Felix: Optimizing Tensor Programs with Gradient Descent

Yifan Zhao
yifanz16@illinois.edu
University of Illinois
Urbana-Champaign, USA

Hashim Sharif
hsharif3@illinois.edu
University of Illinois
Urbana-Champaign, USA

Vikram Adve
vadve@illinois.edu
University of Illinois
Urbana-Champaign, USA

Sasa Misailovic
misailo@illinois.edu
University of Illinois
Urbana-Champaign, USA

Abstract

Obtaining high-performance implementations of tensor programs such as deep neural networks on a wide range of hardware remains a challenging task. Search-based tensor program optimizers can automatically find high-performance programs on a given hardware platform, but the search process in existing tools suffer from low efficiency, requiring hours or days of time to discover good programs due to the size of the search space.

We present ***Felix***, a novel gradient-based compiler optimization framework for tensor-based programs. Felix creates a *differentiable* space of tensor programs that is amenable to search by gradient descent. Felix applies continuous relaxation on the space of programs and creates differentiable estimator of program latency, allowing efficient search of program candidates using gradient descent, in contrast to conventional approaches that search over a non-differentiable objective function over a discrete search space.

We perform an extensive evaluation on six deep neural networks for vision and natural language processing tasks on three GPU-based platforms. Our experiments show that Felix surpasses the performance of off-the-shelf inference frameworks – PyTorch, Tensorflow, and TensorRT – within 7 minutes of search time on average. Felix also finds optimized programs significantly faster than TVM Ansor, a state-of-the-art search-based optimizer for tensor programs.

ACM Reference Format:
Yifan Zhao, Hashim Sharif, Vikram Adve, and Sasa Misailovic. 2024. Felix: Optimizing Tensor Programs with Gradient Descent. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651348

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04…$15.00
https://doi.org/10.1145/3620666.3651348

1 Introduction

Computationally intensive deep neural networks (DNNs) are ubiquitous in a wide range of applications such as autonomous driving, augmented reality, and language translation. DNNs are increasingly deployed in resource-constrained edge computing environments, making it challenging to perform inference with low latency. Existing deep learning frameworks, such as PyTorch [26] and TensorFlow [1], map operators in DNNs to kernel libraries to manually-optimized implementations for specific hardware architectures. This approach requires significant expertise and manual effort, and the performance of the optimized code does not carry across the increasingly diverse edge platforms.

Search-based automatic code generation of tensor programs [2–4, 7, 28] is a recent and more scalable approach to finding efficient implementations of tensor operators. A search-based code generation framework typically defines a search space of *schedules* – sequences of program transformations such as loop tiling, vectorization, parallelization, and unrolling, that apply to the user-given initial program. The code generator then searches for a schedule that delivers high performance for a given program on the target hardware.

To achieve good performance for a wide range of different hardware architectures, the search space needs to include a large number of candidate schedules with different *schedule variables* (e.g., tiling factors, loop unroll factors). However, searching in a large space is fundamentally difficult. Existing Machine Learning (ML) compilers rely on combinatorial discrete search techniques such as beam search and/or genetic algorithms to explore the search space, and suffer from excessively long tuning time of hours or days per program [28, 38]. Some existing approaches therefore resort to covering only part of the search space, using manually-written templates [4, 7] or aggressive pruning [2].

We instead start from the insight that turning a discrete search problem into a *differentiable optimization* problem has a potential to produce a more efficient decision-making algorithm. Compared to even sophisticated discrete search techniques, gradient-based methods are more informed about the shape of the objective function which can become a significant advantage in search time and result optimality. While gradient-based search techniques are routinely used for tasks such as DNN training [17] or neural architecture search [22, 35], they are rarely applied to selecting the

program schedule, because it is formulated as an inherently discrete problem.

Formulating the program schedule search as a differentiable optimization problem poses significant challenges:

- The search space of schedules is discrete, with many of the tunable parameters constrained in a subset of integers. For example, the tiling sizes of loop tiling optimization must be integers, but also must be factors of the loop extent to not introduce conditional branches.
- The *objective function* – the performance of the program as a function of the schedule – is highly complex, often discontinuous, and non-differentiable. To evaluate a schedule accurately, the compiler needs to generate a program from the schedule and empirically measure the performance of the program on the target hardware.
- Analytical [34] or learned [19, 23, 39] performance models approximate the performance objective and are generally faster to evaluate than empirical measurements. However, to maintain generality, these models typically take as input the generated program or a program *feature vector* representative of the program's performance, instead of taking the schedule as the input. Even when these models are differentiable themselves, differentiation of the feature vector with regard to the variables in the program schedule remains tremendously challenging as program generation exercises multiple components of the compiler.

Our Work. To address these challenges, we develop **Felix**, a novel gradient-based compiler optimization framework for tensor programs. Felix deviates from the common practice of relying on combinatorial search algorithms for schedule tuning, and instead applies gradient descent over a *differentiable performance prediction function* to optimize schedules.

Felix first partitions the computation graph of the input program into subgraphs, each representing one or several tensor operators, and searches for a schedule for each subgraph independently. For each subgraph, Felix creates a performance predictor differentiable against schedule variables (e.g., tiling factors, loop unroll factors), and optimizes the value of these variables with gradient descent. This performance predictor is composed of two parts: (1) program features differentiable against schedule variables, and (2) a pretrained DNN-based cost model, which takes values of program features as input and predicts its execution time. Our novel contribution of auto-generating differentiable program features enables Felix to apply to a wide range of tensor operators and compiler transformations, because the DNN-based cost model can be trained once on a set of program features and applied to various tensor programs.

Auto-generating differentiable program features introduces several key technical challenges that we address and discuss in Section 3. A major challenge is to automatically derive program features as mathematical expressions of schedule variables. Felix creates *symbolic schedules* that contains schedule variables as parameters of program transformations, and applies these schedules to the subgraph to create *symbolic programs*. Lastly, Felix's feature extraction component analyses symbolic programs to produce program feature expressions.

To enable gradient descent, Felix relaxes the space of schedules to be continuous, and tracks *validity constraints* that ensure the potential code transformations are legal. For an example loop of size N, Felix relaxes a loop unrolling size variable $k \in \{1, 2, ..., N\}$ into $k \in [1, N] \subset \mathbb{R}$, and adds the legality constraint $1 \leq k \leq N$. The gradient descent search then produces relaxed candidate schedules in the continuous space. Felix discretizes schedules (e.g., by rounding the variables to the nearest integer) and validates their legality by checking the original constraints. Then, Felix confirms the legality of candidate schedules empirically on the target hardware platform to yield the final schedules for each subgraph. Finally, Felix combines these optimized subgraphs to produce an optimized full program.

Results. We evaluate Felix on six diverse neural networks for vision and natural language processing tasks on three hardware GPU platforms that represent server (NVIDIA A10G), desktop (NVIDIA RTX A5000), and edge (NVIDIA Xavier NX) use cases. We compare Felix against (1) recent versions of off-the-shelf inference frameworks PyTorch, TensorFlow, and TensorRT, which all have code generation and autotuning capabilities, and (2) state-of-the-art search-based compiler framework TVM Ansor [38], extended with TenSet pretraining [39]. Our experiments show that the speedup of Felix-optimized networks is 2.2× compared to PyTorch, 1.7× to TensorFlow, and 1.5× to TensorRT within just 7 minutes of tuning on average (geometric mean). across the six networks and three hardware platforms, it reaches a 95% performance of the best discovered code 3.4× faster and reaches 99% performance 2.8× faster on average (geomean). Felix is particularly effective for time-constrained tuning or tuning on resource-constrained edge devices, as Felix can quickly find schedules with high performance.

Contributions. We make the following contributions:

- We present Felix, a novel gradient-based compiler optimization framework for tensor programs. Felix carefully models the program performance as a differentiable function of the schedule to find good schedules efficiently.
- We present a novel algorithm to define a symbolic schedule template for a tensor program subgraph and transform it into a differentiable formula of program's run time, which enables using gradient descent to tune program schedules.
- We implement Felix on top of the Apache TVM tensor compiler [7] and its Ansor optimizer [38]. Felix's optimization approach is general and can be implemented over other tensor-based compilers.

- We present a thorough evaluation of Felix that shows Felix delivers significantly higher performance in limited search time compared to the state-of-the-art systems on multiple benchmarks running on server, desktop and edge GPU-based hardware platforms.

Felix code is available at https://github.com/uiuc-arc/felix.

2 Problem Statement

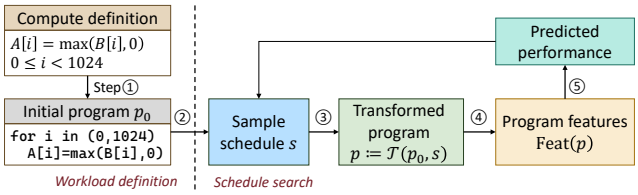


Figure 1. Typical workflow of a search-based tensor compiler.

Figure 1 outlines the typical workflow of a search-based tensor program compiler such as Ansor, the Halide auto-scheduler, and the Tiramisu auto-scheduler [2, 3, 7]. Users define the computation using an API or a declarative language supported by the compiler, for which the compiler does a simple 1:1 translation of the language constructs to its intermediate representation equivalents (Step 1). This initial program p_0 is a naïve implementation of the user-given mathematical definition and usually has low performance. The goal of the search-based compiler optimizer is to find a sequence of program transformations s – often called a *schedule* – that produces an optimized program $\mathcal{T}(p_0, s)$. Here, $\mathcal{T}$ is the program transformation pipeline in the compiler that applies the sequence of transformations s to the initial program p_0. The optimization problem is formulated as:

$$\max_{s \in \mathbb{S}(p_0)} \quad \text{Performance}\left(\mathcal{T}(p_0, s)\right) \tag{1}$$

where p_0 is the initial program, $\mathbb{S}(p_0)$ is the search space of legal schedules applicable to p_0, $\mathcal{T}(p_0, s)$ is the optimized program produced by applying the schedule s to p_0, and Performance is measured as execution time on the target machine.

Search space definitions. To achieve performance better than hand-tuned implementations, tensor compilers need to apply multiple different kinds of optimizations including but not limited to loop tiling, loop unrolling, vectorization, parallelization, as well as hardware-specific transformations such as CUDA blocking/threading on GPUs. Each kind of transformation can also contain tunable (usually boolean or integer) parameters; for example, tiling a loop level into n levels introduces $n-1$ integer parameters for loop sizes. The choice and ordering of these transformations and their tunable parameters create an extremely large number of possible schedules, but typically the vast majority of them do not produce valid programs. To combat this, existing compilers generally adopt schedule templates (user-written or auto-generated) or sequentially construct and prune the schedule to disallow arbitrary sequences of transformations. The search space of schedules depends on the input program p_0 and includes discrete parameters since the compiler parameters being tuned are integer values (e.g., tile sizes, SIMD parallelization factors).

Cost prediction models and program features. Since the search space of schedules is discrete, a number of classic discrete-space search techniques such as genetic search and beam search have been applied for this optimization problem. Due to the large size of the search space, the search usually requires a large number of tuning iterations each empirically evaluating one candidate program. To mitigate the overhead of empirically measuring too many programs, existing compilers typically use some *cost model*, such as feed-forward networks, LSTMs, and decision trees, to predict the performance of the searched programs, and may reduce the number of empirical program evaluations. The cost models do not directly take schedule s as the input, but often a **vector** of K *program features* extracted from $p := \mathcal{T}(p_0, s)$:

$$\textbf{Feat}(p) := (\text{Feat}_1(p), \ldots, \text{Feat}_k(p), \ldots, \text{Feat}_K(p))$$

These program features, such as the number of floating point add/multiply operations in the program and the reuse distance of buffer access in loops, are often hand-selected by the compiler designer and extracted with static analysis.

Existing objective functions are not differentiable. As shown in Figure 1, there are 3 major steps (Steps 3-5) to estimating the performance of a schedule: generating the program $\mathcal{T}(p_0, s)$, extracting the program features $\textbf{Feat}(\mathcal{T}(p_0, s))$, and feeding them to the cost model. When using this performance estimator, the optimization problem becomes the following (compare Eqn. 1):

$$\max_{s \in \mathbb{S}(p_0)} \quad \text{CostModel}\left(\textbf{Feat}(\mathcal{T}(p_0, s))\right) \tag{2}$$

The objective function is not differentiable, because conventionally applying the schedule and extracting the program features are non-differentiable procedures. To address these problems, we will next present how Felix makes this procedure differentiable by generating symbolic schedules, symbolically transformed programs, and formulas of program features.

3 Felix Design

Figure 2 illustrates the workflow of the Felix compiler. Our approach is described as follows:

- Felix analyzes the input program as a computation graph whose nodes are tensor operators and edges indicate data flow, and *partitions* it into subgraphs such that optimization happens within the subgraph but not inter-subgraph (§3.1). Each subgraph is optimized independently and gets its own schedule.

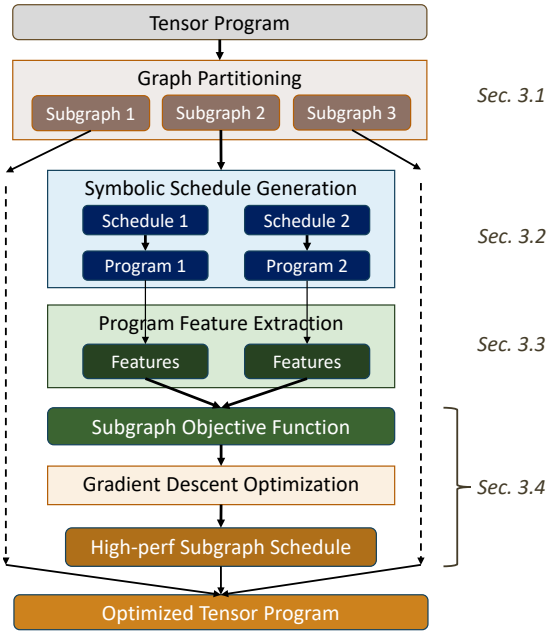


Figure 2. Felix High-level Workflow.

- For each subgraph, a symbolic schedule s^* is a schedule whose transform steps can use variables as parameters (in contrast to non-asterisk concrete schedule s). Felix generates N symbolic schedules $s_1^*, \ldots, s_i^*, \ldots, s_N^*$, each with M_i *schedule variables* $\mathbf{x}_i := (x_{i1}, \cdots, x_{ij}, \cdots, x_{iM_i})$ (§3.2) that can come from multiple code transformations (e.g., tiling and vectorization). For example,

  `[Unroll(loop_id=1, max_step=(n))]`

  is a symbolic schedule with one variable (n). A symbolic schedule spans the search space, because it can generate many concrete schedules when given different values for the schedule variables. The subgraph's search space is the Cartesian product of its symbolic schedules' search spaces.
- For each symbolic schedule s_i^* in each subgraph, Felix creates a differentiable performance estimator EstmPerf_i, and combines all estimators of the subgraph into one *subgraph objective function* $O(\mathbf{x}_i)$ (§3.3). The estimator composes two functions:
  1. Felix first applies the symbolic schedule s_i^* as a sequence of transformations to the subgraph to get a transformed *symbolic program* p_i^* (§3.2), which contains the schedule variables in its loop bounds, array indices, etc. An analysis pass then extracts *a vector of feature formulas* for each program p_i^* as differentiable functions of $\mathbf{x}_i$:

  $$\begin{aligned}\mathbf{Feat}_{p_i^*}(\mathbf{x}_i) := (&\text{Feat}_{1;p_i^*}(x_{i1}, x_{i2}, \cdots, x_{iM_i}), \ldots,\\ &\text{Feat}_{k;p_i^*}(x_{i1}, x_{i2}, \cdots, x_{iM_i}), \ldots,\\ &\text{Feat}_{K;p_i^*}(x_{i1}, x_{i2}, \cdots, x_{iM_i})\ )\end{aligned}$$

  that capture characteristics of the program, such as the number of float operations and the reuse distance of buffer accesses.
  2. Felix pretrains a feed-forward neural network-based *cost model* C that maps values of the program features to the predicted performance (a scalar). The cost model is trained offline on a dataset of artificially generated schedules and only needs to be trained once per target hardware architecture.
- Felix minimizes $O(\mathbf{x}_i)$ with regard to $\mathbf{x}_i$ using gradient descent to discover high-performance schedules for each subgraph (§3.4), and finally combines the optimized subgraphs to generate an optimized tensor program (§3.5).
- Lastly, we demonstrate in §3.6 how developers can use Felix's programming interface to optimize tensor programs with little effort.

3.1 Computation Graph Partitioning

Felix partitions the computation graph of the input tensor program into *subgraphs* of fused tensor operators. These subgraphs are to be optimized in rounds of search, where each round chooses one subgraph and optimizes it independently (detailed in §3.5). The idea of subgraph partitioning [38] is to reduce the search space of optimization, as opposed to jointly optimizing the entire program, which can be intractable. The graph partitioning fuses operators in fixed patterns, e.g., a Conv followed by a ReLU can be fused into a Conv-ReLU subgraph which is optimized as a fused construct.

3.2 Symbolic Schedule and Symbolic Program Generation

As a concrete example, Figure 3 shows a `Dense-Add` subgraph with its mathematical definitions and the corresponding naïve program, and the two symbolic schedules s_1^*, s_2^* generated for the graph when compiling to GPU. `Dense-Add` performs a matrix multiplication with bias add, a common operator in many DNNs. The simpler schedule s_1^* has 2 schedule variables `TILE0`, `UNROLL0` and the more complex s_2^* has 12: $\texttt{TI}_{0..3}$, $\texttt{TJ}_{0..3}$, $\texttt{TK}_{0,1}$, `SV0`, and `UNROLL0`.

Felix extends Ansor's [38] notion of (concrete) *sketch* and *annotation* to generate these *symbolic* schedules. A sketch is a list of program transformations with unfilled tunable parameters; Ansor generates multiple sketches for each subgraph. A sketch can be annotated by filling in these parameters to produce a valid schedule. Instead of annotating the sketches with concrete integer and boolean values (the only ones Ansor can do), Felix defines *symbolic schedule variables* $\mathbf{x}_i$ and annotates the sketch with schedule variables to produce symbolic schedules s_i^*. Felix also keeps track of constraints c_{iq} $(1 \le q \le C_i)$ over the value of schedule variables. For example, in the schedule s_1^* in Figure 3, `TILE0` satisfies $c_{11}(\texttt{TILE0}) = (1 \le \texttt{TILE0} < \texttt{K})$. The number of constraints C_i varies for different symbolic programs. Felix can convert any sketch that Ansor generates into a symbolic schedule, and for a given subgraph, Felix's search space has the same dimension as Ansor's. In terms of schedule variables, this includes variables in a number of positions, such as loop sizes

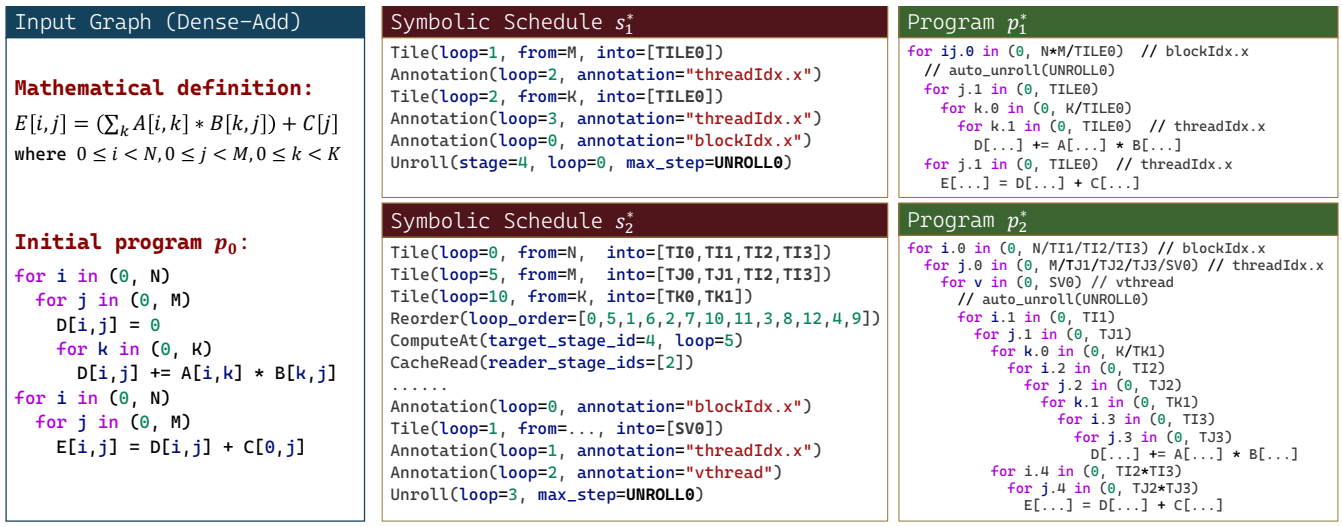


Figure 3. Example of symbolic schedule generation on a `Dense-Add` graph, showing 2 generated symbolic schedules and their corresponding programs. Some transformation steps in the schedules and long expressions in the programs are omitted for brevity. Variables that Felix introduces into the schedules are shown in bold font.

in loop tiling ($\texttt{TI}_i, \texttt{TJ}_i, \texttt{TK}_i$, `TILE0` in Figure 3), the number of virtual threads (`SV0`), sizes of loop unrolling (`UNROLL0`) / vectorization / parallelization, and location of computation inlining (known as `ComputeAt` transformation in TVM [7]).

Symbolic program generation. Felix generates a *symbolic program* p_i^* from each symbolic schedule s_i^* by applying the transformation steps in s_i^* on the initial program of the subgraph p_0; i.e., $p_i^* := \mathcal{T}(p_0, s_i^*)$. These symbolic programs contain the schedule variables $\mathbf{x}_i$ – the loop bounds, annotations, and buffer access indices are expressions of x_{ij}. The right column in Figure 3 shows the two programs p_1^*, p_2^* derived from two symbolic schedules s_1^*, s_2^*.

3.3 Feature Formula Extraction and Rewriting

Felix contains a program analysis pass that runs on a symbolic program p_i^* to extract a number of program features, such as the total number of add / mul / div operations in the program and the reuse distance of memory buffers. Felix uses a total of 82 distinct features to capture computation and memory access characteristics of the program. Because these features are dependent on the loop bounds and buffer access indices in p_i^* which contains the schedule variables $\mathbf{x}_i$, the feature formulas are functions of the schedule variables. The following table shows a few of the features extracted from the first program p_1^* of the `Dense-Add` graph:

Operator	Feature
`float_add`	$\texttt{N} \cdot \texttt{M} \cdot \texttt{K}$
`blockIdx_len`	$\texttt{N} \cdot \texttt{M} / \texttt{TILE0}$
`int_add`	$\texttt{NMK}(\textbf{select}(\texttt{TILE0} > 1, 5, 2))$

Here, $\textbf{select}(b, x, y)$ is a piecewise function that equals x when the boolean expression b evaluates to true and y otherwise. In general, each feature expression extracted by Felix can only contain schedule variables, constants, and a fixed set of operators that the analysis pass uses, such as $+, -, *, /$, **pow**, **min**, **max**, **select**, etc.

Extracted features are non-differentiable. The extracted formulas contain discontinuous and non-differentiable operators such as **select**, **min**, and **max**, due to the discrete nature of some program features. A frequent scenario where discontinuity arises is when the value of the feature depends on if a loop level is trivial with a bound of 1. The formula of `int_add` feature exhibits this behavior and contains the **select**() function.

Felix makes expressions differentiable. To obtain differentiable formulas of program features, Felix uses smoothing kernels to create smooth differentiable approximation for each non-differentiable operator it encounters.

To automatically rewrite whole formula that contains non-differentiable operators, an expression rewriter in Felix applies a built-in library of *rewrite rules* each mapping one non-differentiable operator into its differentiable version. We derived each approximated function $f_{\text{smooth}}(x)$ by convolving the non-differentiable function $f(\cdot)$ with a smoothing kernel $\phi(\cdot)$:

$$f_{\text{smooth}}(x) := \int_{-\infty}^{x} f(x-t)\phi(t)\mathrm{d}t$$

In this work we used $\phi(t) := 1/\sqrt{1+t^2}$, which makes the gradient of the smoothed functions more numerically stable compared to other common alternatives such as Gaussian or bump kernels.

As the result of this process, all generated functions will be *smooth* (infinitely differentiable). Figure 4 compares two examples of non-differentiable functions ($\textbf{select}(x > 0, 5, 2)$

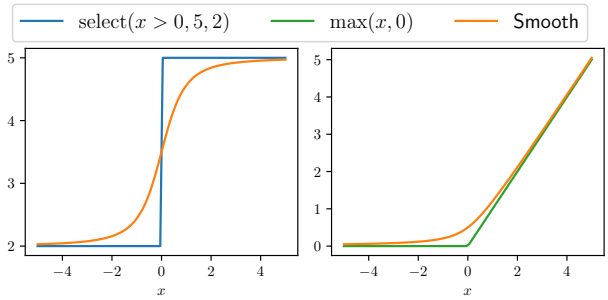


Figure 4. Comparison between two non-differentiable functions and their smooth differentiable version respectively. The nondifferentiable functions are **select**$(x > 0, 5, 2)$ (left), and **max**$(x, 0)$ (right).

and **max**$(x, 0)$) and their smooth approximations used by Felix. The rewrite rules can automatically rewrite any feature expressions emitted from the analysis pass with smooth functions' versions. We derived smooth versions manually as the number of non-differentiable operators is small (less than 10), and it is a one-time effort in the entire development of the compiler and does not depend on the transformations. Conceptually, this step can be readily automated, e.g. by smooth interpretation [5].

Gradient stability. Some program features have exponentially growing expressions and large value ranges. For example, `float_add` is often the product of a few variables and can grow up to $10^8 \sim 10^9$ for ordinary input programs. Gradient vanishes when program features take on large values because a delta change is insignificant. To avoid vanishing gradients, Felix takes logarithm of the smooth program features, and performs exponential variable substitution $x = e^y$ for each schedule variable x so that y is the new variable to be optimized. These two rewrite steps convert multiplicative terms to additive ones, and produce feature expressions that exhibit linear growth and have more stable gradients.

Constraint penalty functions. Felix's expression rewriter also rewrites the variable constraints c_{iq} $(1 \le q \le C_i$; see §3.2) into differentiable constraint penalty functions $g_{ir}(\mathbf{x}_i)$ $(1 \le r \le G_i)$. One constraint can produce one or multiple penalty functions, meaning that $G_i \ge C_i$. For example, $c_{11} =$ $(1 \le$ `TILE0` $\le$ `K`$)$ becomes $g_{11}($`TILE0`$) :=$ `1` $-$ `TILE0` and $g_{12}($`TILE0`$) :=$ `TILE0` $-$ `K`. The schedule variables are in valid range if and only if all $g_{ir}(\mathbf{x}_i) \le 0$.

Some size variables, such as loop sizes in loop tiling, may have divisibility constraints of the form $N \bmod x = 0$. N is the size of the loop to be tiled (a constant), and x equals e^y due to Felix's logarithm substitution. Felix replaces each such constraint with a size constraint $y \le \ln N$ and addresses divisibility by value rounding post-optimization. Instead of rounding x to the closest integer, Felix rounds y to the closest $\ln N_i$ where N_i is an integer factor of N. A list of N_i is easily computable as N is the size of a dimension of some tensor and typically smaller than 10^6.

3.4 Optimizing Schedules with Gradient Descent

Objective function. In this stage, Felix creates performance estimators per symbolic program p_i^* by composing two functions together: the program features $\mathbf{Feat}_{p_i^*}(\mathbf{x}_i)$ and a pre-trained, neural network-based cost model C, which takes the program feature values as input and outputs a scalar for the predicted performance of the schedule, i.e., $\text{EstmPerf}_{p_i^*}(\mathbf{x}_i) := C(\mathbf{Feat}_{p_i^*}(\mathbf{x}_i)) \in \mathbb{R}$. Felix then tunes the values of the symbolic variables across all the symbolic programs of a subgraph using the following optimization problem formulation:

$$\max_{p_i^*} \max_{\mathbf{x}_i} \quad C(\mathbf{Feat}_{p_i^*}(\mathbf{x}_i)) \quad \text{s.t.} \quad \forall i, r.\ g_{ir}(\mathbf{x}_i) \le 0 \tag{3}$$

We relax the variables $\mathbf{x}_i$, which are mostly booleans and integers, into real-valued variables, and rewrite Equation 3 into the following *objective function* to be minimized:

$$O(\mathbf{x}) := \sum_i \left(-C(\mathbf{Feat}_{p_i^*}(\mathbf{x}_i)) + \lambda \sum_q \max\left(g_{ir}(\mathbf{x}_i), 0\right)^2 \right) \tag{4}$$

Here we sum over i to simultaneously optimize all $\mathbf{x}_i$ of a subgraph, and negate the higher-better performance estimator to make a lower-better objective function. The penalty term $\max(g_{ir}(\mathbf{x}_i), 0)^2$ is 0 when g_{ir} is not violated, and is $g_{ir}(\mathbf{x}_i)^2 > 0$ otherwise, so minimizing this term pushes $\mathbf{x}_i$ towards satisfying the constraints. Adding penalty terms to the function to be minimized converts a constrained optimization problem into an unconstrained one, and is commonly used on many constrained optimization problem such as DNN weight regularization [11] and structured pruning [35, 40]. Here, λ is a hyperparameter that controls the strength of penalty functions.

The objective $O(\mathbf{x})$ is differentiable because (1) $\mathbf{Feat}_{p_i^*}(\mathbf{x}_i)$ is differentiable by our construction; (2) the $\max(x, 0)^2$ function is differentiable with derivative $2\max(x, 0)$; (3) each $g_{ir}(\mathbf{x}_i)$ is differentiable by our construction, and thus the composition $\max(g_{ir}(\mathbf{x}_i), 0)^2$ is also differentiable; finally, (4) the sum of differentiable functions is differentiable.

Schedule optimization for subgraph. Algorithm 1 outlines how Felix optimizes one subgraph with gradient descent. To explore more parts of the search space and avoid being trapped in local minima, Felix optimizes multiple (`nSeeds`) schedules simultaneously. First, Felix extracts the symbolic variables, symbolic schedules and objective functions on lines 10 and 11. On line 12, Felix randomly samples `nSeeds` sets of initial values ($\sigma_{\mathbf{x}}$) for the schedule variables that satisfies all constraints using rejection sampling. Lines 14 to 19 initializes an Adam [17] optimizer and runs it for `nSteps` optimization steps to minimize the objective function given as Eqn. 4. Each step evaluates the gradient of the objective function $O(\mathbf{x})$ at the current values $\sigma_{\mathbf{x}}$ of the schedule variables,

Algorithm 1: Felix schedule tuning for each subgraph.

1 **Inputs:**
2 p_0: subgraph to be optimized
3 C: pretrained performance predicting model
4 nSeeds: number of schedules to optimize simultaneously
5 nSteps: number of steps to run gradient descent for
6 nMeasure: number of schedules to evaluate on hardware
7 λ: coefficient of the constraint penalty term
8 **Outputs**: $\mathbf{s}_{\text{best}}$, measuredPerfs, C_{upd}
9 **Function** *ExtractAndOptimize*
10 $\mathbf{s}^*$, $\mathbf{x}$ = GenerateSymSchedulesAndVars(p_0);
11 O = MakeObjectiveFunc($\mathbf{s}^*$, C, λ);
12 $\sigma_{\mathbf{x}}$ = RandomInitSchedVars($\mathbf{x}$, nSeeds);
13 varValuesHistory = list();
14 optimizer = Adam();
15 **for** i *in range(0,* nSteps*)* **do**
16 costs = $O(\sigma_{\mathbf{x}})$;
17 varValuesHistory.append(($\sigma_{\mathbf{x}}$, costs));
18 gradients = AutoDiff(O, costs, $\sigma_{\mathbf{x}}$);
19 $\sigma_{\mathbf{x}}$ = optimizer.step($\sigma_{\mathbf{x}}$, costs, gradients);
20 $\mathbf{s}_{\text{valid}}$ = GetValidSchedules($\mathbf{s}^*$, varValuesHistory);
21 $\mathbf{s}_{\text{best}}$ = TopNSchedulesByPredPerf($\mathbf{s}_{\text{valid}}$, nMeasure);
22 $\mathbf{p}_{\text{best}}$ = $\{ \mathcal{T}(p_0, s) \mid s \in \mathbf{s}_{\text{best}} \}$;
23 measuredPerfs = MeasurePerfOnDevice($\mathbf{p}_{\text{best}}$);
24 C_{upd} = UpdateCostModel(C, $\mathbf{s}_{\text{best}}$, measuredPerfs);
25 **return** $\mathbf{s}_{\text{best}}$, measuredPerfs, C_{upd};

and calls the Adam optimizer to move $\sigma_{\mathbf{x}}$ in the direction of the gradient. Each step updates all the nSeeds schedules.

To get a list of valid concrete schedules $\mathbf{s}_{\text{valid}}$, Felix then takes all the variable values traversed in the optimization process, rounds them to integers (we use the nearest rounding in our implementation), and removes invalid rounded schedules (line 20). Felix then takes the best nMeasure schedules $\mathbf{s}_{\text{best}}$ (sorted by cost model-predicted performance), generates their corresponding concrete programs $\mathbf{s}_{\text{best}}$, and empirically evaluates them on the target hardware. Thus, Felix does only a small set of (often expensive) evaluations on the target hardware. Felix also uses the schedules and their measured performance to update the cost model C (which is an inexpensive process) so that the cost model is better fitted to the subgraphs of this input program in the next rounds.

3.5 Full Graph Tuning

To optimize an entire tensor program, Felix first decomposes the program into subgraphs (§3.1), and adopts Ansor's round-based tuning algorithm [38] to tune these subgraphs in rounds of search. In each round, Felix applies its subgraph optimization algorithm (Algorithm 1) to one subgraph. Algorithm 2 shows this iterative tuning process. We adopt Ansor's task scheduler to select which subgraph to optimize in each round, but any other selection algorithm would work as well. Finally, after nRounds of tuning, Felix selects the best schedule of each subgraph and *combines* them into a full schedule for the whole tensor program.

Algorithm 2: Felix full program schedule tuning.

1 **Inputs:**
2 **p**: subgraphs to be optimized
3 nRounds: number of rounds of tuning among all subgraphs
4 C_0: pretrained cost model for target hardware
5 **Outputs**: optimized full program
6 **Function** *TuneFullProgram*
7 tuneHistory = list(); $C = C_0$;
8 **for** i *in range(0,* nRounds*)* **do**
9 p_{next} = SelectNextSubgraph(**p**, tuneHistory);
10 **s**, measuredPerfs, C =
ExtractAndOptimize(p_{next}, costModel, ...);
11 tuneHistory.append((p_{next}, **s**, measuredPerfs));
12 bestFullSchedule = CombineBestSchedules(tuneHistory);
13 **return** bestFullSchedule;

3.6 Felix Programming Interface

Felix is developed as a Python library with an easy-to-use API. Figure 5 presents an example of using Felix to optimize ResNet-50 to run on the Nvidia Xavier NX GPU. The key interface functions in the example are explained within the comments. As the figure shows, using Felix interface to optimize a DNN requires only a few lines of code. Developers only need to provide the network to optimize and the network's input shape, and specify the total tuning time budget in terms of the number of tuning rounds. This is a common practice in existing autotuning works, as automatic stopping criteria (e.g. stop on convergence) are rarely satisfied.

4 Implementation

Felix is implemented on top of TVM [7] and its Ansor [38] optimizer. Felix represents its symbolic programs in the TVM TIR intermediate representation language and uses TVM's program transformation passes and hardware backends. It also reuses two algorithms from Ansor: its graph partitioning algorithm and sketch generation algorithm. Felix extends Ansor's graph partitioning algorithm to track additional information for creating symbolic variables (§3.1), and processes the sketches produced by Ansor's sketch generation algorithm to generate symbolic programs (§3.2). We implement Felix to utilize all hardware-independent and GPU-specific optimizations in TVM, and support the tuning of all the program transformation parameters that Ansor tunes on GPU.

Felix's search space includes the following tunable parameters: (1) CUDA thread/block sizes, (2) loop tiling sizes, (3) vectorization dimension sizes, (4) parallelization dimension sizes, and (5) loop unrolling factors. This search space also includes the options of skipping a transformation, as Felix treats a size/factor of value 1 as a no-op. In Felix and Ansor, some optimizations are applied without tuning; these

```python
import felix
from torchvision import resnet50
# Define the hardware target to optimize for
device = felix.device.cuda("xavier-nx")
# Define the DNN to optimize and its input shape
dnn, input_shape = resnet50(), [1, 3, 256, 256]
# Extract subgraphs to tune from the DNN
graphs = felix.extract_subgraphs(dnn, input_shape)
# Get pretrained cost model for the target device
cost_model = felix.pretrained_cost_model(device)
# Felix `Optimizer` sets up the search space
# and the objective function for each subgraph
opt = felix.Optimizer(graphs, cost_model, device)
# Run the search process for 100 rounds
opt.optimize_all(n_total_rounds=100,
    measure_per_round=16, save_res="resnet50.json")
# Apply the best schedules found for each subgraph
# and generate a compiled module
compiled_lib = opt.compile_with_best_configs(
    configs_file="resnet50.json")
# The module can be called as a function in Python
# or saved to a file and loaded later
compiled_lib.save("resnet50_xavier_nx.pkl")
```

Figure 5. Example code for using Felix's programming interface to optimize a ResNet-50 network defined in PyTorch.

decisions are delegated to built-in rules in TVM and Ansor. For example, Felix and Ansor use a rule-based loop reordering and apply operator fusion greedily when applicable. For each subgraph, the search space of Felix (and Ansor) only include orderings of transformations suggested by Ansor's sketch generation algorithm. A search that includes arbitrary orderings of transformations is out of scope of this work.

Felix uses PyTorch [26] for gradient back-propagation and invoking the Adam optimizer used in Algorithm 1, and TenSet [39] for the cost model definition and the dataset for training the cost model. We choose the multi-layer percepton (MLP) architecture provided by TenSet, which is a network with 4 linear layers and approximately 250K parameters.

Felix is implemented as 13K lines of C++, Python, and Rust code. Since TVM requires concrete integer-valued parameters in some program transformations and some parts of the program, e.g., certain loop bounds, we patch 1.5K lines of TVM code to add support for Felix's symbolic schedule and symbolic program representation. A large part of the symbolic program generation and feature extraction is implemented in C++, and Felix's expression rewriter is written in Rust to interface with egg [36], an equality saturation-based rewriting framework we use to enable efficient rule-based expression rewriting described in Section 3.3. A developer can extend Felix to tune an existing transformation in TVM by modifying the source code of the transformation to use symbolic parameters. Moreover, retraining Felix's cost model and adding simplification rules to Felix's expression rewriter are optional and can help achieve better search results.

5 Experimental Methodology

Tensor programs to optimize. We choose 6 neural networks to evaluate Felix: ResNet-50 [15], MobileNet-v2 [30], R3D-18 [14], DCGAN [27], Vision Transformer (ViT) [12], and LLaMA [33]. Each of these networks contains different types of operators, such as 2d/3d convolutions, transposed convolutions, and batched matrix multiplications, and together they cover a large part of operators commonly seen in today's deep learning workloads. We optimize all the networks for inference at full (`float32`) precision. In all experiments except §6.4, the networks run on a batch size of 1, i.e., the input to each network is 1 image, video, or sentence, depending on the network. For LLaMA, the length of each input sentence is 100 tokens. We study the effect of other input batch sizes in §6.4.

Hardware platforms. We choose three GPU devices for Felix to optimize programs for: Nvidia A10G, Nvidia RTX A5000 and Nvidia Xavier-NX. Nvidia A10G represents devices in a server setting [8], A5000 represents desktop devices [10], while Xavier-NX is representative of devices seen in an edge computing setting [9]. For Xavier-NX, we run the code of Felix and TVM themselves on a separate machine with a 32-core AMD Ryzen 3975WX CPU and 512 GB of RAM, and use remote-procedure calls to run the tensor programs on the Xavier-NX.

Baseline frameworks. We include PyTorch 2.2, TensorFlow 2.15, TensorRT 8.6, and Ansor (commit hash: `95aac9224`) as baseline frameworks. PyTorch, TensorFlow, and TensorRT are off-the-shelf inference frameworks that contain manually optimized kernel libraries and can mix and match them with JIT-generated code. Ansor is a state-of-the-art search-based tensor compiler. All these frameworks are capable of graph compilation, graph optimization, code generation, and schedule autotuning. We use the TorchInductor backend in PyTorch via the command `torch.compile(..., backend='inductor')`, and XLA with TensorFlow via the command `tf.function(..., jit_compile=True)`. We use Ansor with a cost model pretrained on the TenSet dataset; this method, the architecture of the cost model, and the dataset are provided by the original TenSet work [39], so we refer to this setup as Ansor-TenSet. It achieves equal or better tuning result than Ansor in less search time [39], and is hence a better baseline. Ansor-TenSet has the same search space definition and tuning techniques as Ansor.

Search parameter settings. We use 8 initial schedules, 200 gradient descent steps, and 16 empirical measurements (nSeeds, nSteps, and nMeasure in Algorithm 1) as the default settings for the gradient optimizer of Felix. When comparing against Ansor, we use the recommended settings for Ansor a population size of 2048 and 4 generations for Ansor's evolutionary search, and 64 empirical measurements per tuning round,

since using a lower setting with Ansor produces worse results. While the search parameters of Ansor and Felix are not directly comparable, they indicate that Ansor predicts the performance of approximately $8192 = 2048 \times 4$ schedules per search round, while Felix predicts $1600 = 8 \times 200$ per round. We set both Felix and Ansor to run each candidate schedule for 100 milliseconds when evaluating schedules on hardware, which corresponds to repeating the program 10 – 1000 times depending on the latency of the program.

Cost model training. The cost model is trained on a dataset provided by TenSet [39], which contains over 1000 subgraphs and thousands of schedules for each subgraph. We select a subset of 500 subgraphs, covering all common types of bottleneck workloads such as convolutions and linear layers. For each subgraph, we select 512 schedules (or all the schedules if fewer than 512 is provided for that subgraph), such that our selected dataset contains ~250,000 schedules in total. We don't use the entire dataset provided by TenSet, as doing so has negligible benefits to both the accuracy of the cost model and tuning results. We split our dataset into 90% training set and 10% validation set, and use the same hyperparameters as in TenSet code, such as learning rate and batch size.

6 Evaluation

We evaluate the efficacy of Felix on both entire neural networks and single operator benchmarks, while targeting multiple hardware platforms. For each setting, we compare Felix against state-of-the-art search framework (Ansor-TenSet) and off-the-shelf inference frameworks (PyTorch, TensorFlow, TensorRT). In our experiments, we consider the following questions:

RQ1: Can Felix provide higher performance than off-the-shelf inference frameworks? How much optimization search time needs to be allocated for Felix to surpass the performance offered by these inference frameworks?

RQ2: Does Felix produce better or similar level of performance improvements in shorter search time compared to today's practice of discrete-space program search?

RQ3: How do individual operators optimized by Felix compare to those provided by kernel libraries or optimized by Ansor-TenSet?

6.1 Felix vs. Off-the-shelf Inference Frameworks

Figure 6 present the inference performance of 6 DNNs on 3 hardware platforms when using Felix vs. PyTorch, TensorFlow, and TensorRT. Felix improves the performance of the 6 networks on average (geometric mean) by 1.41× on A5000, 1.50× on A10G and 1.70× on Xavier NX, and up to 4.48×, 5.40×, and 10.8× respectively. A few configurations do not produce results: TensorFlow is unable to run the high memory-footprint vision transformer (ViT-B/32) on Xavier NX due to insufficient memory; LLaMA does not run on Xavier NX with any framework due to insufficient memory to hold its parameters; LLaMA is not yet supported in TensorFlow [13] and produces a segmentation fault error with TensorRT.

Table 1. Tuning time (seconds) Felix takes to exceed the performance of the best-performing manual library on different DNNs and hardware devices.

Network	RTX A5000	A10G	Xavier NX
ResNet-50	278 s	416 s	416 s*
MobileNet-v2	496 s	527 s	496 s
DCGAN	495 s	144 s	145 s
ViT	144 s	496 s	145 s*
LLaMA	389 s	—	—

Insights on performance improvements. We observe that Felix usually provides much higher performance compared to PyTorch, TensorFlow, and TensorRT, on network with smaller layers, such as MobileNet-v2 and DCGAN. One reason we identify is smaller neural network layers are harder to parallelize in a way that does not under-utilize GPU compute resources. Even though PyTorch, TensorFlow, and TensorRT have some code generation and autotuning capabilities, Felix explores a larger space of schedules to increase parallelism and cache reuse (among other performance characteristics) and is hence more effective at optimizing in these scenarios. For example, Felix gets significant speedup (3.42×) for MobileNet-v2 and DCGAN (2.25×) on A5000 because MobileNet-v2 consists of many relatively small layers, which is harder to schedule with enough parallelism to fully utilize the 8192 cores of A5000.

Meanwhile, the speedup of Felix over off-the-shelf inference frameworks can depend on the type of operators in the tensor program. Felix did not outperform the baselines on R3d-18 due to 3d convolutions, which make up more than 99% of computation in R3d-18 and are highly optimized in the baselines. Our study in §6.3 confirms that the low performance of Felix on R3d-18 is due to 3d convolutions, and show that Felix still outperforms off-the-shelf frameworks for all other types of operators in the evaluated networks.

Amount of tuning time to surpass the baselines. Table 1 shows the amount of tuning time Felix takes to surpass the best performance among the kernel libraries (PyTorch, TensorFlow, TensorRT). The efficient gradient-based search in Felix allows it to generate programs that outperform PyTorch, TensorFlow, and TensorRT in as little as 2.4 minutes (144 seconds) and on average 6.9 minutes (413 seconds). The asterisk for Xavier NX indicates that we compare the search time against the second best-performing inference framework, as Felix slightly under-performed TensorRT on ViT (by 6%) and ResNet-50 (by 8%). These off-the-shelf inference frameworks can also perform some tuning to achieve

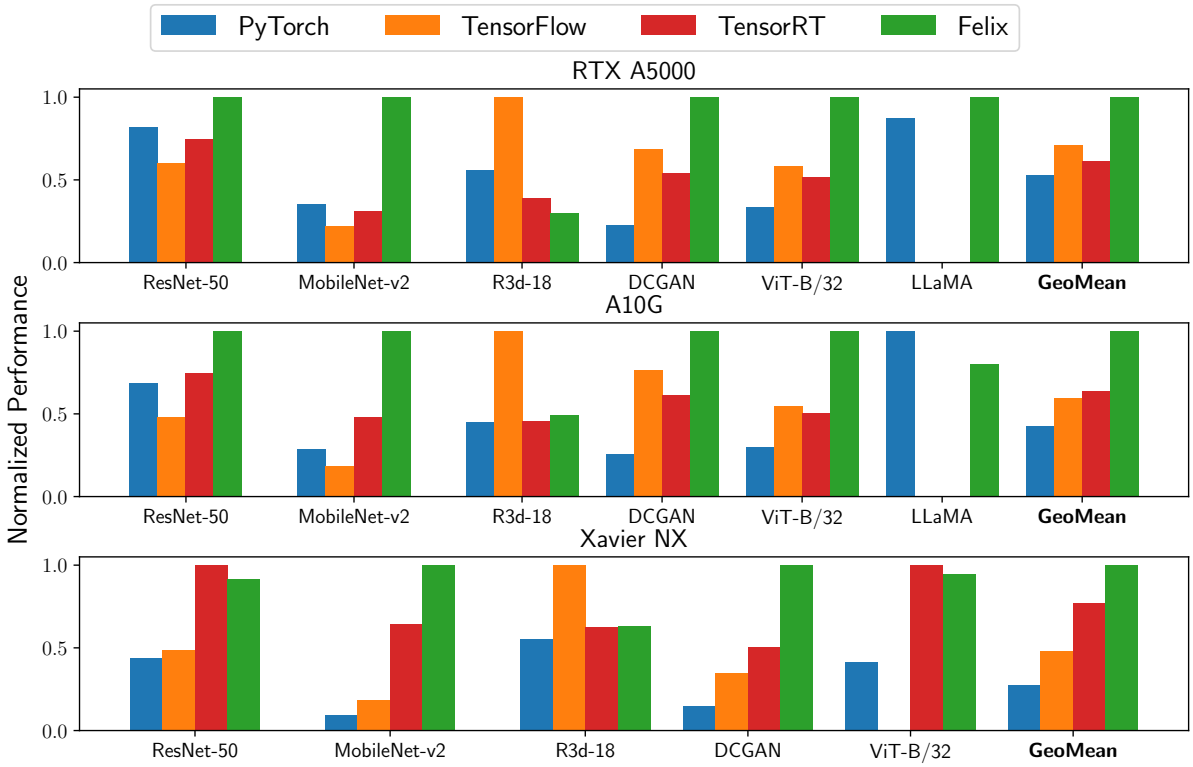


Figure 6. DNN inference performances when using Felix vs. deep learning frameworks (PyTorch, TensorFlow) on three hardware platforms. The y-axis is the performance of one framework normalized to the best performance across all frameworks for a network.

their best performance. When these frameworks invoke autotuning, it takes ~20 seconds to ~7 minutes to optimize one network (depending on the GPU architecture and the tensor operators).

Overall, Felix generates high-performance tensor program that outperforms off-the-shelf inference frameworks in most cases, and does so with low tuning overhead due to its efficient gradient descent search technique.

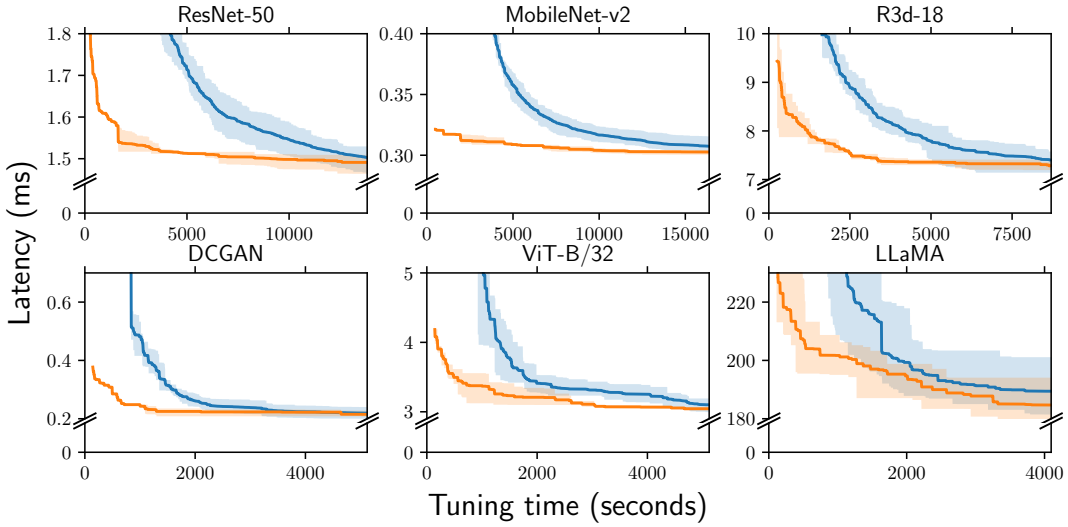
6.2 Felix vs. Ansor-TenSet Search-based Compiler

To answer **RQ2**, we now compare Felix against a state-of-the-art search-based tensor program compiler, Ansor-TenSet. Ansor-TenSet achieves equal or better tuning result than Ansor in less search time (confirmed in the TenSet paper [39] and in our preliminary experiments) and is hence a better baseline.

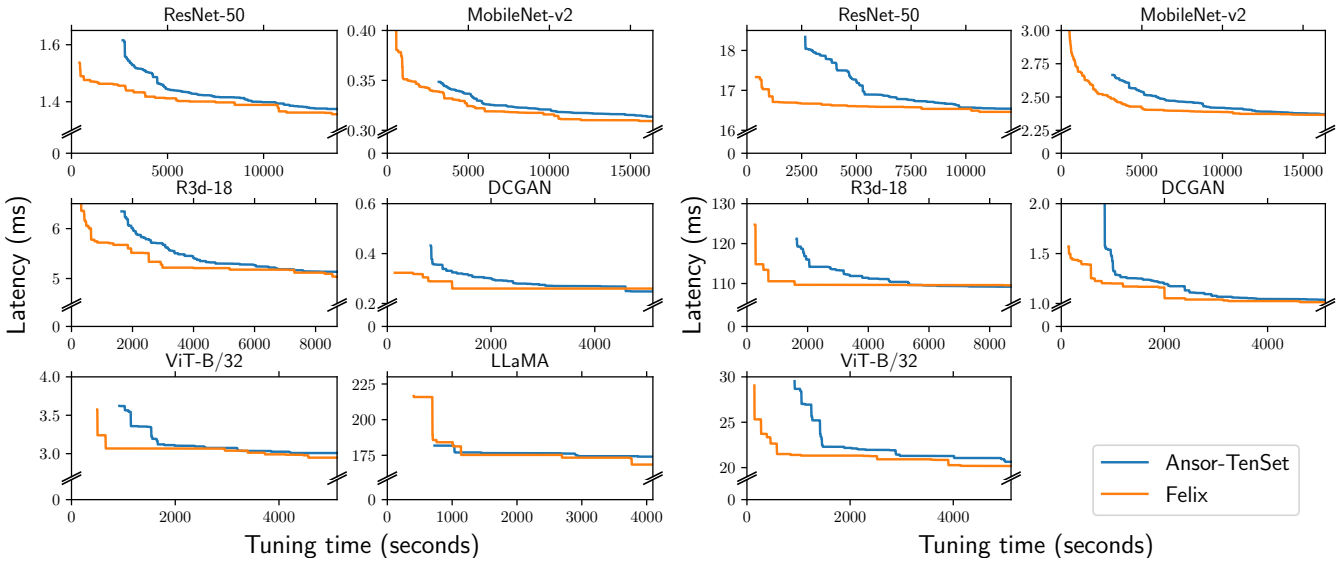
Figure 7 shows how Felix and Ansor's best result improves (latency decreasing, y-axis) over increasing tuning time (in seconds, x-axis) on three hardware architectures. Figure 7a additionally shows the minimal and maximal latency among five runs as a band, where the curve in the band is the mean latency. While Felix and Ansor eventually converge to the same or similar performance levels, Felix finds better schedules in significantly less tuning time owing to its gradient-based optimization process. The reason Ansor and Felix have similar eventual performance improvements is because both use the same program transformations and schedule sketch (described in §3.2).

We summarize the speedup of Felix search compared to Ansor-TenSet in Table 2a. This table compares the search time for Felix and Ansor-TenSet to reach performance milestones: 90%, 95%, and 99% of the best performance achieved in the entire search, which we call 90%, 95%, and 99% *peak performance*. On average, Felix reaches 90% peak performance in 3.4× less search time than Ansor, and 95% peak performance in 2.8× less search time, across the three hardware architectures. DCGAN on A10G is the only case where Felix does not achieve 99% of Ansor's best performance (indicated in the table with a dash), which may not be statistically significant given the stochasticity of tuning as in our other experiments.

Insight on Felix's fast convergence. Felix discovers high-performance program schedules much more early-on than Ansor-TenSet because Felix's gradient-based search optimizes the schedules according to its cost model more efficiently and finds a population of many high predicted-performance schedules in significantly fewer tuning iterations. Moreover, we find that the cost model does not need to perfectly reflect the empirical performance of schedules. If the cost model is a good indicator of performance, some



(a) RTX A5000



(b) A10G **(c)** Xavier NX

Figure 7. Best performance (shown as network inference latency) vs. search time during the schedule search of Felix and Ansor-TenSet.

of the high predicted-performance schedules Felix finds will have high performance in empirical evaluation.

Figure 8 shows how the *predicted* performance of the population of schedules (y-axis) improves as Felix's (orange) and Ansor's (blue) search technique searches an increasing number of schedules (x-axis). At $x = n$, the search technique has searched n schedules $[s_1, s_2, \ldots, s_n]$ whose predicted performances are $p_i := \text{EstmPerf}(s_i)$. The blue and orange lines in the Figure shows the best predicted performance seen among s_i, i.e., $\mathbf{max}([p_1, p_2, \ldots, p_n])$ when $x = n$, and the blue and orange shaded portion shows the 64th best performance, i.e., $\mathbf{sorted}([p_1, p_2, \ldots, p_n])[64]$ when $x = n$. The goal of showing the best 64 predicted schedules is to show the spread of the predicted performance of the search population (closer to the best is better). We run both search techniques on 3 tensor subgraphs taken from the set of DNNs we use for our evaluation, including Conv2d (left), Conv3d (middle), and Dense (right). For a fair comparison, Felix and Ansor have the same search space of program transformations in each subgraph.

The key takeaway of Figure 8 is that the gradient-based search technique in Felix finds more schedules with higher predicted-performance compared to Ansor's evolutionary

Table 2. Tuning speedup of Felix compared to Ansor, measured by how much time Felix and Ansor take to converge to 90%, 95%, and 99% of the best Ansor performance achieved.

(a) Batch size = 1

Network	A5000			A10G			Xavier NX		
peak perf.:	90%	95%	99%	90%	95%	99%	90%	95%	99%
ResNet-50	9.6×	5.2 ×	3.3×	7.4×	1.8×	1.1×	6.4×	11.2×	5.9×
MobileNet-v2	12.0×	16.6×	2.9×	1.9×	1.2×	1.3×	2.5×	2.0×	1.5×
R3d-18	4.2×	2.9 ×	2.7×	2.2×	1.6×	1.4×	5.8×	7.3×	3.4×
DCGAN	2.4×	2.6 ×	1.0×	2.3×	3.7×	—	1.2×	1.5×	1.6×
ViT	3.3×	2.9 ×	1.8×	3.1×	2.5×	1.0×	3.2×	5.0×	1.3×
LLaMA	3.9×	1.5 ×	1.4×	1.0×	0.7×	2.6×	—	—	—
Geomean	5.0×	3.2×	2.0×	2.5×	1.7×	1.4×	3.2×	4.1×	2.3×

(b) Batch size = 16

Network	A5000		
peak perf.:	90%	95%	99%
ResNet-50	10.9×	11.8×	1.8×
MobileNet-v2	6.4×	6.9×	2.0×
R3d-18	3.1×	1.8×	1.5×
DCGAN	14.7×	7.0×	6.6×
ViT	2.0×	2.8×	3.3×
LLaMA	—	—	—
Geomean	5.8×	4.9×	2.6×

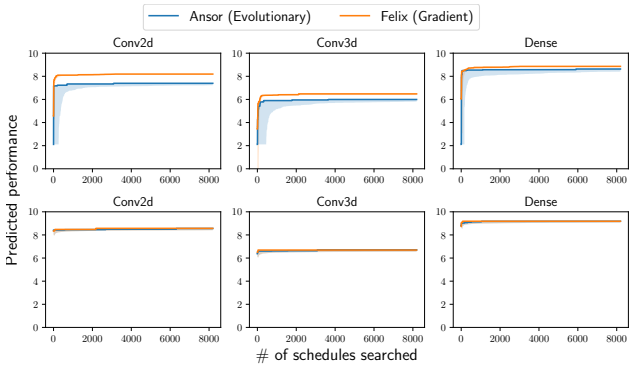


Figure 8. Predicted performance of the candidate schedule population as the search progresses in Felix and Ansor. The top row shows earlier tuning iterations close to the beginning of the tuning session. and the bottom row shows later tuning iterations close to convergence (both tools' lines overlap).

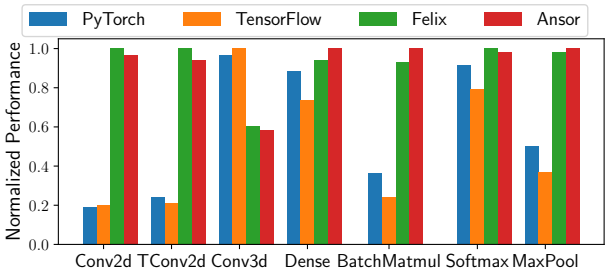


Figure 9. Single operator performance of Felix, Ansor, and manually-optimized libraries (PyTorch, TensorFlow) on RTX A5000. The y-axis is the performance of one framework normalized to the best performance across all frameworks for an operator.

search. Felix's search converges more uniformly than Ansor's search, as indicated by the much smaller orange shaded area (barely visible in the figure) compared to the blue shaded area, and provides a pool of better candidate schedules to be evaluated on hardware. The larger blue shaded area for Ansor shows the randomness in Ansor's search which may sometimes find good configurations since it covers a wider spread over the search space, but is less effective in following the cost model. Most other tensor operator subgraphs follow similar convergence patterns for Felix vs. Ansor.

6.3 Single Operator Performance of Felix vs. Other Tools

Figure 9 shows the performance of search-based compilers (Felix, Ansor) and manually-optimized libraries (PyTorch and TensorFlow) for different types of tensor operators on the RTX A5000 hardware platform. All of these operators are taken from the DNNs we include in our evaluation. Felix outperforms PyTorch and TensorFlow on 7 out of 8 types of operators, and delivers similar performance as Ansor on every type of operator. The only exception is 3D convolution where both PyTorch and TensorFlow outperform Felix and Ansor.

As discussed in 6.1, Felix usually outperforms off-the-shelf inference frameworks on small and uncommon neural network layers/operators. The hand-tuned implementation for 3D convolution is quite efficient because its a commonly occurring tensor operation, hence a significant amount of manual development and optimization effort has been spent to optimize it. Moreover, 3D convolutions usually work with large tensor shapes and is thus easier to parallelize using manual tuning. However, the performance of 3D convolution in PyTorch and TensorFlow significantly differs across the evaluated hardware platforms which is because the code has to be tuned differently to get high performance for each platform. In contrast, Felix automates the hardware-specific tuning of tensor operators and does not need any manual tuning.

6.4 Felix vs. Ansor under Different Batch Sizes

To demonstrate that Felix is also effective in bulk inference scenarios, we now compare Felix against Ansor-TenSet on the same 6 networks using a input batch size of 16. Figure 10 shows how Felix and Ansor's best result improves over

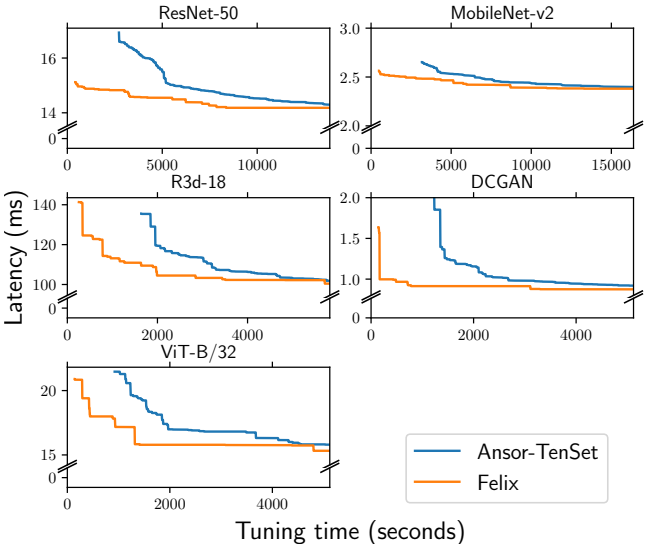


Figure 10. Best performance (network inference latency) vs. search time during the schedule search of Felix and Ansor-TenSet. The networks are optimized for RTX A5000 with a input batch size of 16.

increasing tuning time on RTX A5000. This experiment excludes LLaMA, as a batch size of 16 with LLaMA requires more GPU memory for inference than is available on RTX A5000. Similar to §6.2, Table 2b shows the speedup of Felix search over Ansor-TenSet for reaching 90%, 95%, and 99% peak performance. In this setting, Felix on average (geomean over the networks) reaches 90% peak performance in 5.8× less search time than Ansor-TenSet and 95% peak performance in 4.9× less search time. These results indicate that Felix provides fast convergence to high-performance schedules even when the input batch size is increased.

7 Related Work

Automatic tensor program generation. A number of tensor program compilers, including Halide [28], TVM [7], and Tiramisu [4], provide the capability to define the computation and the schedule separately. These compilers provide a schedule language or API to specify optimization primitives, and then the schedule that includes these optimization primitives is either developer-specified or provided by automatic search (autotuning). Halide includes three search-based auto-schedulers [2, 21, 25], based on different tuning techniques, each of which automatically discovers good schedules. Similarly, TVM has three different tuning systems: AutoTVM [7], Ansor [38], and MetaSchedule [31]. These auto-schedulers utilize combinatorial discrete search algorithms, such as genetic algorithm (e.g., in [31, 38]), tree search (e.g., in [2]), or brute force in a limited search space (e.g., in [25]), to discover schedules with good performance. These tuning techniques have also been extended to exploring trade-offs between performance and accuracy of tensor computations, while still using discrete search algorithms [32, 37]. In contrast to all of these works, Felix instead applies gradient descent on the tensor program schedule directly as the search technique, based on the insight that gradient descent is commonly efficient for smooth functions. While we have shown Felix to be effective for tensor program optimization, the efficacy of gradient descent depends on the domain of the problem [29], and there is no single best optimization algorithm.

Among TVM tuning systems, we used Ansor [38] as our baseline for comparison, since it outperforms AutoTVM as shown in their evaluation and now supersedes AutoTVM in the TVM ecosystem. MetaSchedule supports automatic sketch generation for new program transformations, and thus allows the optimizer to easily adapt to the growing search space. MetaSchedule uses a genetic search algorithm for optimization and is thus orthogonal to the contribution of this paper. Importantly, Felix is complementary to these existing tensor program compilers. While we implement Felix on top of the Ansor framework, the gradient-based search techniques are also applicable to other optimization systems.

Gradient-based optimization for DNN search problems. MindMappings [16] introduces a gradient-based algorithm for *mapping space search* for custom accelerators: optimizing schedule parameters involved in mapping a tensor algorithm to custom accelerators, such as data tile size, to improve performance. MindMappings trains a differentiable surrogate cost model from the schedule parameters directly to the estimated cost, and applies gradient descent on the surrogate. The cost model is tied to the schedule parameters of the specific operator. Thus unlike Felix, for *every type of operator*, MindMapping has to collect a new dataset and train a separate cost model, since the schedule parameters change significantly across operators (e.g., Conv2d has different loop tiling and loop ordering parameters from Matmul). This process is time-consuming and requires manual efforts. Felix instead uses a single pretrained cost model to optimize all tensor operators fully automatically. As a second distinction, the search space of MindMappings is tailored to programmable accelerators with parameters such as the percentage of banks allocated to each tensor, and does not apply to pre-existing GPUs and CPUs. Finally, MindMappings is not a tensor compiler (no code generation capabilities, no empirically measured costs on hardware), while Felix produces CUDA code (using TVM) that runs on commodity GPUs.

Gradient techniques are routinely used for optimization problems, such as DNN training or hyperparameter tuning. While gradient techniques are mostly applied to optimization problems that are continuous and differentiable by formulation, some techniques relax discrete optimization problems into continuous ones and apply gradient techniques to replace discrete search. DARTS [22] is a neural architecture search algorithm which relaxes a discrete set of candidate architectures into a continuous space, and uses gradient

descent to optimize network architecture and weights simultaneously. SSL [35] optimizes structured pruning choices in DNNs by relaxing discrete masks over DNN weights into continuous ones and optimize these masks by gradient descent. Although these techniques also create differentiable optimization problems from discrete ones, the objective functions are often already analytical expressions that are readily differentiable after some hand-written mathematical reformulation of the problem. Felix is unique in that it automatically creates symbolic expressions for program features and converts these to differentiable counterparts for optimization using gradient descent.

Automatic differentiation. Automatic differentiation (AD) is a technique that automatically generates gradients for a given program. All deep-learning frameworks that allow training provide AD capability on DNNs, such as PyTorch [26], TensorFlow [1], and MXNet [6]. A Halide scheduler [21] automatically derives gradients for image processing pipelines developed in Halide. General AD frameworks, such as Enzyme [24], can derive gradients at the LLVM IR level. These AD frameworks generate gradients for a given program where the program itself represents an objective function to be optimized with regard to its inputs. Thus, these systems have entirely different goals from Felix, which generates the gradients of the *schedule parameters* of the given program to optimize its execution.

8 Conclusion

We present Felix, a novel gradient-based compiler optimization framework for tensor-based programs. We show how to create differentiable space of program transformation schedules and use gradient descent to find high-performance schedules. The novel aspects of Felix optimization algorithm are the careful continuous relaxation of the space of schedules and the generation of a differentiable cost function, that is amenable to optimization with gradient descent. Our evaluation shows that the efficient gradient-based search in Felix can quickly find better optimized tensor programs than those offered by PyTorch, TensorFlow, and TensorRT. Moreover, we show the benefits of Felix over state-of-the-art evolutionary search: it generates high-performing programs significantly faster than TVM Ansor. Our evaluation demonstrates that Felix can be used widely, across commodity GPUs from server, desktop, and resource-constrained edge devices.

We believe that the main idea behind Felix – optimizing computation graphs by variable relaxation and numerical optimization – is general and can be implemented on top of other tensor compilers that operate on computation graphs (e.g. Halide [28], MLIR [20], TACO [18], Tiramisu [4]). We see our approach as the first step toward gradient-based optimization for a broader class of programs in general programming languages, with more complicated cost features and control flow.

Acknowledgements

We thank the anonymous reviewers and our shepherd Albert Cohen for their comments. This research was supported in part by the National Science Foundation (Grants No. CCF-1846354, CCF-2217144, CCF-2313028).

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38, 2019.

[3] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman P. Amarasinghe. A deep learning based cost model for automatic code optimization. *CoRR*, abs/2104.04955, 2021.

[4] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.

[5] Swarat Chaudhuri and Armando Solar-Lezama. Smooth interpretation. *ACM Sigplan Notices*, 45(6), 2010.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[8] NVIDIA Corporation. NVIDIA A10 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/products/a10-gpu/, 2024.

[9] NVIDIA Corporation. NVIDIA Jetson Xavier. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/, 2024.

[10] NVIDIA Corporation. NVIDIA RTX A5000 Graphics Card. https://www.nvidia.com/en-us/design-visualization/rtx-a5000/, 2024.

[11] Corinna Cortes, Mehryar Mohri, and Afshin Rostamizadeh. L2 regularization for learning kernels. *CoRR*, abs/1205.2653, 2012.

[12] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *CoRR*, abs/2010.11929, 2020.

[13] Hugging Face. Transformer Supported Frameworks (2023). https://huggingface.co/docs/transformers/index#supported-frameworks.

[14] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. Learning spatio-temporal features with 3d residual networks for action recognition. *CoRR*, abs/1708.07632, 2017.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[16] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W Fletcher. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *ASPLOS*, 2021.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[18] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[19] Paolo Sylos Labini, Marco Cianfriglia, Damiano Perri, Osvaldo Gervasi, Grigori Fursin, Anton Lokhmotov, Cedric Nugteren, Bruno Carpentieri, Fabiana Zollo, and Flavio Vella. On the anatomy of predictive models for accelerating gpu convolution kernels and beyond. *ACM Trans. Archit. Code Optim.*, 18(1), jan 2021.

[20] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020.

[21] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.*, 37(4), 2018.

[22] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018.

[23] Charith Mendis, Cambridge Yang, Yewen Pu, Saman Amarasinghe, and Michael Carbin. Compiler auto-vectorization with imitation learning. *Advances in Neural Information Processing Systems*, 32, 2019.

[24] William Moses and Valentin Churavy. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

[25] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4), jul 2016.

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[27] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.

[28] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[29] R. Salomon. Evolutionary algorithms and gradient search: similarities and differences. *IEEE Trans. on Evolutionary Computation*, 2(2), 1998.

[30] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[31] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

[32] Hashim Sharif, Yifan Zhao, Maria Kotsifakou, Akash Kothari, Ben Schreiber, Elizabeth Wang, Yasmin Sarita, Nathan Zhao, Keyur Joshi, Vikram S Adve, Sasa Misailovic, and Sarita V Adve. Approxtuner: a compiler and runtime system for adaptive approximations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[33] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[34] Yao Wang, Xingyu Zhou, Yanming Wang, Rui Li, Yong Wu, and Vin Sharma. Tuna: A static analysis approach to optimizing deep neural networks. *CoRR*, abs/2104.14641, 2021.

[35] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *CoRR*, abs/1608.03665, 2016.

[36] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[37] Yifan Zhao, Hashim Sharif, Peter Pao-Huang, Vatsin Ninad Shah, Arun Narenthiran Sivakumar, Mateus Valverde Gasparino, Abdulrahman Mahmoud, Nathan Zhao, Sarita Adve, Girish Chowdhary, Sasa Misailovic, and Vikram Adve. Approxcaliper: A programmable framework for application-aware neural network optimization. In *Proceedings of Machine Learning and Systems 5*, 2023.

[38] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020.

[39] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. Tenset: A large-scale program performance dataset for learned tensor compilers. In *NeurIPS; Datasets and Benchmarks Track*, 2021.

[40] Tao Zhuang, Zhixuan Zhang, Yuheng Huang, Xiaoyi Zeng, Kai Shuang, and Xiang Li. Neuron-level structured pruning using polarization regularizer. In *Advances in Neural Information Processing Systems*, volume 33, 2020.