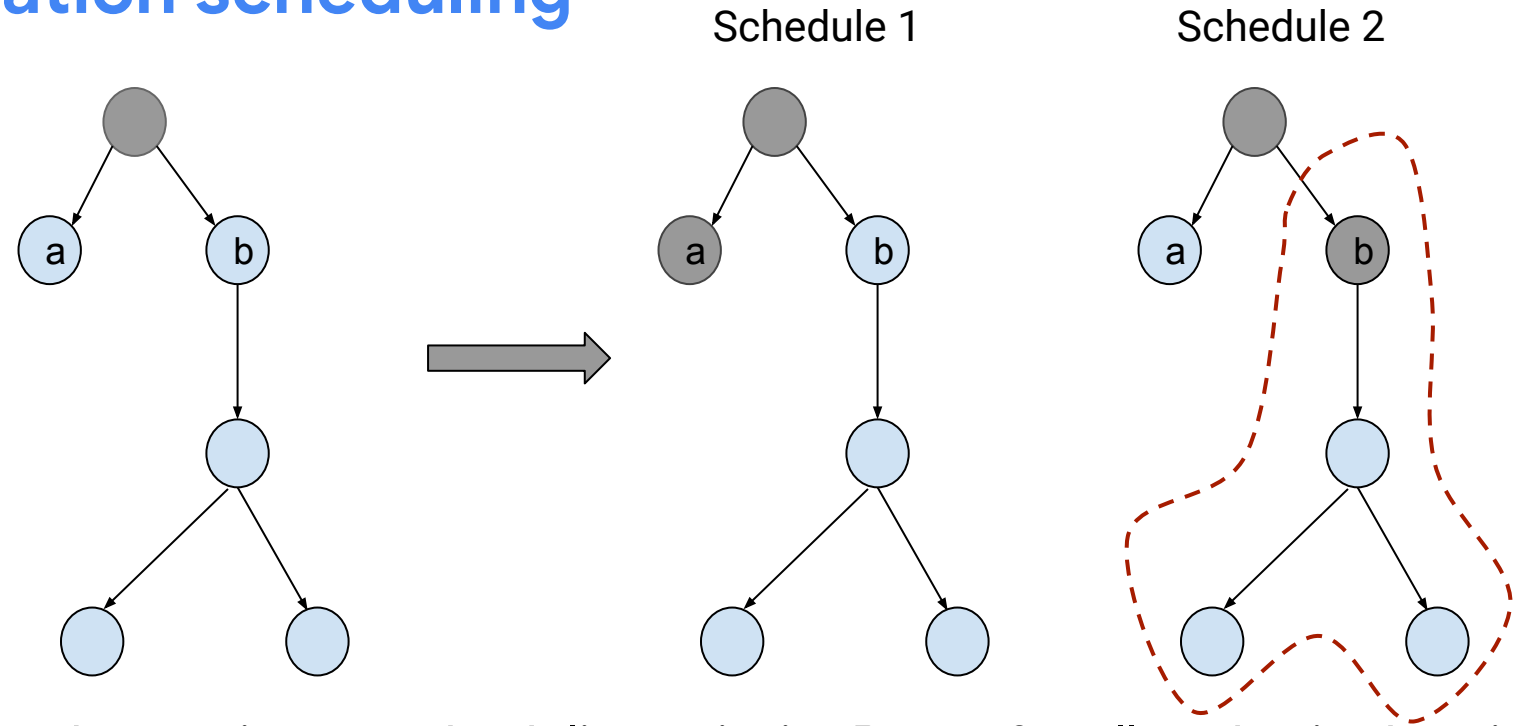
- Apply machine learning to learn a generalized approach for compiler graph optimizations
- Tasks
  - Automatic device placement
  - Operation fusion
  - Operation scheduling
  - ...
- Goals
  - Minimize graph runtime by optimizing node attributes
  - Handle large graphs over 10k nodes
  - Generalization on unseen data
  - Transferable across tasks

Device placement



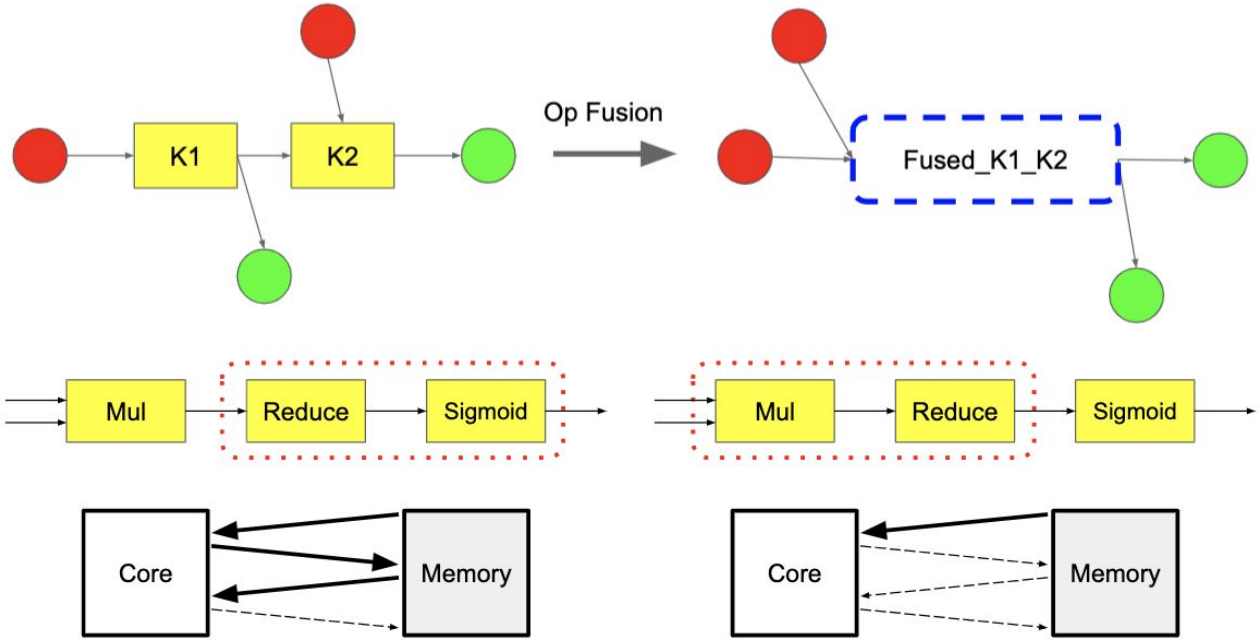
π: $\mathcal{G} \rightarrow \mathbb{D}$ that assigns a device D $\in \mathbb{D}$ for all nodes in the given graph G $\in \mathcal{G}$ to maximize a reward r(G, D).

Operation scheduling



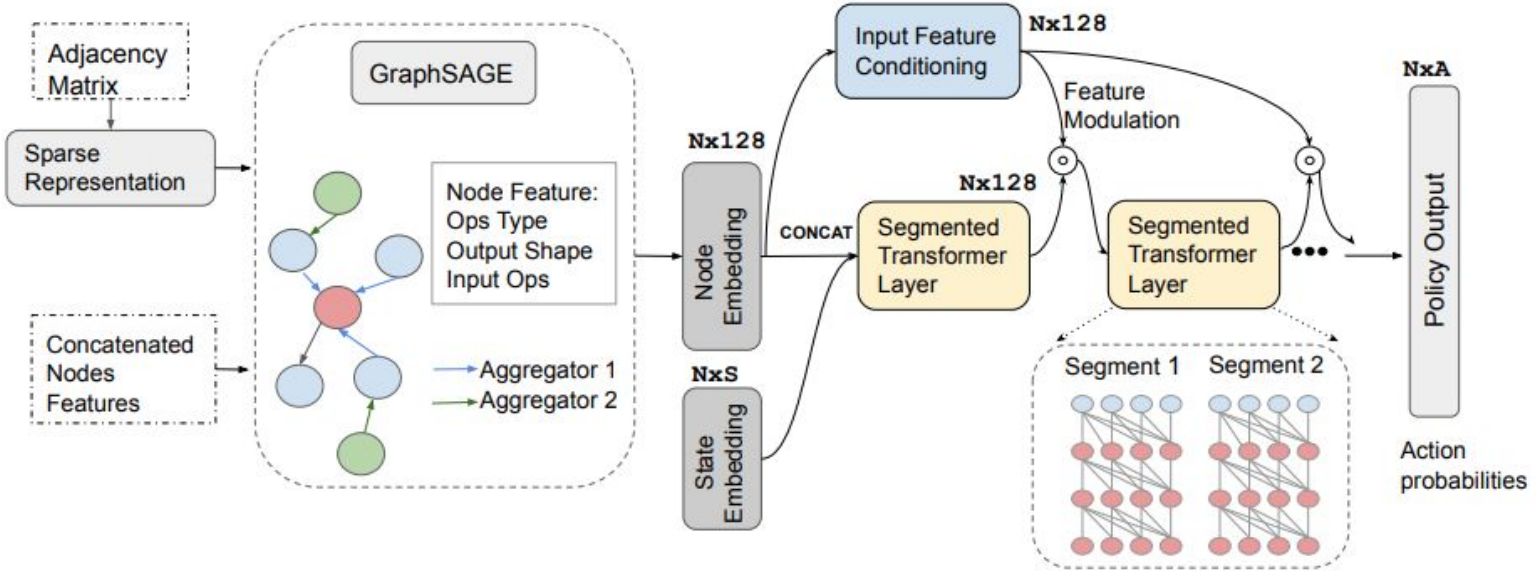
π: $\mathcal{G} \rightarrow \mathbb{P}$ that assigns a scheduling priority $P \in \mathbb{P}$ for all nodes in the given graph $G \in \mathcal{G}$ to maximize a reward r(G, P).

Operation fusion



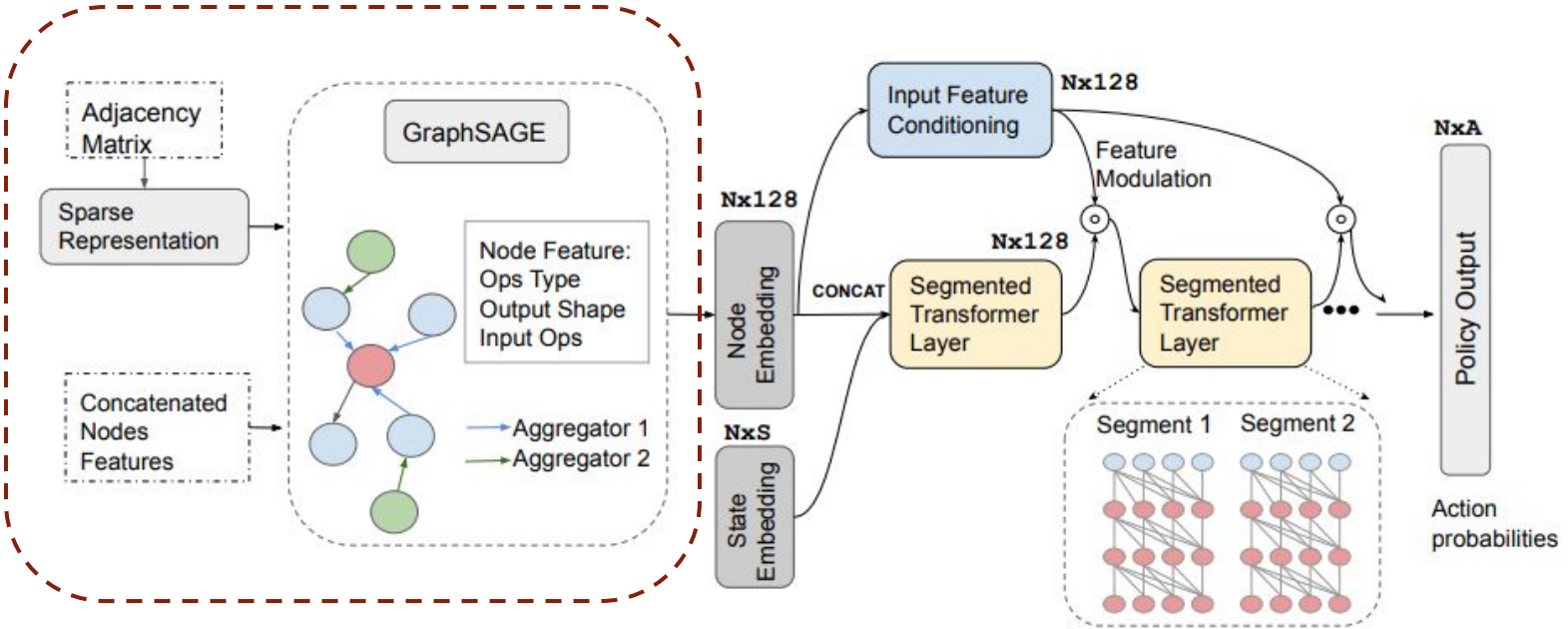
π: $\mathcal{G} \rightarrow \mathcal{F}$ that assigns a fusion priority $F \in \mathcal{F}$ for all nodes in the given graph $G \in \mathcal{G}$ to maximize a reward r(G, F).

GO: Transferable graph optimizers



GO: Transferable graph optimizers

- Generalization across graphs->GraphSAGE embedding layers

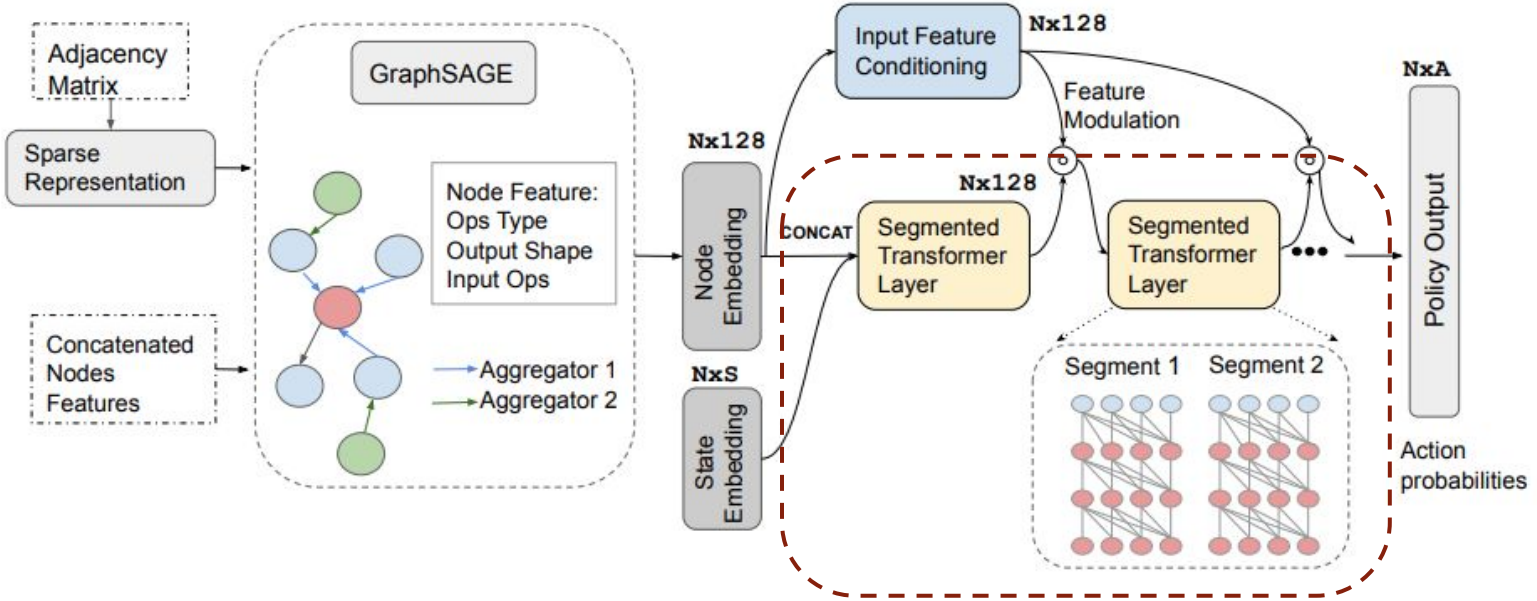


GO: Transferable graph optimizers

- Generalization across graphs->GraphSAGE embedding layers
  - Invariant to node order.
  - Generalize to unseen nodes, graphs.

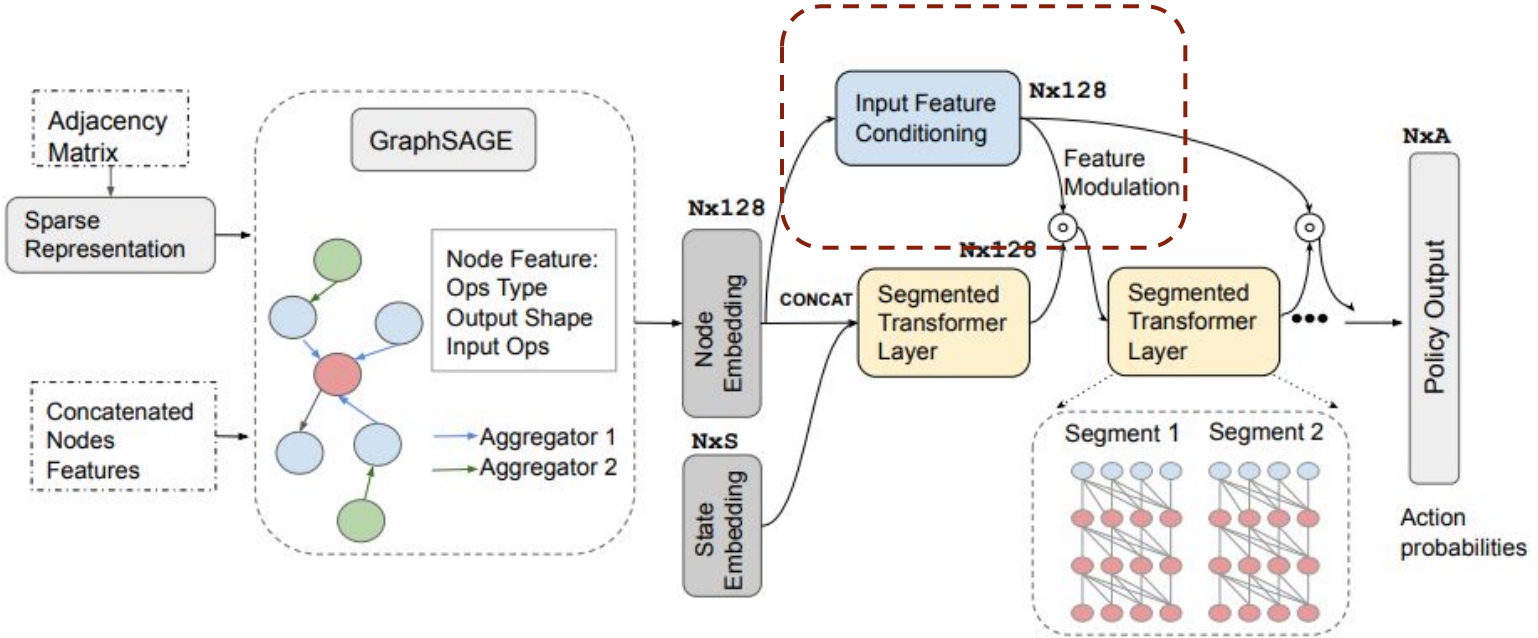
GO: Transferable graph optimizers

- Long-range dependences->Segmented recurrent attention



GO: Transferable graph optimizers

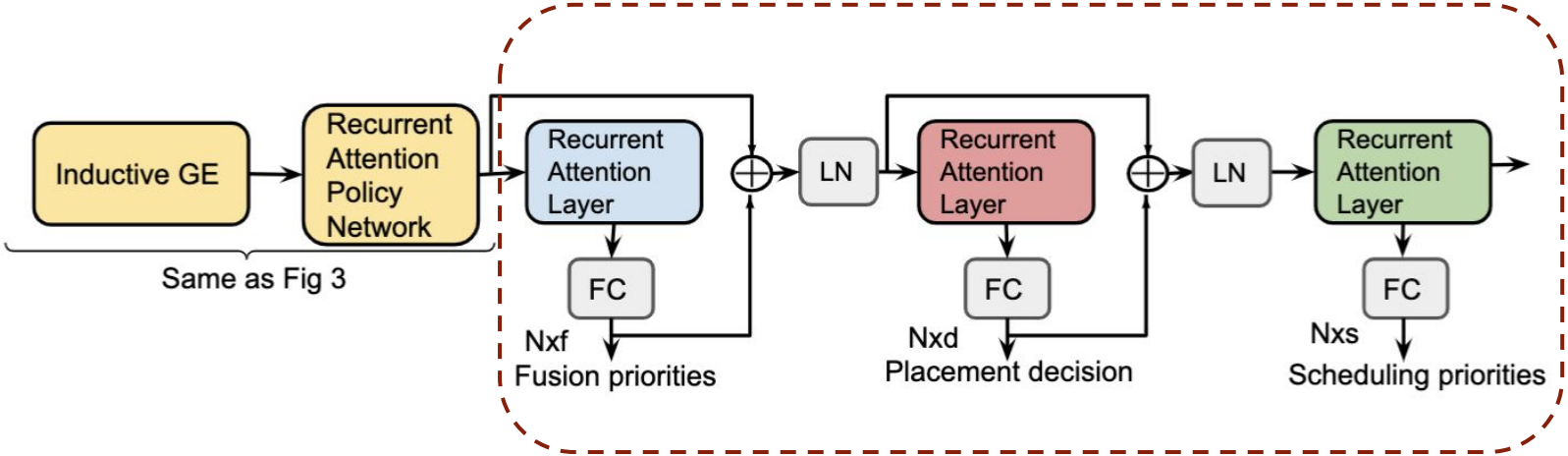
- Specialization for graph types->Input feature conditioning



GO: Transferable graph optimizers

- Multiple dependent tasks->Recurrent attention task heads

$$H^t = \mathrm{LN}(\mathrm{Concat}(A^{t-1}, H^{t-1}))$$
$$A^t = \mathrm{FC}(\mathrm{MultiHeadAttn}(H^t))$$



GO: Transferable graph optimizers

- Generalization across graphs
  - T∈{D, P, F}: task
  - θ: parameters of the RL policy
  - $\mathcal{G}$: empirical distribution of the dataflow graphs.

$$J(\theta) = \mathbb{E}_{G\sim\mathcal{G},T\sim\pi_\theta(G)}[r_{G,T}]$$

- Joint optimization across tasks
  - Parameters across the three tasks can be partially shared.
  - The shared policies can be parameterized in the form of multiple recurrent attention layers.

$$J(\theta) = \mathbb{E}_{G\sim\mathcal{G},D\sim\pi_{\theta_D}(G),P\sim\pi_{\theta_P}(G),F\sim\pi_{\theta_F}(G)}[r_{G,D,P,F}]$$

GO: Iterative but non-autoregressive node decisions

- Autoregressive
  - Ideally, compute the action distribution of the node under consideration based on the actions of all previous nodes:

$$p(\mathbf{y}|G) = \prod_{i=1..N} p\left(y_i|h_G, y_{i-1}, y_{i-2}, ...\right)$$

  - Infeasible as N can be as large as 10K.
- Iterative but non-autoregressive approximation:
  - N sampling procedures are carried out in parallel within each iteration t.
  - Decisions of N nodes are allowed to mutually influence each other.

$$p(\mathbf{y}^{(t)}|G) = \prod_{i=1..N} p\left(y_i^{(t)}|h_G, \mathbf{y}^{(t-1)}\right)$$

GO: Experiment Setup

- Workloads
  - RNNLM, GNMT, Transformer-XL, Inception-v3, AmoebaNet, WaveNet.
  - Varying architectural parameters (e.g. layers, hidden dim)
- Run time measurement
  - Real-hardware measurement for device placement
  - Industry-standard performance model for the rest tasks
- Baselines
  - Default heuristics in TF (GPU)
  - Human expert solution
  - Simulated annealing
  - Learning-based strategy like HDP

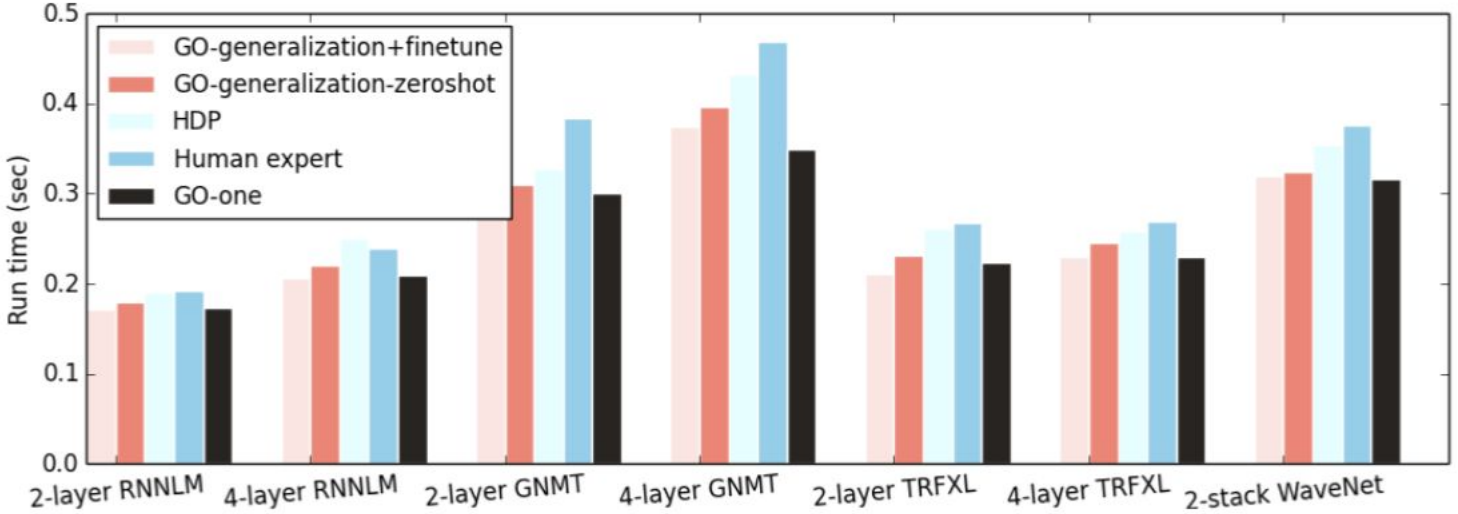
GO: Speedup on device placement

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

Table 2: Run time comparison between GO-one, human expert, TensorFlow METIS, and hierarchical device placement (HDP) on six graphs (RNNLM, GNMT, Transformer-XL, Inception, AmoebaNet, and WaveNet). Search speed up is the policy network training time speed up compared to HDP (reported values are averages of six runs).

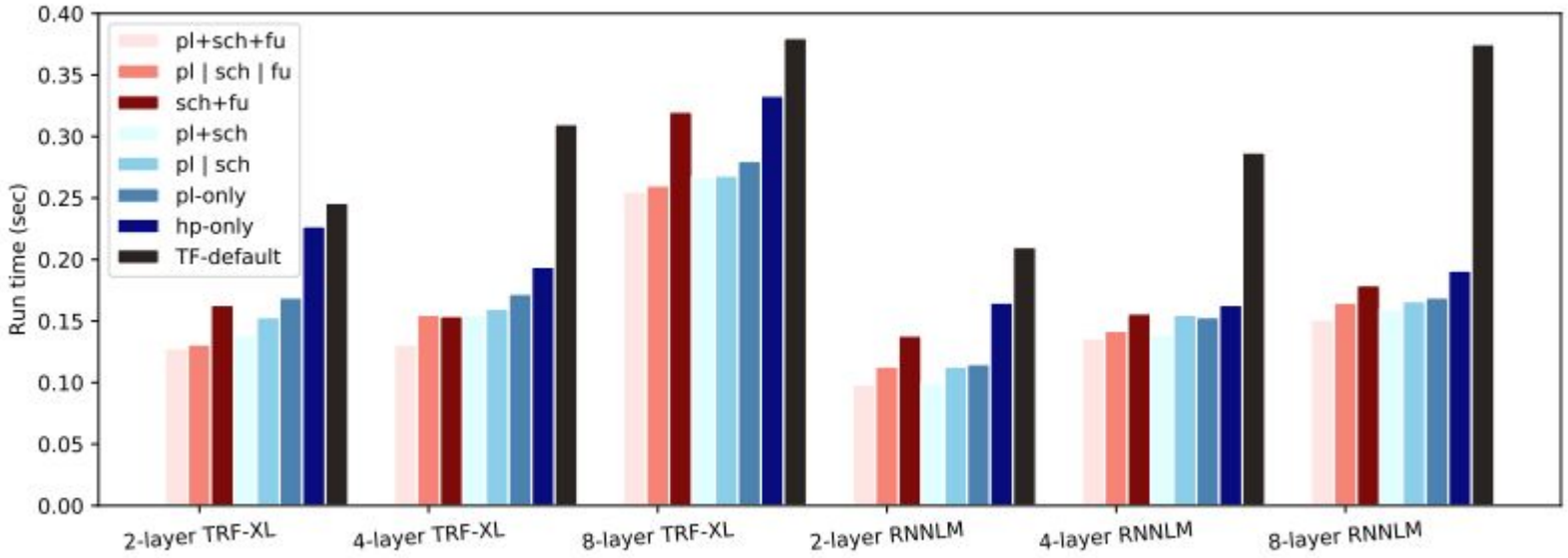
GO: Generalization on device placement

- Real-hardware measurement
- GO-finetune matches GO-one
- GO-zeroshot better than human placement and HDP



GO: Multi-task speedup

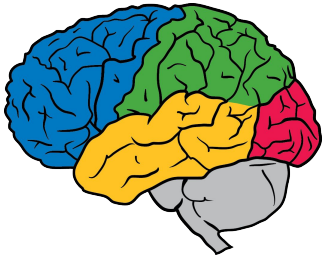
- 33%-60% speedup jointly optimizing three tasks, over TF default



Today's agenda

- Supervised Learning
  - A Learnt Performance Model for TPUs, MLSys 2021
- Reinforcement Learning
  - GO: Transferable Graph Optimizers for ML Compilers, NeurIPS 2020
- **Production**
  - Partitioning ML Models on Multi-Chip Modules, MLSys 2022

Google



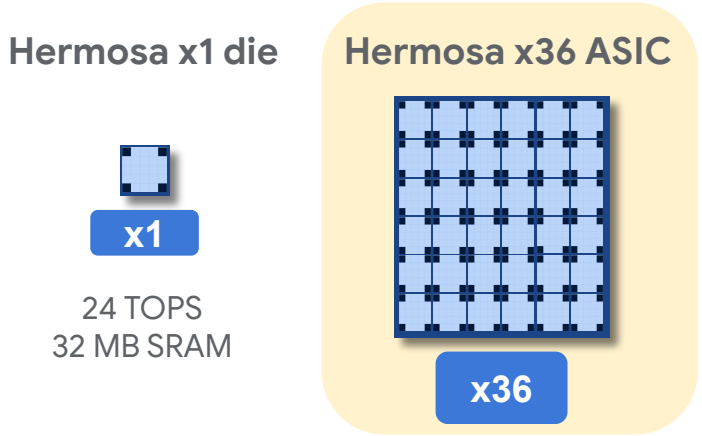
A Transferable Constrained RL for Partitioning Modules on Multi-Chip-Modules

xinfeng@ucsb.edu, sudipr@, mangpo@, prakashp@,

ulysse@, azalia@, ebrevdo@, jlaudon@, yanqiz@

Gyrfalcon: A Multi-Chip-Module Package for ML Acceleration

- Multi-Chip-Module
  - Combines smaller chiplets
  - Reduce cost and improve yields
- Our target hardware: Gyrfalcon
  - 6x6 Hermosa Chips
  - 432 TFLOPS / 864 TOPS
  - 1 GB SRAM; 290 W; 16 GB/s die-to-die 1D Ring
- Graph Partitioning is critical
  - Balanced
  - Lower inter-chip communication

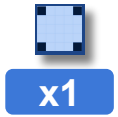


[source]

Why a constrained solver?

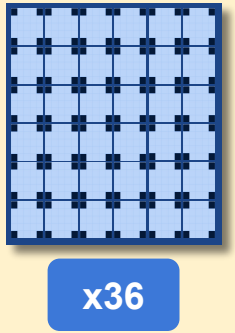
- Placement Constraints
  - Acyclic dataflow
  - No skipping chips
  - Chip triangle dependency
  - Uncaptured dynamics

Hermosa x1 die



24 TOPS
32 MB SRAM

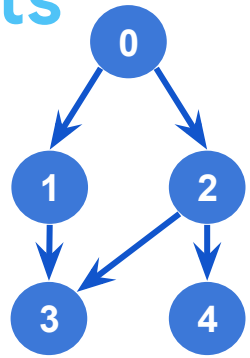
Hermosa x36 ASIC



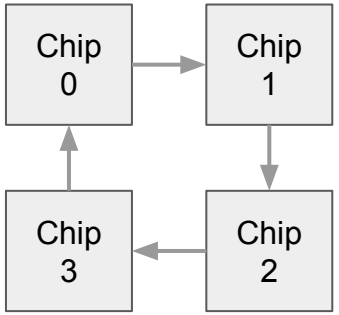
[source]

Device Placement Constraints

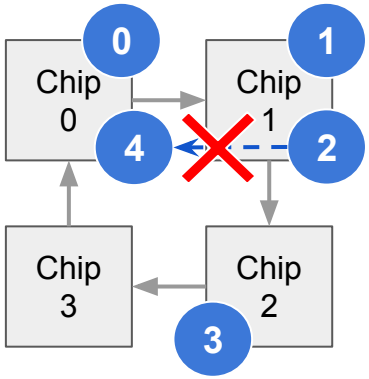
- Acyclic dataflow constraint
  - Data only flows from lower chip ID to higher chip ID.
  - Reason: (hardware) 1D ring for inter-chip communication.



ML graph



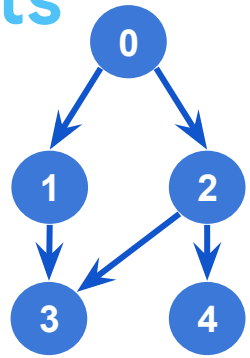
An example of 4 chips



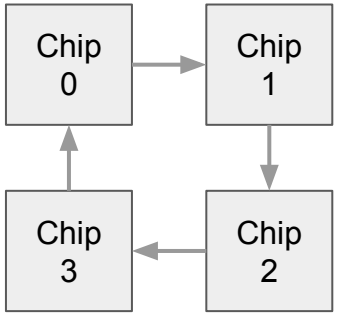
An invalid placement

Device Placement Constraints

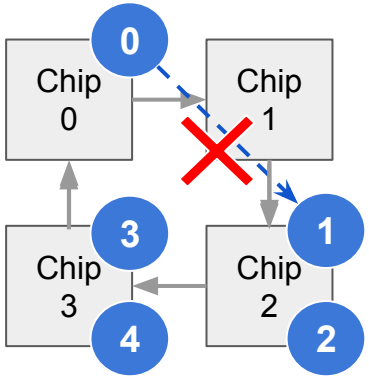
- No skipping chip constraint
  - Every chip should have at least one TF node assigned to it.
  - Reason: (software) virtual device groups are always contiguous.



ML graph



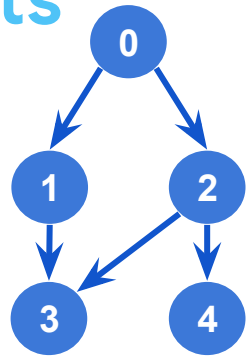
An example of 4 chips



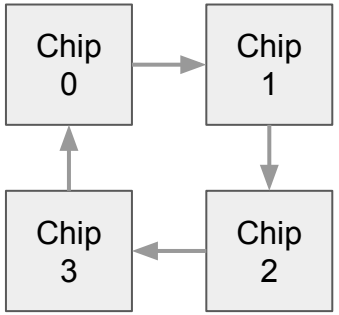
An invalid placement

Device Placement Constraints

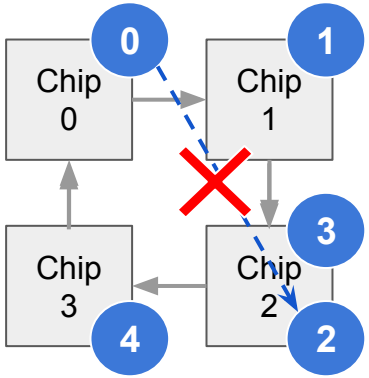
- Chip triangle dependency constraint
  - A -> B -> C and A -> C are not allowed
  - Example: {Chip 0 -> Chip 1 -> Chip 2} and {Chip 0 -> Chip 2}
  - Reason: (hardware) prevent inter-chip communication deadlocks.



ML graph



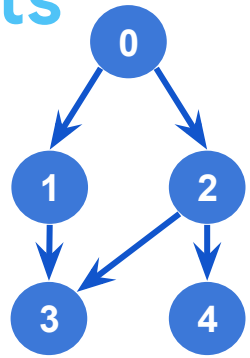
An example of 4 chips



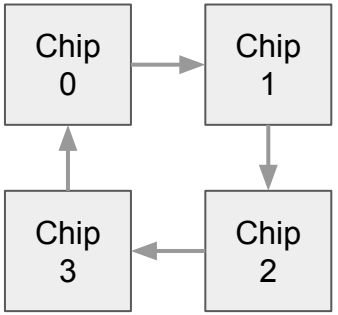
An invalid placement

Device Placement Constraints

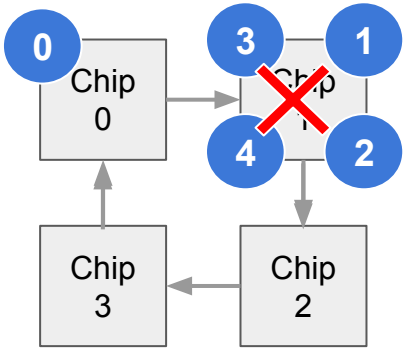
- Static constraints:
  - Acyclic dataflow constraint
  - No skipping chip constraint
  - Chip dependency triangle constraint
- Dynamic constraints:
  - Memory allocation constraint (OOM)



ML graph



An example of 4 chips



An invalid placement

Device Placement Constraints

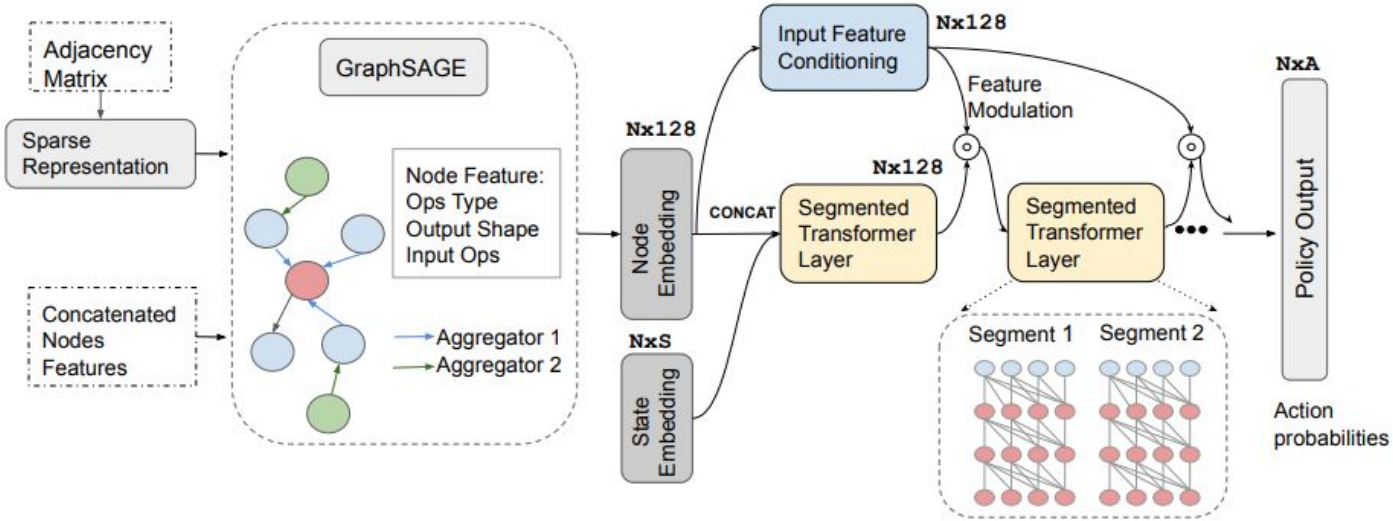
- Static constraints:
  - Easy to mathematically formulate.
  - Constraint solvers can identify invalid placement statically
- Dynamic constraints:
  - Hard to formulate before compilation.
  - RL is able to learn through environment dynamics.

Our contribution: a transferable deep RL working with a constraint solver.

Google

GO Framework: An RL Approach

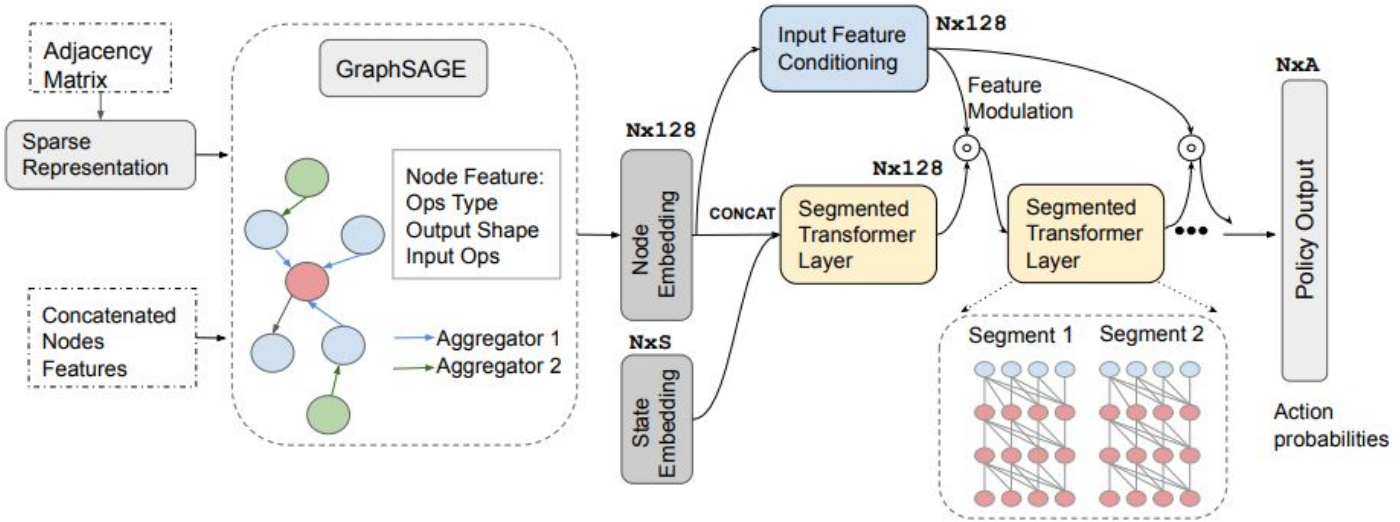
Pros: GO can generate placement actions in a non-autoregressive way.



GO Framework: An RL Approach for Device Placement

Pros: GO can generate placement actions in a non-autoregressive way.

Cons: GO is not aware of placement constraints, and it is infeasible to rely on RL to learn constraints because the reward space is too sparse.



Telamon: A Constraint Solver

“Telamon is a wrapper around a constraint solver that encodes combinatorial decision problems and exposes them to external search heuristics.”

Pros: Telamon can work with external search strategies to find valid placement under static constraints.

Google

Telamon: A Constraint Solver

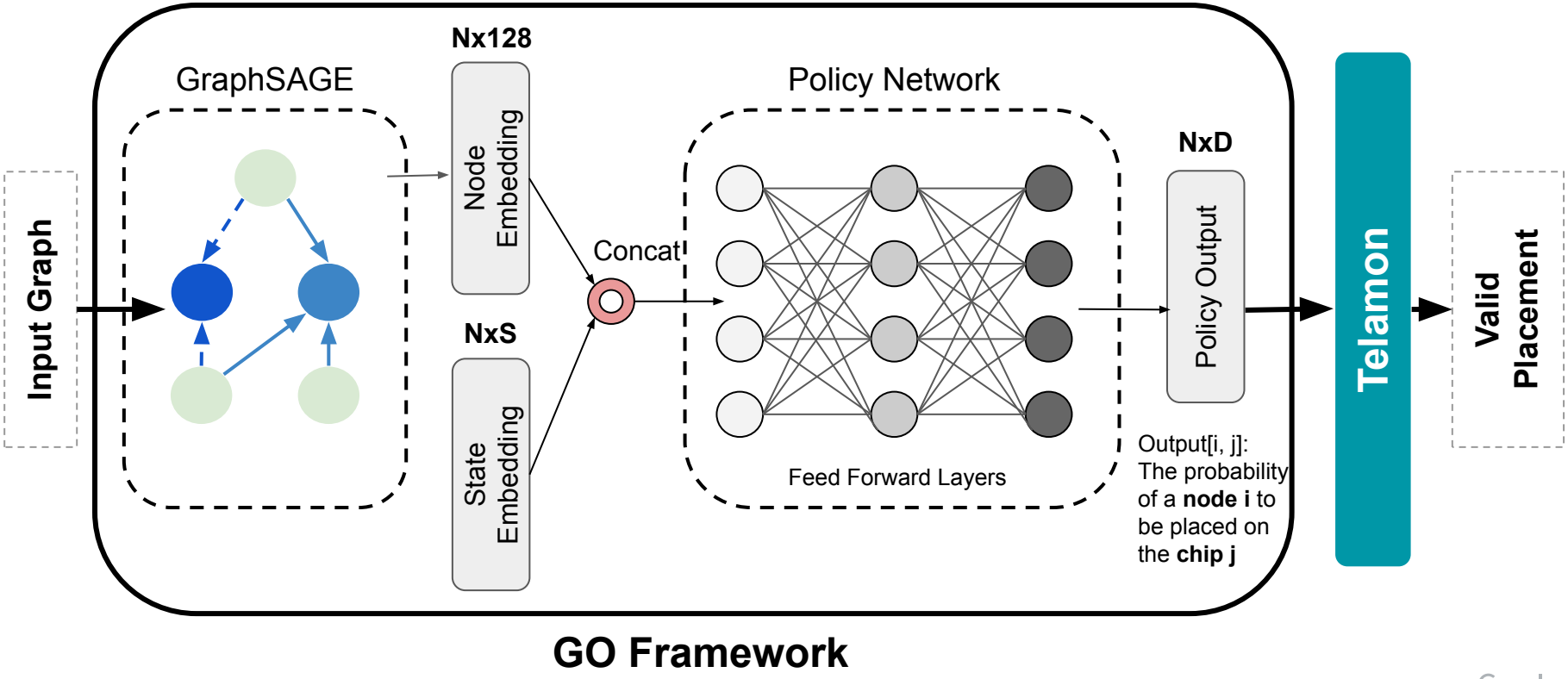
"Telamon is a wrapper around a constraint solver that encodes combinatorial decision problems and exposes them to external search heuristics."

Cons:

(1) Need a closed-form formulation of the objective function for finding the optimal.
(2) Does not learn any bias or policies from input data.
(3) Not every constraint can be statically formulated (e.g., memory allocation).

Google

GO + Telamon: A Constrained RL Approach



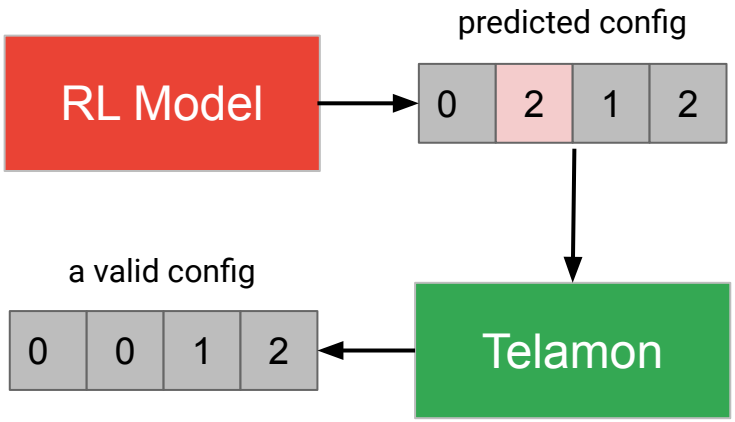
Inside a Telamon

- Maintains the range of valid values for every variable yi (the domain of yi).
- Query the current domain of variables and set variable domains.
- Constraint propagation that recursively prunes the domain of other variables when setting a variable.
- If detects an invalid assignment during constraint propagation, backtrack to to a previous state, undoing previous decisions.

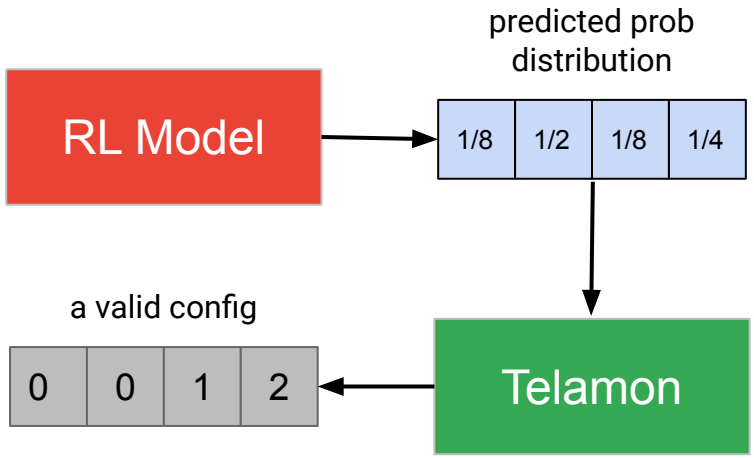
Google

GO + Telamon: Two Modes

- Telamon + RL:
  - FIX mode / SAMPLE mode

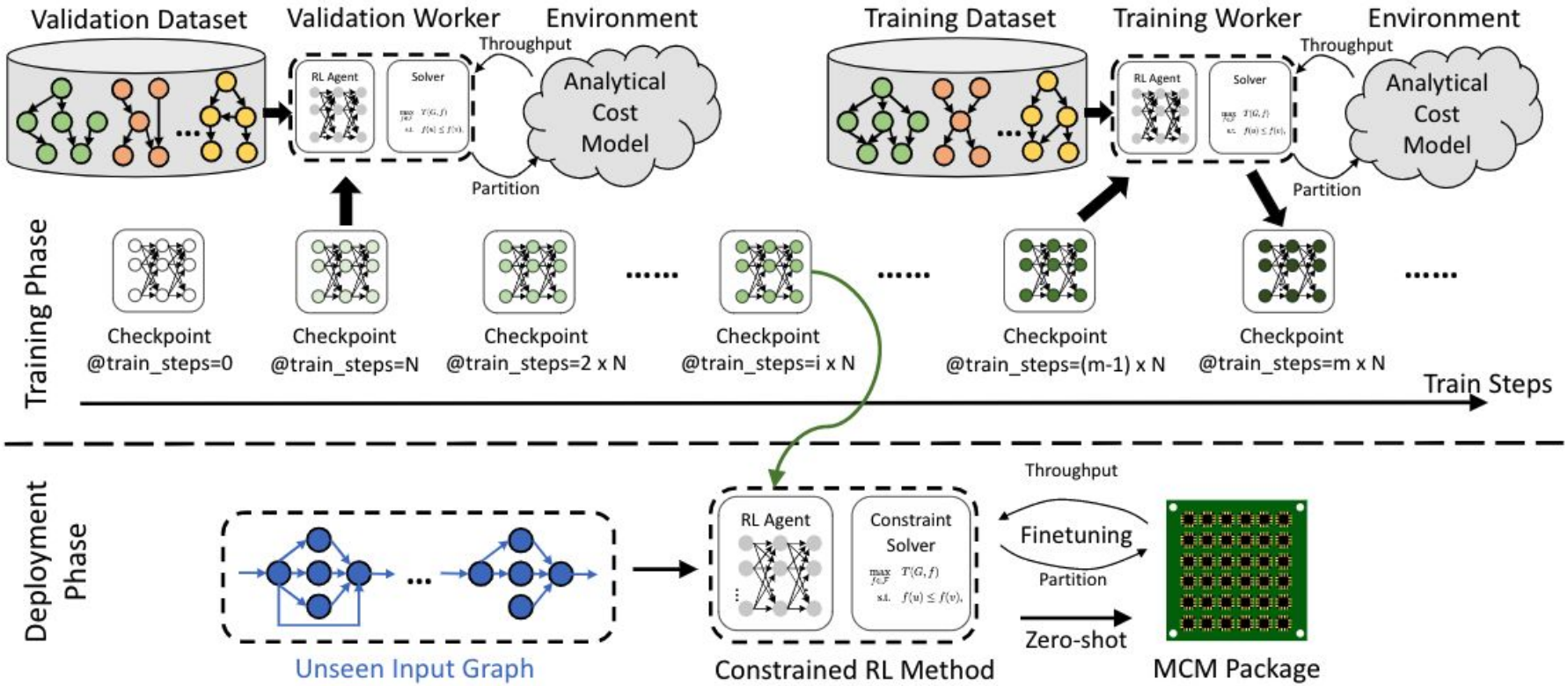


FIX mode



SAMPLE mode

Pretraining and Generalization

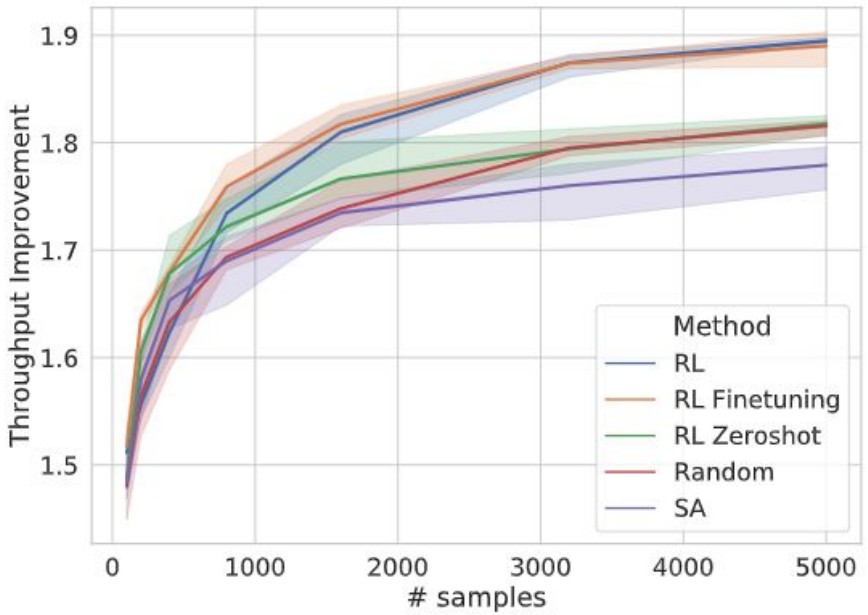


Pretraining Setups

- Model Zoo: 87 ML graphs in total
  - Training dataset: 66 ML graphs
  - Validation dataset: 5 ML graphs
  - Test dataset: 16 ML graphs
- Environment:
  - An analytical model to estimate the latency of each chip
  - Tmax = max(T0, T1, ..., TD-1) where D is the number of devices
  - Throughput = 1 / Tmax
- Reasons for using analytical model:
  - Fast: real hardware evaluation is expensive.
  - Strong correlation between analytical model and real hardware evaluation.

Throughput Improvement

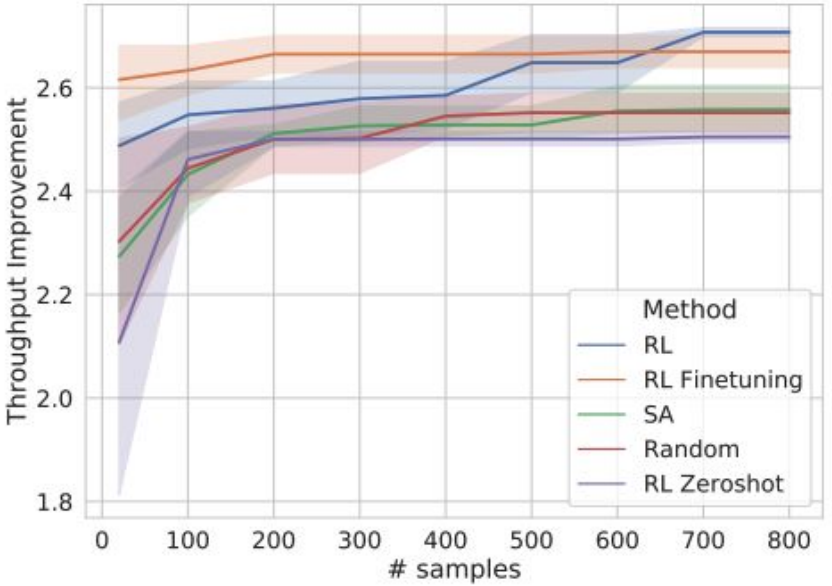
- Average on 16 test graphs, over compiler heuristics.
  - RL from scratch
  - RL finetuning
  - RL zeroshot
  - Random search
  - Simulated annealing



Throughput Improvement	$\geq 1.60\times$	$\geq 1.70\times$	$\geq 1.80\times$
Random	305 (1.08×)	915 (0.74×)	3612 (0.41×)
SA	255 (1.29×)	979 (0.69×)	N.A. (N.A.)
RL	330 (1.00×)	676 (1.00×)	1496 (1.00×)
RL Zeroshot	196 (1.68×)	600 (1.13×)	3652 (0.41×)
RL Finetuning	**171 (1.93×)**	**503 (1.34×)**	**1362 (1.10×)**

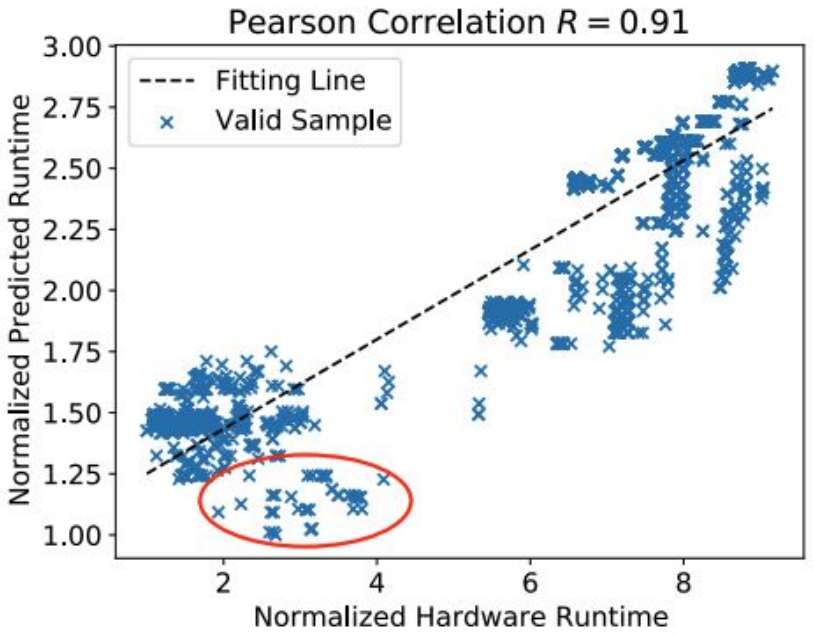
Real Hardware Evaluation on BERT

- Better throughput at convergence
  - RL (GO) outperforms Random and SA by 6.11% and 5.86% respectively.
- RL fine-tuning yields best results
- Search time reduction
  - 20 samples (9 minutes) vs. 800 samples (6 hours) to get on-par with training from scratch



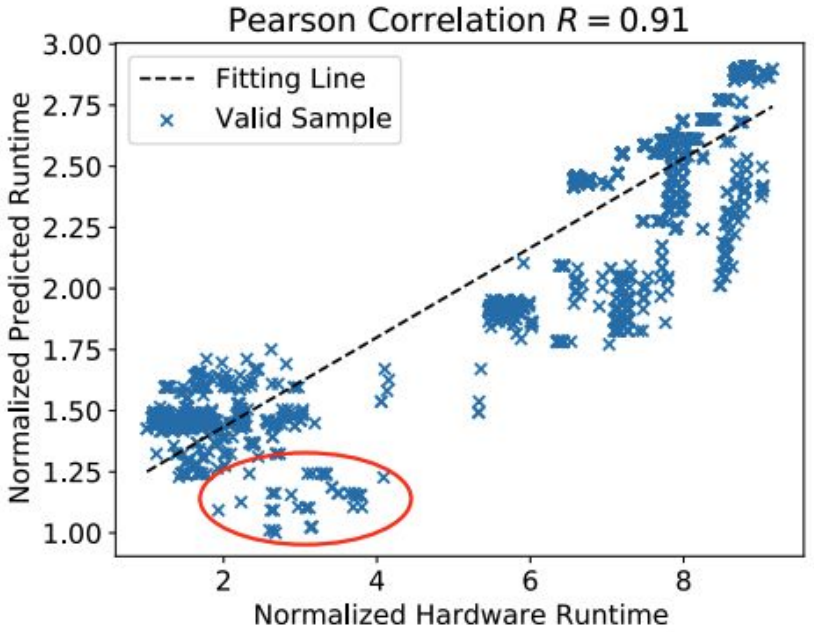
Analytical Model Regression

- Generate 2000 placement configs randomly.
- Evaluate the placement configs on both analytical model and real hardware.
- Correlation score: 0.91



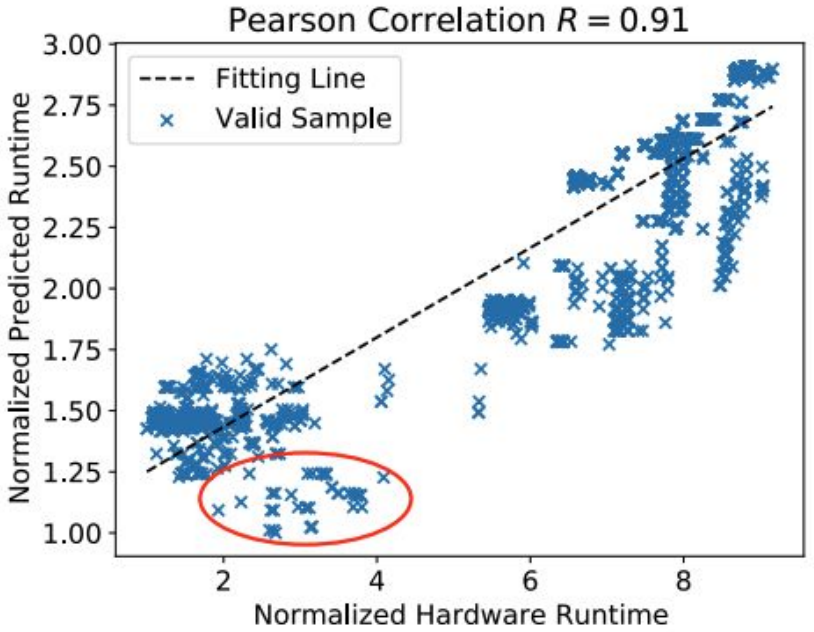
Generalization Results

- GO Fine-tuning works well:
  - Strong correlation (0.91)
- Pre-training on analytical model
  - Learns a bias on model balancedness
- Fine-tuning on real hardware
  - Learn additional dynamic constraints (dynamic memory)



Generalization Results

- GO Fine-tuning works well:
  - Strong correlation (0.91)
  - Fine-tuning allows the model to learn additional dynamic constraints.
- GO Zeroshot does not work well:
  - Some false positives from the analytical model
  - some runtime constraints can't be captured the analytical model (13.5% failures).

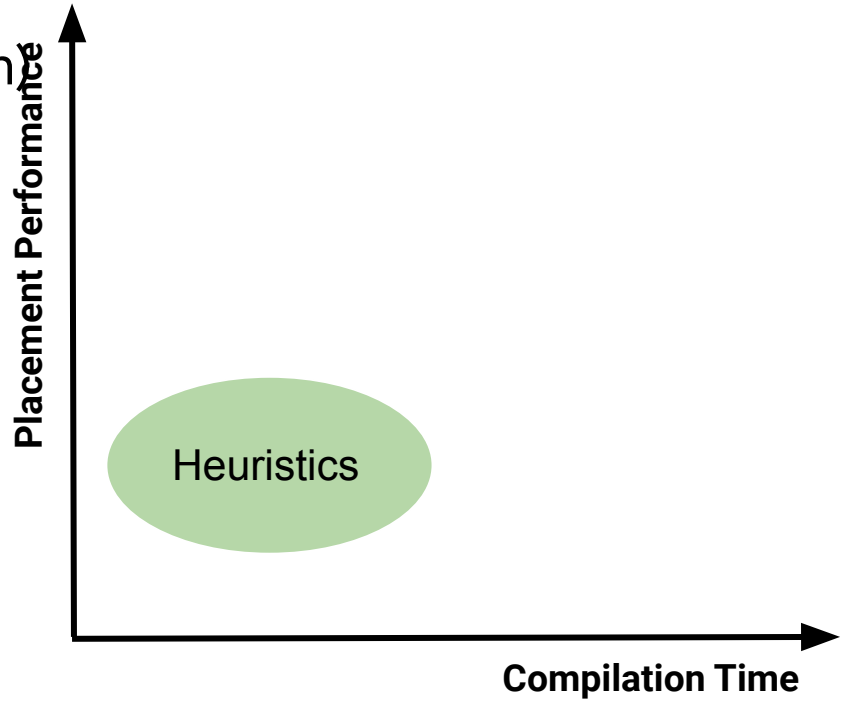


Takeaways

- Transferable graph optimizer using deep RL
  - Non-autoregressive method that is fast and scalable to more than 10k nodes.
  - Generalization across unseen graphs and transfer learning for multiple tasks.
- Productionized in a multi-chip placement problem
  - Better throughput than production compilers.
  - Reduce search time from 6 hours to 9 minutes.

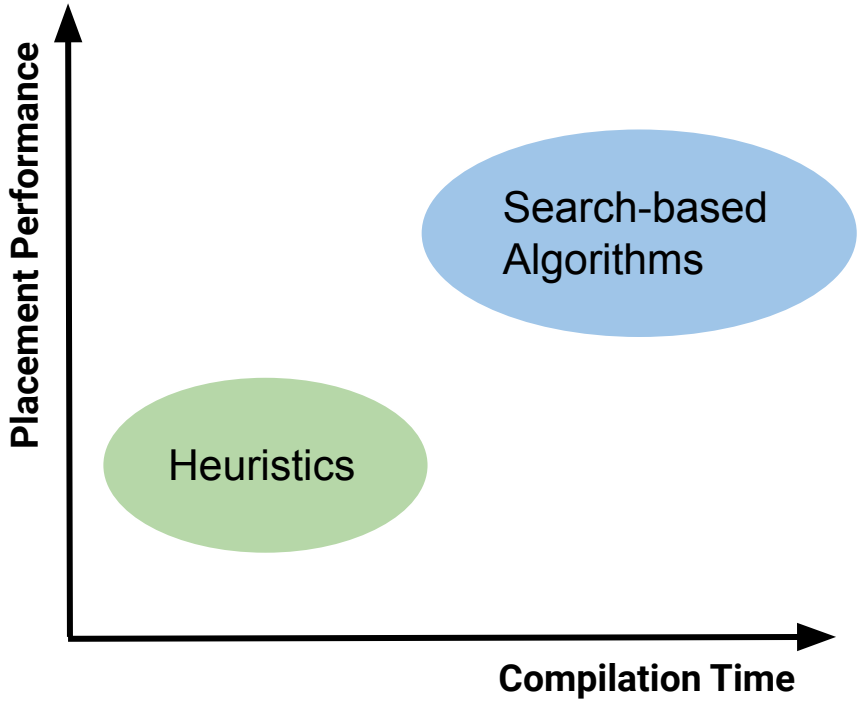
Existing Solutions in Compilers

- Heuristics (e.g. Greedy Algorithm)
  - **Pros:** Does not need to evaluate generated placement configs.
  - **Cons:** Achieve lower performance (throughput)



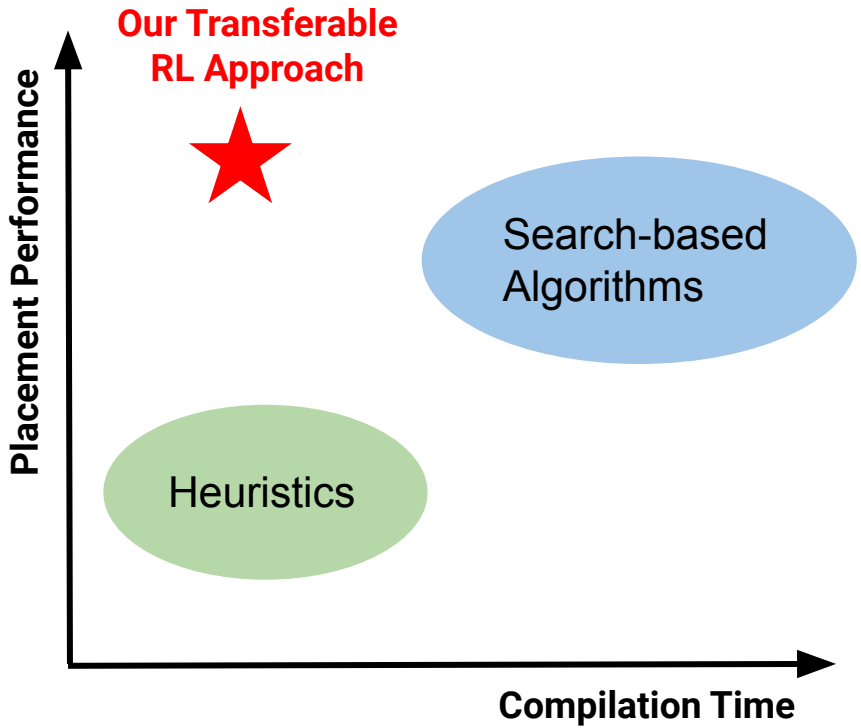
Existing Solutions in Compilers

- Heuristics (e.g. Greedy Algorithm)
  - **Pros:** Does not need to evaluate generated placement configs.
  - **Cons:** Achieve lower performance (throughput)
- Search-based Algorithms (e.g. Simulated Annealing)
  - **Pros:** Achieve higher performance (throughput)
  - **Cons:** Needs a number of samples to discover a good placement (~27 secs per sample)



Our Solutions in Compilers

- Better Performance:
  - Higher throughput for discovered placement config.
  - Improved Sample Efficiency:
- Transferable from training data to unseen input graphs
- Ultra good performance with fine-tuning.



Thank you & QA