ML For Systems - I

- Shaurya Patel
- Shashwat Singh

Learning to Optimize Tensor Programs

Paper by -Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, Arvind Krishnamurthy

Presenter: Shaurya Patel

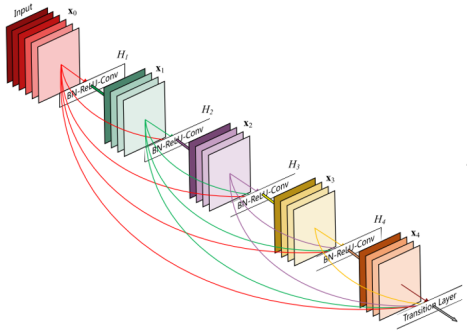
Outline

- Problem description and formalization
- Related and prior work
- Proposed solution
- Experiments
- Conclusion

Basic problem - Performance is hard

- Julius Caesar, year 42.

Learning to optimize tensor programs - Problem formalization



$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \otimes \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}$$

Low-level code

s1

s2

s3

Edge FPGA

Cloud FPGA

ASIC

Why is this not trivial?

e compute expression

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024),
    lambda y, x:
    t.sum(A[k, y] * B[k, x], axis=k))
```

x_0 default code

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

s_1 loop tiling

```
yo, xo, yi, xi = s[C].title(y, x, ty, tx)
s[C].reorder(yo, xo, k, yi, xi)
```

$x_1 = g(e, s_1)$

```
for yo in range(1024 / ty):
  for xo in range(1024 / tx):
    C[yo*ty:yo*ty+ty][xo*tx:xo*tx+tx] = 0
    for k in range(1024):
      for yi in range(ty):
        for xi in range(tx):
          C[yo*ty+yi][xo*tx+xi] +=
            A[k][yo*ty+yi] * B[k][xo*tx+xi]
```

s_2 tiling, map to micro kernel intrinsics

```
yo,xo,ko,yi,xi,ki = s[C].title(y,x,k,8,8,8)
s[C].tensorize(yi, intrin.gemm8x8)
```

$x_2 = g(e, s_2)$

```
for yo in range(128):
  for xo in range(128):
    intrin.fill_zero(C[yo*8:yo*8+8][xo*8:xo*8+8])
    for ko in range(128):
      intrin.fused_gemm8x8_add(
        C[yo*8:yo*8+8][xo*8:xo*8+8],
        A[ko*8:ko*8+8][yo*8:yo*8+8],
        B[ko*8:ko*8+8][xo*8:xo*8+8])
```

Why is this not trivial?

- Search space for low level programs could be billions for a single operator.
- Hardware backends are becoming increasingly complicated and different.
- Lots of variables controlling performance on specific hardware backend - threading, memory reuse, pipelining etc.

Problem Formalization

index expression : $C_{ij} = \Sigma A_{ki} B_{kj}$

ε denotes the space of all index operators

For a give e ϵ ε

We could have s ϵ S_e

Where S_e denotes space of all possible low level programs.

let $x = g(e, s)$ where g represents a compiler framework

$f(x)$ represents the real runtime cost on the hardware.

For a given tuple (g, e, S_e, f) we want

$argmin_{s \epsilon S_e} f(g(e, s))$

Similarity to hyperparameter optimization

- The problem we are trying to solve is similar to hyperparameter optimization with some key differences
- Low experiment cost - Hyperparameter optimization in most cases uses grid or random search methods which can take a while. Cost of training and compiling a tensor program is a few seconds on the other hand.
- Domain specific problem structure - Hyperparameter optimization is a black-box but rich structure of programs can be leveraged.
- Similar operators - A large quantity of similar operators gives way to transfer learning.

Prior work and current methods

- Current tensorflow, pytorch etc. all use cuDNN for this optimization.
- cuDNN contains hand optimized low-level code for target architectures.
- The engineering effort behind this is a lot and its manual.

Proposed Solution - AutoTVM

- The compiler framework used here to generate

  The space of all low-level programs is TVM.

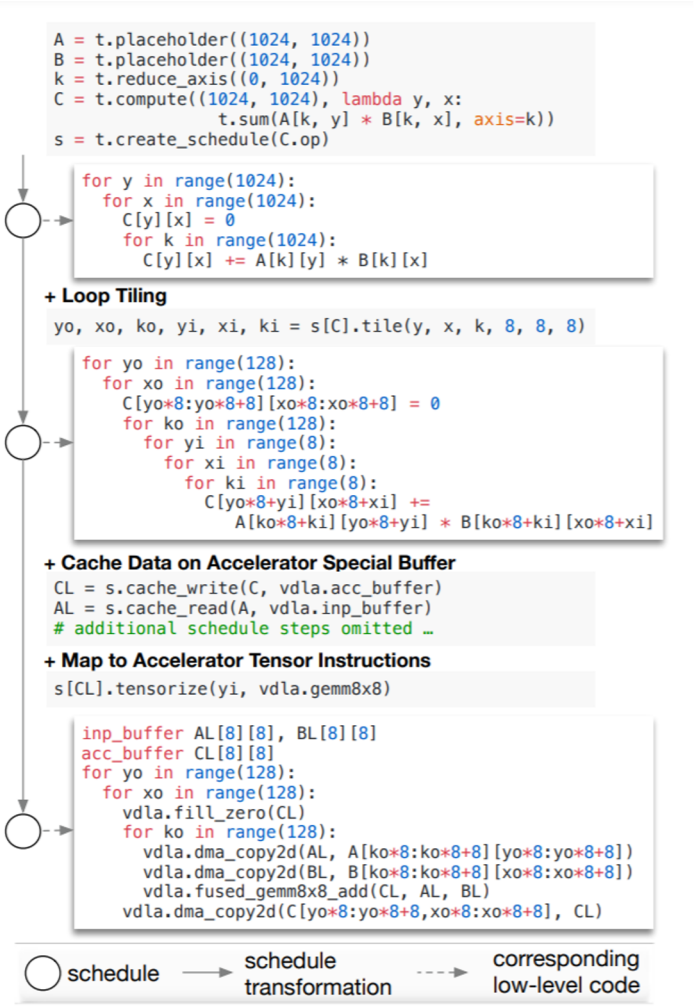
  The search space for possible schedules includes

  multi-level tiling on each loop axis, loop ordering,

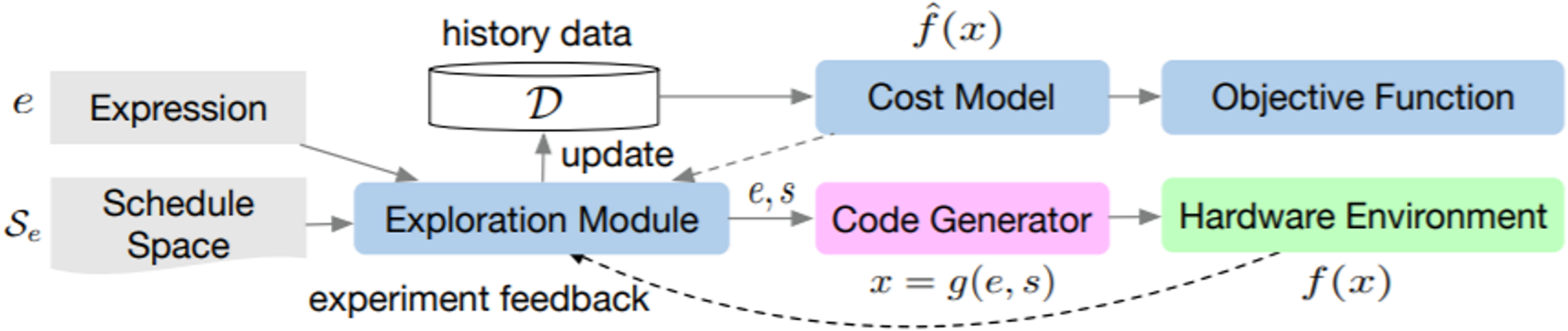
  shared memory caching for GPUs,

  and annotations such as unrolling

  and vectorization.

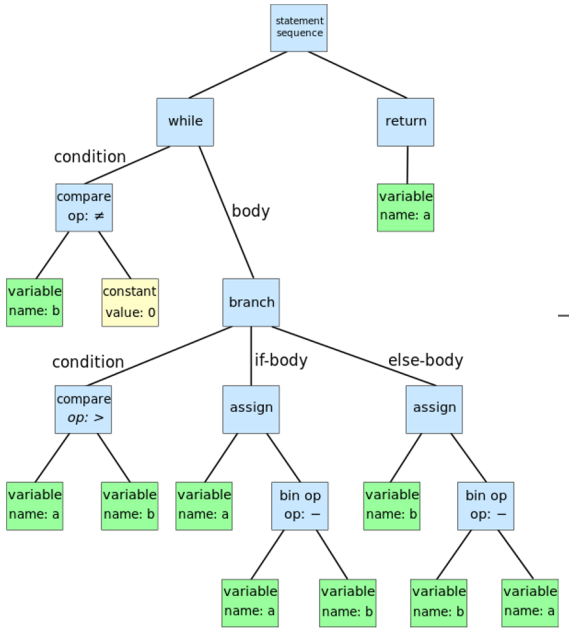


Proposed Solution - High level implementation



Deep Dive - Statistical Model XGBoost

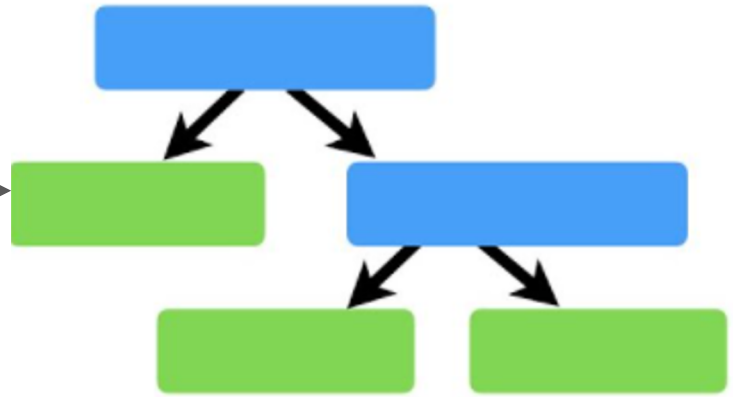
Abstract syntax tree



Features

memory
accesses
data reuse ratio
etc.

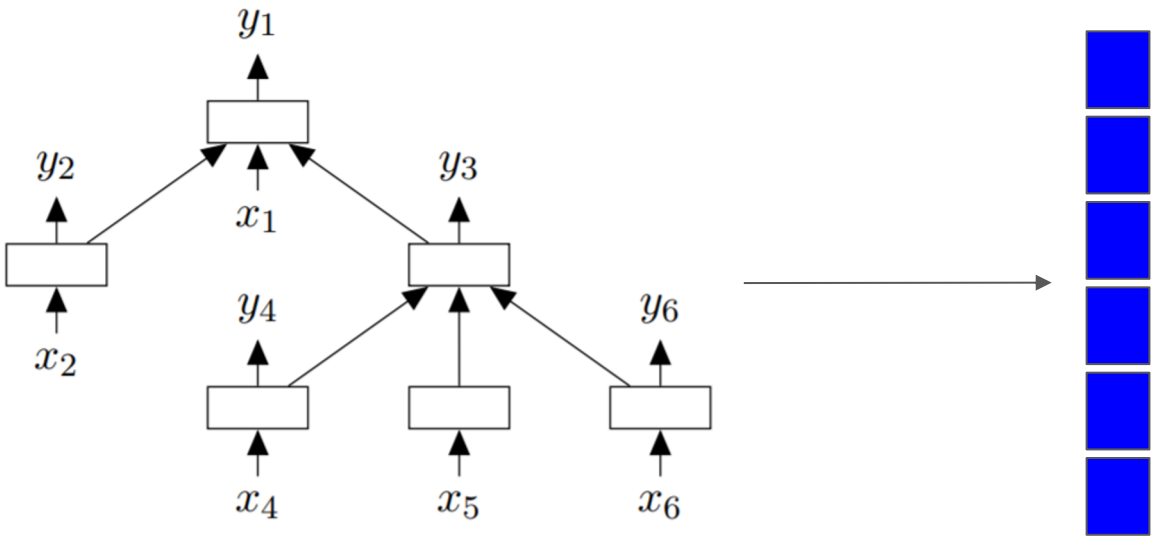
XGBoost



DeepDive - Statistical models TreeGRU

TreeGRU - for better semantic relatedness

Embedding vector



Objective function for training

Here e and s represents the tensor operator and the low level program and c represents the actual run time on the hardware.

F cap represents the predicted runtime of the preceding models that we saw.

$$\mathcal{D} = \{(e_i, s_i, c_i)\}$$

$$\sum_{i,j} \log(1 + e^{-\operatorname{sign}(c_i - c_j)(\hat{f}(x_i) - \hat{f}(x_j))}).$$

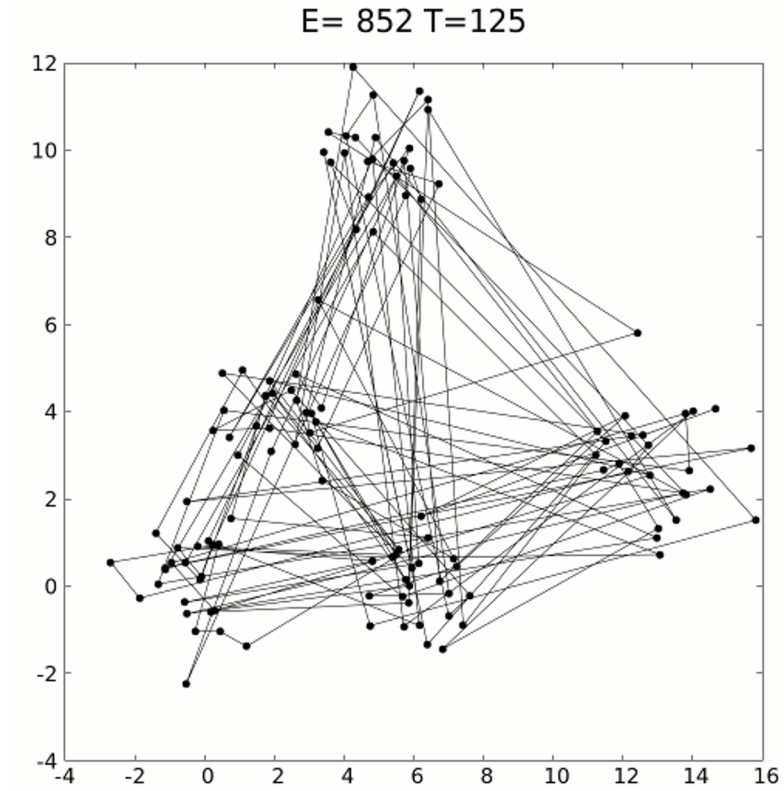
Exploration module - Simulated annealing

The paper uses simulated annealing as the

Exploration strategy. They use the cost model

Output as the energy function for the

algorithm.



Algorithm

Algorithm 1: Learning to Optimize Tensor Programs

Input : Transformation space $\mathcal{S}_e$
Output : Selected schedule configuration s^*
$\mathcal{D} \leftarrow \emptyset$
while *n_trials* $<$ *max_n_trials* **do**
    `// Pick the next promising batch`
    $Q \leftarrow$ run parallel simulated annealing to collect candidates in $\mathcal{S}_e$ using energy function $\hat{f}$
    $S \leftarrow$ run greedy submodular optimization to pick $(1-\epsilon)b$-subset from Q by maximizing Equation 3
    $S \leftarrow S \cup \{$ Randomly sample ϵb candidates. $\}$
    `// Run measurement on hardware environment`
    for s ***in*** S **do**
        $c \leftarrow f(g(e, s)); \mathcal{D} \leftarrow \mathcal{D} \cup \{(e, s, c)\}$
    end
    `// Update cost model`
    update $\hat{f}$ using $\mathcal{D}$
    n_trials $\leftarrow$ n_trials $+ b$
end
$s^* \leftarrow$ history best schedule configuration

Diversity aware exploration - Used with an aim to explore more of the searchspace. We try to maximize the following objective when selecting the set b to run on hardware. Here the L(S) is a submodular function and a greedy algorithm is used to optimize it.

$$L(S) = -\sum_{s \in S} \hat{f}(g(e,s)) + \alpha \sum_{j=1}^{m} |\cup_{s \in S} \{s_j\}|$$

Uncertainty estimators - certainty estimators are available for the cost model output. This didn't prove to be useful for their search.

Transfer Learning

- Focus so far has been on a single tensor operator.
- In the real world we have multiple operators and different input and data shapes.
- We want to use this historical data to speed up the optimization we talked about.
- The problem is to define a transferrable representation that is invariant to the source and target domain.
- To make this learning transferrable we use a low-level AST of s.

Possible ways to encode a low level AST

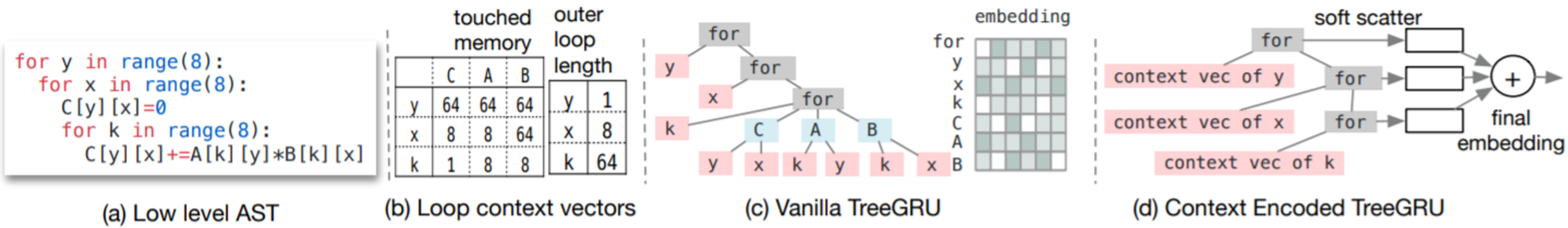


Figure 3: Possible ways to encode the low-level loop AST.

Encoding of the AST

The AST needs to be encoded into a vector to perform predictions, this is done in two ways based on the models.

Context relation features for GBT - Context features represent direct loop vectors using their ranges but these can’t generalize across different loop patterns. Context relation features treat the context features as a bag of words and create high level features from that. The way they convert it is using beta-thresholds over a log2-space and encode the context vector using that threshold.

Encoding of the AST

Context Encoded TreeGRU - One way is to directly obtain an embedding vector from the low-level AST. But this isn't transferable because loop variables can change across different domains. So they instead encode each loop variable using a context vector. They then scatter the context vectors into m vectors and use softmax on it to calculate the out rule for each vector. They then sum the scattered vectors to get the final embedding.

Once we have a transferable representation the learning problem can just change to -

$$\hat{f}(x) = \hat{f}^{(global)}(x) + \hat{f}^{(local)}(x).$$

Experiments

The chosen model for the experiments is ResNet18. First set of experiments use no historical data.

Workload Name	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
H, W	224,224	56,56	56,56	56,56	56,56	28,28	28,28	28,28	14,14	14,14	14,14	7,7
IC, OC	3,64	64,64	64,64	64,128	64,128	128,128	128,256	128,256	256,256	256,512	256,512	512,512
K, S	7,2	3,1	1,1	3,2	1,2	3,1	3,2	1,2	3,1	3,2	1,2	3,1

Table 1: Configurations of all conv2d operators in a single batch ResNet-18 inference. H,W denotes height and width, IC input channels, OC output channels, K kernel size, and S stride size.

Experiments - Prior blackbox methods vs Cost based

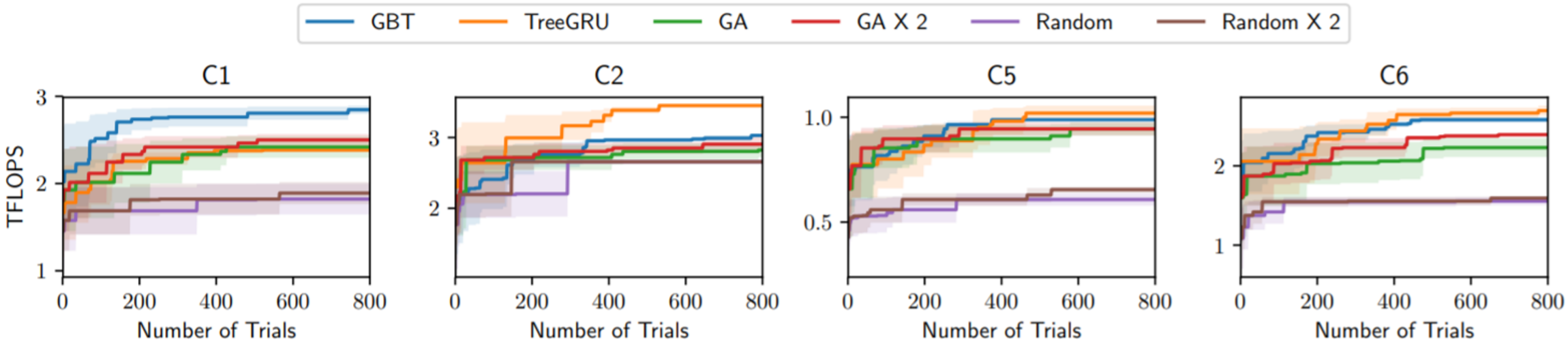


Figure 4: Statistical cost model vs. genetic algorithm (GA) and random search (Random) evaluated on NVIDIA TITAN X. 'Number of trials' corresponds to number of evaluations on the real hardware. We also conducted two hardware evaluations per trial in Random $\times 2$ and GA $\times 2$. Both the GBT- and TreeGRU-based models converged faster and achieved better results than the black-box baselines.

Experiments - Rank vs Regression based objective

Interpretation unclear - Assuming TFLOPS means actual runtime performance.

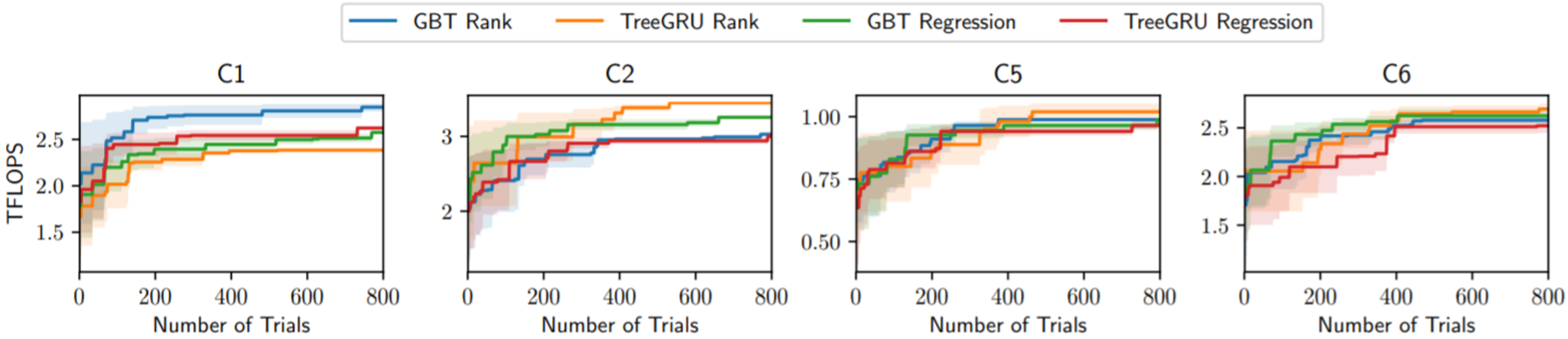


Figure 5: Rank vs. Regression objective function evaluated on NVIDIA TITAN X. The rank-based objective either outperformed or performed the same as the regression-based objective in presented results.

Experiments - Diversity aware selection

No positive or negative impact - My view is that diversity doesn't really matter given enough trials eventually the quality (runtime cost) should have the maximum weight.

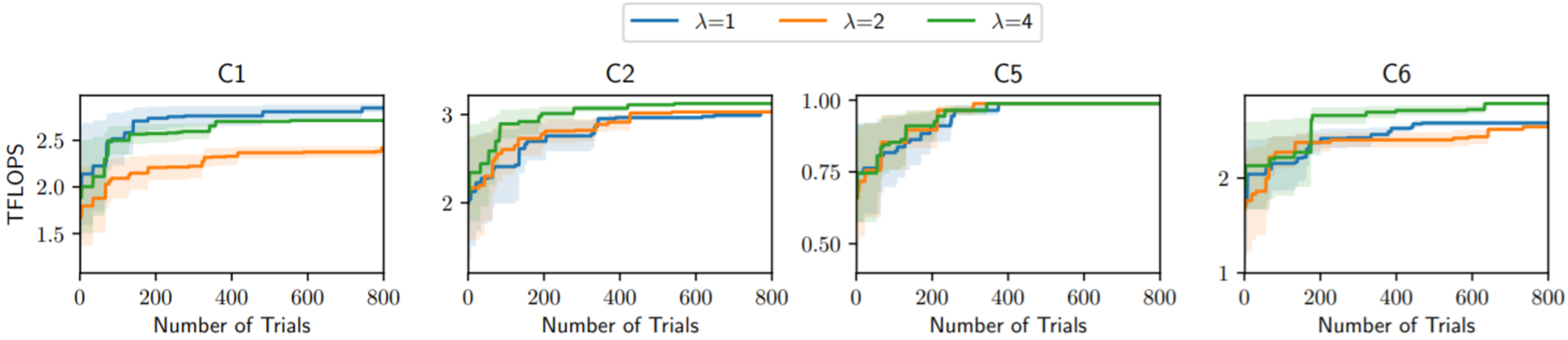


Figure 6: Impact of diversity-aware selection with different choices of λ evaluated on NVIDIA TITAN X. Diversity-aware selection had no positive or negative impact on most of the evaluated workloads.

Experiments - Uncertainty aware scheduling.

No improvement over traditional mean method.

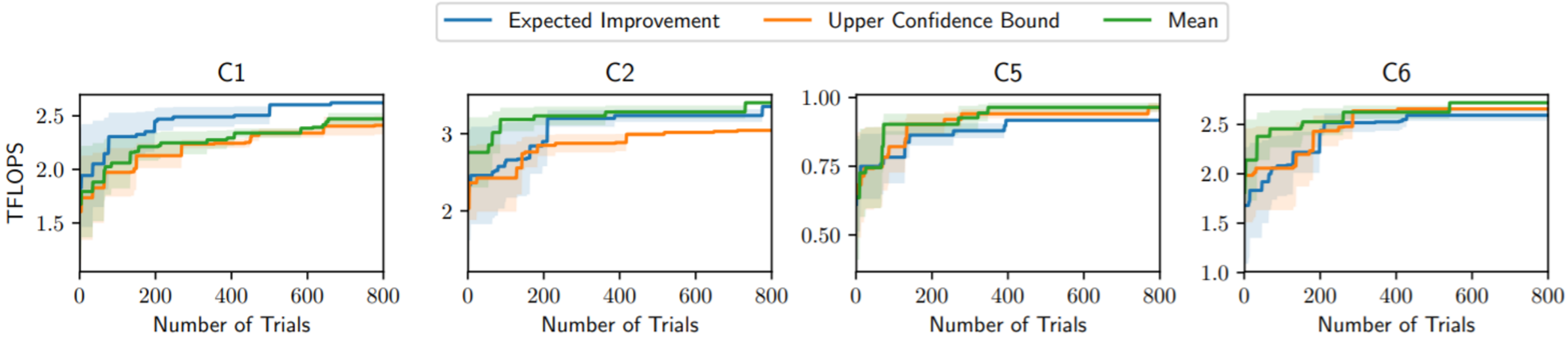


Figure 7: Impact of uncertainty-aware acquisition functions evaluated on NVIDIA TITAN X. Uncertainty-aware acquisition functions yielded no improvements in our evaluations.

Experiments - Historical data and transfer learning

To test transfer learning the authors used the historical training experiments from previous experiments. 30000 samples from Titan X and 20000 samples from ARM gpus. Transfer learning improved speedup by 2-10x.

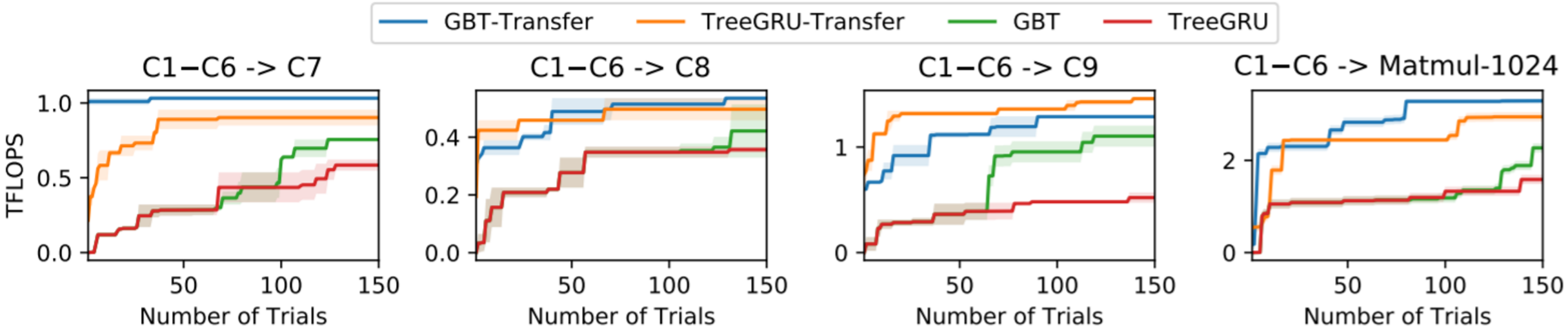
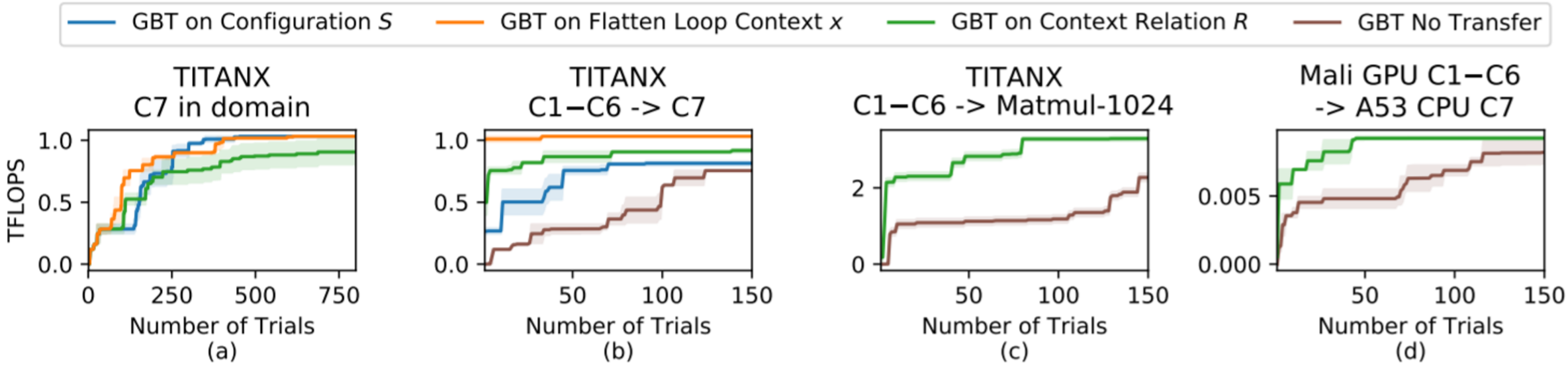


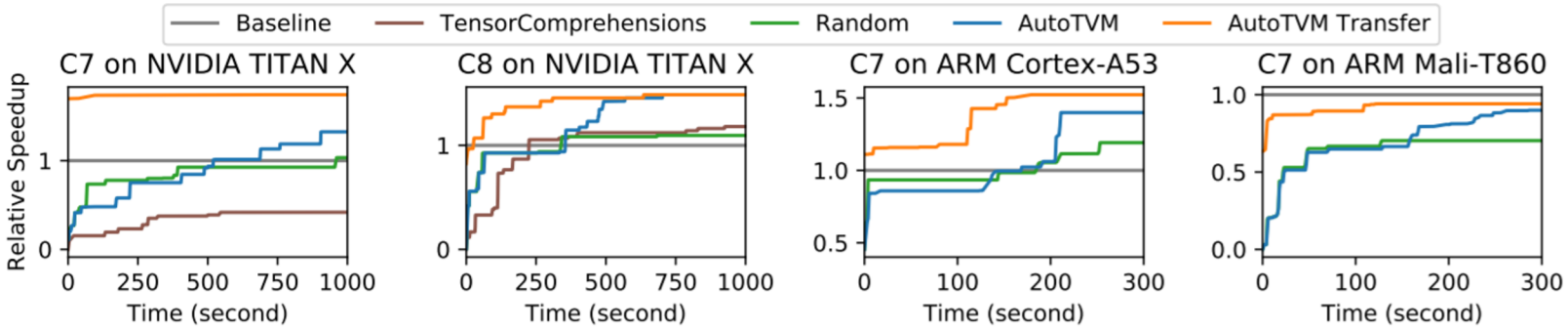
Figure 8: Impact of transfer learning. Transfer-based models quickly found better solutions.

Experiments - Invariant representation and domain distance.

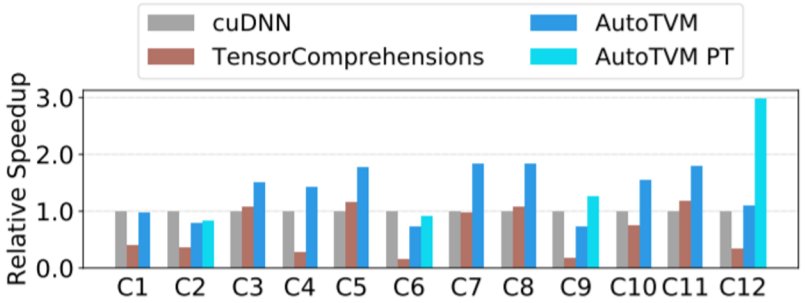
In my opinion, these results are very interesting. The real test would be to transfer from TitanX to ARM which isn't covered. Context Relation should be the most invariant across different domains but they didn't show Blue and orange on C and D.



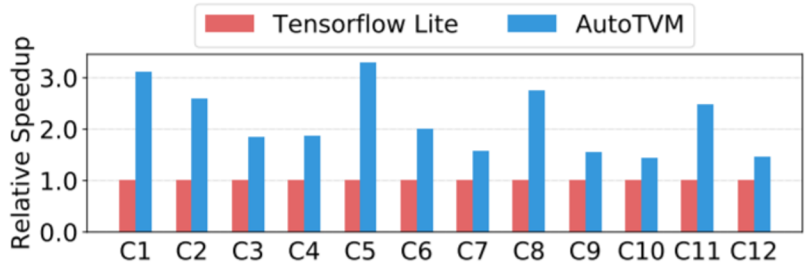
Experiments - End to end evaluation



(a) Optimization curves in wall clock time. (We set cuDNN v7, Tensorflow Lite and ARM ComputeLibrary v18.03 as the baselines for TITAN X, ARM A53 and ARM Mali-T860, respectively.)



(b) NVIDIA TITAN X Single Op

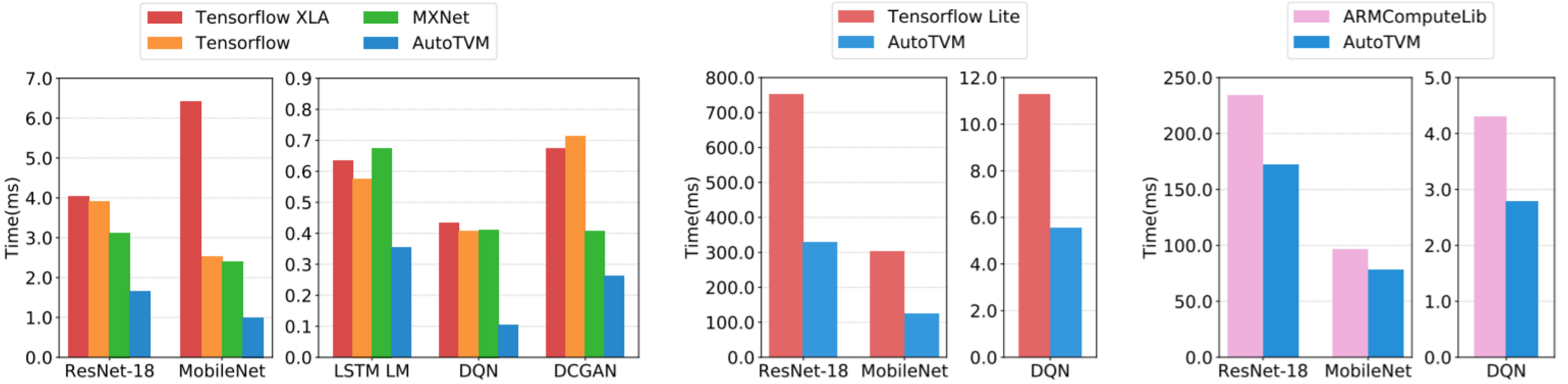


(c) ARM Cortex-A53 Single Op

Figure 10: Single operator performance on the TITAN X and ARM CPU. (Additional ARM GPU (Mali) results are provided in the supplementary material.) We also included a weight pre-transformed Winograd kernel [24] for 3×3 conv2d (AutoTVM PT). AutoTVM generated programs that were competitive with hardware-specific libraries.

Experiments - End to end evaluations time taken.

1.2x - 3.8x improvements.



(a) NVIDIA TITAN X End2End (b) ARM Cortex-A53 End2End (c) ARM Mali-T860 End2End

Figure 11: End-to-end performance across back-ends. [2]AutoTVM outperforms the baseline methods.

Pros

1. Transfer learning is a great concept that allows previous experiments to be transferable. I think by adding some context into the actual hardware (to the cost model) that it was run in the data the portability of the experiments will increase. For example - If the experiments are run on titan X then they wouldn’t be as relevant to ARM.
2. The two cost models used are a good tradeoff. XGBoost would be faster but would require feature engineering, TreeGRU would be more general but would run slower.

Cons

1. The paper at times leaves a lot to imagination.
2. Some of the omitted results were important. Like transfer learning between Titan X examples to ARM.
3. Featurization of the low level program is not well explained and this still feels like an open problem.

Conclusion

AutoTVM is able to automatically search a space of possible programs and select the best performing one for a targeted backend. Additionally it achieves a speedup to existing deep learning models vs hardware accelerated implementations.

Learning to Optimize Halide with Tree Search and Random Programs

ANDREW ADAMS, Facebook AI Research KARIMA MA, UC Berkeley LUKE ANDERSON, MIT CSAIL RIYADH BAGHDADI, MIT CSAIL TZU-MAO LI, MIT CSAIL MICHAËL GHARBI, Adobe BENOIT STEINER, Facebook AI Research STEVEN JOHNSON, Google KAYVON FATAHALIAN, Stanford University FRÉDO DURAND, MIT CSAIL JONATHAN RAGAN-KELLEY, UC Berkeley

Presenter: Shaurya Patel
Shashwat Singh

University of Massachusetts Amherst

OUTLINE

- Problem description
- Prior work
- Proposed solution
- Results
- Conclusion

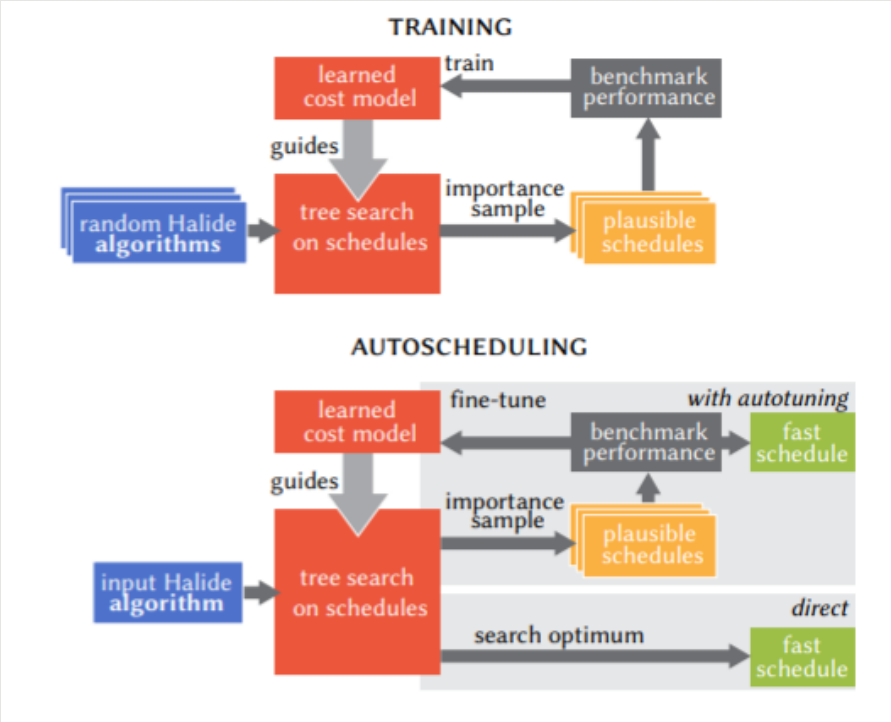
PROBLEM DESCRIPTION

- **Given a compute definition how do we find the best low-level implementation for our target hardware without hand-optimizing the solutions.**

PRIOR WORK

- Works by Mullapudi et al. 2016, 2015; Sioutas et al. 2018 are constrained by only considering a small subset of possible schedules, exploration is tightly coupled to key choices made and the cost models they use are hand designed.
- The polyhedral compiler PolyMage [Mullapudi et al. 2015] uses an algorithm similar to that of the current Halide autoscheduler. This has also been extended with backtracking and a richer cost model to some benefit [Jangda and Bondhugula 2018], but without changing the fundamentally restricted search space.
- TVM follows a similar philosophy but focuses on locally optimizing individual operators in isolation.

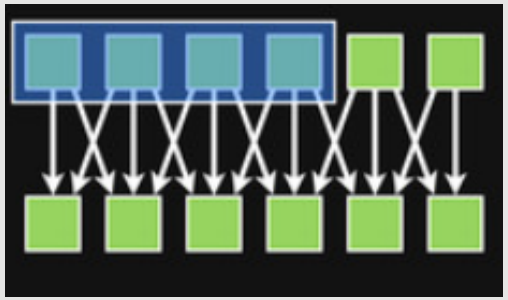
PROPOSED SOLUTION – HALIDE AUTO SCHEDULER

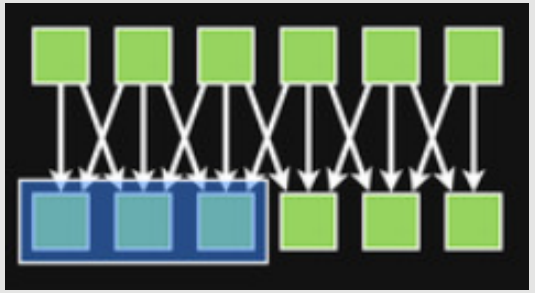


HALIDE SCHEDULING SPACE

- **Halide separates the compute definition from the actual schedule. Unlike traditional programs the schedule decisions aren't made when writing the code. The major tradeoff is between parallelism, locality and amount of work performed.**

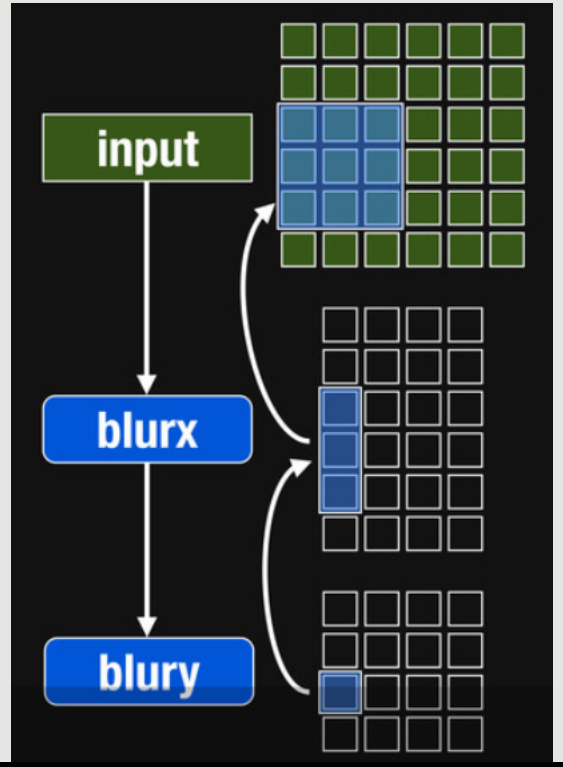
Intra-stage order – First choose what order to evaluate each stage in. This includes tiling and arbitrary dimension order. The parallel loops can then be executed over threads, SIMD etc.





HALIDE SCHEDULING SPACE

- **Cross stage granularity – Here we choose granularity at which each stage is computed with respect to its consumers. When should we compute parts of g with parts of f that it uses.**



PARAMETERIZING THE SEARCH SPACE

A loop nest for each stage contains –

- a nested set of n-dimensional tilings
- a compute and storage granularity
- annotations of any levels of the tiling to be unrolled or spread across parallel threads or SIMD lanes.

(1) We select a compute and storage granularity at which to insert the new stage, optionally adding an extra level of tiling to the consuming stage's existing loop nest to create additional options.

(2) We add inner and outer tilings to the newly-added stage for parallelism. The outer tiling is annotated to be spread across parallel threads, and the inner tiling across SIMD lanes

THE SEARCH ALGORITHM

- Beam search is used as a primary algorithm which is an optimized best-first search algorithm. The heuristic used consists of the cost of scheduled states so far and an optimistic underestimate of the true cost of unscheduled stages.

Schedule pruning –

1. All multicore parallelism must occur at outer class level.
2. SIMD loop sizes are always at the native vector width.
3. Parallel loops should not exceed number of cores * 16.
4. No value must be redundantly computed more than 10 times.
5. Loops between storage and compute sites should be 1-D.

COARSE TO FINE SCHEDULE REFINEMENT

- To enforce more diversity in the beam the authors enforce a hash function.
- They hash loop nests upto some fixed nest d.
- This attempts to preserve the backtracking ability of beam search for top-level scheduling decisions.
- This algorithm is implemented as a two step process.

PREDICTING RUNTIME

- **Beam search requires a metric to evaluate the state but this can't be actual benchmarked time because the search space is very large.**

The 3 important parts of predicting the runtime are –

1. Featurizing a schedule.

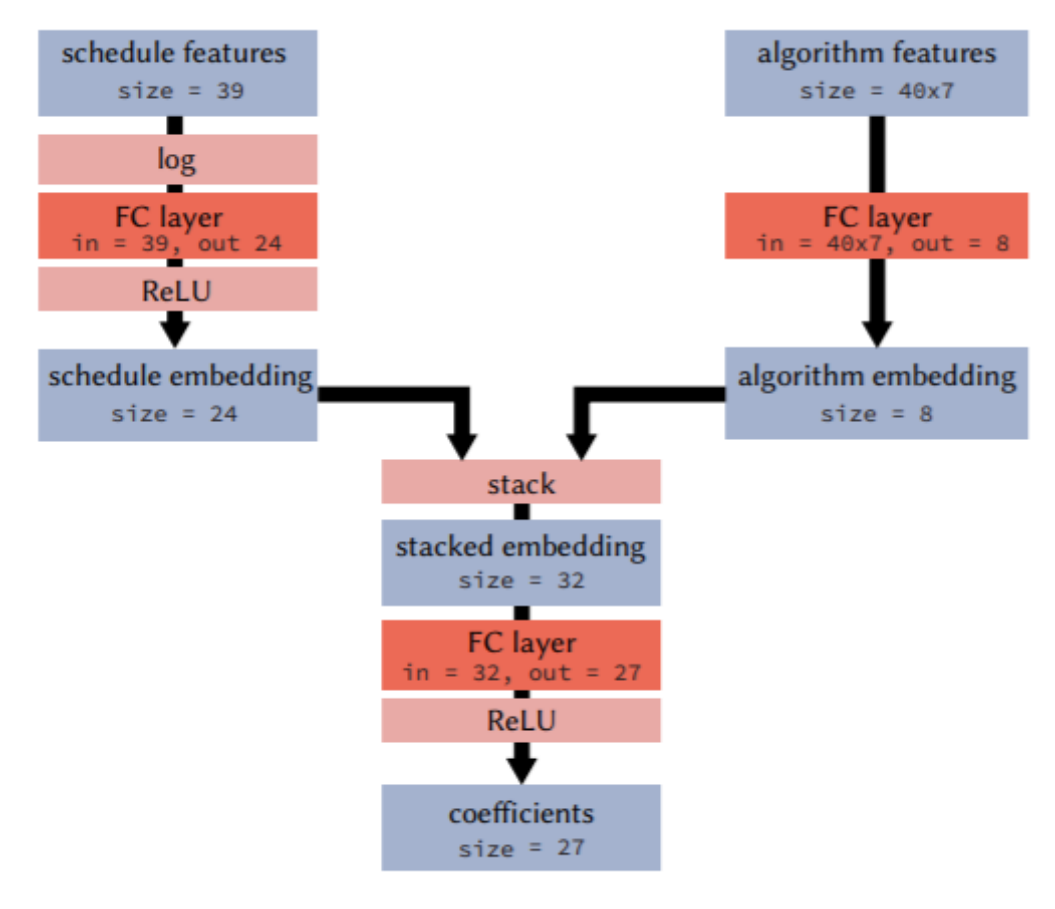
2. Cost model design.

3. Autotuning.

FEATURIZING A SCHEDULE

- How do we represent a schedule as a feature for the model?
- For halide they model each stage separately and divide it into algorithm-dependent and schedule-dependent features.
- For algorithm specific features they use histograms of operations used to compute a point in the algorithm. They create jacobians to identify memory dependencies across consumers.
- Schedule dependent features either count events of various types or characterize memory footprints.

COST MODEL DESIGN



COST MODEL DESIGN

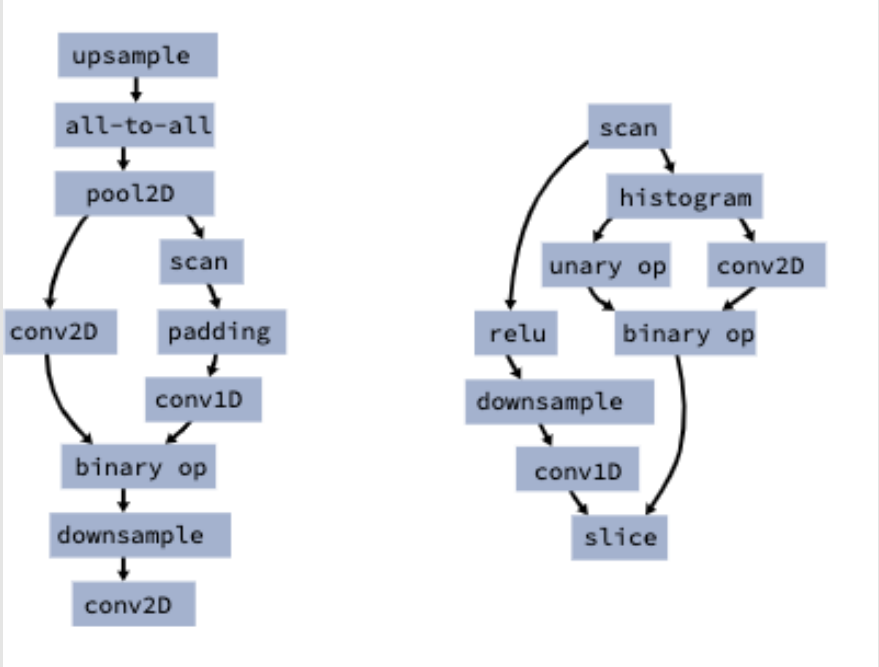
- L2 error is used on throughput while training to make the model focus on difference between parsing differences between two faster implementations.
- They use a training batch size of 32 and use gradient descent and adam for minimizing with standard hyper-parameters.

AUTOTUNING

- The idea is to add samples and benchmarking to make the search more accurate.
- Cost model is not accurate and beam search doesn't globally minimize it.
- At each stage of beam search they expand the best state with a probability p, otherwise they discard it and expand the next-best state with p. This allows them to benchmark because of the modest number of samples.

Generating Training Data

1. Generated through a random algorithm generator
2. This generator synthesizes a diverse range of programs that exhibit most of the patterns of computation present in real applications
3. Programs must have some notion of the units on each spatial dimension to safely mix stages. For e.g. adding an image to a smaller copy of itself, creates shift-varying memory footprints, which invalidates some of the analysis



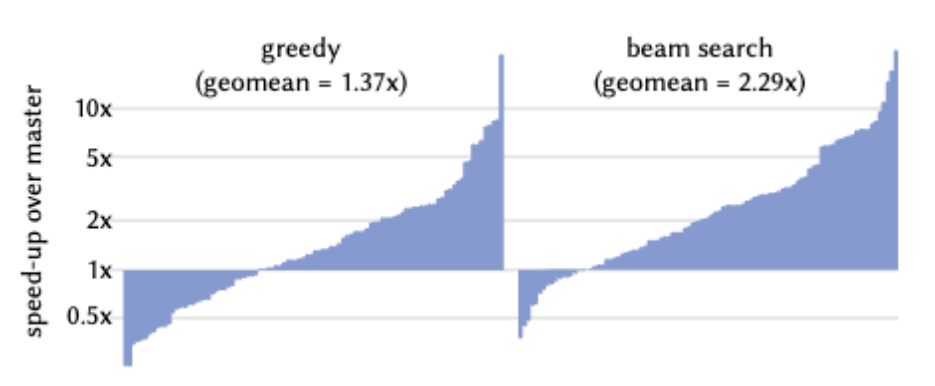
Example Random Pipelines.
DAG-structured pipelines out of a library of stage types commonly used in image processing and deep learning.

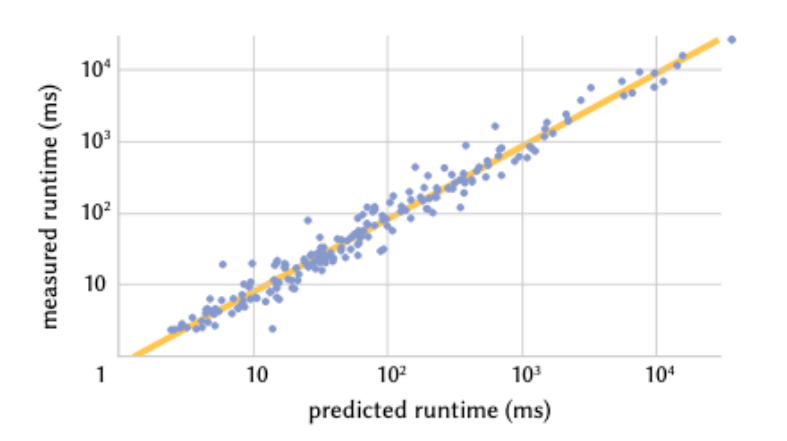
Generating Training Data

- For each algorithm, a batch of 32 schedules is synthesized using the auto tuning process
- The featurizations and runtimes produced in this way are harvested to form the training set.
- For generating an initial round of training data, uniform random weights in $[-\frac{1}{2}, \frac{1}{2}]$ are used.
- For further rounds of training newer weights are used.
- It has been found that five rounds of this boot-strapping procedure, with ten thousand random algorithms times thirty-two random schedules per round, is sufficient to converge.
- The data generation process is distributed over a cluster of machines, each of which searches for and compiles multiple programs in parallel, then benchmarks each one sequentially.

Results

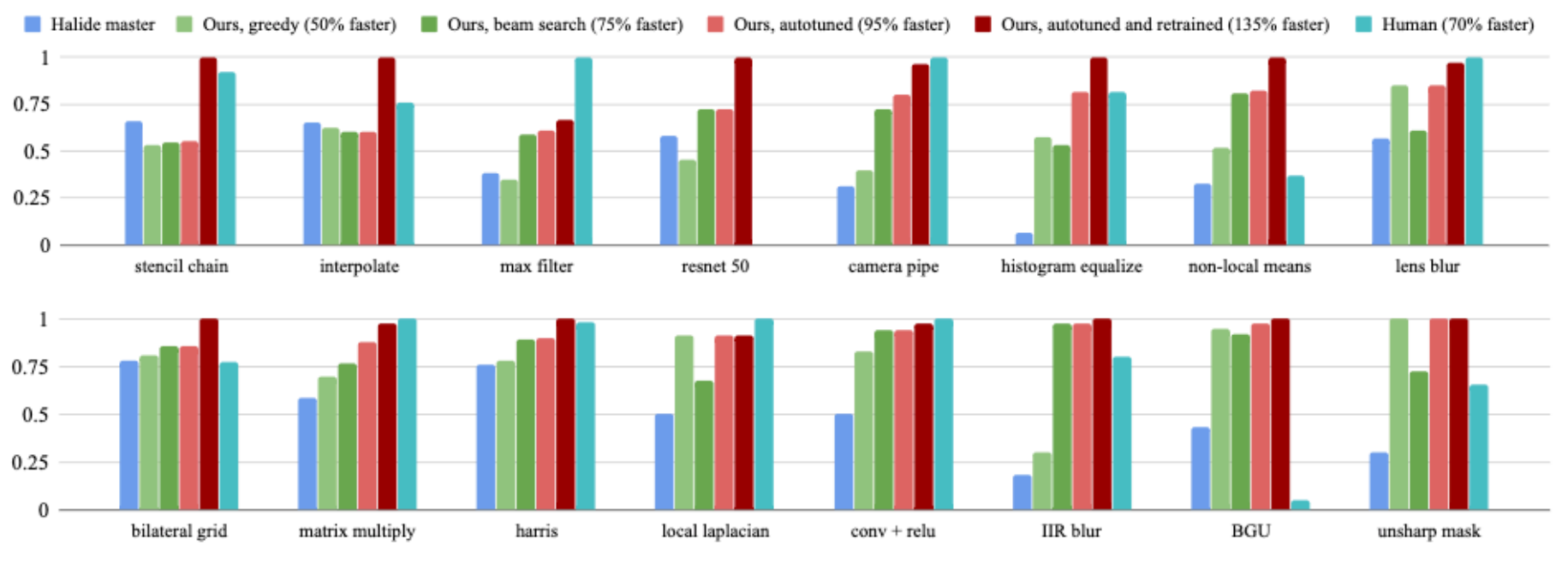
Halide produces good schedules in short compile times and better schedules given longer compile times.





The results are compared with the current auto scheduler used by Halide (referred by the term master)

Results



Throughput relative to the throughput of the best-known schedule on a suite of real applications, running on a high-end x86 CPU.

Limitations And Future Work

1. Expanding the training set: The current cost model produces excellent schedules on real applications. Next step is to contribute samples harvested from real applications back to the training set.

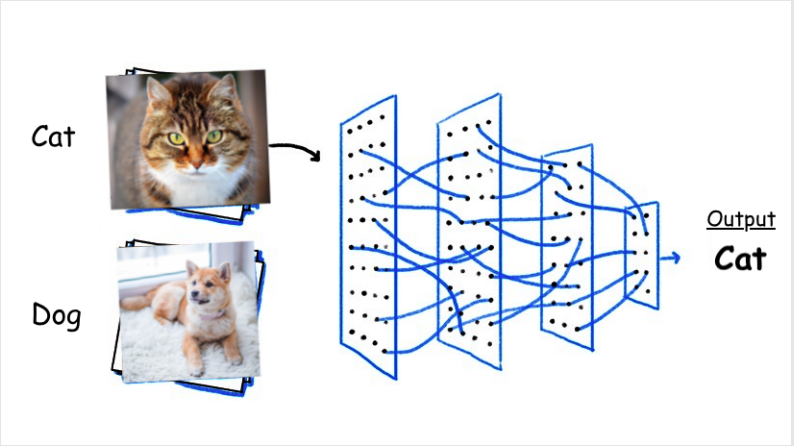
2. Expanding the search space: Adding a new architecture involves training the cost model on hundreds of thousands of fresh random programs and triage the results to ensure that the featurization correctly captures the factors that matter for performance on that architecture.

FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System (Size Zheng et al. @Peking University)

Presenter: Shashwat Singh

University of Massachusetts Amherst

Tensor Computations are Everywhere

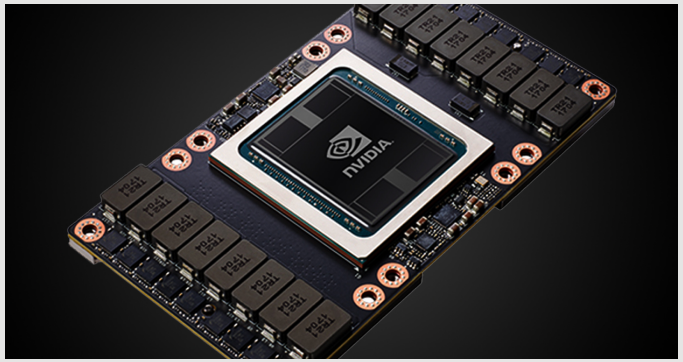


Tensor: Multi Dim Array
Tensor Computation:

- GEMM
- Conv2d
- Conv3d
- ……

Heterogeneous systems (Beneficial or Challenging?)

CPU



GPU

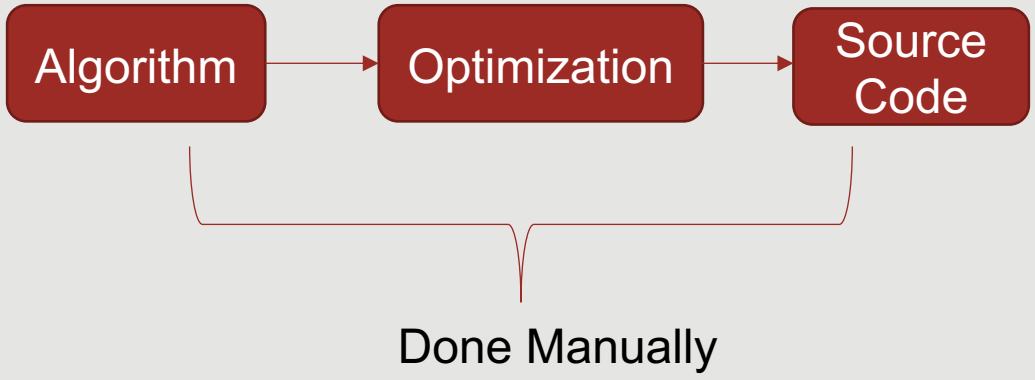
FPGA

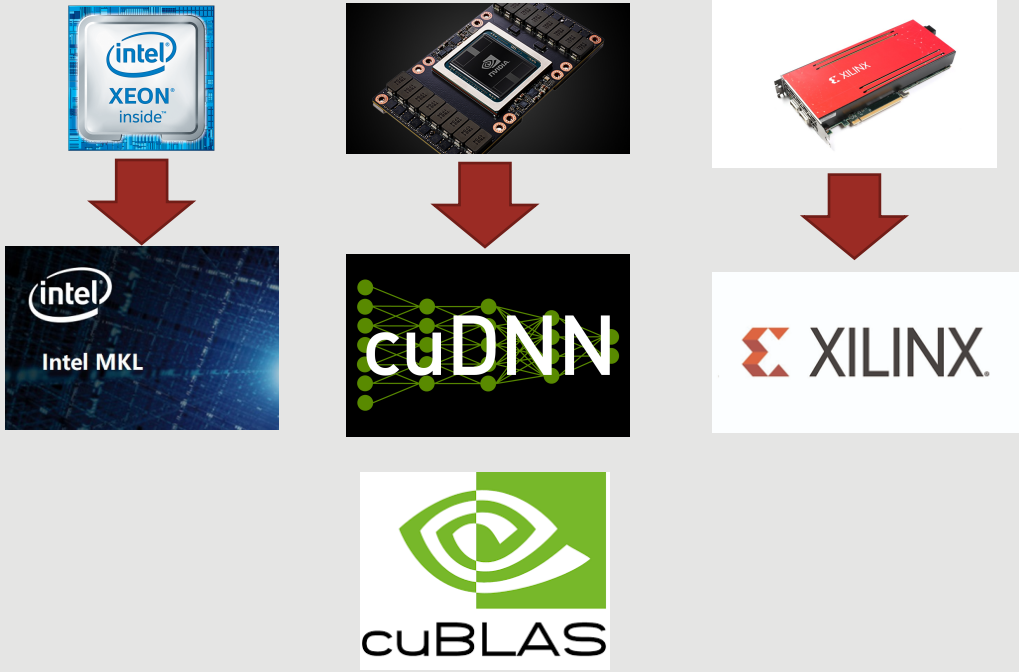
The wide adoption of tensor computation and its huge computation cost has led to high demand for flexible, portable, and high-performance library implementation on heterogeneous hardware accelerators such as GPUs and FPGAs

Challenges in using hand optimized libraries

1. Requires programmers to manually design low-level implementation and optimize from the algorithm, architecture, and compilation perspectives.

2. Manual development process often takes months or even years, which falls far behind the rapid evolution of the application algorithms.

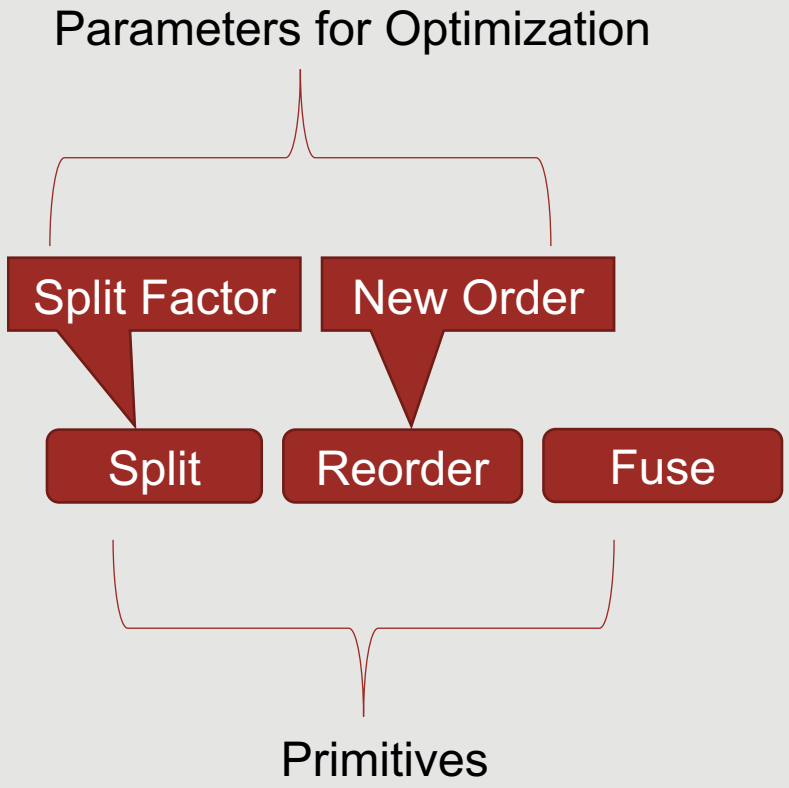




Code Generation with Scheduling

Compute Description:
- Is high level
- Includes algorithm
- Includes Mathematical expressions

Scheduling:
- Includes primitives
- Is hardware specific
- Involves parameter for optimization



Limitations of writing schedules manually

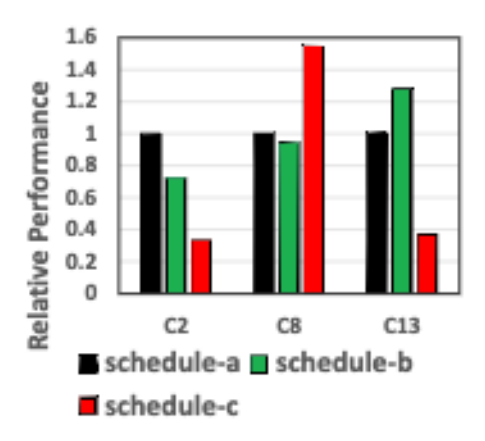
- Many primitives to choose

Split Reorder Fuse

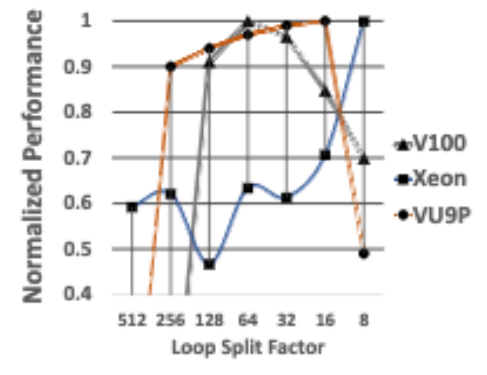
- Complex combinations and huge parameter space

Split + Reorder + Fuse + . . .

Limitations of writing schedules manually



- Different types of schedules are best for different shapes, for e.g. for shape C2, schedule a is better
- Each schedule Involves different primitive combinations



- Different parameters
- Different hardware

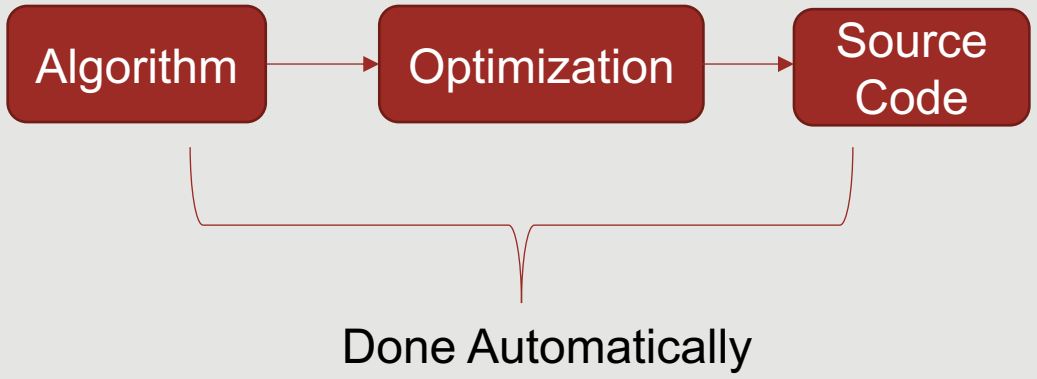
Motivation behind FlexFlow

An Ideal framework:

1. Auto optimization
2. High performance
3. Portable
4. No low-level programming
5. Short development time

FlexFlow

1. User just focuses on writing the algorithm
2. It hides the hardware details from the user



Key features of FlexFlow

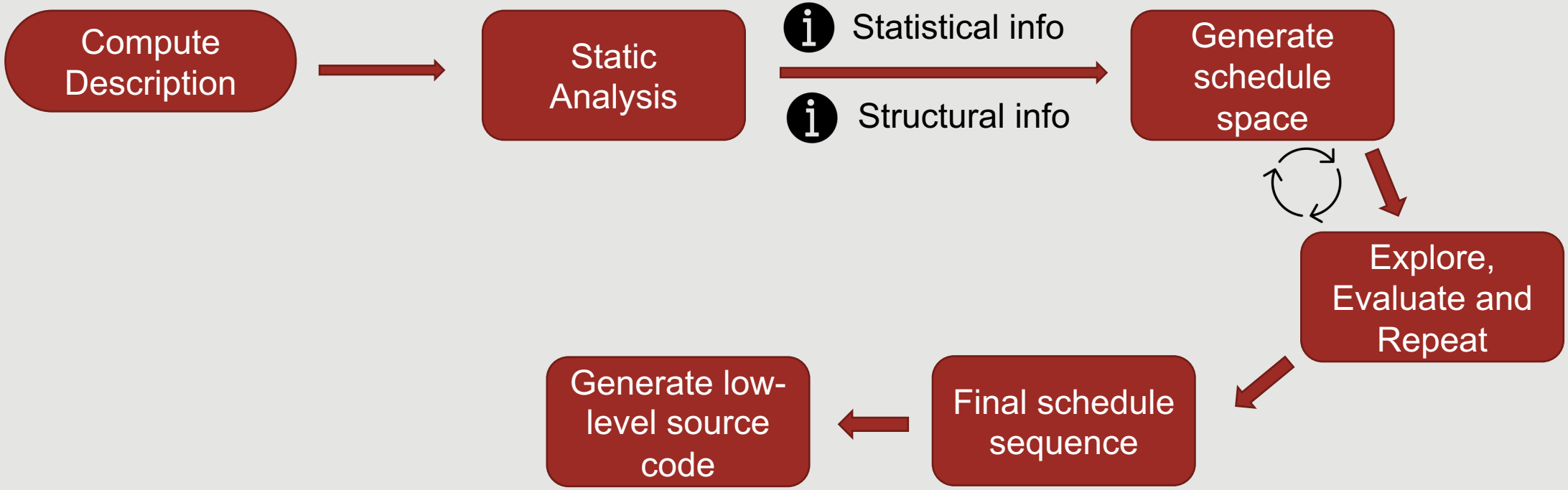
Feature	
Portable	Supports CPU, GPU, FPGA
Optimization	Automatic schedule exploration
Good performance	Speedup: 1.83x on GPU, 1.72x CPU, 1.5x on FPGA
Involves high level programming	Compute description written in Python

Related Works

Halide: A compiler for optimizing parallelism, locality and recomputation

TVM: An automated end-to-end optimizing compiler for deep learning

FlexTensor Workflow



Schedule space generation

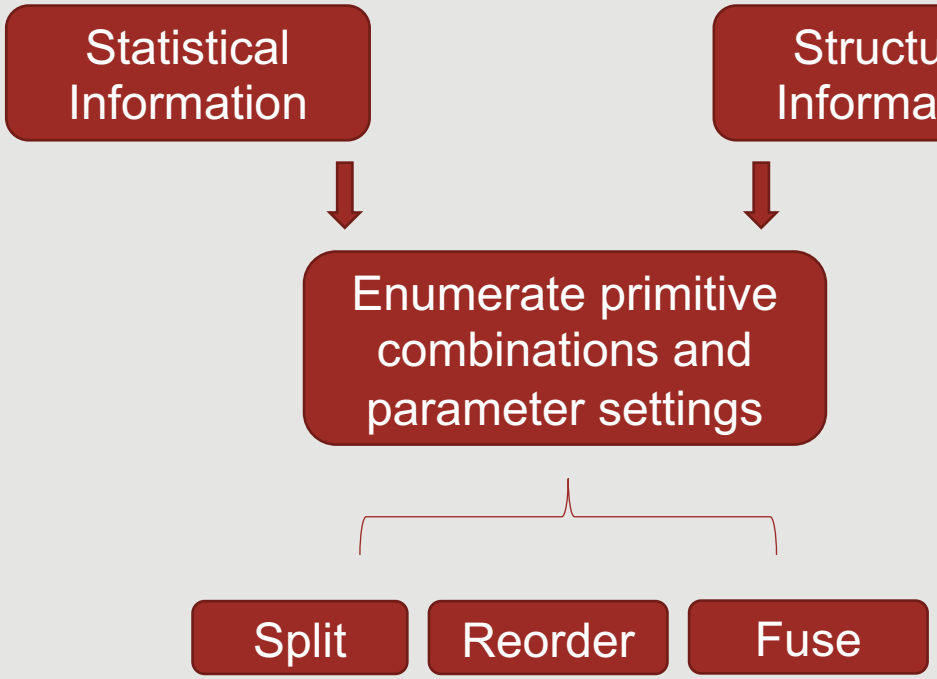
Statistical Information

- Loop trip counts
- Number of loops
- Loop order

Structural Information

Computation graph structure

Schedule space generation



1. Limit the depth of primitive combination

2. Pre-determine certain decisions for different hardware

Effective Exploration

What point to start
-- Heuristics: Simulated annealing

Which direction to search along
-- Machine learning: Q learning

How to evaluate each point
-- Run on target device

Heuristics: Simulated Annealing

Evaluated Points

Known Value: v^1, v^2, v^3
Best known value: v^*
Choose according to possibility:
$e^{\frac{-\gamma(v^* - v^i)}{v^*}}$, i=1,2,3
Allows choosing of multiple points

Machine Learning : Q-Learning

- Used for finding best directions
- Keeps records of visited points
- Uses DQN algorithm to predict Q-value for each direction, q^1, q^2, q^3
- Choose the largest q value, $q^* = \max(q^i)$, i=1,2,3

Schedule Generation

Type of Hardware	Approaches taken
CPU	Multi-level tiling, multi-threading, efficient vectorization
GPU	Multi-level tiling, bind loops to blocks, shared memory configuration(Each block loads data to memory before computation)
FPGA	Three-stage pipeline(Data read, compute, write), bandwidth, resource constraints

Evaluation Methodology

Benchmarks

12 widely used operators

Evaluation

- Geometric mean speedup
- Absolute performance

Baseline

- CPU:MKL-DNN
- GPU: cuDNN
- FPGA: hand-optimized

Overall Performance on GPUs

- Three GPU platforms
- Speedup over cuDNN
  - 1.83x on V100
  - 1.68x on P100
  - 1.71x on Titan X

Case Study: Conv2d-GPU

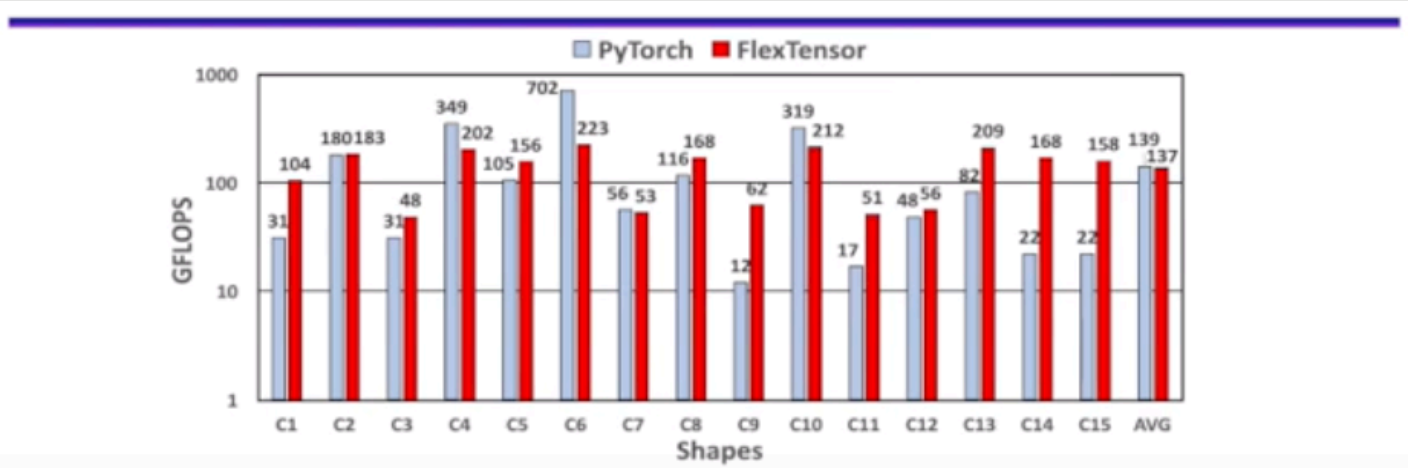
Speedup (Geometric mean):

- **1.5x** speedup over cuDNN
- **1.56x** speedup over PyTorch (native)

Absolute performance (average):

- **PyTorch:** 2438.74 GFLOPS
- **cuDNN:** 3076.70 GFLOPS
- **FlexTensor:** 3810.96 GFLOPS

Case Study: Conv2d-CPU



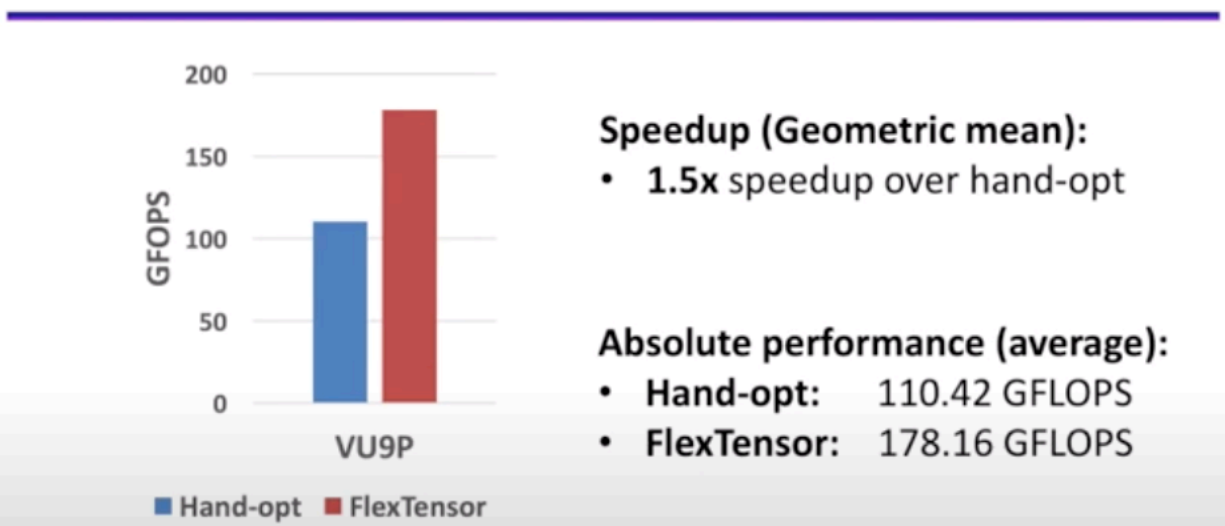
Speedup (Geometric mean):

- **1.72x** to MKL-DNN

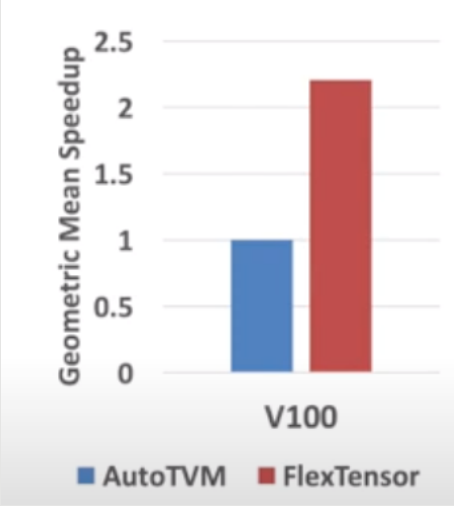
Absolute performance (average):

- **MKL-DNN:** 139.49 GFLOPS
- **FlexTensor:** 136.91 GFLOPS

Case Study: Conv2d-FPGA



Comparison with State of the Art



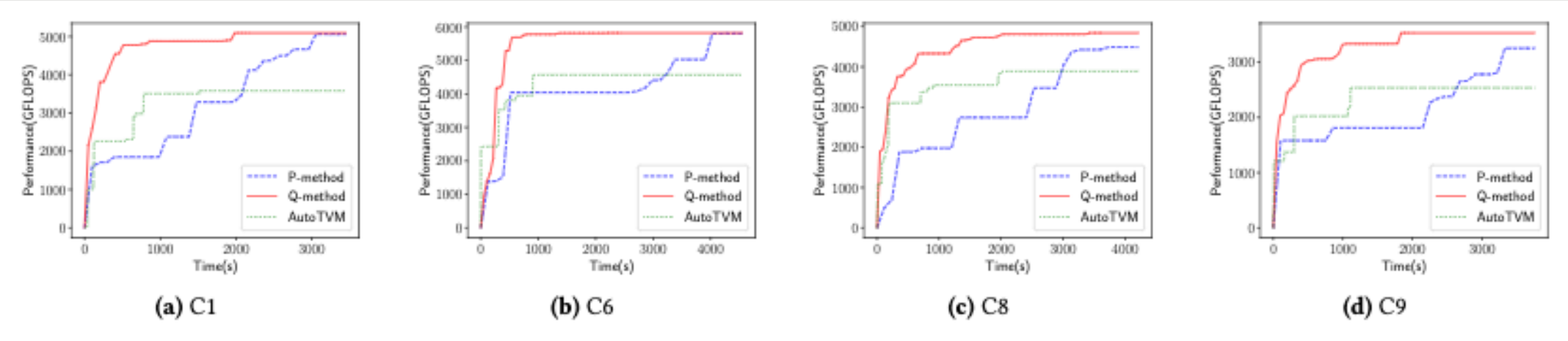
AutoTVM:

- Auto tuning tool for TVM
- Requires template

Benchmarks

Overall speedup of 2.21x

Comparison with State of the Art



Q-method: Used by FlexTensor
P-method: Implemented without Q learning
Baseline: AutoTVM

Q-method only uses 27.6% of time of AutoTVM

THANK YOU !

QUESTIONS ?

University of Massachusetts Amherst