Google Research

Autotuning Production ML Compilers

Phitchaya Mangpo Phothilimthana
mangpo@google.com

Search-Based ML Compilers

Google Research

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

Search-Based ML Compilers

Google Research

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

Google Research

Search at Subgraph Level is Suboptimal

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.

Google Research

Search-Based ML Compilers

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

Google Research

Search Approaches: Long Compile Time

optimization scope

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

compile time (for ResNet like inference)

Google Research

Production Compilers: Multi-Pass

- Models evaluated by research compilers: up to 1,000 node
- Industrial-scale models: up to **500,000 nodes!**
- That's why **production ML compilers** still decompose the compilation into **multiple passes**.
- **None** of the existing approaches **support** autotuning different optimizations in a **multi-pass compiler**.
    - **Challenge**: search space of a pass is highly dependent on decisions made in prior passes.

Our Goal

Bring the benefits of **search-based** exploration to **multi-pass compilers**:

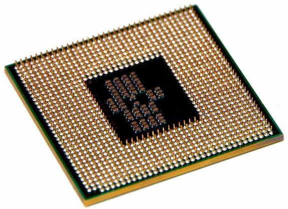
- for both graph and subgraph levels
- with flexibility via configurable search to tune subset of optimizations of interest

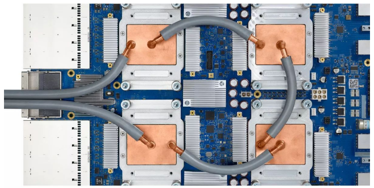
Ref: Phothilimthana et al., A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers, PACT 2021.

Production ML Compilation Stack at Google

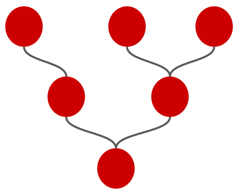
PyTorch

• • •





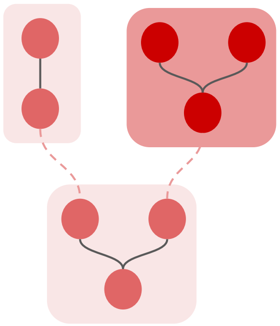
XLA TPU Compiler



tensor computation graph

Graph-Level Optimizations

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

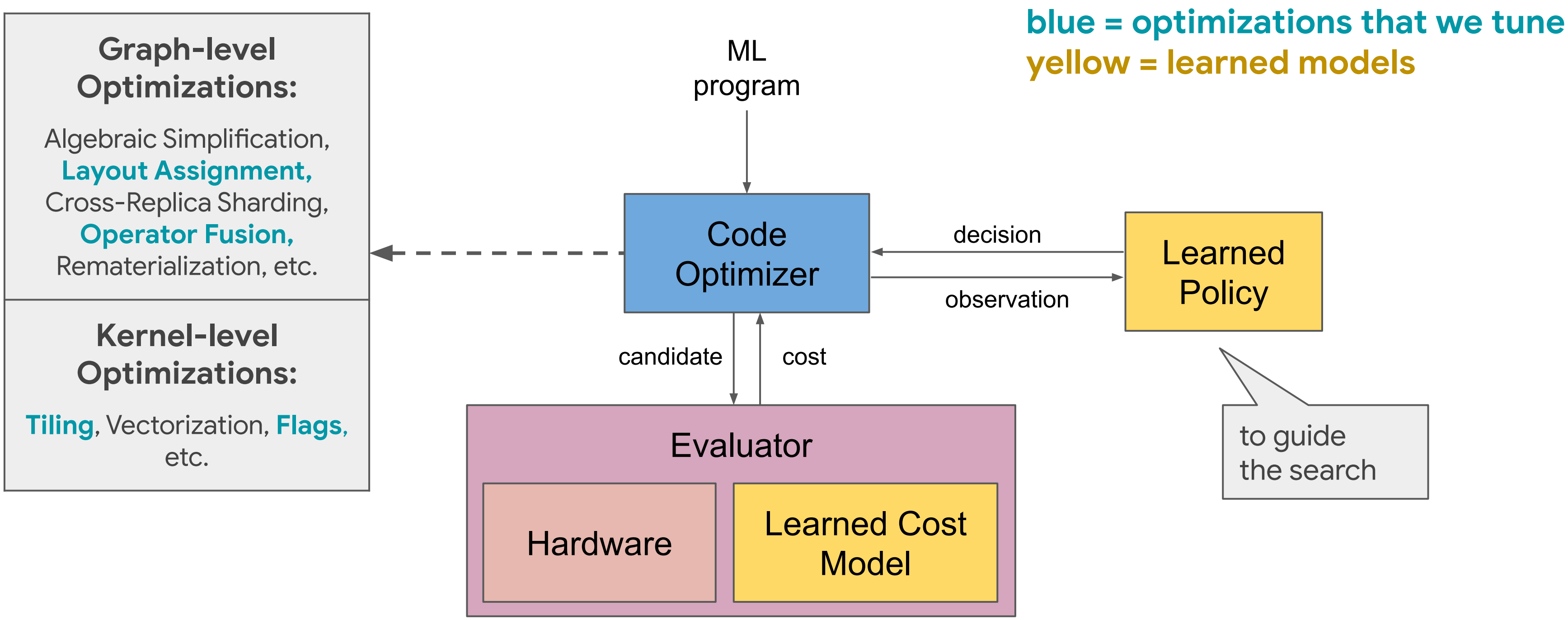


kernels / subgraphs

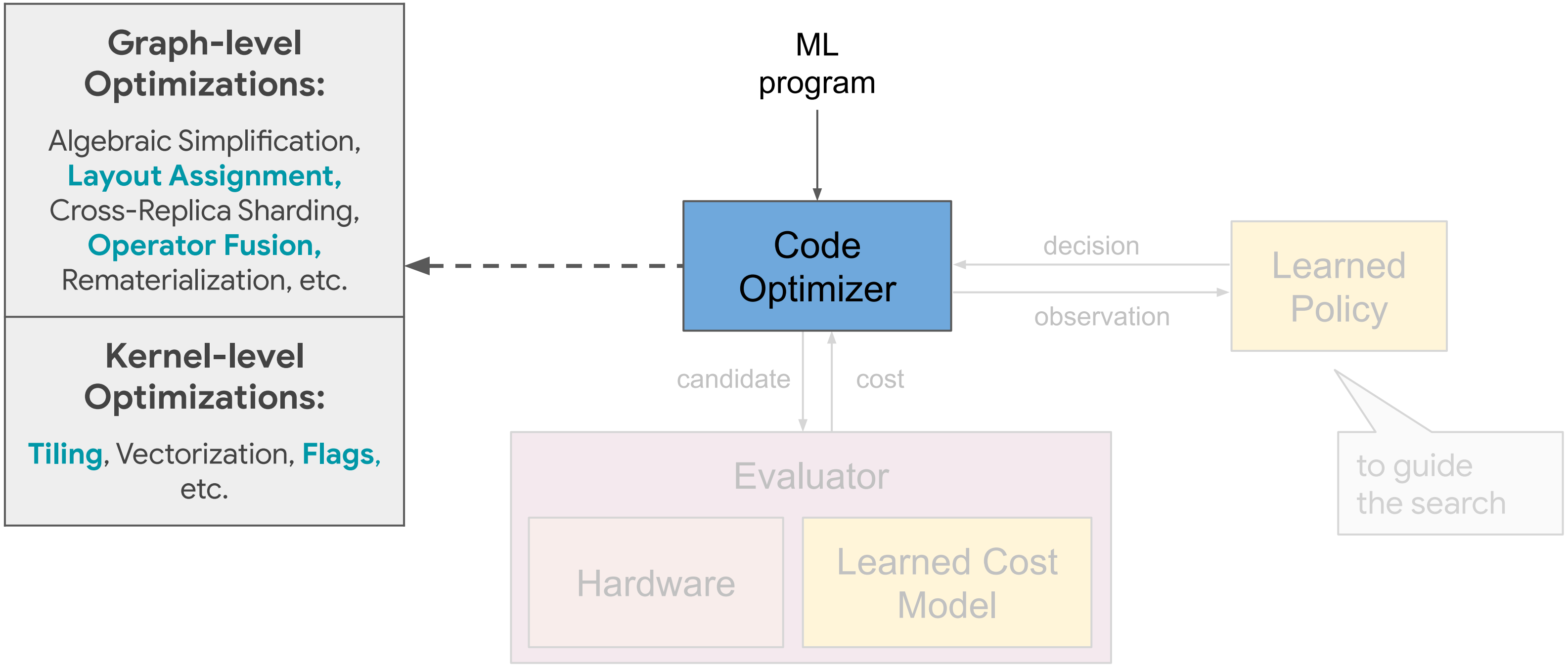
Kernel-Level Optimizations

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

XTAT: XLA TPU Autotuner



XTAT: XLA TPU Autotuner

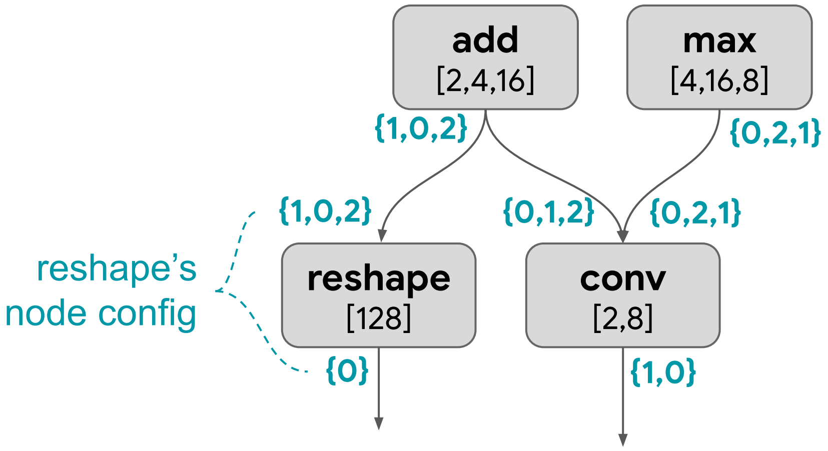


Pass Configuration

configuration on a tensor graph
for an **optimization pass**
is
a collection of per-node configurations that control
how the pass transforms each node in the graph

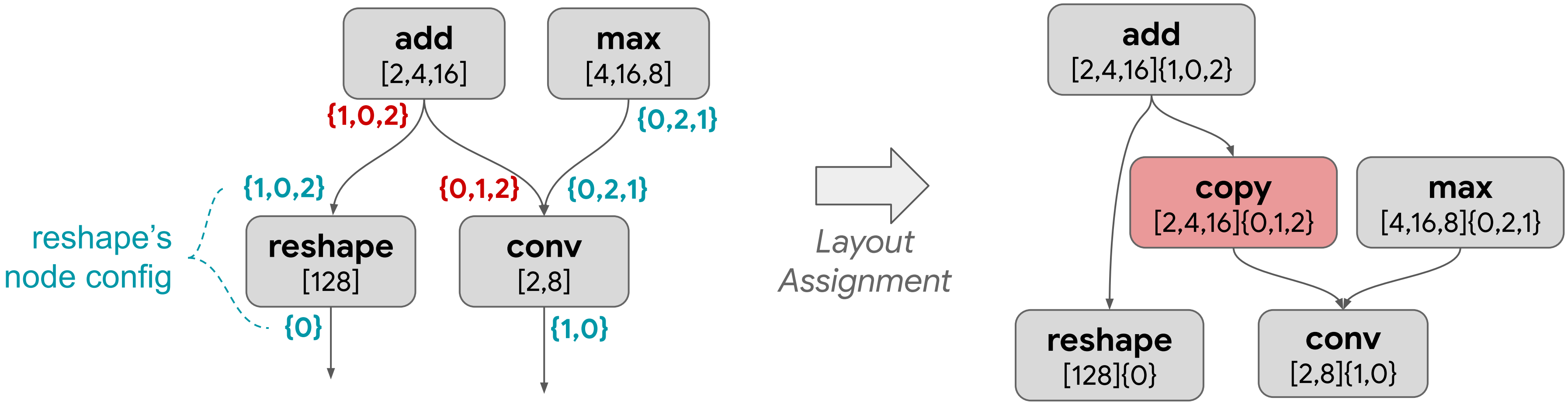
Layout Assignment

Example:



Layout Assignment

Example:



Layout Search Space

Option #1: Naive

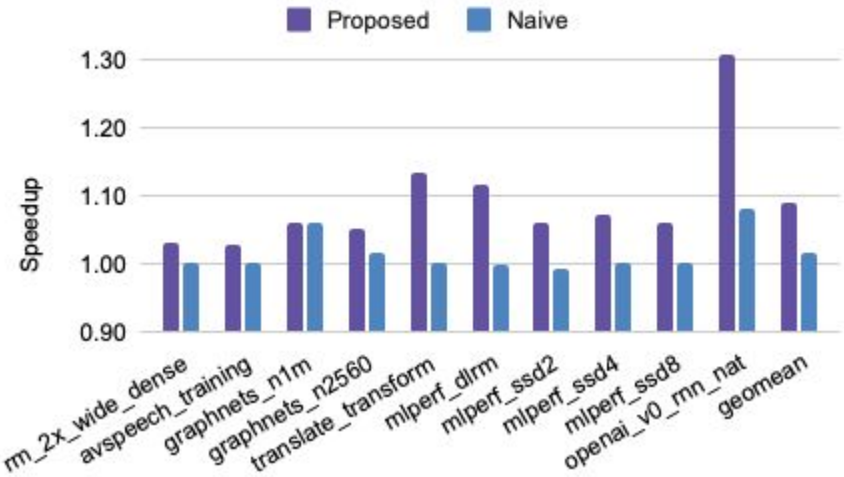
- Layout options for **each input/output** are **permutation** of its dimensions.
- Many **invalid configs** because there are constraints between tensors.

Option #2: Proposed

- Tune **layout options for important ops** (convolution and reshape).
- For each important op, get valid input-output layouts from XLA.
- Leverage XLA **layout propagation** algorithm.

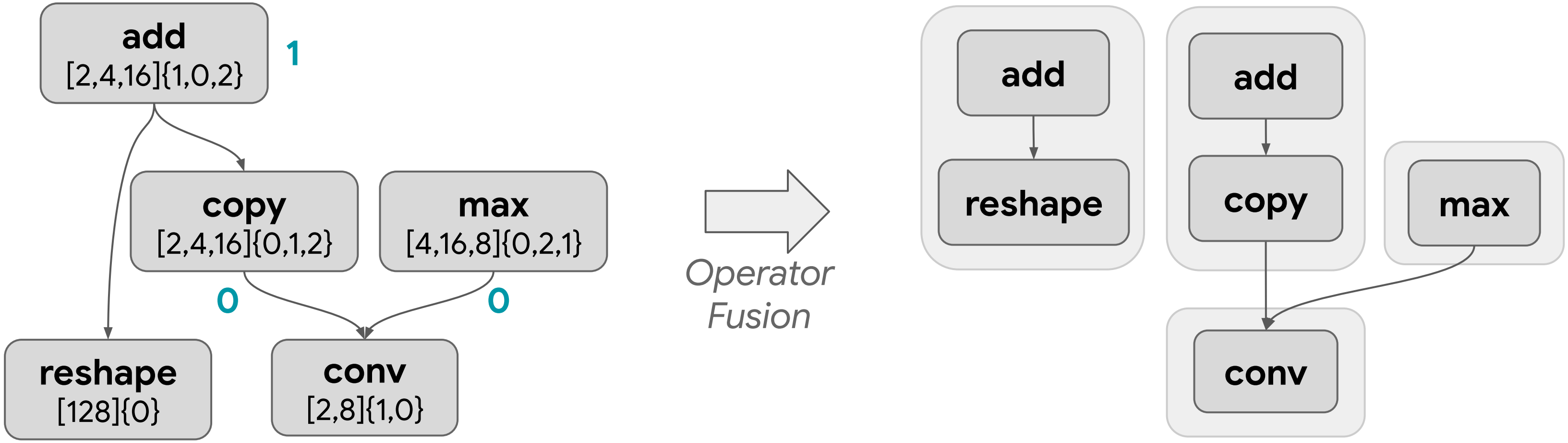
Layout Search Space: Result

Max speedup across 10 simulated annealing runs.



Operator Fusion

Example:



Google Research

Operator Fusion Search Space

Per-Node: assign a boolean value to each fusible node to control whether it is fused with its consumers

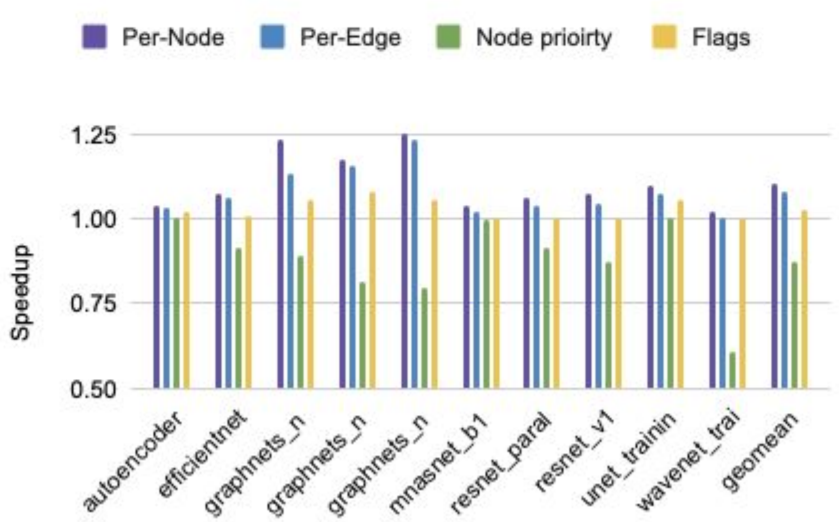
Per-Edge: assign a boolean value to each fusible edge

Node Priority: assign a node a priority value instead of controlling its fusion behavior explicitly

Fusion Parameter Flags: flags that
(1) limit fusions of inputs into convolutions,
(2) limit fusions of outputs into convolutions and
(3) parameterize the fusion cost model

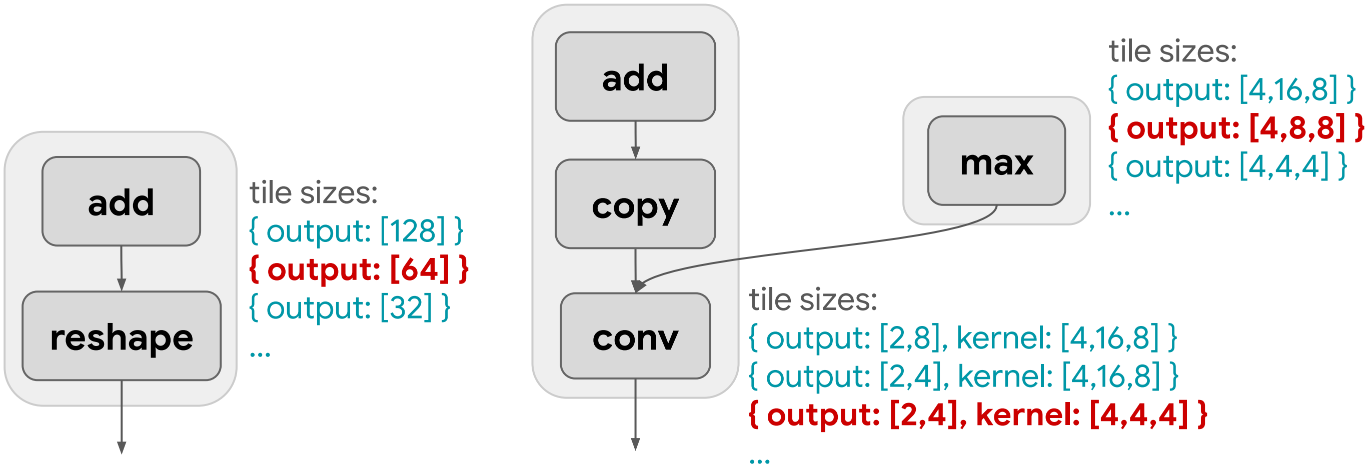
Operator Fusion Search Space: Result

Max speedup across 10 simulated annealing runs.



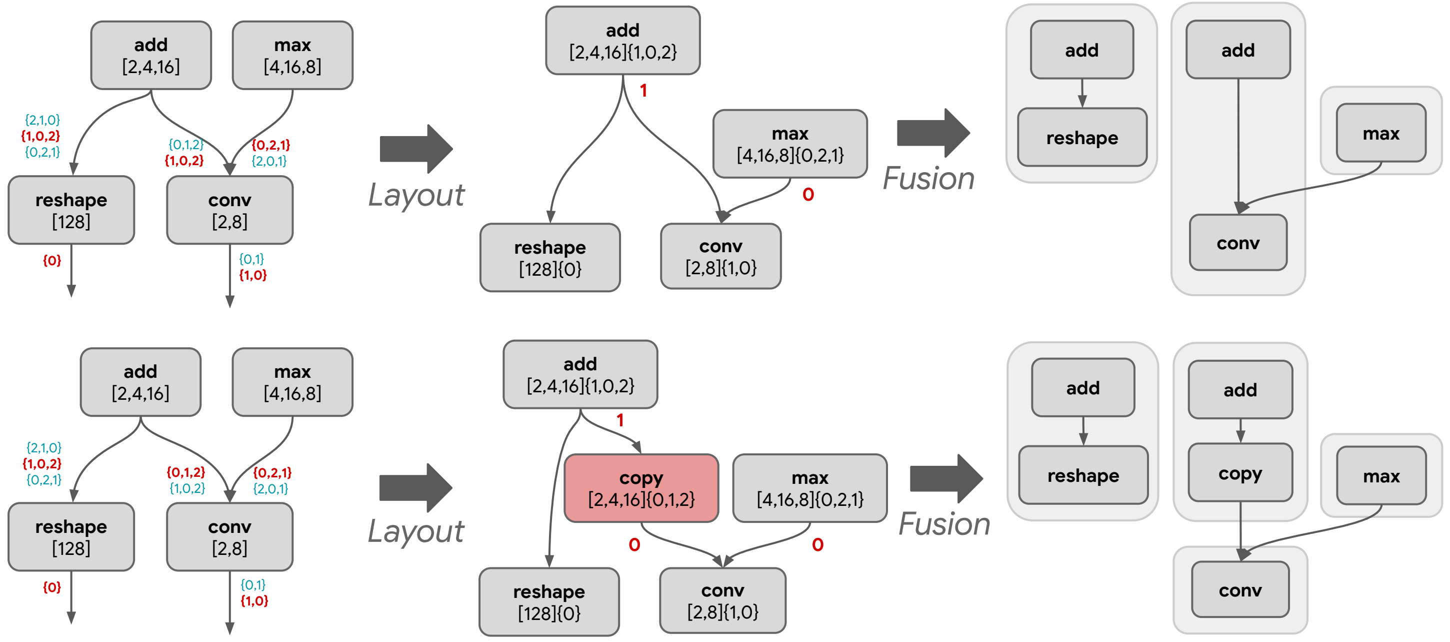
Tile Size & Code Gen Flags Search Space

Tune config for each fused node (kernel) independently.



Joint Autotuning: Challenges

g_A --- $A(config_A)$ ---> g_B --- $B(config_B)$ ---> g_{out}



$config_A$ determines the input graph g_B to pass B and its search space

When we change $config_A$ to $config_A$', g_B is changed, and $config_B$ is no longer valid.

How to not start the search for B from scratch when $config_A$ is changed?

Methodology for Joint Autotuning

```
function SEARCHSTEP(C)
    opt_id ← SelectOpt(Opts)
    C' ← GenerateCandidates(opt_id, C)
    for c : C' do
        UpdateAndApplyCandidate(opt_id, c)
        Evaluate(c)
    end for
    return SelectCandidates(C, C')
end function
```

Returns:
A, B, C, A, B, C, ... **(joint tuning)**
A, A, ..., B, B, ..., C, C, ... **(sequential)**
or some combinations of them

Methodology for Joint Autotuning

function $\textsc{SearchStep}(C)$
    $opt_{id} \leftarrow$ **SelectOpt**$(Opts)$
    $C' \leftarrow$ **GenerateCandidates**(opt_{id}, C)
    for $c : C'$ **do**
        $UpdateAndApplyCandidate(opt_{id}, c)$
        $Evaluate(c)$
    end for
    return **SelectCandidates**(C, C')
end function

Candidate c:
c.graphs = [g_A, g_B, g_{out}]
c.configs = [$config_A$, $config_B$]

Change $config_A$:
c.graphs = [g_A, g_B, g_{out}]
c.configs = [$config_A$', $config_B$]

Fix c to be *well-formed*:
c.graphs = [g_A, g_B', g_{out}']
c.configs = [$config_A$', $config_B$']

Google Research

Construct Well-Formed Candidate

Key ideas:

- Update subsequent graphs
- Update config_B' to have configurations for all nodes in g_B' from:
  - **config_B**
  - **global configuration store** (maintaining the best config per node)
  - **default value**

Change config_A:
c.graphs = $[g_A, g_B, g_{out}]$
c.configs = $[\text{config}_A', \text{config}_B]$

↓

Fix c to be *well-formed*:
c.graphs = $[g_A, g_B', g_{out}']$
c.configs = $[\text{config}_A', \text{config}_B']$

Google Research

Update Global Configuration Store

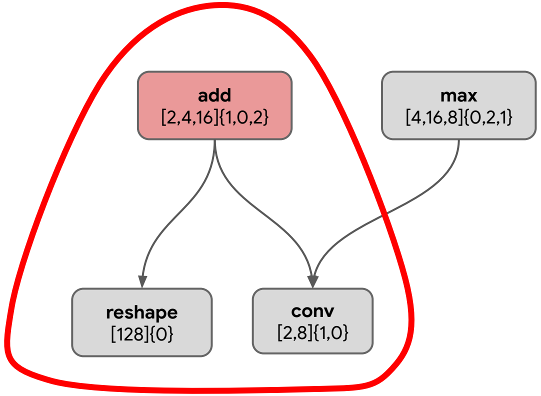
```
function EVALUATE(c)
    c.cost ← ExecuteGraph(c.graphs[final])
    if c.cost < best_candidate.cost then
        best_candidate ← c
        UpdateStore(ConfStore, c.configs)
    end if
end function
```

Global *ConfStore*

Key	Value
fp(n_0)	config of node n_0
fp(n_1)	config of node n_1
...	....

Update Global Configuration Store

function EVALUATE(c)
  $c.cost \leftarrow ExecuteGraph(c.graphs[final])$
  if $c.cost < best\_candidate.cost$ **then**
    $best\_candidate \leftarrow c$
    UpdateStore($ConfStore, c.configs$)
  end if
end function



Global *ConfStore*

Key	Value
fp(n_0)	config of node n_0
fp(n_1)	config of node n_1
...	....

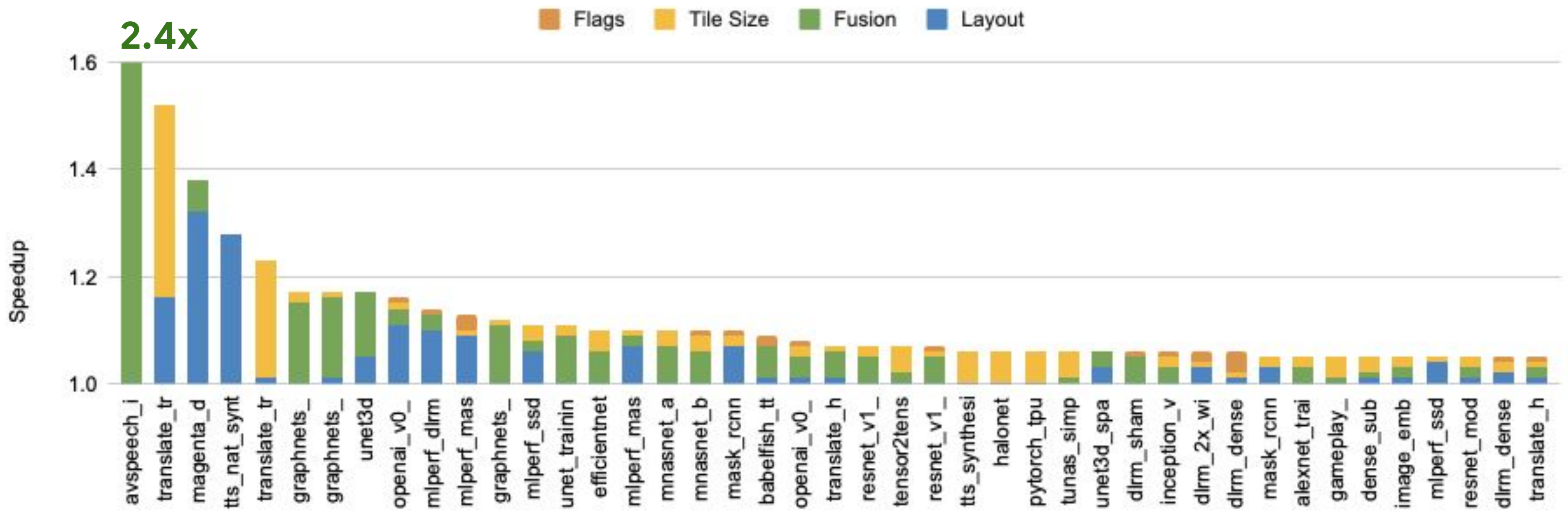
End-to-End Search Schedule

- Separate tuning graph-level and kernel-level optimizations for scalability
- Tuning **layout + fusion jointly is better** than sequentially
- Tuning **tile size + flag jointly is worse** than sequentially

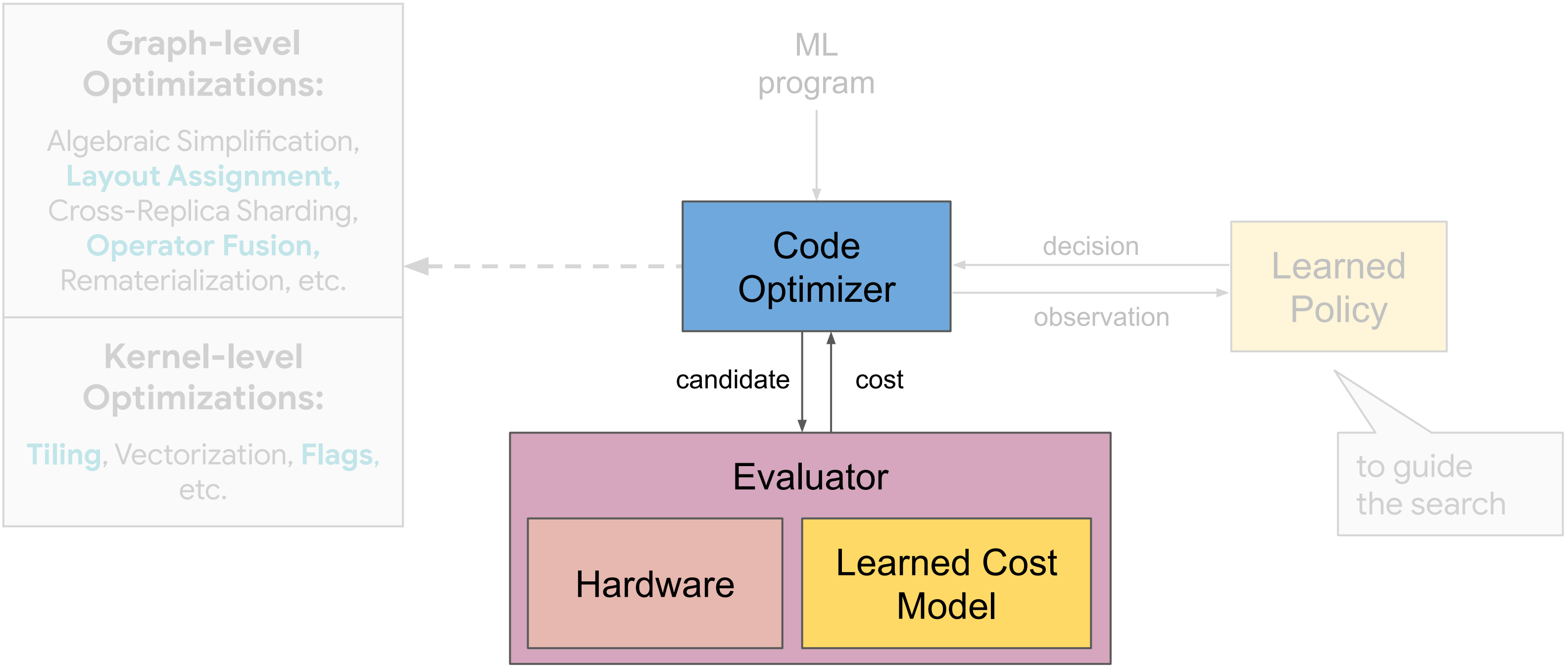
Tune **layout-fusion jointly** (simulated annealing)
→ then tune **tile size** (exhaustive)
→ then tune code gen **flags** (exhaustive)

End-to-End Runtime Speedup

We measured end-to-end model speedups from autotuning **150 ML models**.
The figure shows models that achieve 5% or more improvement.



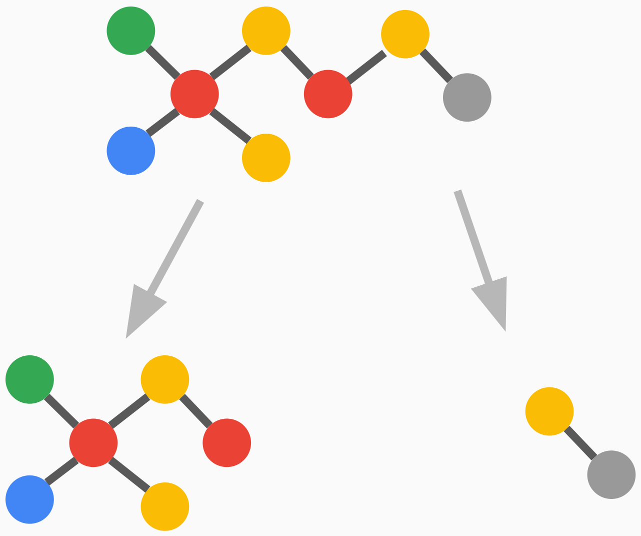
Learned Cost Model



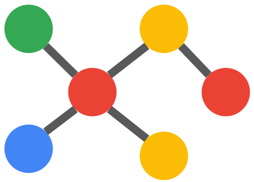
Ref: Kaufman and Phothilimthana et al., A Learned Performance Model for Tensor Processing Units, MLSys 2021.

Overview of Cost Model

1. Decompose Into Kernels



2. Regression Per Kernel

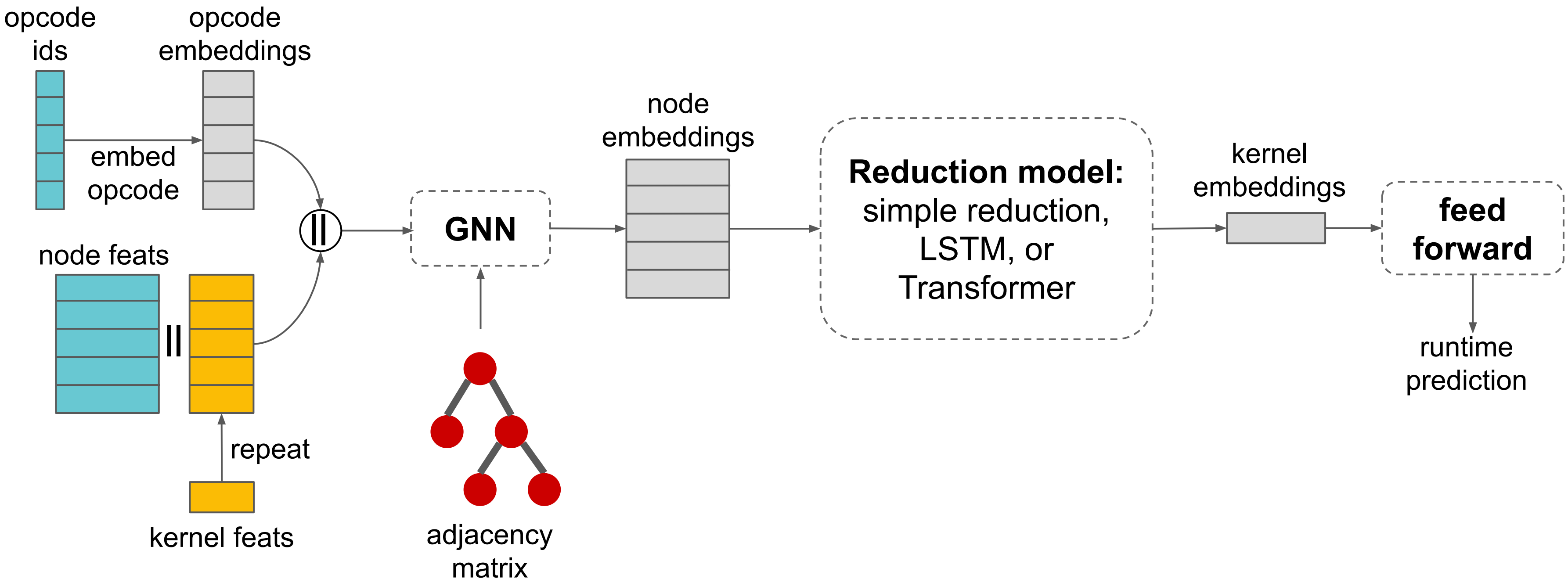
$f($  $)$

KERNEL

$\cong 5.2s$

RUNTIME

Model Architecture



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).

Google Research

Training for All Optimization Tasks

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

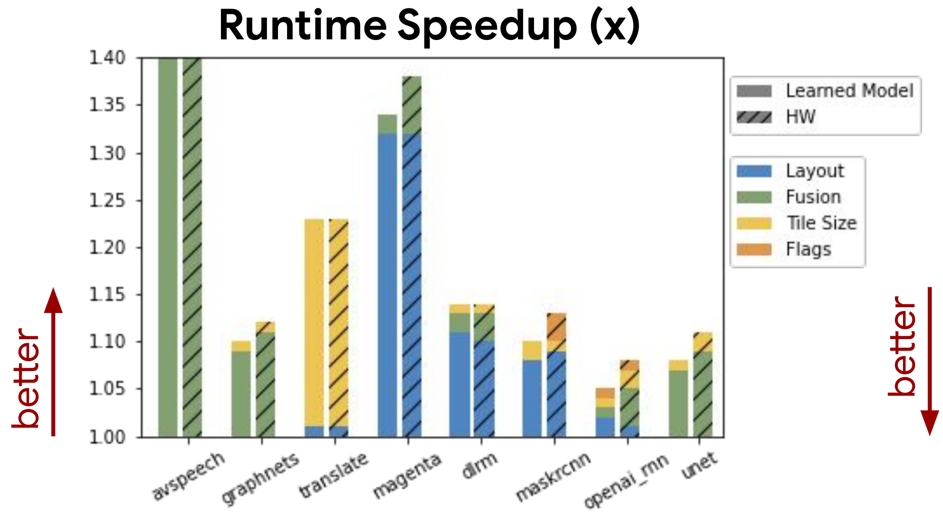
- Train:
  - one model for all **graph-level** optimizations to predict **absolute runtime**
  - one model for **tile-size** to predict **relative runtime**
  - one model for **flags** to predict **relative runtime**

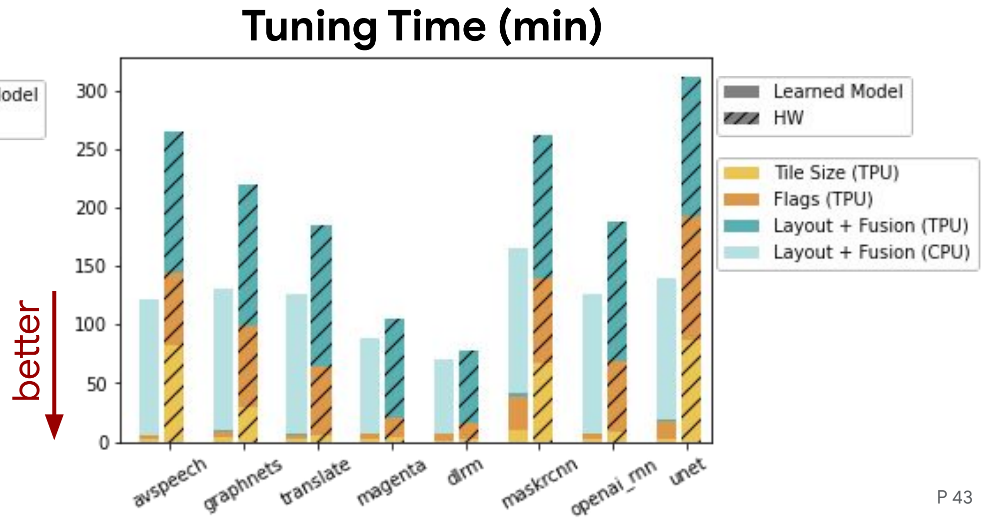
- The graph embedding network is shared between tile-size and flags models.

Tuning with Learned Cost Model

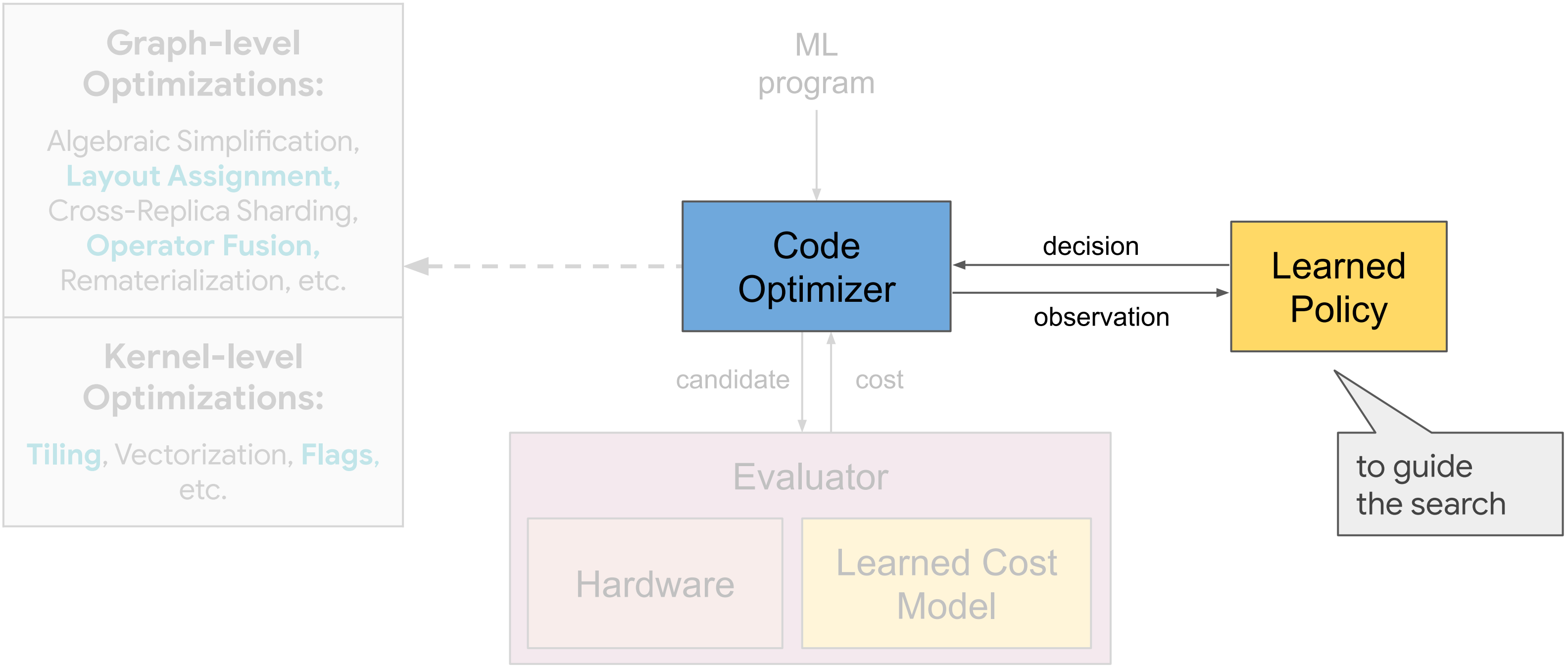
Execute the top k configurations from each worker according to the model on real hardware and pick the best.

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





Search Strategies



Search Strategies

- Exhaustive
- Simulated annealing (SA)
- Evolutionary (EVO)
- Model-based optimization (MBO)
- Deep reinforcement learning (RL)

Model-Based Optimization (MBO)

- At each optimization round, a set of candidate **regression models are fit to the acquired data**.
- Good models are **assembled to define an acquisition function.**
- The acquisition function is then **optimized by EVO to generate a new batch of samples**.
- Candidate models: ridge regression, random forests, gradient boosting, and neural networks

Ref: Angermueller et al., Model-based reinforcement learning for biological sequence design, ICLR 2019

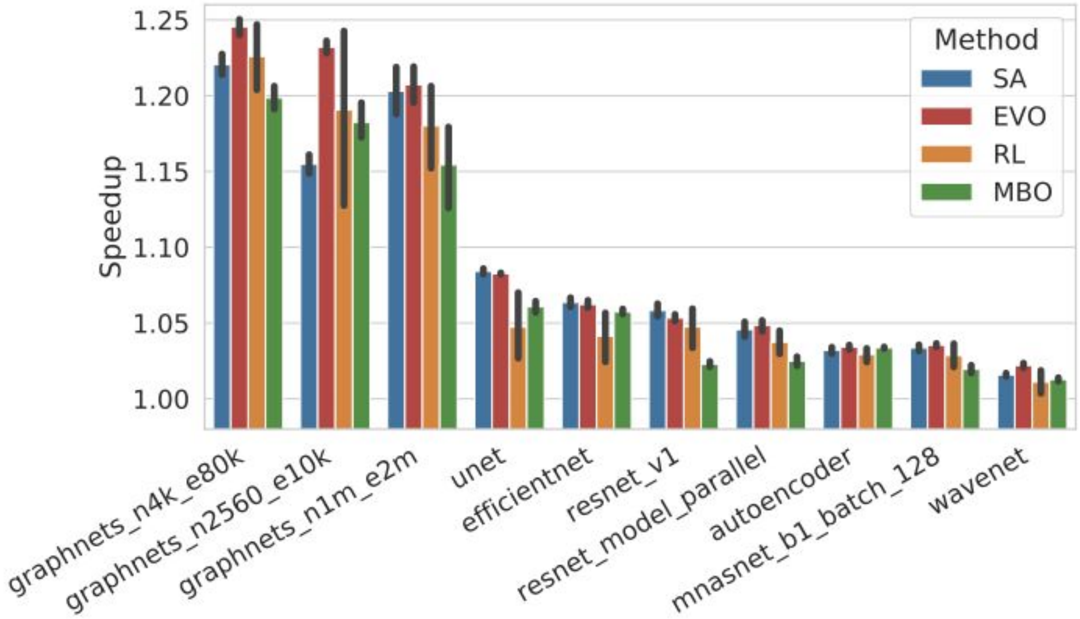
Deep Reinforcement Learning (RL)

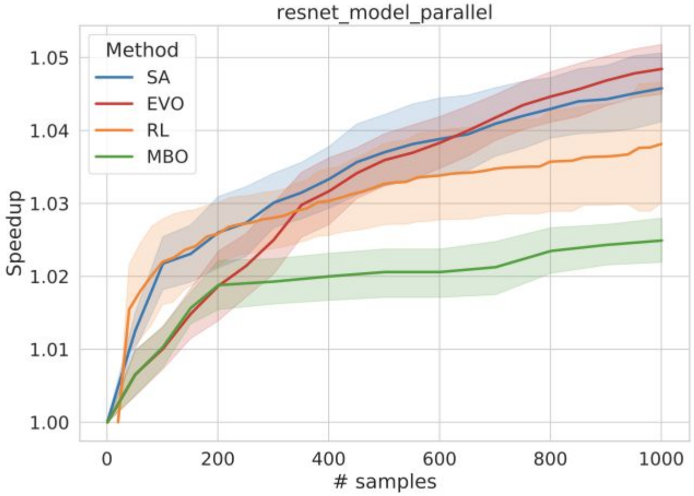
- Designed specifically for ML compiler's graph optimizations
- Uses a **graph neural network** to create node embeddings and segmented **recurrent attention layers** to capture long-range dependencies
- **Non-autoregressive**
  - N node decisions are done in parallel
  - Conventional autoregressive approach is infeasible as N can be as large as 100k

Ref: Zhou et al., Transferable graph optimizers for ml compilers, NeurIPS 2020

Search Strategies: Fusion Autotuning

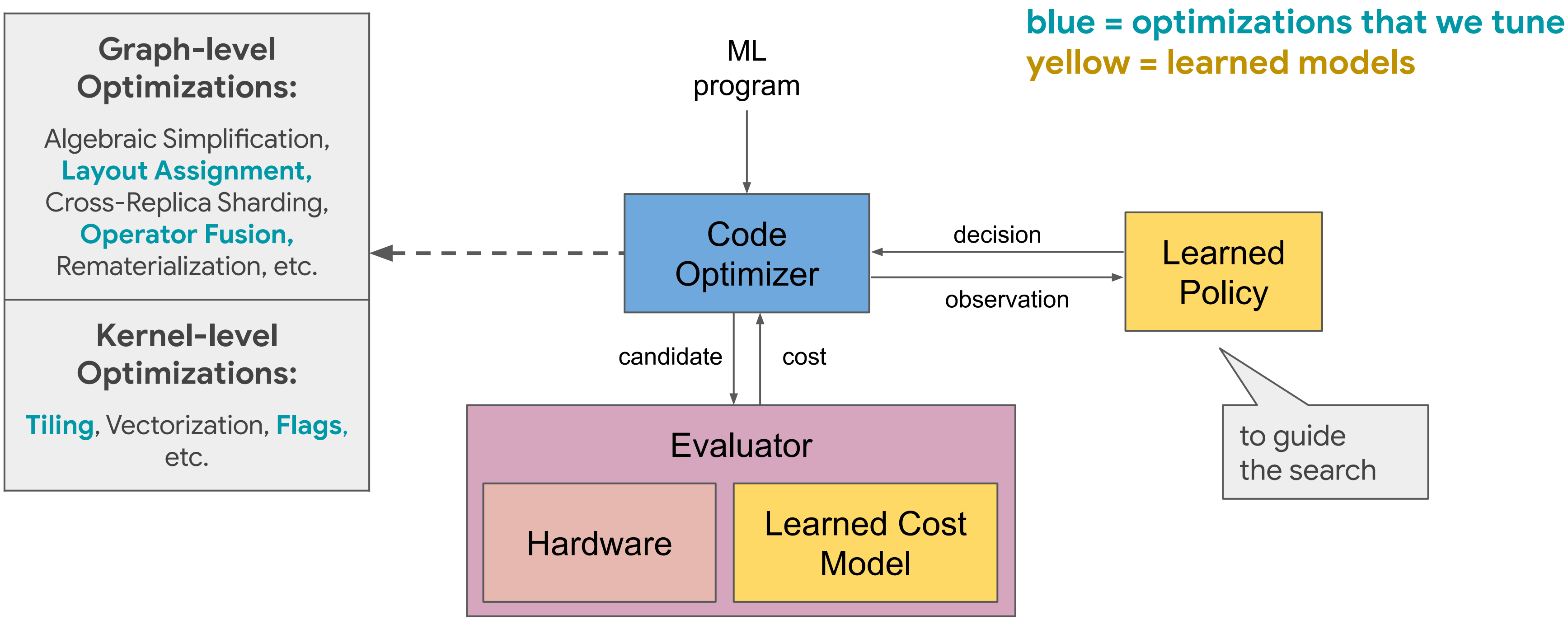
Average speedup across 10 runs. Each run evaluated 10,000 candidates.





Google Research

XTAT: XLA TPU Autotuner



Google Research

Data-Center Scale Deployment

Deployment Strategies

Google Research

Offline autotuning

Users run autotuner offline on their workloads. Save best configs and use them in future compilation.

Pros

Fast compilation.
More time for autotuning.

Cons

Require more user's effort.

Deployment Strategies

Online autotuning

Compiler runs autotuner automatically during compilation.

Pros

Easy to use.

Cons

Small time budget for autotuning.

Reproducibility issue.

Deployment Strategies

Profile-guided autotuning

System runs autotuner automatically **on top workloads** and saves best configs in shared database. Compiler uses configs from database if exist.

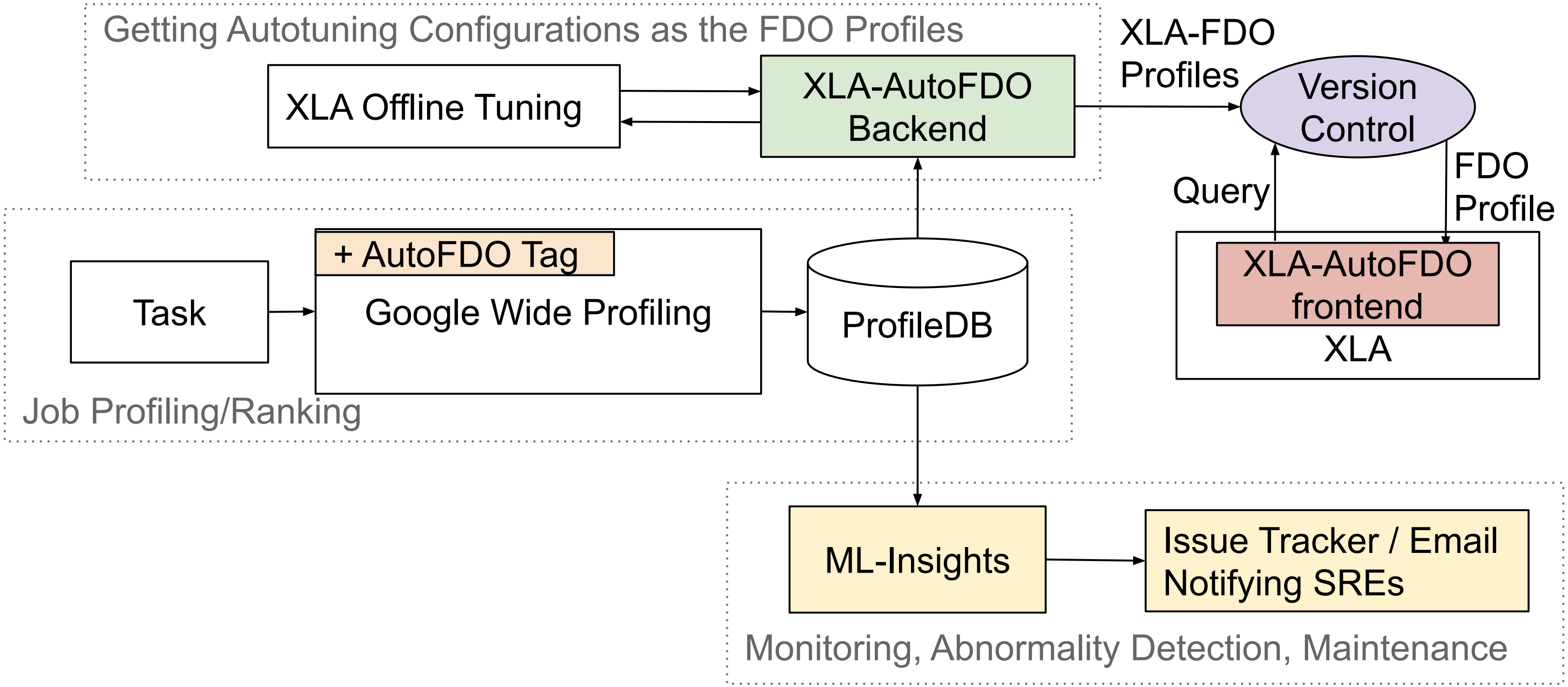
Pros

Easy to use.
Fast compilation.
More time for autotuning.

Cons

Some workloads won’t get benefit.

Fleet-Wide TPU Autotuning at Google

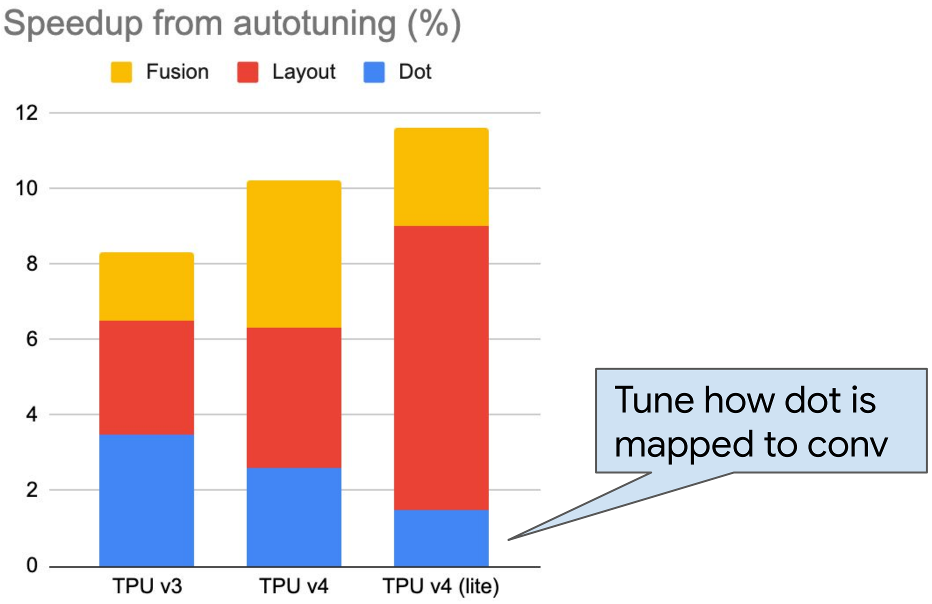


Fleet-Wide TPU Autotuning at Google

- Have deployed the tile size and flags autotuning to optimize top workloads in the TPU fleet daily
- Learned cost model enabled tuning 20x more kernels per day
- Save >2% of total TPU consumption
- Savings / tuning cost: ~40x

Graph-Level Autotuning on the Fleet

Tune top 1000 graphs from Google fleet. 1 hour limit.



Google Research

References

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

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

Google Research

Contributors

Aleksey Orekhov
Amit Sabne
Arissa Wongpanich
Berkin Ilbeyi
Bjarke Roune
Blake Hechtman
Charith Mendis
Christof Angermueller
Emma Wang
Jeremy Wilke
Jose Baiocchi Paredes
Karthik Srinivasa Murthy
Ketan Mandke
Mangpo Phothilimthana
Mike Burrows
Nacho Cano
Nikhil Sarda
Peter Ma
Rezsa Farahani
Sam Kaufman
Shen Wang
Sudip Roy
Yanqi Zhou
Yuanzhong Xu