# A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers

Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou,
Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, Yu Emma Wang, Berkin Ilbeyi,
Blake Hechtman, Bjarke Roune, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman*
Google, *University of Washington
Email: mangpo@google.com

***Abstract*—Search-based techniques have been demonstrated effective in solving complex optimization problems that arise in domain-specific compilers for machine learning (ML). Unfortunately, deploying such techniques in production compilers is impeded by two limitations. First, prior works require factorization of a computation graph into smaller subgraphs over which search is applied. This decomposition is not only non-trivial but also significantly limits the scope of optimization. Second, prior works require search to be applied in a single stage in the compilation flow, which does not fit with the multi-stage layered architecture of most production ML compilers.**

**This paper presents XTAT, an autotuner for production ML compilers that can tune both graph-level and subgraph-level optimizations across multiple compilation stages. XTAT applies XTAT-M, a flexible search methodology that defines a search formulation for joint optimizations by accurately modeling the interactions between different compiler passes. XTAT tunes tensor layouts, operator fusion decisions, tile sizes, and code generation parameters in XLA, a production ML compiler, using various search strategies. In an evaluation across 150 ML training and inference models on Tensor Processing Units (TPUs) at Google, XTAT offers up to 2.4× and an average 5% execution time speedup over the heavily-optimized XLA compiler.**

***Keywords*-compiler, autotuning, machine learning**

## I. INTRODUCTION

Machine learning (ML) compilers solve multiple optimization problems to translate an ML program, typically represented as a tensor computation graph, to an efficient executable for a hardware target. Recent works have demonstrated that search-based techniques can be used to solve many of these problems effectively [1]–[10]. However, production ML compilers (e.g., XLA [11] and Glow [12]) still rely on heuristics to solve these problems quickly, albeit often sub-optimally. Current search-based techniques [1]–[10] have at least one of the two key shortcomings that prevent them from being deployed in production ML compilers.

First, they rely on the assumption that performance-critical optimization decisions are localized within a subgraph and hence can be made independently from the rest of the graph [1]–[8]. This is often not the case. For example, decisions to fuse[1] tensor operations affect memory requirements, which in turn affect decisions to rematerialize[2] tensors in different portions of the graph. Furthermore, the subgraph-focused solutions assume that they can easily partition a tensor computation graph into suitable subgraphs. However, finding an optimal partitioning for a particular optimization task is a non-trivial combinatorial optimization problem by itself. A common strategy is to partition a graph according to the neural network layers [3], [5], ignoring cross-layer optimization opportunities. We empirically observed a regression of up to 2.6× and 32% on average across 150 ML models by limiting fusions in XLA to be within layers. Furthermore, prior approaches [1]–[10] have primarily studied inference graphs. Training graphs, on the other hand, can be up to two orders of magnitude larger than inference graphs in terms of number of nodes, rendering subgraph decomposition less effective.

Second, the optimizations enabled in prior works [1]–[10] are applied at the same stage in the compilation flow, specifically in the loop transformation stage. However, this is impractical in production compilers because compiler transformations are organized as passes to reduce complexity, and have strict ordering constraints. For example, in the XLA compiler, tensor layout assignment occurs before operator fusion. This is required because operator fusion's goal is to reduce memory traffic, which is impossible to estimate without layouts. The automatic cross-replica sharding [13] happens between the layout and fusion passes. It requires layouts to calculate parallelization overheads correctly, and must execute before the fusion pass because otherwise the distribution granularity becomes coarse, lowering performance gains. Therefore, joint optimization of layout and fusion cannot occur at the same compilation stage. To perform joint layout-fusion optimization using the previous research approaches, all intermediate transformations must be co-optimized too, which does not scale in practice.

We next discuss our approach to overcome these shortcomings. To address the issue of the optimization scope, we develop an autotuner for ML compilers that supports tuning decisions made at both whole graph and subgraph levels. Specifically, we implement **XTAT** (pronounced "stat"), an autotuner for the XLA compiler. **XTAT** can tune tensor layouts

[1]When operators are fused, intermediate tensors can be used by the consuming operator directly without saving them to the slow main memory.

[2]Rematerialization reduces memory usage by recomputing a tensor when it is needed instead of saving it in memory.

978-1-6654-4278-7/21/$31.00 ©2021 IEEE
DOI 10.1109/PACT52795.2021.00008

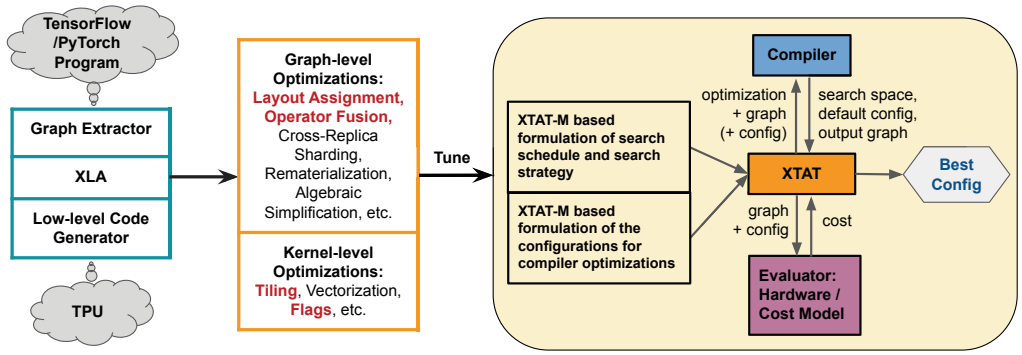



Fig. 1: Overview of **XTAT**, an autotuner for XLA. **XTAT** takes as inputs: (a) a program/graph to optimize, (b) a formulation of the search strategy in **XTAT-M**, and (c) a formulation of the compiler optimization configurations in **XTAT-M**. **XTAT** currently supports tuning optimization passes highlighted in red. **XTAT** uses the learned cost model and/or hardware to evaluate the performance of different optimization configurations and outputs the best configuration found.

and fusion decisions at the graph level, and tune tile sizes[3] and critical code generation parameters at the kernel (subgraph) level for TPUs [14], [15]. Employing search at the graph level is challenging due to search spaces that are exponentially large in the number of nodes. Therefore, it is important to ensure that the search space for an optimization pass is well designed, as the design can greatly affect the quality of the final solution found by a search method. To this end, we present effective search spaces for optimizations tuned by **XTAT**.

To tune multiple compiler optimizations at different stages, we develop a flexible methodology to apply search to multiple compiler passes, called **XTAT-M**. The complexity of multi-stage joint optimizations tuning comes from the fact that decisions made in one optimization pass affect input graphs to subsequent passes and, thereby, their configurations and search spaces. To address this complexity, **XTAT-M** defines configuration-update specifications, which co-relate the state of the intermediate graph left behind by one compiler optimization to a partial solution for a subsequent compiler optimization. Through such specifications, **XTAT-M** allows composing search-based strategies in certain passes (e.g., for layout and fusion) with existing heuristic-based strategies in intermediate passes (e.g., for cross-replica sharding). Additionally, **XTAT-M** enables flexibility to trade off time spent searching for a solution and solution optimality through a configurable *search schedule*. This flexibility is critical to achieve the best performance given a time budget. Lastly, **XTAT-M** allows us to apply a wide range of search techniques including exhaustive search, simulated annealing, evolutionary search, model-based optimization, and reinforcement learning.

To summarize this paper's contributions:

- We develop **XTAT-M**, a novel methodology to formulate search for multiple optimizations at different stages of an ML compiler with a flexible search schedule.
- We build the **XTAT** autotuner based on **XTAT-M** to tune tensor layouts, operator fusion decisions, tile sizes, and code generation parameters in XLA, as shown in Fig. 1.
- **XTAT** is the first autotuner that can jointly tune multiple optimizations at different stages of an ML compiler.
- **XTAT** handles much larger search spaces compared to prior works since **XTAT** optimizes at both graph and subgraph levels. For instance, the number of valid fusion configurations alone for the EfficientNet training model with approximately 40,000 nodes is $2^{40,000}$. In contrast, prior works [1]–[8] tune one subgraph, which typically has fewer than 10 nodes, at a time.
- We demonstrate how to incorporate advanced techniques such as a learned cost model and various search strategies into **XTAT** to reduce autotuning time.
- We evaluate **XTAT** on 150 ML models (comprising both training and inference models from Google's production and research workloads) on TPUs and achieve significant improvement: up to 2.4× speedup and 5% on average over the heavily-optimized XLA compiler.

[3]A tile of tensor is processed at a time to effectively utilize fast memory (e.g., scratchpad and cache).

## II. XTAT-M: Methodology & Formulations

### A. Overview

We define a *configuration* on a graph for an optimization pass as a collection of per-node configurations that control how the pass transforms each node in the graph. For example, consider the pass that determines a tensor layout (i.e., a physical ordering of tensor dimensions in memory). A configuration for this pass is a collection of physical layouts assigned to tensors at each node in the graph.

Searching for the best configuration for a compiler optimization involves (i) exploration of candidate configurations, (ii) application of the optimization pass according to the candidates, and (iii) evaluation of the output graphs. Implementing these steps for a single optimization is relatively straightforward, while the quality of results depends on the expressiveness and compactness of the search space as well as the capability of the search technique.

However, searching for optimal configurations for multiple optimizations requires that the search satisfies a critical

condition: *the search space and the configuration chosen for an optimization pass must be consistent with decisions made in prior passes.* For example, consider Fig. 2 with two optimization passes, *layout assignment* (which determines tensor layouts) and *operator fusion* (which fuses several operators together). Since the performance of a fused node depends on the layouts of the relevant tensors, the decisions in the fusion pass must take into account the layouts decided in the prior pass. Further, the search space for the fusion pass (i.e., the set of node configurations to consider for search) depends on the layout assignment since the layout pass may transform the graph. As shown in Fig. 2, two layout configurations lead to different input graphs for the fusion pass: one contains a `copy` node, while the other does not; as a result, the fusion search space for the left graph must contain configurations for `copy`, but the search space for the right must not.

How can we orchestrate searches for joint optimizations to satisfy the above condition? To do so, we identify two key properties that a search should implement. To describe these properties, consider the scenario where an optimization pass, say $A$, is applied before another optimization pass, say $B$.

1) The ordering between the produced graphs is $g \xrightarrow{A(config_A)} g' \xrightarrow{B(config_B)} g''$. This ordering implies that: (i) the graph $g'$ is a result of applying a candidate configuration $config_A$ selected for $A$ and (ii) the search space of candidate configurations and a selected candidate $config_B$ for $B$ is based on $g'$.

2) All configurations are *well-formed*; a configuration $config_B$ contains valid configurations for all configurable nodes in $g'$. Further, when $config_A$ is changed, causing graph $g'$ to change, $config_B$ must be updated to be compatible with $g'$ using the best nodes' configurations observed so far for $B$. This ensures that the search for $B$ does not begin the exploration from scratch.

To realize this orchestration, we develop a generic methodology **XTAT-M** to formulate the interactions between multi-stage compiler optimizations, and search strategies. Our proposed search methodology is applicable to compilers that apply multiple optimization passes in sequence.

### B. Methodology

The search formulation of **XTAT-M** is displayed in Fig. 3. We model the search process as a sequence of search steps. Each search step takes a set of current candidates $C$ and produces a new set of candidates $C'$ for the next round. When tuning $n$ optimizations, a candidate $c$ in $C$ captures $n$ graphs ($c.graphs$) and $n$ configurations ($c.configs$), where $c.graphs[id]$ and $c.configs[id]$ are an input graph and a configuration for an optimization pass $id$ respectively. As color coded in Fig. 3, **XTAT-M** allows one to configure its routines to implement a desired search schedule (red) and search strategy (green), and tailor the configuration-update procedures to specific optimizations (purple). Each search step (*SearchStep* lines 8–16) performs the following actions:

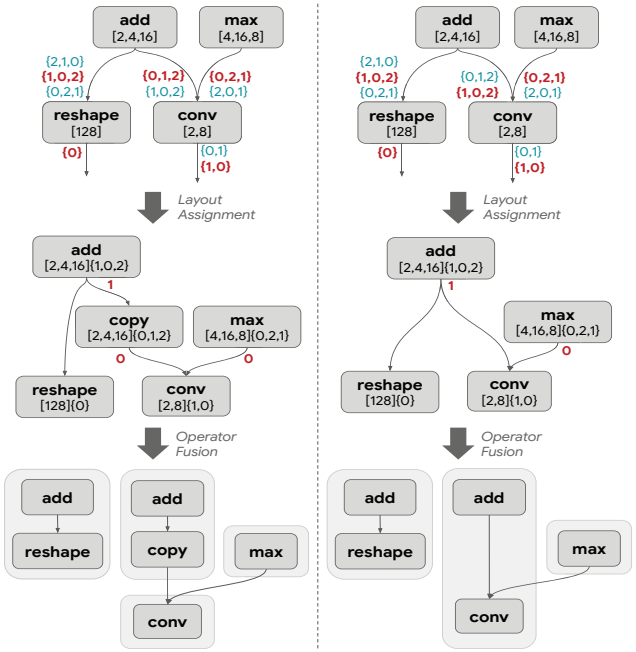



Fig. 2: **Layout configurations determine the input graphs to the operator fusion pass.** A node is annotated with its output tensor shape $[n_0, n_1, ...]$, where $n_i$ is the size of dimension $i$. Layout $\{d_0, d_1, ...\}$ represents minor to major ordering in memory. Applied configurations are highlighted in red, and other valid configurations are highlighted in blue. A layout configuration specifies the layouts of inputs and outputs of influential operators (i.e., convolution and reshape). The compiler propagates layouts from the influential nodes to others. A copy operator is inserted when there is a layout mismatch. A fusion configuration specifies which nodes are fused, where a node with decision value 1 must be fused with all of its consumers. If a node being fused has multiple consumers (e.g., `add`), it is duplicated (recomputed). **The fusion configuration for `add` is the same (fused) between left and right, but the outcomes differ; `conv` is fused in the right but not in the left scenario.** This figure shows operator fusion directly following layout assignment, but other intermediate passes can alter the input to the fusion pass further.

- *SelectOpt*: select an optimization to tune
- *GenerateCandidates*: develop search candidates
- *FixAndApplyCandidate*: fix and apply a candidate
- *Evaluate*: evaluate a candidate
- *SelectCandidates*: select candidates for the next step

*Terminate* (line 3) determines when the search stops, e.g., when all candidates are explored, or a certain number of steps have been performed, or the time limit is reached. The search process also maintains a global *ConfStore* that captures the best configurations observed for all optimizations.

*1)* **SelectOpt***:* The flexibility to control the *search schedule* is critical to achieve the best performance given a desired time budget. Tuning optimizations jointly often enables more performance improvement opportunities only if we are able to explore the search space sufficiently. **XTAT-M** allows users to configure the search schedule by controlling the return values of *SelectOpt*. To tune one optimization pass, say, $B$, after another, say $A$, *SelectOpt* can return a series of $A, A, ..., B, B, ..., C, C, ...$ across calls. To enable joint op-

```
Global variables: Opts, ConfStore, best_candidate
 1: function MainSearch(g_input)
 2:     C ← Init(g_input)
 3:     while !Terminate() do
 4:         C = SearchStep(C)
 5:     end while
 6: end function
 7:
 8: function SearchStep(C)
 9:     opt_id ← SelectOpt(Opts)
10:     C' ← GenerateCandidates(opt_id, C)
11:     for c : C' do
12:         FixAndApplyCandidate(opt_id, c)
13:         Evaluate(c)
14:     end for
15:     return SelectCandidates(C, C')
16: end function
17:
18: function FixAndApplyCandidate(opt_id, c)
19:     g ← ApplyOpt(opt_id, c.graphs[opt_id], c.configs[opt_id])
20:     // Update configs of later passes to be compatible with g
21:     for id : SubsequentOpts(Opts, opt_id) do
22:         c.graphs[id] ← g
23:         c.configs[id] ← InferConfig(c.configs[id], g, ConfStore[id])
24:         g ← ApplyOpt(id, g, c.configs[id])
25:     end for
26:     c.graphs[final] ← g
27: end function
28:
29: function Evaluate(c)
30:     c.cost ← ExecuteGraph(c.graphs[final])
31:     if c.cost < best_candidate.cost then
32:         best_candidate ← c
33:         UpdateStore(ConfStore, c.configs)
34:     end if
35: end function
36:
37: function Init(g)
38:     for id : Opts do
39:         config ← GetInitConfig(id, g)
40:         g_combo[id] ← g
41:         config_combo[id] ← config
42:         g ← ApplyOpt(id, g, config)
43:     end for
44:     g_combo[final] ← g
45:     cost ← ExecuteGraph(g)
46:     C[0] ← {graphs : g_combo, configs : config_combo, cost : cost}
47:     return C
48: end function
```

Fig. 3: Formulation of **XTAT-M** search methodology. Routines in red are configured by users to control the search schedule. Routines in green are implemented by search strategies. Routines in purple are optimization-specific and essential for joint optimizations. Routines in blue are interfaces to retrieve information from a compiler.

timizations, we can configure *SelectOpt* to alternate passes: $A,B,C,A,B,C,\ldots$. **XTAT-M** also supports a mixture of sequential and joint tuning, e.g., tuning $A$ and $B$ jointly followed by tuning $C$ and $D$ jointly: $A,B,A,B,\ldots,C,D,C,D,\ldots$. Note that a mixed search schedule alone does not enable a joint optimization, it is the combination of a mixed search schedule and candidates *fixing* (described later in *FixAndApplyCandidate*) that enables a joint optimization.

*2)* **GenerateCandidates***:* **XTAT-M** lets one apply various search strategies by implementing *GenerateCandidates*, which generates a new set of candidates from the current set of candidates. Note that a candidate contains configurations for all optimizations being tuned. *GenerateCandidates* primarily focuses on generating new configurations for a specific optimization $opt_{id}$, and optionally new configurations for the subsequent interacting optimizations. Note that a new candidate created from this routine may be *ill-formed*; i.e., *c.configs* may not be compatible with *c.graphs* because this routine changes configurations without updating or considering the graphs. Consider Fig. 2 as an example, let the left layout and fusion configurations compose a candidate $c$. If we mutate only the layout of $c$ to be the layout shown on the right side of the figure, the fusion configuration is no longer valid for the right graph because the `copy` node has been removed. We allow *GenerateCandidates* to generate ill-formed candidates in this step and then fix them in the next step. This is how we enable search strategy implementers to develop their search algorithms without having to deal with complex compiler transformation effects between interacting optimizations.

*3)* **FixAndApplyCandidate***:* In this step, we update the intermediate graphs and the configurations of all optimizations to be well-formed using *ApplyOpt* and *InferConfig*. $ApplyOpt(opt_{id}, g, config)$ transforms $g$ by applying optimization $id$ and potentially more optimizations (using compiler's heuristics) between $id$ and $id+1$. For example, when $opt_{id}$ is layout, to obtain the input graph for the fusion pass, we apply layout assignment pass with respect to *config* and many more transformations such as conditional code motion and cross-replicas sharding. Note that not all of these transformations are worthy of tuning; **XTAT-M** enables applying the heuristic-based decisions for them naturally.

When a configuration for an optimization, say $A$ (e.g., layout), is changed by *GenerateCandidates*, the configurations for subsequent interacting optimizations, say $B$ (e.g., fusion), must be updated to be well-formed. *InferConfig* (line 23) does so by fixing $c.configs[B]$ to be compatible with the graph $g$, generated from applying $c.configs[A]$ for pass $A$. *InferConfig* first identifies the nodes that are *unchanged* between $g$ and $c.configs[B]$ and carries forward the configurations in $c.configs[B]$ for such nodes. For the other nodes in $g$, it pulls the best configurations found so far from the global *ConfStore* if present; otherwise, it generates optimization-specific defaults using $GetInitConfig(B,g)[n]$. By default, *GetInitConfig* returns the default configuration generated by the compiler's heuristic, but one can override it to return a random configuration or others for more exploration. The semantics of *InferConfig* is shown in the `infer-config` rule in Fig. 12 in the appendix.

$ConfStore[opt_{id}][fp(n)]$ stores the best configuration for optimization $opt_{id}$ for node $n$ with fingerprint $fp(n)$. Two nodes are considered to be the same or *unchanged* if they have the same fingerprint. This approach of reusing configurations of *unchanged* nodes assumes that the configurations of these nodes still work well in a new context, provided we can define the *unchanged* relation appropriately. A simple strategy is to compare only the nodes' attributes (e.g., operator type,

TABLE I: Implementations of search-strategy-specific functions in **XTAT-M**. Column $|\mathbf{C}|$ describes the number of candidates returned by the function **SelectCandidates**. Parameters $M$ and $K$ in EVO, MBO, and RL are configurable by users.

| **Strategy** | Function **GenerateCandidates** | Function **SelectCandidates** | $\lvert\mathbf{C}\rvert$ |
|---|---|---|---|
| Exhaustive | Return the next candidate (not visited before). | N/A | N/A |
| SA | Mutate configurations of some nodes in the current candidate. | Return either the old or new candidate depending on their costs and the annealing temperature. | 1 |
| EVO | Generate $M$ new candidates by crossing over parent candidates and mutate some nodes' configurations. | Return the $K$ (where $K > M$) most recent (unique) candidates, using costs for tie breaking. | $K$ |
| MBO | Generate $M$ new candidates from the model's latent state. | N/A | N/A |
| RL | Generate $M$ new candidates from the learned policy at the current state (i.e., the current candidate). | Return the best candidate so far. | 1 |

input/output shapes, layouts etc). Fig. 2 shows how this strategy might infer the fusion configuration on the right from the left one. The `add` is identified as unchanged, so `add` will be fused with both `reshape` and `conv`, but `add` and `conv` are not fused together in the left scenario.

To define *unchanged* more conservatively, we consider a context around a node by computing a node's fingerprint $fp(n)$ from both its attributes and its neighborhood within $k$ hops. If $k$ is set to one, only `reshape` remains unchanged between the middle left and the middle right graphs in Fig. 2, so we will not reuse the configuration for `add`. In practice, $k$ is set to five for the fusion pass (from hyperparameter tuning). For a node-level optimization where an optimal configuration for a node is independent from the rest of the graph, we can simply set $k = 0$ to ignore the neighborhood.

*4)* **Evaluate***:* This step evaluates the performance of a candidate. If the new best candidate is found, we update the global *ConfStore* (line 33). The update overwrites a previous configuration for a node with respect to $fp(n)$ if its configuration changes, while retaining configurations for nodes that do not belong to the candidate, as formalized in the `update-store` rule in Fig. 12 in the appendix. It is important to retain configurations for nodes that do not belong to the candidate because the current best candidate may not be optimal and future search steps taken by the search strategy may favor graphs that contain these nodes.

*5)* **SelectCandidates***:* The last action is selecting a set of generated candidates to pass to the next round.

To support tuning an additional pass, one can simply add the optimization to the *Opts* list, configure the search schedule through *SelectOpt*, set the neighborhood size for *fp* for that specific optimization (which can be set through hyperparameter tuning), and modify the compiler pass to apply the optimization (*ApplyOpt*) according to a given configuration.

Note that fixing illegal combinations of configurations is the key to our joint autotuning methodology. Simply dropping illegal combinations is insufficient for joint autotuning because changes applied to one pass configuration without the fix are often illegal. This means one has to fall back to using default configurations for later passes, resulting in tuning one pass after another, but not a joint optimization.

### C. Search Strategies

Below, we describe a wide spectrum of search strategies that we have evaluated. Table I summarizes how the following search strategies implement routines *GenerateCandidate* and *SelectCandidate*.

*1) Exhaustive search within a node:* The exhaustive strategy generates a new candidate by selecting the next option for the current node's configurations. When all configurations have been explored for the current node, it moves on to the next node. This strategy can be used for node (kernel) level optimizations, such as tile size selection.

*2) Simulated annealing (SA):* SA mutates a candidate and probabilistically accepts a new one based on annealing temperature that decreases over time. We have found the following types of mutation operators to be effective:

- Single-node: mutates the configuration of one randomly selected node.
- Multi-node: mutates configurations of all nodes independently with probability $p$, where $p$ decreases as a function of temperature in simulated annealing.
- Group: mutate configurations of a randomly-selected set of nodes grouped by a certain criterion.

We have empirically observed that a multi-node mutation works well for fusion autotuning and a combination of group and single-node mutations works best for layout autotuning.

*3) Regularized evolution (EVO):* EVO performs evolutionary search using a population of $K$ individuals. Each new individual is generated by selecting two parents from the population using binary tournament selection, recombining them with some crossover rate $\gamma$, and mutating the recombined individual with some probability $\mu$. Following [16], to promote exploration, the population is updated by replacing the oldest individuals by newly evaluated individuals. In our experiment, we use the parameters $K = 100$, $\gamma = 0.2$, and $\mu = 0.01$. We also considered population-based evolutionary optimization using P3BO [17] but did not find clear performance improvement over a single evolution optimizer.

*4) Model-based optimization (MBO):* MBO performs model-based optimization with automatic model selection [18]. At each optimization round, a set of candidate regression models are fit to the data acquired thus far, and their hyperparameters are optimized by randomized search and five fold cross-validation. Models with a cross-validation score $\geq 0.4$ are ensembled to define an acquisition function. This function is then optimized by regularized evolution to generate a new batch of samples. Candidate models include ridge regression, random forests, gradient boosting, and neural networks.

*5) Deep reinforcement learning (RL):* A deep RL method (GO) [19] is designed specifically for ML compiler's graph optimizations. GO uses a graph neural network (GNN) to create node embeddings and segmented recurrent attention layers to capture long-range dependencies that appear in a computation graph. The policy network transforms the graph representation into optimization decisions with soft attention. The learning objective is optimized using Proximal Policy Optimization (PPO) [20]. Instead of using conventional RL algorithms, we leverage the existing compiler heuristics by initializing the search with the default heuristic solution, and modify the RL sampling stage to encourage more exploitation. Concretely, we modify the state transition function such that RL only traverses to a state with a higher reward than the default configuration's: $S(t) = S(t-1)$ if $R(t) < R(0)$, where state $S(t)$ is the embedding of the fusion configuration at step $t$, and $R(0)$ is the reward for the default configuration.

### D. Limitations

There are some important optimizations, such as graph rewrites, that do not naturally lend to a node-configuration-based representation. In such cases, we believe that **XTAT-M** can still be used as a subroutine, where the outer loop applies rewrites, and for each candidate change, we use **XTAT-M** to optimize other optimization decisions. Notice that **XTAT-M** does not have to start the search from scratch every iteration because the global *ConfStore* persists across all iterations. Our formulation also does not support tuning an unbounded number of passes, for example, passes that run until a fixpoint.

## III. XTAT: IMPLEMENTATION IN XLA

We develop **XTAT** based on **XTAT-M** to tune tensor layouts, operator fusion decisions, tile sizes, and code generation parameters in XLA. **XTAT**'s target accelerator is TPUs [15], energy-efficient ML accelerators.

### A. Background on XLA

XLA is a ML compiler capable of generating code for various hardware targets [11]. Its workflow can be split into three stages. In the graph-level passes, XLA uses a graph-based intermediate representation named High-Level Operation (HLO) to describe a tensor computation graph. Various compiler passes transform HLO so that the output graph is algebraically optimized, and is ready to be mapped onto the target hardware. At the end of the graph-level phase, the optimized HLO graph consists of nodes that represent fusions of multiple operations. We refer to a fused node as a *kernel*. Next is the hardware lowering phase; the compiler converts each individual kernel into instructions that can be executed on the target hardware. In the third phase, low-level architecture-specific optimizations are applied, such as VLIW instruction scheduling, peephole optimizations, and register allocation.

### B. XTAT's Optimization-Specific Search Formulations

We instantiate **XTAT-M** to tune the most performance-critical optimization passes in XLA including layout assignment, operator fusion, and tile size selection, as well as compiler's flags used during the lowering phase. Layout assignment and operator fusion effect graph-level changes and hence are optimized globally. Tile size and flags selection are kernel-level optimizations and hence are optimized for each kernel independently. Apart from these optimizations, **XTAT** employs the other heuristic-based passes used in XLA. Our choice of passes to tune was influenced by expert XLA developers who suggested these optimizations as ones that significantly affect performance of most ML models. Note that layout and/or fusion decisions heavily influence other optimizations such as cross-replica sharding and rematerialization. While rematerialization and operator scheduling are important for making a program fit in available memory, they do not typically reduce program's execution time.

TABLE II: Instantiation of **XTAT-M** for specific optimization passes tuned by **XTAT**.

| Optimization | Applicable node | Node's $config$ |
|---|---|---|
| layout assignment | conv. & reshape | input and output layouts |
| operator fusion | fusible node | fusion control bit |
| tile size | kernel w/ tiling | input and output tile sizes |
| flags | non-comm kernel | lowering-phase flag values |

For each optimization pass we tune, we aim to define a search space that is: (1) *expressive*, i.e., contain diverse candidates that lead to optimal outcomes; (2) *compact*, i.e., include only valid candidates, few candidates that have the same behavior, and not too many more bad candidates than good ones. We specialize **XTAT-M**'s generic formulation to specific optimizations as summarized in Table II. The rest of this section details the search formulation for each optimization. We also describe existing XLA heuristics and alternative search formulations, which are the baselines in our evaluations.

**Layout Assignment:** The layout assignment pass chooses the physical layouts of the input and output tensors of each node to satisfy constraints from the user's program, the compiler backend, and the underlying hardware, while minimizing the program's execution time. An example layout constraint for convolution on TPUs is that the input and output must have input feature, output feature, or batch dimensions as their most minor dimensions. Figure 2 displays the valid input layouts of `conv` in blue. Layout $\{d_0, d_1, ...\}$ represents minor to major dimensions, where elements in the most minor dimension are physically consecutive. If an edge connects an output to an input with a different layout, the compiler inserts a *copy* (transpose) operator to convert the layout. In Fig. 2 (left), the compiler assigns layout $\{1,0,2\}$ to the output of `add` but $\{0,1,2\}$ to the first input of `conv`, causing a layout mismatch, and the insertion of a copy operator. The compiler must trade-off between selecting the best layouts for each specific operator and the overhead from copy operators.

*Compiler's heuristic:* XLA performs layout assignment in multiple rounds. Initially, the pass includes only constraints from the program's inputs and outputs. In each round, the *layout propagation* algorithm propagates layouts from the constrained operators to others through element-wise, pad, and slice operators. This optimistic propagation may cause layout

constraint violations at some operators. After each round, the pass uses various heuristics — ranging from a rough cost model to hard-coded decisions depending on the operators — to rectify the violations. This process continues until all constraints are satisfied.

*Naïve search formulation:* We could define the search space to cover all permutations of the dimensions of input and output tensor of every node. However, this leads to an extremely large search space with many invalid and inefficient candidates.

*Our search formulation:* We define the search to configure only the most layout-performance-critical nodes, which are convolution[4] and reshape operations because they are common operations and have the most constrained implementations for TPUs. The search queries the compiler for valid input-output layout combinations for these nodes. Once layouts of reshape and convolution nodes are assigned, we leverage the existing layout propagation algorithm to propagate layouts from these nodes to others. This search space contains only valid and relatively efficient candidates.

**Operator Fusion:** When operators are fused, intermediate tensors can be used by the consuming operator directly without saving them to the slow main memory. Fusion also reduces kernel launch overheads. While most ML compilers [1], [3], [4], [6], [10], [21] support fusions of a limited set of operations, XLA can fuse more complex operations (e.g., gather, scatter, reshape, reduce, etc.) resulting in more optimization opportunities as well as more decisions to make. Typically, when an operator is fused into multiple consumers, it must be duplicated (recomputed) in each consumer because consumers can have different iteration spaces (loop structures). This is illustrated in Fig. 2, where `add` is fused into both `reshape` and `copy`. The compiler must trade between recomputation and reduced memory communication.

*Compiler's heuristic:* The XLA fusion algorithm maintains a priority queue of all nodes in the graph. A cost model computes the priority of a node as the benefit of fusing that node into its consumers. The algorithm iteratively fuses a node with the highest priority value until all nodes have negative priority values. The fused node and its consumers are removed from the queue, newly formed fused nodes are inserted, and the priority values of affected nodes are updated.

*Our search formulation:* We assign a boolean value to each fusible node to control whether it is fused with its consumers.

*Alternative formulation (edge):* We can assign a control bit per edge (instead of per node) such that a parent node is fused with only a consumer whose edge is marked. However, when a parent node $u$ is fused with a consumer node $v$ but not a consumer node $w$, the compiler saves the intermediate output of $u$ in the slow memory; as a result, we do not get the full benefit of fusion. While this formulation is more expressive than the per-node formulation, we believe this additional coverage is unnecessary.

*Alternative formulation (node priority):* Another approach is to assign a node a priority value instead of controlling its fusion behavior explicitly, similar to GO [19]. We can adapt the existing heuristic to use these priorities as its initial node priorities, instead of using the cost model. However, the heuristic dynamically adjusts the priorities when nodes are fused; a configuration cannot define these values in advance because we do not know how many adjustments will occur. A workaround is to set the priority of the newly fused node to the sum of its constituents' priorities. This search space is larger than ours, but covers the same set of behaviors.

*Alternative formulation (flags):* Another approach is to tune the compiler's fusion-related flags. We try tuning flags that (1) limit fusions of inputs into convolutions, (2) limit fusions of outputs into convolutions, and (3) parameterize the fusion cost model. This search space is small, and does not grow as the number of nodes increases. However, this approach provides less control than the other formulations.

**Tile-Size Selection:** The goal is to pick an optimal tile size for each kernel's input and output tensors such that they fit in scratchpad memory.

*Compiler's heuristic:* XLA enumerates all possible tile sizes and chooses the optimal one according to hand-written analytical cost models.

*Search formulation:* We query the compiler to get a set of valid tile sizes for each kernel to form the search space. The number of tile sizes we autotune ranges from 2 to 500,000.

**Lowering Flags:** Apart from tile size selection, the lowering phase is also responsible for many performance-critical decisions. Many of these decisions are determined by heuristics that are controlled by flags. Some examples include instruction window size for hoisting load/store instructions, enabling overlaps of input/output DMAs, and scratchpad limit to allocate to an operator.

*Compiler's heuristic:* The compiler developers set these flag values to "magic" numbers that work well on most benchmarks in the regression suite.

*Search formulation:* For an integer flag, we limit the search space to four sensible values. **XTAT** tunes the total of eight integer and boolean flags.

### C. XTAT's Search Schedule

We configure *SelectOpt* in **XTAT-M** to explore different search schedules for tuning our four target optimizations. For scalability, we choose to decouple graph-level optimizations (layout and fusion) from kernel-level optimizations (tile size and flags) for scalability.

*1) Joint layout-fusion autotuning:* Following the formulation presented in Section II-B, we configure *SelectOpt* to alternate between layout and fusion every $S$ steps, where $S$ is set to five via hyperparameter tuning. Mutating both layout and fusion configurations in one search step performs slightly worse than the alternating strategy.

*2) Joint tile-size-flags autotuning:* The tile size selection pass does not change the graph, merely annotating nodes with tile sizes, so the joint tile size and flags optimization is less complex. Further, the joint search space of tile sizes and flags

[4]All tensor contraction operations (e.g., multiplication and einsum) are converted into convolutions before layout assignment in XLA.

is small enough for enumeration. Hence, we performed an exhaustive search within each node on the cross-product of tile size and flags search spaces until the time limit is reached. We also explore tuning tile size and then flags sequentially. We configure *SelectOpt* to tune tile size until exhaustion before switching to flags. The result in our evaluation (Section IV-B) reveals that tuning them in sequence is superior to tuning them jointly because we can only explore a small fraction of the entire joint search space before timeout. Thus, we configure **XTAT** to tune tile size and flags separately.

### D. Execution Time Measurement

*1) Measurement on hardware:*

*a) Kernel execution time:* Most kernels' runtimes do not depend on input data, so they can be measured reliably using random inputs; this is an approximation in a few cases such as gather and scatter kernels. We do not tune tile sizes for kernels that communicate across multiple devices; their optimal choices are trivial since there is no complex tradeoff between communication and compute. We parallelize runtime measurements across machines, and use multi-threading to further parallelize compilation.

*b) Graph execution time:* We measure the runtime of a graph by summing its kernels' runtimes. We ignore kernels that communicate across multiple devices, as layout and fusion decisions rarely affect these kernels' runtimes, enabling tuning a multi-device model on a single device. We ignore loops and conditionals, considering only the kernels in their bodies. We found these approximations to be accurate enough for ranking in autotuning. One can improve the fidelity of the estimates using program traces to weight kernels' runtimes by their execution counts. Finally, we ignore interactions between kernels as they do not execute concurrently on a TPU, leaving only negligible effects like code prefetching and device temperature. We use caching to avoid measuring identical kernels repeatedly since small configuration changes leave most kernels in a graph unchanged.

*2) Learned cost model:* To avoid expensive candidate evaluations (compiling and executing on real hardware), we train a learned cost model to predict execution time, following the TPU learned performance model [22]. The model uses a GNN to encode operation features and the structure of a kernel subgraph. Node embeddings outputted from the GNN are summarized using a simple reduction function to generate a kernel embedding, which is fed into a feed-forward layer to produce a final prediction. We train one model for all graph-level optimization tasks, one for tile-size selection, and one for flags selection. The GNN is shared between tile-size and flags models. For graph-level tasks, the model predicts the absolute runtime of a kernel based on the default tile size and flag values, selected by the compiler's heuristics. The entire program's runtime is the summation of the kernels' predictions. For kernel-level optimizations, the model predicts relative runtimes of different configurations of a given kernel.

To ensure accuracy of the model for newer types of workloads and to keep up with compiler's changes, we finetune the model periodically; we freeze the GNN layers and retrain only the feed-forward head. Since the model can never be 100% accurate, we execute the top $k$ configurations on real hardware and pick the best.

## IV. Evaluations

Our benchmarks comprise 150 machine learning models from the XLA TPU regression suite representing both production and research usage at Google. They include both inference and training graphs, with sizes ranging from 100 to 56,000 nodes. We report execution time speedup of each benchmark over its *default* execution time when compiled using the XLA compiler's heuristics, which have been continuously improved by a large team of experts since 2016 for production usage. Execution time is measured on TPU v3 [15].

### A. End-to-End Autotuning

*1) Tuning on real hardware:* According to the results from Section IV-B, we set our end-to-end autotuning schedule as follows: first, jointly tune layout and fusion via SA, followed by exhaustive search for tile-size autotuning, and ending with exhaustive search for flags tuning. The joint layout-fusion optimization was tuned for two hours on 10 TPU machines (each with a host and an accelerator) or at most 10,000 candidate evaluations on each machine for each model. The exhaustive search for tile-size and flags autotuning is sharded to run on 10 machines with one hour timeout for each task. The results in this section show what **XTAT** can achieve given a moderate amount of resources and time; techniques to reduce tuning time and resources will be evaluated later.

Figure 4 reports the real execution time speedup using representative program inputs and complete control flow. The figure breaks down the speedup contributions from autotuning different passes. We display 43 models that achieve performance improvement of 5% or more. Overall, **XTAT** offers 5% speedup on average over the production compiler across 150 models, where tile size and fusion autotuning contributes the most to the total speedup, followed by layout, and flags autotuning. We see a huge speedup on AVSpeech inference (2.4×) and Translate Transformer inference (1.5×). Nine models also exhibit more than 15% improvement. Note that the XLA compiler has been heavily optimized for most of these models; nevertheless, the autotuner still provides substantial speedup: 14% on MLPerf DLRM training (recommendation model), 13% on MLPerf Mask RCNN training, 11% on MLPerf SSD training, and 7% on a few ResNet training models.

*2) Tuning with learned cost model:* Next, we evaluate the learned cost model in terms of its efficacy on reducing tuning time. We generated training data for the learned cost model from compiling and running the 150 ML models using random layout, fusion, tile size, and flag configurations. Eight benchmarks were held out for testing.

On the hold-out benchmarks, we compared: (1) using only real hardware for evaluations and (2) using the learned cost model to select top $k$ configurations to evaluate on real hardware. Both settings employed the same autotuning schedule

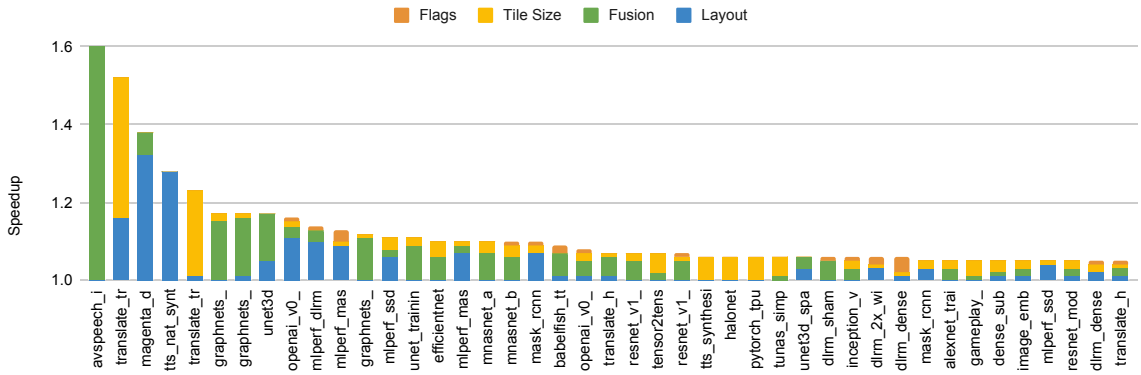


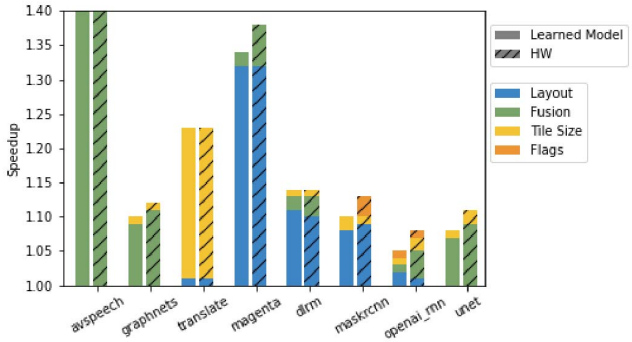

Fig. 4: End-to-end model speedups from autotuning 150 ML models. The figure shows only models that achieve 5% or more improvement. The first bar (AVSpeech inference) has 2.36× speedup (2.34× from fusion and 2% from tile size). There is no performance regression on the rest of the benchmarks; in the worst case, the model's execution time remains the same.

(joint layout fusion, followed by tile size, and ended with flags tuning). For (1), we used the same autotuning setup as in Section IV-A. For (2), we also used 10 TPU machines to perform real evaluations on hardware but used 10 CPU machines at the beginning to select top layout and fusion configurations in order to cut down time using TPU machines, which are in high demand. In particular, we first ran the joint fusion-layout autotuning on 10 CPU machines for two hours or at most 10,000 steps using the learned cost model, selected the top ($k = 10$) candidates from each machine, and ran them on 10 TPU machines. Then, we used the learned cost model to pick the best ($k = 5$) tile sizes and flags from each shard to execute on a TPU. Here, we ran both cost model evaluations and real hardware evaluations on TPU machines as time spent on cost model evaluations are negligible.

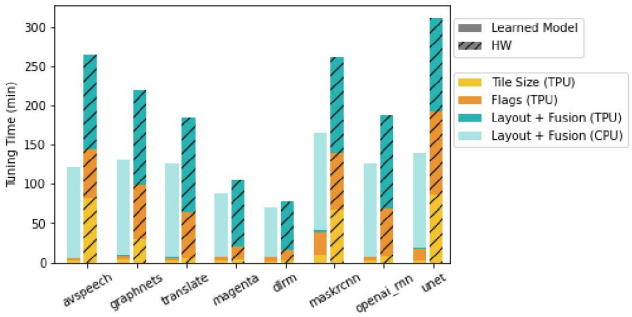
Figure 5a reports the execution time speedup using representative program inputs and complete control flow, and Fig. 5b reports the tuning time. Overall, using the learned cost model achieves almost the same speedup as using real hardware alone, while drastically reducing tuning time. On average, it reduces the tuning time of tile sizes and flags by 6× and 8× respectively. Notice that although it does not help reduce the total layout-fusion tuning time significantly, it reduces the tuning time on TPU machines by 240×.

This improved tuning time makes it possible to run the autotuner at scale. We have deployed the tile size and flags autotuning to automatically optimize the top workloads in Google's fleet daily. The learned cost model enabled **XTAT** to tune 20× more kernels per day. In the past 10 days, **XTAT** has sped up the most heavily-executed kernels by 3.5% and 1.6% on average from tile size and flags tuning respectively.

*3) Performance analysis:* The speedup from autotuning generally comes from higher FLOPs utilization. Good tile sizes balance computation and memory bandwidth utilization. **XTAT** doubles the tile sizes of many copy operations in the Translate Transformer benchmark, increasing the memory bandwidth utilization of these operations from 45% to 70%, resulting in an overall speedup of 1.3×.




(a) Execution time speedup on benchmarks in the hold-out set when using the learned cost model vs. using hardware evaluations alone. Higher is better. Using the learned cost model achieves the same speedup of 2.4× on AvSpeech as using the real hardware.




(b) Tuning time in minutes (maximum across 10 machines). Lower is better. Light blue is tuning time on CPU machines, which are cheaper and more abundant resources.

Fig. 5: End-to-end autotuning: the learned model drastically reduces tuning time while achieving almost the same speedup as using real hardware alone.

Fusion autotuning generates better fused operators, enabling more efficient tile sizes for corresponding operations. E.g., in AVSpeech inference, fusion tuning moves one reshape operation out of a convolution fusion operation, enabling it to use a tile size that is 12.5× larger than before. As a result, this convolution fusion is sped up by 12.5× with over 2×

FLOPs utilization and over 3× memory bandwidth utilization, resulting in an overall speedup of 2.3× for that benchmark.

Good layout assignment generally reduces the overall data-formatting overheads. However, autotuning Translate Transformer reveals an intricate relationship between different optimizations. Layout tuning adds a copy operation, removing a bitcast from a long-running fused operation. As a result, the FLOPs and memory bandwidth utilization of the fusion operation increases by over 2× and over 1.5× respectively, yielding an overall speedup of 1.16× for that benchmark.

*4) Result discussion:* Our performance improvements may at first seem modest. However, our benchmarks are representative of large, real world deployments, so even small improvements translate to significant resource savings. Further, the baseline compiler we compare against is actively being tuned against the same benchmarks (and sometimes in the light of our own tuning results!), making it a constant battle to stay ahead. When **XTAT** was applied to top workloads from the TPU fleet (beyond the benchmark suite), we saw higher speedups (approximately 15% on average).

Prior works [1]–[7], [10] evaluate only on inference models, and show impressive performance improvements. However, their baselines are library kernels (from CuDNN, MKL, etc) that are generally optimized for training workloads. In contrast, our baseline is the XLA TPU compiler, which is optimized for both inference and training workloads.

### B. Search Formulations

To evaluate the search formulations presented in Section III-B, we used SA as the search strategy, measured execution time using real hardware, and selected 10 benchmarks that showed significant speedup from autotuning for a particular optimization problem. We ran 10 replicas of SA with random seeds in parallel and reported the best configuration found among all the replicas. On each replica, we ran the search for 10,000 steps with a two-hour time limit. We experimented with two modes: starting the search from a default configuration (from the compiler's heuristic) and starting from random configurations (different replicas starting from different random configurations). This subsection reports speedup with respect to execution time measured as described in Section III-D1 (ignoring control flow and using randomly generated data).

*1) How effective is our layout assignment formulation?:* As shown in Fig. 6, when starting the search from a default configuration, our proposed search formulation drastically outperforms the naïve formulation, as we hypothesized. When starting from random configurations, the search using the naïve formulation is unable to find any valid layout configuration for any of the benchmarks. In contrast, the search using the proposed formulation is able to find valid configurations for all benchmarks with an average speedup of 8.6%, which is almost the same as the average speedup of 9% when starting from the default configuration.

*2) How effective is our operator fusion search formulation?:* Figure 7 shows the results when starting the search from a default configuration. The per-node control bit formulation

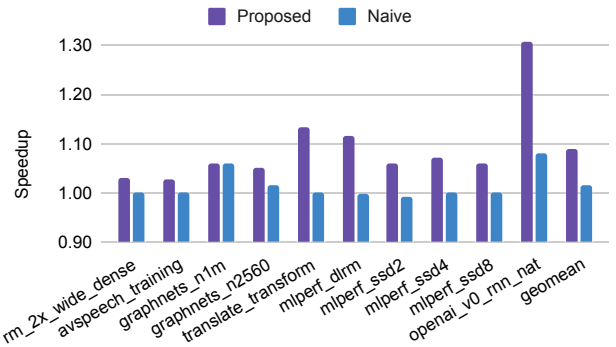



Fig. 6: Layout autotuning: execution time speedup using different search spaces (starting from a default configuration). Our proposed formulation outperforms the naïve formulation.

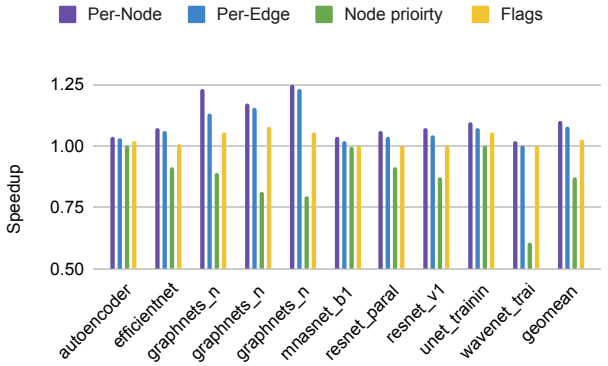



Fig. 7: Fusion autotuning: execution time speedup with different search spaces (starting from a default configuration). The per-node search space offers the most speedup.

is slightly (2 percentage points) better than the per-edge alternative, as hypothesized. The difference is more pronounced when starting from random configurations (not shown in the figure), where the per-node formulation's average speedup is 8 percentage points higher than that of the per-edge formulation.

Another search formulation uses node priority, and is significantly worse than the control bit formulations. Even when starting from default, the best candidates found in this search space are worse than the default; it is impossible to faithfully reconstruct the default configuration in this formulation as priority values change dynamically during the fusion pass.

The last alternative search formulation is tuning flags. Because the entire search space has only 52 candidates, we performed an exhaustive search here. As we hypothesized, this formulation is worse than the control bit formulations due to its limited expressivity, offering tiny speedup on all benchmarks.

*3) How much benefit does joint layout and fusion autotuning offer?:* To investigate the potential of joint autotuning of multiple passes, we compared two strategies: tuning layout and then fusion sequentially (each for two hours), and tuning them jointly (for four hours). Figure 8 shows the results for benchmarks where joint autotuning improves over tuning layout and then fusion. For the remaining six benchmarks, sequential and joint tuning offer the same speedup. According

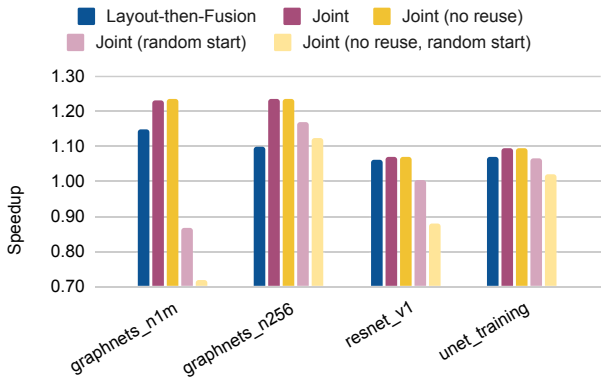



Fig. 8: Autotuning layout then fusion vs. autotuning them jointly (with/without configuration reuse). The first three in each cluster start from a default configuration; the last two start from random configurations. 'No reuse' does not employ the configuration reuse mechanism (the global *ConfStore*). Joint tuning with the reuse mechanism offers the most speedup.

to Fig. 8, joint tuning is better than sequential tuning. This is because a better layout configuration may disable an even better fusion configuration.

In addition to evaluating the achievable speedups, we also compared tuning time between sequential tuning and joint tuning. We measured the time taken to reach within 0.1 percentage point of the highest speedup. On average, joint tuning took 2.7× less time than sequential tuning to reach the highest speedup across 10 benchmarks. This is likely because, in some benchmarks, the sequential search schedule wasted time optimizing layouts (in the first half of the search) when there was not much room for improvement.

This experiment also suggests that our simple search technique is still effective in an extremely large search space, thanks to our strategy of leveraging compiler heuristics.

*4) How important is the configuration reuse mechanism for joint autotuning?:* The configuration reuse mechanism (Section II-B3) enables the autotuner to reuse parts of the best fusion configurations found so far when a layout configuration changes. Without this mechanism, the autotuner must reset to a default or random fusion configuration when a new graph is generated by a new layout configuration. According to Fig. 8, when the search starts from a default configuration, there is no difference between using and not using the reuse mechanism. However, if the search starts from a random configuration and *GetInitConfig* in Fig. 3 returns a random configuration, we observe significant benefit from employing the reuse mechanism: up to 15% execution time improvement.

In terms of time to reach the highest speedup (within 0.1 percentage point) when starting from a default configuration, reuse or no reuse mechanism took similar time on all but two benchmarks. On MLPerf DLRM and GraphNets n4k, the reuse mechanism reached the highest speedup 4.5× and 1.8× faster than no reuse mechanism did. Hence, the reuse mechanism is useful for reducing tuning time for some models and essential if one does not have good default configurations.

*5) How much benefit does joint tile size and flags autotuning offer?:* Similar to joint layout-fusion, we compared tuning tile size and then flags, and tuning them jointly. Since tile size and flags search space is small, we enumerated all candidates and evaluated them in a random order up to a time limit (10 minutes). For the sequential schedule, we tuned tile size and then flags on every kernel. For the joint strategy, the time limit per kernel is 20 minutes. Tuning tile size then flags offers 6.6% average speedup, while tuning them jointly offers only 1.9% average speedup. This is because while the individual tile size and flags search spaces are small, their joint search space is too large; as a result, traversing the search space in a random order fails to discover good candidates by the time limit.

### C. Search Strategies

To show that **XTAT-M** allows various search strategies without changing the search formulation, we ran the search techniques from Section II-C (excluding exhaustive search) on fusion autotuning. This also evaluates the search strategies at finding good candidates in a large search space. We measured execution time on real hardware, using 1,000 or 10,000 candidate evaluations, depending on the benchmark's time for one evaluation. We ran each search strategy on 10 replicas starting from the default configuration. We report speedup using randomly generated data and ignoring control flow.

Figure 9 shows the average speedup over the default configuration. Fig. 10 in the Appendix shows the optimization trajectories for one example benchmark. We find rather simple evolution-based techniques (SA and EVO) reliably find better solutions than advanced model-based optimization techniques (MBO and RL). We hypothesize that the high-dimensional search spaces ($2^{960}$ for AutoEncoder; up to $2^{39568}$ for EfficientNet) make it challenging to fit an accurate model without prior knowledge. Furthermore, the evolution-based techniques take less time to propose new candidates, reducing compute costs and overall tuning time. While SA and EVO are comparable for most benchmarks, SA is worse than EVO and RL on GraphNet models. With more samples, however, SA achieves

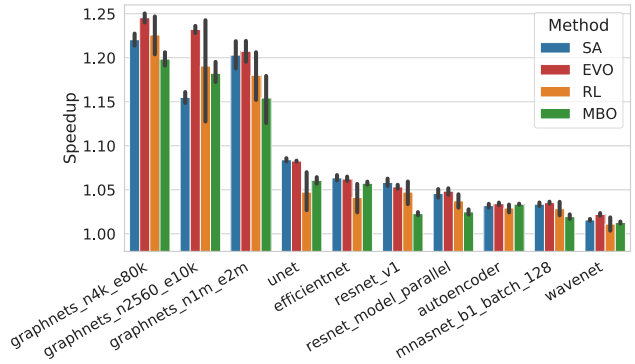



Fig. 9: Fusion autotuning: average execution time speedup using different search strategies (starting from a default configuration) when allowing up to 1,000 or 10,000 samples (depending on benchmarks). Error bars show 95% confidence intervals over ten replicas.

the highest speedup. We conclude that when resources are limited (one or a few machines available for autotuning), EVO finds good configurations faster than other algorithms.

## V. Related Work

### A. Autotuning in ML Compilers

Autotuning has been effective at optimizing code in various domains [23]–[28]. We apply this technique to a multi-pass ML compiler. Recent search-based ML compilers [1]–[8] apply autotuning at the kernel or subgraph level. Template-based approaches [1]–[7] are similar to XLA's lowering algorithm; based on a kernel's subgraph, a loop structure template is used to generate code. Ansor [3] proposed a template-based approach that allows more flexible loop fusions compared to prior work. However, their tiling structure is still fixed, and their fusion capability is more limited than XLA's. Halide [5], [29]–[31] covers a larger space of possible loop implementations compared to template-based approaches, but performed worse than FlexTensor [4] and Ansor [3]. Mind Mappings [32] and AKG [33] focus on operator-level optimizations and code generation for custom hardware accelerators. Unlike **XTAT**, these prior works do not tune layout decisions.

The value learning approach [9] first applied search to loop optimizations for an entire graph at once. However, its search must be applied in a single compilation stage, making it inapplicable to multi-pass production ML compilers, and it does not tune tensor layouts. This approach has been applied to inference graphs with up to 400 nodes, while our approach has been evaluated on both inference and training graphs with up to 500,000 nodes. DeepCuts [34] applies a greedy exploration guided by an analytical cost model to tune graph-level fusion decisions along with some GPU kernel parameters. However, its fusion capability is limited, and it is not obvious how to extend DeepCuts to support layout tuning at the graph level. Graph substitution approaches [10], [35] optimize entire tensor computation graphs, but work in a limited search space reachable via graph rewrites. Without exploding the number of rewrite rules, they cannot represent arbitrary fusions of tensor operations or change layouts of arbitrary tensors. GO [19] optimizes device placement, operator fusion, and operator scheduling decisions for an entire TensorFlow graph. However, it does not consider the effect of multiple compilation passes on the intermediate computation graph, so we cannot adopt their search formulation when dealing with a full compiler stack. Rammer [36] can jointly optimize inter- and intra-operator parallelism, but it does not address other kinds of optimizations, such as layout and fusion, addressed in this paper. Rammer's approach is also inapplicable to TPUs, since TPUs do not support concurrent execution of multiple kernels. Many existing works use heuristics or analytical cost models to tackle a specific graph-level optimization, including operator fusion [21], [37], [38] and layout assignment [39].

### B. Joint Autotuning Capability

In a multi-pass compiler, changing a configuration for one optimization pass changes the search spaces for subsequent optimizations, as explained in Section II-A. Generic autotuning frameworks, such as OpenTuner [28], require the entire search space to be specified upfront, thus disallowing search spaces that change dynamically. Our joint autotuning methodology is a generic method to formulate the interactions between multi-pass compiler optimizations. It is applicable to any multi-pass compiler and is not specific to the XLA compiler or **XTAT**. Therefore, it can be implemented in existing autotuning frameworks (if they can be extended to support dynamic search spaces), enabling them to perform joint optimizations.

The approach used in most search-based ML compilers [1]–[9], inspired by Halide [29], supports joint optimizations by default. However, when a search space is large, it requires a cost model that can accurately evaluate a (partial) schedule or configuration, which is extremely challenging to create for an entire tensor graph. Therefore, most of these compilers optimize only one small subgraph at a time. While the value learning approach [9] can optimize an entire inference graph, it requires carefully-crafted feature engineering, which is not easily applied to new hardware. Furthermore, one cannot easily apply a Halide-like approach to multi-pass production compilers without completely reimplementing them. Our goal is to provide autotuning capability to such multi-pass compilers.

### C. Learned Cost Model

Similar to prior works [2], [5], [7], [9], [22], [40]–[42], we use a learned cost model to accelerate autotuning. We additionally propose a pretraining-finetuning method to reduce training time while keeping the model up-to-date.

## VI. Conclusion

We proposed a search methodology that enables autotuning various graph-level and subgraph-level optimizations at different compilation stages for production ML compilers. Our search formulation allows solving optimization problems jointly or one-at-a-time, as well as using a spectrum of search strategies to solve them. Based on this, we developed an autotuner to tune the key optimization passes in the XLA TPU compiler. The execution time speedups on 150 ML models from Google's workloads found by the autotuner averaged 5%, with many cases over 15%, and one improving by a factor of 2.4. We substantially reduced tuning time via a learned cost model.

**XTAT** has been used in different ways to optimize production models. Compiler developers have used **XTAT** to detect opportunities to improve the baseline heuristics multiple times. For instance, fusion autotuning led to a fix in the heuristics, yielding 18% latency reduction on a model served in production. Such improvements are invisible in our experiments as they have already been incorporated into the compiler. Besides being used by compiler developers, **XTAT** has been deployed to automatically tune tile sizes and flags for the most heavily-used production models in Google's fleet everyday.

## Acknowledgment

We thank Albert Cohen, Charith Mendis, Jacques Pienaar, Jason Ansel, and the reviewers for their insightful feedback.

## REFERENCES


[1] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '18. USA: USENIX Association, 2018, p. 579–594.

[2] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to Optimize Tensor Programs," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NeurIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 3393–3404.

[3] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating High-Performance Tensor Programs for Deep Learning," in *14th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '20. USENIX Association, Nov. 2020, pp. 863–879.

[4] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 859–873.

[5] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley, "Learning to Optimize Halide with Tree Search and Random Programs," *ACM Trans. Graph.*, vol. 38, no. 4, Jul. 2019.

[6] B. H. Ahn, P. Pilligundla, A. Yazdanbakhsh, and H. Esmaeilzadeh, "Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation," in *International Conference on Learning Representations*, 2020.

[7] M. Li, M. Zhang, C. Wang, and M. Li, "AdaTune: Adaptive Tensor Program Compilation Made Efficient," in *34th Conference on Neural Information Processing Systems*, ser. NeurIPS'20, 2020.

[8] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[9] B. Steiner, C. Cummins, H. He, and H. Leather, "Value Learning for Throughput Optimization of Deep Learning Workloads," in *Proceedings of MLSys Conference*, 2021.

[10] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 47–62.

[11] TensorFlow, "XLA: Optimizing Compiler for TensorFlow," https://www.tensorflow.org/xla, [Online; accessed 19-September-2019].

[12] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, J. Montgomery, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, M. Smelyanskiy, and M. Wang, "Glow: Graph Lowering Compiler Techniques for Neural Networks," *arXiv preprint arXiv:1805.00907*, 2019.

[13] Y. Xu, H. Lee, D. Chen, H. Choi, B. A. Hechtman, and S. Wang, "Automatic cross-replica sharding of weight update in data-parallel training," *CoRR*, vol. abs/2004.13336, 2020.

[14] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17, 2017.

[15] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A Domain-Specific Supercomputer for Training Deep Neural Networks," *Commun. ACM*, vol. 63, no. 7, p. 67–78, Jun. 2020.

[16] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the aaai conference on artificial intelligence*, vol. 33, 2019, pp. 4780–4789.

[17] C. Angermueller, D. Belanger, A. Gane, Z. Mariet, D. Dohan, K. Murphy, L. Colwell, and D. Sculley, "Population-Based Black-Box Optimization for Biological Sequence Design," *arXiv preprint arXiv:2006.03227*, 2020.

[18] C. Angermueller, D. Dohan, D. Belanger, R. Deshpande, K. Murphy, and L. Colwell, "Model-based reinforcement learning for biological sequence design," in *International Conference on Learning Representations*, 2019.

[19] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. Ma, Q. Xu, H. Liu, P. Phothilimtha, S. Wang, A. Goldie, A. Mirhoseini, and J. Laudon, "Transferable Graph Optimizers for ML Compilers," in *Proceedings of International Conference on Neural Information Processing Systems*, ser. NeurIPS'20, 2020.

[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[21] J. Roesch, S. Lyubomirsky, M. Kirisame, L. Weber, J. Pollock, L. Vega, Z. Jiang, T. Chen, T. Moreau, and Z. Tatlock, "Relay: A High-Level Compiler for Deep Learning," *arXiv preprint arXiv:1904.08368*, 2019.

[22] S. J. Kaufman, P. M. Phothilimthana, Y. Zhou, C. Mendis, S. Roy, A. Sabne, and M. Burrows, "A Learned Performance Model for Tensor Processing Units," in *Proceedings of Machine Learning for Systems*, 2021.

[23] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98, 1998.

[24] M. Frigo and S. G. Johnson, "FFTW: an adaptive software architecture for the FFT," in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, 1998.

[25] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, and M. O'Boyle, "Milepost GCC: Machine Learning Enabled Self-tuning Compiler," *International Journal of Parallel Programming*, vol. 39, pp. 296–327, 2011.

[26] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A Language and Compiler for Algorithmic Choice," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. Association for Computing Machinery, 2009.

[27] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable Performance on Heterogeneous Architectures," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. Association for Computing Machinery, 2013.

[28] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, and S. Amarasinghe, "OpenTuner: An extensible framework for program autotuning," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques*, ser. PACT '14, 2014, pp. 303–315.

[29] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. Association for Computing Machinery, 2013.

[30] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, "Differentiable programming for image processing and deep learning in halide," *ACM Trans. Graph.*, vol. 37, no. 4, Jul. 2018.

[31] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically Scheduling Halide Image Processing Pipelines," *ACM Trans. Graph.*, vol. 35, no. 4, Jul. 2016.

[32] K. Hegde, P.-A. Tsai, S. Huang, V. Chandra, A. Parashar, and C. W. Fletcher, "Mind Mappings: Enabling Efficient Algorithm-Accelerator Mapping Space Search," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 943–958.

[33] J. Zhao, B. Li, W. Nie, Z. Geng, R. Zhang, X. Gao, B. Cheng, C. Wu, Y. Cheng, Z. Li, P. Di, K. Zhang, and X. Jin, "AKG: Automatic Kernel Generation for Neural Processing Units Using Polyhedral Transformations," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1233–1248.

[34] W. Jung, T. T. Dao, and J. Lee, "DeepCuts: A Deep Learning Optimization Framework for Versatile GPU Workloads," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 190–205.

[35] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken, "Optimizing DNN Computation with Relaxed Graph Substitutions," in *Proceedings of MLSys Conference*, 2019.

[36] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, "Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks," in *14th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '20. USENIX Association, Nov. 2020, pp. 881–897.

[37] Nvidia, "Nvidia TensorRT: Programmable Inference Accelerator," 2017.

[38] Z. Zheng, P. Zhao, G. Long, F. Zhu, K. Zhu, W. Zhao, L. Diao, J. Yang, and W. Lin, "FusionStitching: Boosting Memory Intensive Computations for Deep Learning Workloads," *arXiv preprint arXiv:2009.10924*, 2020.

[39] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing CNN Model Inference on CPUs," in *2019 USENIX Annual Technical Conference)*, ser. ATC '19. Renton, WA: USENIX Association, Jul. 2019, pp. 1025–1040.

[40] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast Compiler Optimisation Evaluation Using Code-feature Based Performance Prediction," in *Proceedings of the 4th International Conference on Computing Frontiers*, ser. CF '07, 2007.

[41] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc," in *Proceedings of MLSys Conference*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 187–198.

[42] R. Baghdadi, M. Merouani, M.-H. Leghettas, K. Abdous, T. Arbaoui, K. Benatchba, and S. Amarasinghe, "A Deep Learning Based Cost Model for Automatic Code Optimization," in *Proceedings of MLSys*, 2021.

## APPENDIX

### A. Additional Evaluation Results on Search Strategies

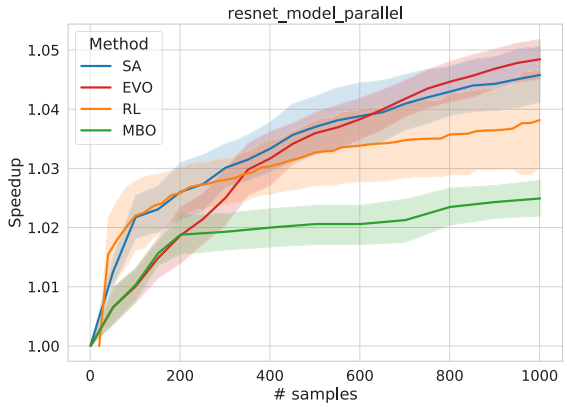



Fig. 10: Fusion autotuning: optimization trajectories for one example benchmark. Shaded areas show 95% confidence intervals over ten replicas.

In the experiment that compares different search strategies in Section IV-C, we collect the speedup trajectories of various search strategies over time on one example benchmark, shown in Fig. 10. We notice that RL has very high variance on most benchmarks compared to the rest, as also evidenced in Fig. 10. This means in the setting where we can run 10 search replicas in parallel and take the best, RL often achieves the most speedup compared to the others at the same number of samples, especially earlier in the search. Future work may consider incorporating prior knowledge, e.g. pretraining the models on a diverse set of graphs, to improve the performance and the consistency of model-based optimizations.

### B. Semantic Rules for XTAT-M

In this section, we describe (a) the implementation of *GenerateCandidates* for simulated annealing, and (b) briefly present the semantics of operations in **XTAT-M** and derive two search steps in a joint layout-fusion tuning for the graph in Fig. 2.

*a) Implementation of GenerateCandidates for simulated annealing:* As described in Section II-C, simulated annealing uses a mutation operator, e.g., single-node, to mutate a given configuration and accept a new one based on annealing temperature. Fig. 11 presents an implementation of *GenerateCandidates* for simulated annealing. This implementation utilizes a routine, *Mutate*, which implements the mutation operator, generating a new configuration from the current configuration.

*b) Semantics of operations in* **XTAT-M***:* Fig. 12 presents the semantics of *InferConfig*, *UpdateStore*, *Mutate* (based on Fig. 11), and *ApplyOpt*. Specifically, to keep the discussion concise, we focus on a single-node mutation-based search strategy applied to tuning *layout assignment* and *operator fusion* optimizations.

- *InferConfig*: This routine updates a configuration *config* for an optimization $opt_{id}$ to be compatible with the graph $g$, an input to the optimization pass. *InferConfig* identifies the nodes in $g$ and *config* that have the same fingerprint $fp(n)$ and carries forward the configurations in *config* for such nodes. For the other nodes in $g$, it pulls the best configurations found so far from the global *ConfStore* if present; otherwise, it generates optimization-specific defaults.
- *UpdateStore*: When a search candidate configuration is evaluated and it performs superior to prior observed evaluations, then the candidate configuration is captured by updating the global *ConfStore*.
- *Mutate*: *Mutate* constructs a new configuration from the current configuration. We present a simplified rule for a single-node mutation, so *Mutate* returns a configuration change for a single node in the graph.
- *ApplyOpt*: As described in Section II-B, *ApplyOpt* transforms an input graph by applying the optimization.

```
1: function GENERATECANDIDATES(opt_id, C)
2:     c ← C[0] // C has one candidate.
3:     c' ← c    // Init configs for all optimizations.
4:     // Mutate only the config for optimization opt_id.
5:     c'.configs[opt_id] ← Mutate(c.configs[opt_id])
6:     return {c'}
7: end function
```

Fig. 11: Implementation of *GenerateCandidates* for simulated annealing.

A derivation of the joint layout-fusion tuning performed on the graph in Fig. 2 is presented in Fig. 13. For conciseness, we do not describe all the tensors involved in the graph. Specifically, we focus on the mutation of the layout configurations for tensors, $t_5$ and $t_6$, in the search step for layout, and focus on the `add` node in the search step for fusion. In each search step, the *Mutate* operation for layout begins with the graph $g$, applies a layout mutation, and infers the *config* for fusion. Additionally, the first step performs the *Mutate* operation for fusion. Both steps apply the inferred fusion configuration and execute the output graph. Based on the evaluation, each search step updates the global store, i.e., *ConfStore*. In the derivation, we note that applying a search candidate for layout in step 1 results in the addition of a `copy` node, while no `copy` is introduced when a different search candidate for layout is explored in step 2.

Fig. 12: Semantics of functions *InferConfig*, *UpdateStore*, *Mutate* (for a single node), and *ApplyOpt* in **XTAT-M**.

$$\text{infer-config}\ \frac{\begin{array}{c} from\_config := \quad \{tuple(fp(n), config[fp(n)])\}, \quad \forall(n).(n \in g.Nodes \wedge fp(n) \in config) \\ from\_store := \quad \{tuple(fp(n), ConfStore[opt_{id}][fp(n)])\}, \quad \forall(n).(n \in g.Nodes \wedge fp(n) \notin config \wedge fp(n) \in ConfStore[opt_{id}]) \\ from\_init := \quad \{tuple(fp(n), GetInitConfig(opt_{id}, g)[n])\}, \quad \forall(n).(n \in g.Nodes \wedge fp(n) \notin config \wedge fp(n) \notin ConfStore[opt_{id}]) \end{array}}{E \vdash config' := InferConfig(config, g, ConfStore[opt_{id}]) \Downarrow E\{config' \mapsto \{from\_config \cup from\_store \cup from\_init\}\}}$$

$$\text{update-store}\ \frac{\begin{array}{c} new\_config := \quad \{tuple(fp(n), configs[opt_{id}][fp(n)])\}, \quad \forall(n).(fp(n) \in configs[opt_{id}]) \\ old\_config := \quad \{tuple(fp(n), ConfStore[opt_{id}][fp(n)])\}, \quad \forall(n).(fp(n) \notin configs[opt_{id}] \wedge fp(n) \in ConfStore[opt_{id}]) \end{array}}{E \vdash UpdateStore(ConfStore, configs) \Downarrow E\{ConfStore[opt_{id}] \mapsto \{old\_config \cup new\_config\}, \forall(opt_{id} \in ConfStore)\}}$$

$$\text{Mutate layout}\ \frac{\begin{array}{c} \{n_1\} \in Nodes \quad n_1 := \{\texttt{op}, t_1\} \quad opt_{id} := layoutID \\ config[n_1] := layout_{t_1} \quad layout'_{t_1} \in rand(LayoutGen(t_1)) \quad \wedge \quad (layout_{t1} \neq layout'_{t_1}) \end{array}}{E \vdash config' := Mutate(config) \Downarrow E\{config' \mapsto \{(n_1, layout'_{t_1})\}\}}$$

$$\text{apply-layout1}\ \frac{\begin{array}{c} g; \{n_1, n_2\} \in Nodes; \{(n_1, n_2)\} \in Edges \wedge (n_2.t_{in} == n_1.t_{out}) \\ config[n_1] := l_1 \quad config[n_2] := l_2 \quad E \vdash l_1 == l_2 \Downarrow FALSE \end{array}}{E \vdash g := ApplyOpt(layoutID, g, config) \Downarrow E\{g \mapsto \{g \cup \{n_1.config.layout_{t_{out}} = l_1, n_2.config.layout_{t_{in}} = l_2\} \cup \{\texttt{Insert}(copy, n_1, n_2)\}\}}$$

$$\text{apply-layout2}\ \frac{\begin{array}{c} g; \{n_1, n_2\} \in Nodes; \{(n_1, n_2)\} \in Edges \wedge (n_2.t_{in} == n_1.t_{out}) \\ config[n_1] := l_1 \quad config[n_2] := l_2 \quad E \vdash l_1 == l_2 \Downarrow TRUE \end{array}}{E \vdash g := ApplyOpt(layoutID, g, config) \Downarrow E\{g \mapsto \{g \cup \{n_1.config.layout_{t_{out}} = l_1\}, n_2.config.layout_{t_{in}} = l_2\}\}}$$

$$\text{Mutate fusion}\ \frac{\{n_1\} \in Nodes \quad n_1 := \{\texttt{op}, t_1\} \quad opt_{id} := fusionID \quad config[n_1] := 0}{E \vdash config := Mutate(config) \Downarrow E\{config \mapsto \{(n_1, 1)\}\}}$$

$$\text{apply-fusion}\ \frac{g; \{n_1, n_2\} \in Nodes; \{(n_1, n_2)\} \in Edges \quad config[n_1] := 1}{E \vdash g := ApplyOpt(fusionID, g, config)) \Downarrow E\{g \mapsto \{g \setminus \{n_1, n_2\} \cup \{n'_2 := n_1 \circ n_2\}\}}$$

Fig. 13: Two search steps in a joint layout-fusion tuning for the graphs in Fig. 2: left for step 1 and right for step 2. The neighborhood size for the node's fingerprint calculation used by *ConfStore* is set to 0 (ignoring neighborhood) in this example.

$$\begin{array}{c} \{n_1, n_2, n_3\} \in Nodes \quad \{(n_1, n_3), (n_1, n_4), (n_2, n_4)\} \in Edges \quad n_1 := \{\texttt{add}, t_1\} \quad n_2 := \{\texttt{max}, t_2\} \\ n_3 := \{\texttt{reshape}, t_3, t_4\} \quad n_4 := \{\texttt{conv}, t_5, t_6, t_7\} \quad g := c.graphs(layoutID) \\ LayoutGen(t_1) := LayoutGen(t_3) := LayoutGen(t_5) := \{\{0,2,1\}, \{2,1,0\}, \{1,0,2\}\} \quad LayoutGen(t_2) := LayoutGen(t_6) := \{\{\{0,2,1\}, \{2,0,1\}\} \\ \textbf{Step 1: Tune layout \& fusion} \\ E \vdash config := Mutate(c.configs(layoutID)) \Downarrow E\{config \mapsto \{(n_4, layout_{t5} \mapsto \{0,1,2\}), (n_4, layout_{t6} \mapsto \{0,2,1\})\}\} \\ E \vdash g' := ApplyOpt(layoutID, g, config) \Downarrow E\{g' \mapsto \{g \cup \{n_4.config.layout_{t5} = \{0,1,2\}, n_4.config.layout_{t6} = \{0,2,1\}\} \cup \{\texttt{Insert}(copy, n_1, n_4)\}\} \\ E \vdash config' := InferConfig(c.configs(fusionID), g', ConfStore(fusionID)) \Downarrow E\{config' \mapsto \{(n_1,0), (n_2,0), (n_3,0), (n_4,0), (copy,0)\}\} \\ E \vdash config' := Mutate(config') \Downarrow E\{config' \mapsto \{(n_1,1)\}\} \\ E \vdash g'' := ApplyOpt(fusionID, g', config') \Downarrow E\{g'' \mapsto \{g' \setminus \{n_1, n_3, copy\} \cup \{n_1 \circ n_3, n_1 \circ copy\}\}\} \\ ExecuteGraph(g'') \\ E \vdash UpdateStore(ConfStore\{layoutID : config, fusionID : config'\})^a \\ \Downarrow E\{ConfStore \mapsto \{layoutID : \{(n_4, layout_{t5} \mapsto \{0,1,2\}), (n_4, layout_{t6} \mapsto \{0,2,1\})\}, fusionID : \{(n_1,1), (n_2,0), (n_3,0), (n_4,0), (copy,0)\}\}\} \\ \textbf{Step 2: Tune layout (fusion config is inferred)} \\ E \vdash config := Mutate(c.configs(layoutID)) \Downarrow E\{config \mapsto \{(n_4, layout_{t5} \mapsto \{1,0,2\}), (n_4, layout_{t6} \mapsto \{0,2,1\})\}\} \\ E \vdash g' := ApplyOpt(layoutID, g, config) \Downarrow E\{g' \mapsto \{g \cup \{n_4.config.layout_{t5} = \{1,0,2\}, n_4.config.layout_{t6} = \{0,2,1\}\}\} \\ E \vdash config' := InferConfig(c.configs(fusionID), g', ConfStore(fusionID)) \Downarrow E\{c.configs(fusionID) \mapsto \{(n_1,1), (n_2,0), (n_3,0), (n_4,0)\}\} \\ E \vdash g''' := ApplyOpt(fusionID, g', config') \Downarrow E\{g''' \mapsto \{g' \setminus \{n_1, n_3, n_4\} \cup \{n_1 \circ n_3, n_1 \circ n_4\}\}\} \\ ExecuteGraph(g''') \\ E \vdash UpdateStore(ConfStore, \{layoutID : config, fusionID : config'\})^b \\ \Downarrow E\{ConfStore \mapsto \{layoutID : \{(n_4, layout_{t5} \mapsto \{1,0,2\}), (n_4, layout_{t6} \mapsto \{0,2,1\})\}, fusionID : \{(n_1,1), (n_2,0), (n_3,0), (n_4,0)\}\}\} \\ \hline (g, 2\,steps\,fusion + layout\,tune) \rightsquigarrow Optimal(g'', g''') \end{array}$$

[a] continues in the next line
[b] continues in the next line