# Automated linear algebra program optimization using ML

Learning “Learning to Optimize Tensor Programs”

Daniel Zheng
Purdue University

## Abstract

Traditionally, the deep learning community uses libraries like cuBLAS and cuDNN to achieve high performance. These libraries provide manually optimized linear algebra primitives for a narrow set of hardware architectures. However, there is an increasing demand to bring deep learning to new hardware architectures, each with diverse memory and compute characteristics. Optimizing operations for each architecture requires significant effort and hardware expertise.

An alternative to high performance libraries is compiler-based approaches, which can compile arbitrary computation to various hardware backends and apply hardware-specific optimizations. TVM [3] is one such deep learning compiler: it features scheduling primitives for transforming low-level program execution details. TVM also provides an autotuning framework which uses statistical cost models that predict program performance to search for optimal program schedules.

This project aims to improve TVM autotuning via improvements to the statistical cost model and search procedure. These improvements are evaluated on single operator tuning and transfer learning tasks.

## 1 Introduction

Deep learning has become ubiquitous, achieving success in a wide range of real-world applications from computer vision to natural language processing. Tensor operations, like matrix multiplication and convolution, are essential building blocks for deep learning models.

High-performance libraries like MKL, cuBLAS, and cuDNN are commonly used in the machine learning community for their efficient tensor operation implementations. operations. These libraries are integrated into popular machine learning frameworks and have led to improved performance and reduced memory consumption for production deep learning models [4]. However, such libraries have some limitations:

- Library functions are manually optimized and highly specialized for specific hardware architectures. Thus, they cannot easily be ported to other architectures with different memory and compute capabilities. This is an important challenge given the growing need to support deep learning on new architectures from mobile devices to custom accelerators. Manual optimization approaches requires significant effort and must be done for each architecture.
- Libraries provide only a limited set of primitives. Implementing custom functions that deviate from library-supported primitives requires significant effort and expertise. This makes it difficult for machine learning researchers to bring experimental tensor operations to production-grade performance.

Recently, there has been a trend towards “deep learning compilers”, which take a general and composable approach towards tensor program optimization. TVM [3] is one such deep learning compiler. It features a set of scheduling commands for exploring common program transformations, like loop ordering and loop tiling. Schedules can be parameterized with tunable knobs to create a search space of possible schedules. Thus, program optimization is defined as the problem of searching for optimal schedules. TVM provides an autotuning module with various schedule exploration strategies, including a statistical cost model approach that uses simulated annealing to find good schedules.

This project aims to improve automated program optimization in TVM, specifically via improvements to statistical cost models and the search procedure. These improvements are evaluated in-domain for single operator tuning and cross-domain for transfer learning tasks to test cost model generalization.

## 2 Related work

**Halide** [7] is a high-performance image processing language. It introduced the idea of explicitly decoupling computation from scheduling. This enables programmers to define algorithms and explore optimizations separately and provides a natural setting for program optimization. Halide provides an autoscheduling tool that automatically identify schedules that improve parallelism and locality [6]; however, it currently only works well for a limited set of CPU programs. Halide is not a polyhedral framework: it uses intervals to represent iteration bounds.

**Tensor Comprehensions** (TC) [10] compiles a high-level mathematical language to Halide IR, then lowers to a polyhedral representation and applies polyhedral transformations

*Purdue CS590-DLS, Dec 2019, West Lafayette, Indiana*
2019.

to generate optimized code. Tensor Comprehensions supports autotuning using genetic search rather than cost model based techniques: this limits tuning efficiency. A key feature of TC is automatic scheduling, which alleviates the burden of manual schedule space specification.

**Tiramisu**[1] is a polyhedral compiler that generates efficient code for heterogeneous architectures. It features a four-stage IR with a clear separation of concerns. Staging enables optimizations to be applied in a logical order (scheduling, memory, synchronization), avoiding phase conflicts. Currently, Tiramisu does not provide any mechanism for automatic scheduling.

**PlaidML** [1] is a framework that provides automatic kernel generation for GPUs. It uses a hardware-aware cost model to generate code for CPU and GPU. This is in contrast to TVM, which intentionally performs hardware-agnostic tuning.

## 3 TVM overview

### 3.1 TVM architecture

TVM [3] is a deep learning compiler infrastructure that compiles high-level neural network specifications to various hardware targets. Following from Halide, TVM adopts the principle of separating computation from scheduling. This enables programmers to define and optimize programs separately: the computation of an algorithm is defined once, then optimization is simply a matter of searching over possible schedules.

TVM uses a tensor expression language for defining computation. Tensor operations are defined in the language using index-based formulas, which specify output tensor shapes and expressions for computing output elements.

TVM also features a rich set of scheduling primitives that map high-level computation is to low-level code. Scheduling commands involve loop transformations (e.g. loop ordering and tiling), mapping loop levels to hardware (e.g. vectorization), and memory manipulation (e.g. caching, shared memory, latency hiding), among other transformations.

Additionally, schedules can be parameterized: rather than specifying concrete arguments to scheduling primitives, one can define schedules with tunable knobs to create a schedule search space. An example tunable schedule is shown in Figure 1.

### 3.2 TVM autotuning

Given a schedule search space, program optimization becomes a matter of finding the optimal schedule for a specific input and hardware configuration. One natural strategy is blackbox autotuning: exhaustively evaluate all possible schedules to find the fastest one. However, blackbox autotuning may take many iterations to find fast schedules. It does not leverage any program- or hardware-specific knowledge to accelerate tuning.

[1]https://github.com/plaidml/plaidml

```
# Define computation: specify output expression and shape.
C = tvm.compute((512, 512), lambda x, y:
        tvm.sum(A[x, k] * B[k, y], axis=k))
# Create schedule.
s = tvm.create_schedule(C.op)
# Define parameterized schedule search space:
# split loop axes to perform tiling.
cfg = autotvm.get_config()
cfg.define_split("tile_x", x, num_outputs=2)
cfg.define_split("tile_y", y, num_outputs=2)
# Use schedule from search space.
xo, xi = cfg["tile_x"].apply(s, C, x)
yo, yi = cfg["tile_y"].apply(s, C, y)
s[C].reorder(yo, xo, k, yi, xi)
```

**Figure 1.** An example tunable schedule in TVM. First, computation is defined in the tensor expression language. Then, a schedule space for the computation is defined via tunable parameters. In this example, the schedule space size is $10 \times 10 = 100$. For practical deep learning operations, the schedule space size is on the order of $10^7$.

Another strategy is to develop hardware-specific cost models that predict good schedules based on hardware characteristics that affect performance, like cache sizes and compute/memory bandwidths. This approach is adopted by some autotuning frameworks [2]. Ideally, a perfect hardware cost model can predict schedule costs exactly; but in practice, developing such models is difficult and non-portable due to the complexity and diversity of hardware.

Instead, TVM uses statistical cost models to perform autotuning. Cost models are trained to predict the running time of low-level programs.

The detailed autotuning process works as follows (Figure 2):

- Propose a batch of program schedules for exploration.
- Compile the proposed schedules and measure their running times. Add schedules/running times to data.
- Train the cost model on all collected data.
- Perform simulated annealing to propose new schedules. The cost model is used as the energy function, so programs predicted to have a low cost are more likely to be explored.
- Repeat $n$ iterations. Return the best explored schedule at the end.

Compared with blackbox optimization and hardware-specific cost models, the statistical cost model approach used by TVM is hardware-agnostic, has a relatively low search cost, and learns from historical performance data.

## 4 Experiments

This paper explores three experiments in automated program optimization within the context of TVM:

- A replication of TVM results and comparison with cuDNN baselines.

[2]https://ai.intel.com/automatic-kernel-generation-in-plaidml

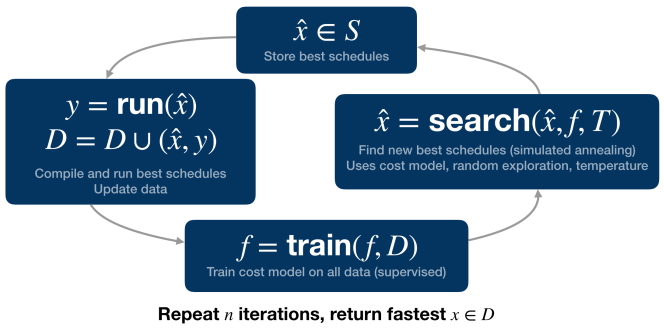
$\hat{x} \in S$
Store best schedules
$y = \mathbf{run}(\hat{x})$
$D = D \cup (\hat{x}, y)$
Compile and run best schedules
Update data
$\hat{x} = \mathbf{search}(\hat{x}, f, T)$
Find new best schedules (simulated annealing)
Uses cost model, random exploration, temperature
$f = \mathbf{train}(f, D)$
Train cost model on all data (supervised)
Repeat $n$ iterations, return fastest $x \in D$

**Figure 2.** TVM autotuning pipeline. Note that the cost model performs supervised learning on schedule/cost data.

- Improvements to the statistical cost model.
- An evaluation of cost model transfer learning between operator domains.

## 4.1 TVM replication

First, the TVM autotuning framework was evaluated on `conv2d` operators from ResNet-18 and compared with cuDNN baselines to identify bottlenecks. `conv2d` configuration details are shown in Table 1.

| Operator name | C7 | C12 |
|---|---|---|
| H, W | 28, 28 | 7, 7 |
| IC, OC | 128, 256 | 512, 512 |
| K, S | 3, 2 | 3, 1 |
| Schedule space size | $7.46 \cdot 10^7$ | $1.04 \cdot 10^7$ |

**Table 1.** `conv2d` operator configurations used in evaluation. These configurations come from ResNet-18 inference. $H, W$: input height and width. $IC, OC$: number of input and output channels. $K, S$: kernel size and stride.

TVM claims that search strategies can readily find `conv2d` operator schedules that surpass performance of cuDNN: random search can surpass cuDNN in 350 trials and ML cost model based tuning can surpass cuDNN in just 60 trials [3]. This is surprising given that cuDNN functions are manually tuned and highly optimized.

| | C7 | C12 |
|---|---|---|
| cudNN baseline | 0.0000744 | 0.000153 |
| ML cost model (after 100 trials) | 0.0002260 | 0.000255 |
| ML cost model (after 1000 trials) | 0.0000752 | 0.000173 |
| ML cost model (after 2000 trials) | 0.0000636 | 0.000126 |

**Table 2.** `conv2d` operator performance comparison (seconds). The average operator performance from three tuning attempts is shown. Tuned operator performance exceeds cuDNN only after 2000 trials. Tuning starts with no training data using the `itervar` feature type.

Experiments confirm the effectiveness of TVM's approach to autotuning: by specializing on input sizes, it is possible to achieve faster than cuDNN performance on specific operator configurations. Cost model based tuning exceeds cuDNN performance on `conv2d` operators, but only after 1000-2000 tuning iterations (Table 2). This is significantly more iterations than reported in the original TVM paper. A likely explanation is that the original experiments used a smaller schedule space, enabling search strategies to find good schedules much more quickly. This suggests that good schedule space specifications are crucial for efficient autotuning. However, TVM currently has no mechanism for automatically extracting good schedule spaces for arbitrary computation: users have the burden of manually selecting schedule knobs.

Experiments also show a high variance in tuning results: tuners take an inconsistent number of tuning iterations to converge on good schedules. This may be due to the nature of simulated annealing, which uses random exploration to propose new schedules. Hyperparameter tuning for simulated annealing (e.g. a temperature cooling schedule) may yield more consistent tuning results.

Operator performance also varies greatly between invocations: operators must be run many times for an accurate cost measurement. Overally, tuning is quite time-intensive: it takes 40 minutes to 2 hours to tune 1000 iterations for specific `conv2d` configuration on a GeForce GTX 1080 Ti.

## 4.2 Cost model improvements

TVM uses statistical cost models to guide schedule space search. These cost models learn to predict schedule costs by extracting features from schedules and generated loop programs ASTs.

Currently, cost models use one of three feature types:

- `knob`, which extracts features directly from a schedule configuration (e.g. loop split sizes, ordering permutations, annotations).
- `itervar`, which extracts features from the loop program AST generated from a schedule (e.g. memory access count and memory buffer reuse ratio).
- `curve`, which samples points from distributions modeling the relation between various AST features (e.g. the relation between loop axis length and memory reuse ratio).

In theory, AST-related features like `itervar` and `curve` have better potential to generalize across domains than `knob`, which uses schedule-space specific features.

The implementation of `curve` did not match the description by TVM authors. In particular, the implementation samples only one point from the feature relation curve: this results in sparse features (with many zeros) that is unlikely to capture much information. The implementation of `curve` was fixed to sample multiple points from relation curves ($n = 30$ was used in experiments).

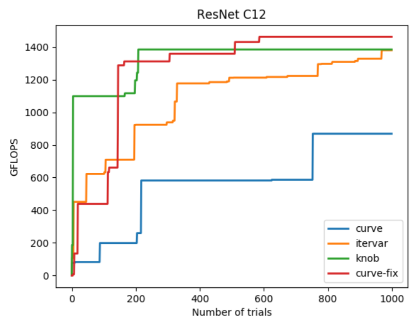

**Figure 3.** Tuning comparison for the C12 operator. The fixed `curve-fix` feature type performs significantly better than `curve`.

| Feature type | C12 |
|---|---|
| `knob` | 34:42.13 |
| `itervar` | 51:40.81 |
| `curve` | 1:21:33.69 |
| `curve-fix` | 1:13:34.58 |

**Table 3.** Feature type tuning time comparison (1000 tuning iterations).

Additionally, an ablation study was performed to find the best combination of features for the `curve` feature. In the end, a combination of orthogonal features yielded the best performance: memory buffer touch count vs memory reuse ratio. The corrected `curve-fix` feature performed significantly better than `curve` during tuning. Though `curve-fix` samples more points from feature relation distributions than `curve`, the impact on tuning time was negligible as tuning time is vastly dominated by operator execution time.

## 4.3 Transfer learning evaluation

An effective deep learning compiler must optimize tensor programs for different input sizes and data layout configurations. The similar structure of tensor programs offers a great opportunity for cost model transfer learning: a statistical cost model trained on one program configuration should be able to accurately predict schedule costs for another program configuration. This cross-domain transfer is important for creating cost models that learn general program features affecting performance.

Cost model transfer learning was evaluated in two scenarios. First, in-domain tuning was performed on the C7 operator without history. Then, transfer learning was performed: a pretrained model on the C12 operator was used as a base for tuning the C7 operator.

Results confirm that cost model transfer learning works well for all feature types: tuning performance improves sharply within the first 100 iterations, suggesting that the transfer learning model can accurately predict schedule costs in the target domain.

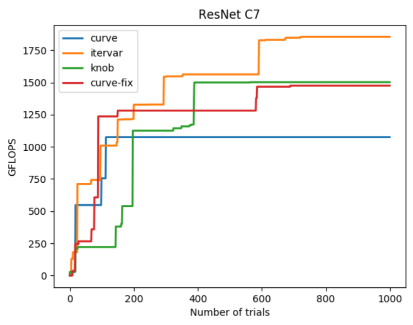

**Figure 4.** In-domain tuning for the C7 operator.

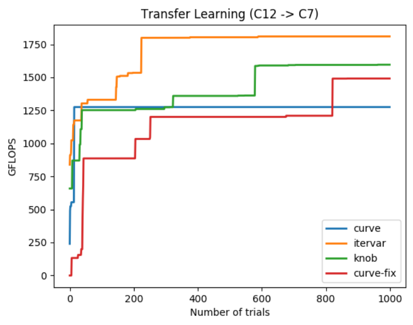

**Figure 5.** Transfer learning from the C12 operator to the C7 operator: initial performance improves sharply. This demonstrates cost model generalization and has the potential to greatly reduce tuning time.

In the transfer learning scenario, the `itervar` and `knob` feature types seem to outperform `curve` and `curve-fix`. This is likely due to the low domain distance between the C7 and C12 operators: specialized features that directly capture schedule-specific information perform better than domain invariant features.

Future work is necessary to investigate cost model transferability across operator domains (e.g. transferring `matmul` cost model to `conv2d`). This is currently difficult in TVM given that feature lengths dffer between schedule spaces for different operators. TVM authors explored the use of recursive program embeddings as cost model features but did not evaluate them on cross-operator transfer tasks.

## 5 Future work

One direction for future work involves improvements to the search procedure. TVM uses a vanilla implementation of parallel simulated annealing to explore new schedules. Optimization techniques like cooling schedules and restarts may help tuning converge more quickly and consistently [5, 8]. Also, the role of cost models in tuning is quite limited: they are used only to predict specific schedule costs to guide search. Expanding the scope of cost models in tuning (e.g.

extending models to directly predict good schedules) can improve transferability and accelerate tuning performance.

Another direction involves improvements to the search specification. Currently, TVM relies on manually defined schedule space templates that capture a wide range of program transformations. Suboptimal templates limit both optimal program performance and tuning speed. There is much prior work on automatically identifying schedules that optimize parallelism and locality [2, 6, 9]: these approaches could help guide automatic schedule space generation and improve autotuning search spaces.

## 6 Conclusion

The project was an exploration of learning to optimize programs within the context of the TVM deep learning compiler. Experiments compared TVM performance with cuDNN baselines and evaluated cost model feature improvements on single operator tuning and transfer learning tasks. Results show that autotuned operators are able to surpass cuDNN performance by specializing on input sizes. Schedule- and AST-specific features outperform relational features on same-operator transfer learning tasks due to the low domain distance. Overall, TVM autotuning is a sound approach for automated program optimization: cost models are able to significantly decrease tuning time without relying on hardware details. Further improvements to the search procedure and search specification have the potential to decrease tuning time and improve cost model transferability.

## References

[1] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2018. Tiramisu: A Code Optimization Framework for High Performance Systems. *CoRR* abs/1804.10694 (2018).

[2] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, Vol. 43. ACM, 101–113.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018).

[4] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014).

[5] F Mendivil, R Shonkwiler, and MC Spruill. 2001. Restarting search algorithms with applications to simulated annealing. *Advances in Applied Probability* 33, 1 (2001), 242–259.

[6] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 83.

[7] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.

[8] D Janaki Ram, TH Sreenivas, and K Ganapathy Subramaniam. 1996. Parallel simulated annealing algorithms. *Journal of parallel and distributed computing* 37, 2 (1996), 207–212.

[9] Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. 2012. Joint scheduling and layout optimization to enable multi-level vectorization. *IMPACT, Paris, France* (2012).

[10] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018).