CMSC 5743
Efficient Computing of Deep Neural Networks

Implementation 06: TVM

Bei Yu
CSE Department, CUHK
byu@cse.cuhk.edu.hk

(Latest update: April 20, 2023)

Spring 2023

These slides contain/adapt materials developed by
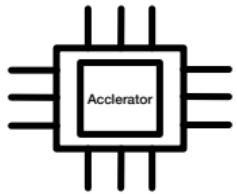
Chen, Tianqi, et al. "TVM: An automated end-to-end optimizing compiler for deep learning." 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018.

Beginning of Story

Beginning of Story

Acclerator

Beginning of Story

Beginning of Story

Goal: Deploy Deep Learning Everywhere

Frameworks

Goal: Deploy Deep Learning Everywhere

Frameworks

Explosion of models and frameworks

Goal: Deploy Deep Learning Everywhere

Frameworks

Explosion of models and frameworks

Explosion of hardware backends

Goal: Deploy Deep Learning Everywhere

Frameworks

Explosion of models and frameworks

Explosion of hardware backends

Goal: Deploy Deep Learning Everywhere

Frameworks

Explosion of models and frameworks

Explosion of hardware backends

Goal: Deploy Deep Learning Everywhere

Frameworks

Explosion of models and frameworks

Explosion of hardware backends

Goal: Deploy Deep Learning Everywhere

Frameworks

Explosion of models and frameworks

Explosion of hardware backends

Goal: Deploy Deep Learning Everywhere

Frameworks

Explosion of models and frameworks

Explosion of hardware backends

Goal: Deploy Deep Learning Everywhere

Frameworks

Explosion of models and frameworks

Explosion of hardware backends

Goal: Deploy Deep Learning Everywhere

Frameworks

Explosion of models and frameworks

Explosion of hardware backends

Goal: Deploy Deep Learning Everywhere

Frameworks

Explosion of models and frameworks

Explosion of hardware backends

Goal: Deploy Deep Learning Everywhere

Frameworks

Explosion of models and frameworks

Huge gap between model/frameworks and hardware backends

Explosion of hardware backends

Acclerator

Existing Approach

Frameworks

Hardware
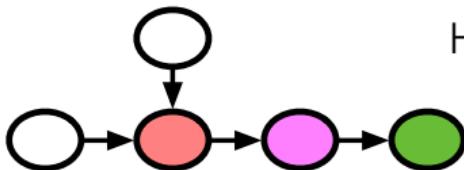
Existing Approach

Frameworks

High-level data flow graph

Hardware
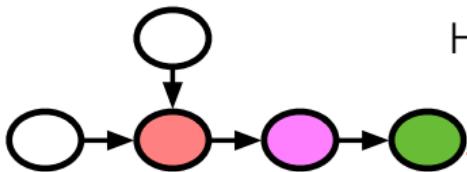
Existing Approach

Frameworks

High-level data flow graph

Primitive Tensor operators such as Conv2D

Hardware
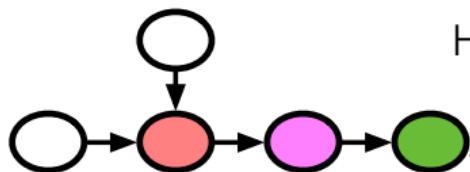
NVIDIA

5

Existing Approach

Frameworks
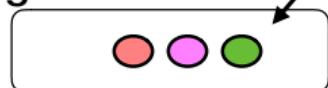
High-level data flow graph

Primitive Tensor operators such as Conv2D

eg. cuDNN
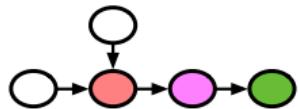
Offload to heavily optimized
DNN operator library

Hardware
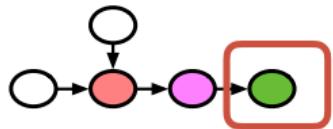
Existing Approach: Engineer Optimized Tensor Operators
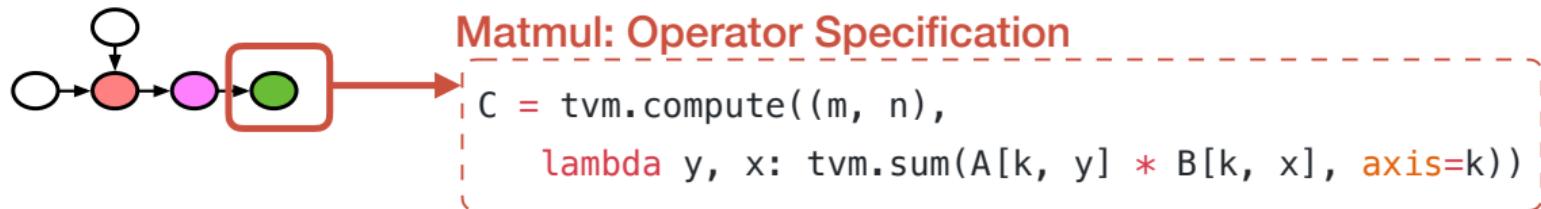
Existing Approach: Engineer Optimized Tensor Operators

Existing Approach: Engineer Optimized Tensor Operators

Matmul: Operator Specification

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Existing Approach: Engineer Optimized Tensor Operators

Matmul: Operator Specification

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Existing Approach: Engineer Optimized Tensor Operators

Matmul: Operator Specification

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```
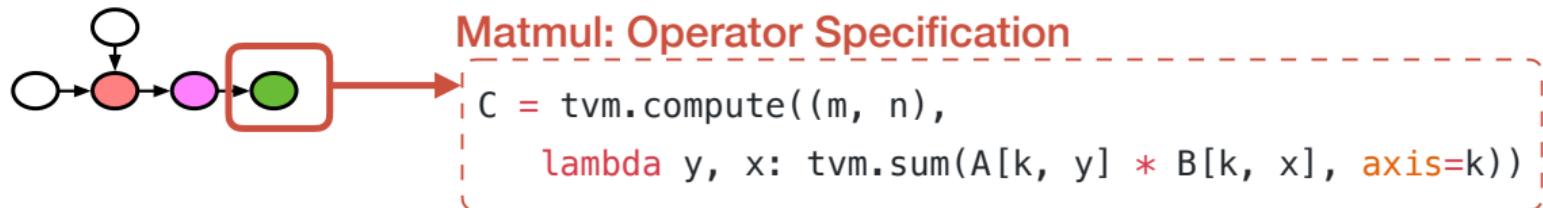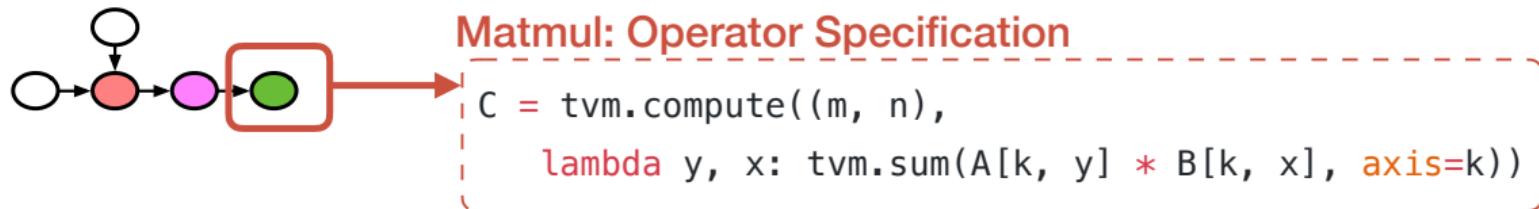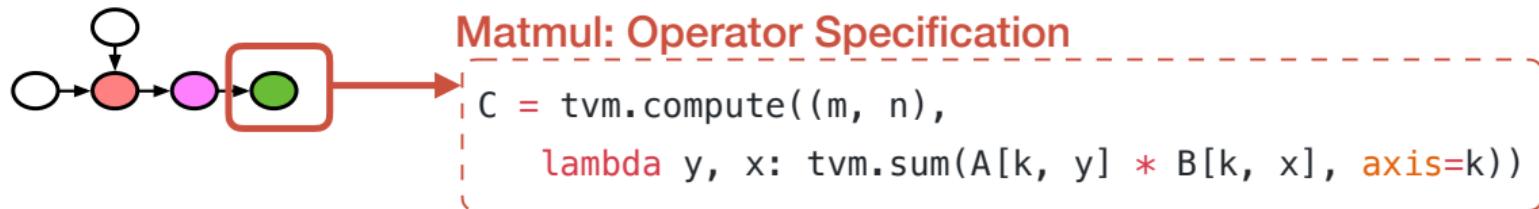
Vanilla Code

```
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]
```

Existing Approach: Engineer Optimized Tensor Operators

Matmul: Operator Specification

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```
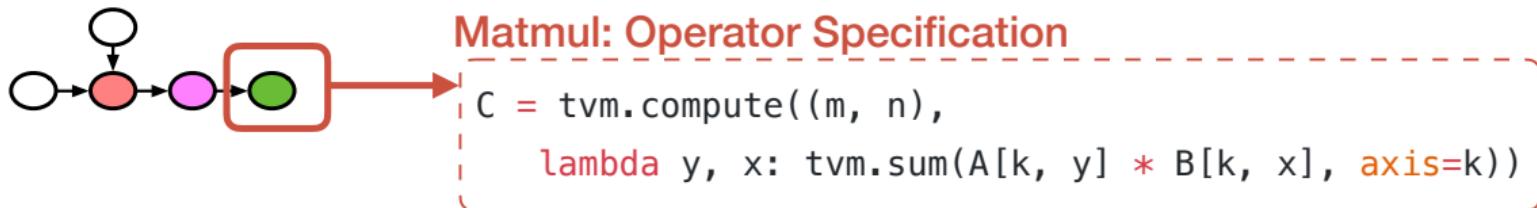
Loop Tiling for Locality

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
                A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

Existing Approach: Engineer Optimized Tensor Operators

Matmul: Operator Specification

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Map to Accelerators
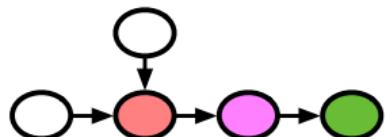
```
inp_buffer AL[8][8], BL[8][8]
acc_buffer CL[8][8]
for yo in range(128):
  for xo in range(128):
    vdla.fill_zero(CL)
    for ko in range(128):
      vdla.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])
      vdla.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])
      vdla.fused_gemm8x8_add(CL, AL, BL)
    vdla.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

Human exploration of optimized code

Limitations of Existing Approach

Frameworks

cuDNN

NVIDIA

Limitations of Existing Approach

Frameworks

cuDNN

Limitations of Existing Approach

Frameworks

Limitations of Existing Approach

Frameworks

cuDNN

NVIDIA

Limitations of Existing Approach

Frameworks

cuDNN

Limitations of Existing Approach

Frameworks

New operator introduced
by operator fusion optimization
potentially benefit: 1.5x speedup

cuDNN

Limitations of Existing Approach

Frameworks

New operator introduced
by operator fusion optimization
potentially benefit: 1.5x speedup

cuDNN

NVIDIA

Limitations of Existing Approach

Frameworks

New operator introduced
by operator fusion optimization
potentially benefit: 1.5x speedup

cuDNN

Limitations of Existing Approach

Frameworks

New operator introduced
by operator fusion optimization
potentially benefit: 1.5x speedup

Engineering intensive
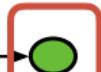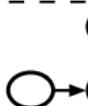
cuDNN

Learning-based Learning System
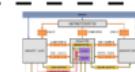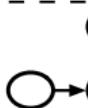
Frameworks

High-level data flow graph and optimizations

Hardware

Learning-based Learning System
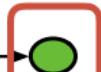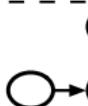
Frameworks

High-level data flow graph and optimizations

Hardware

Learning-based Learning System

Frameworks

High-level data flow graph and optimizations
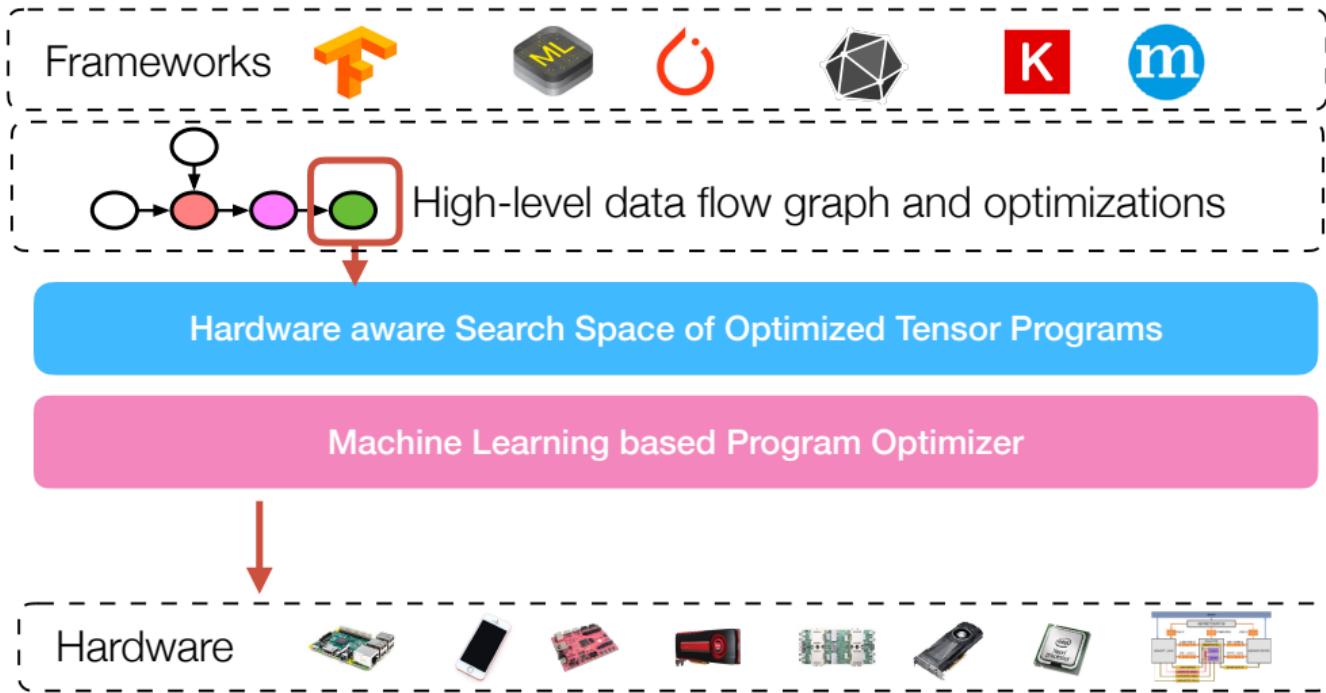
Hardware aware Search Space of Optimized Tensor Programs

Hardware

Learning-based Learning System

Frameworks

High-level data flow graph and optimizations

Hardware aware Search Space of Optimized Tensor Programs

Machine Learning based Program Optimizer

Hardware

Learning-based Learning System

Frameworks

High-level data flow graph and optimizations

Hardware aware Search Space of Optimized Tensor Programs

Machine Learning based Program Optimizer

directly generate optimized program
for new operator workloads and hardware
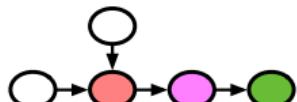
Hardware

Learning-based Learning System
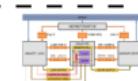
Learning-based Learning System

Hardware-aware Search Space

Tensor Expression Language (Specification)

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Hardware

Hardware-aware Search Space

Tensor Expression Language (Specification)

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Define search space of hardware aware mappings from expression to hardware program

Based on Halide's compute/schedule separation
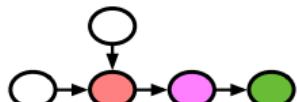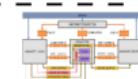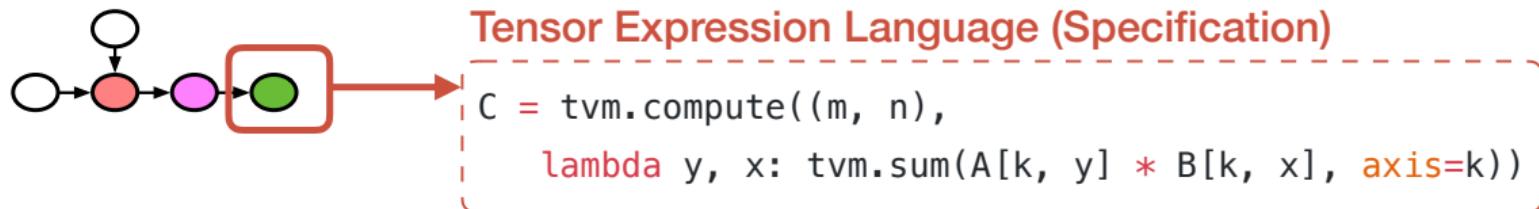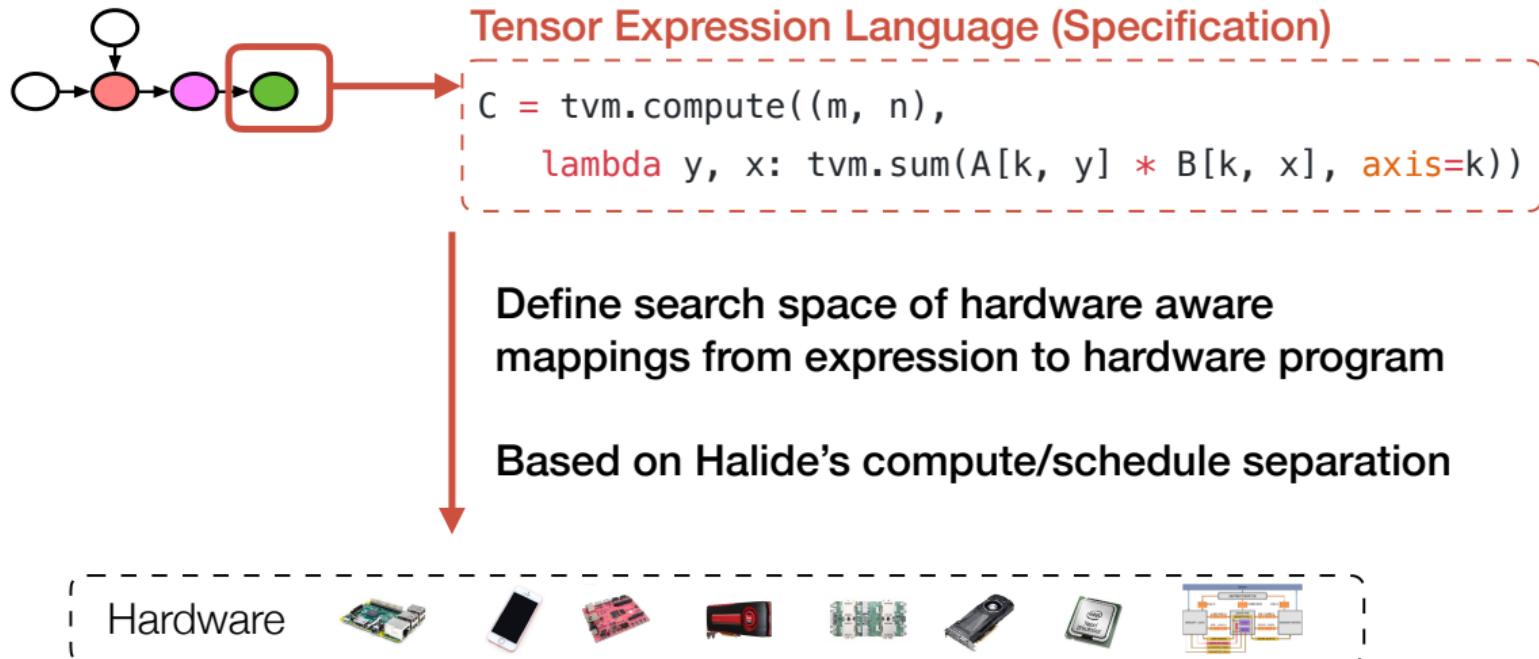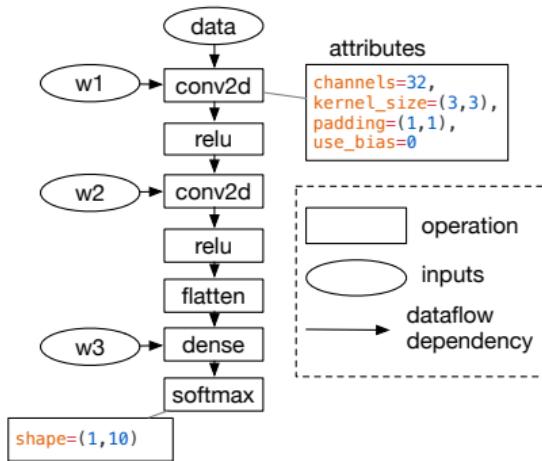
Hardware
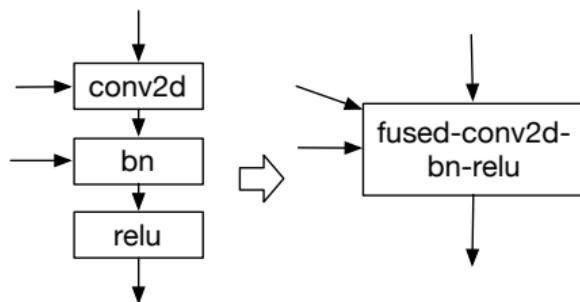
Computational Graph as IR

**Represent High level
Deep Learning Computations**

**Effective Equivalent Transformations
to Optimize the Graph**

Approach taken by: TensorFlow XLA, Intel NGraph, Nvidia TensorRT

The Remaining Gap

Frameworks   Caffe2   CNTK

Computational Graph Optimization

need to build and optimize operators for each hardware, variant of layout, precision, threading pattern …

Hardware

Tensor Level Optimizations

Frameworks

Computational Graph Optimization

Tensor Expression Language

```
C = t.compute((m, n),
    lambda i, j: t.sum(A[i, k] * B[j, k], axis=k))
```

Hardware

Tensor Index Expression

Compute C = dot(A, B.T)

```python
import tvm

m, n, h = tvm.var('m'), tvm.var('n'), tvm.var('h')
A = tvm.placeholder((m, h), name='A')
B = tvm.placeholder((n, h), name='B')


k = tvm.reduce_axis((0, h), name='k')
C = tvm.compute((m, n), lambda i, j: tvm.sum(A[i, k] * B[j, k], axis=k))
```

Inputs

Shape of C

Computation Rule

Tensor Expressions are Expressive

Affine Transformation

```
out = tvm.compute((n, m), lambda i, j: tvm.sum(data[i, k] * w[j, k], k))
out = tvm.compute((n, m), lambda i, j: out[i, j] + bias[i])
```

Convolution

```
out = tvm.compute((c, h, w),
  lambda i, x, y: tvm.sum(data[kc,x+kx,y+ky] * w[i,kx,ky], [kx,ky,kc]))
```

ReLU

```
out = tvm.compute(shape, lambda *i: tvm.max(0, out(*i))
```

Emerging Tools Using Tensor Expression Language

Halide: Image processing language

Loopy: python based kernel generator

TACO: sparse tensor code generator

Tensor Comprehension

Schedule: Tensor Expression to Code

Tensor Expression Language

```
C = t.compute((m, n),
    lambda i, j: t.sum(A[i, k] * B[j, k], axis=k))
```

Schedule Optimizations

Key Idea:
Separation of Compute
and Schedule
introduced by Halide

Hardware

Example Schedule Transformation

```python
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
```

```c
for (int i = 0; i < n; ++i) {
  C[i] = A[i] + B[i];
}
```

Example Schedule Transformation

```python
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
```

```c
for (int xo = 0; xo < ceil(n / 32); ++xo) {
  for (int xi = 0; xi < 32; ++xi) {
    int i = xo * 32 + xi;
    if (i < n) {
      C[i] = A[i] + B[i];
    }
  }
}
```

Example Schedule Transformation

```
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
s[C].recorder(xi, xo)
```

```
for (int xi = 0; xi < 32; ++xi) {
  for (int xo = 0; xo < ceil(n / 32); ++xo) {
    int i = xo * 32 + xi;
    if (i < n) {
      C[i] = A[i] + B[i];
    }
  }
}
```

Example Schedule Transformation

```python
C = tvm.compute((n,), lambda i: A[i] + B[i])
s = tvm.create_schedule(C.op)
xo, xi = s[C].split(s[C].axis[0], factor=32)
s[C].recorder(xi, xo)
s[C].bind(xo, tvm.thread_axis("blockIdx.x"))
s[C].bind(xi, tvm.thread_axis("threadIdx.x"))
```

```c
int i = threadIdx.x * 32 + blockIdx.x;
if (i < n) {
   C[i] = A[i] + B[i];
}
```

Key Challenge: Good Space of Schedule

Should contain any knobs that produces a logically equivalent program that runs well on backend models

Must contain the common manual optimization patterns

Need to actively evolve to incorporate new techniques
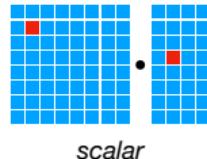
Hardware-aware Search Space
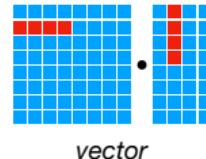
CPUs

Compute Primitives
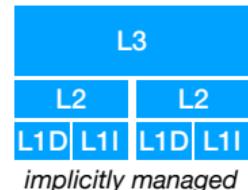
Memory Subsystem

scalar   *vector*

implicitly managed

Loop Transformations   Cache Locality   Vectorization

Reuse primitives from prior work:
Halide, Loopy

Challenge to Support Diverse Hardware Backends
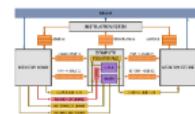
CPUs

GPUs

TPU-like specialized Accelerators
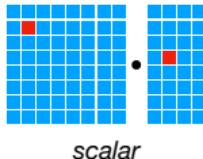
Hardware-aware Search Space
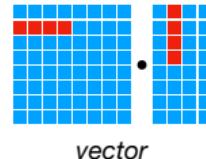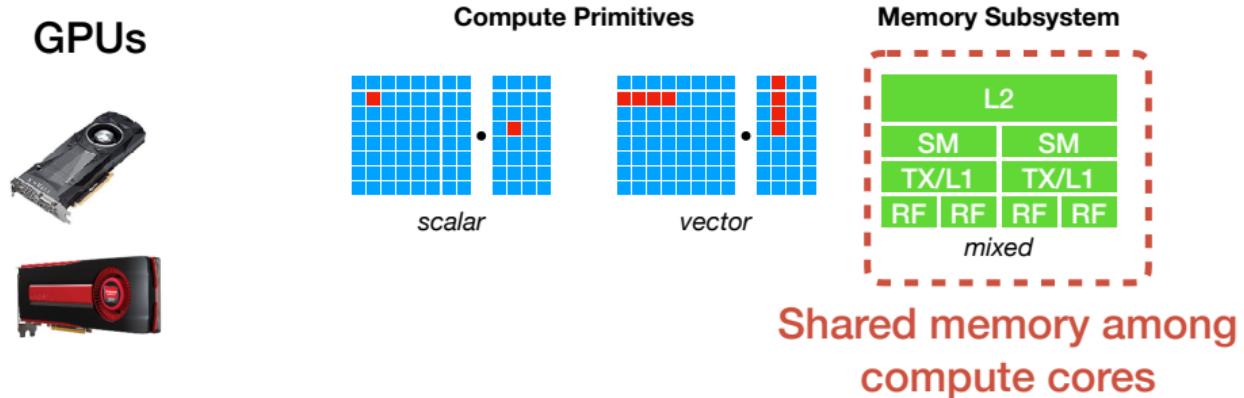
GPUs

Compute Primitives

Memory Subsystem

scalar   *vector*   *mixed*

Hardware-aware Search Space

GPUs

Compute Primitives

scalar

vector

Memory Subsystem

L2

SM   SM

TX/L1   TX/L1

RF   RF   RF   RF

mixed

Shared memory among compute cores

Hardware-aware Search Space

GPUs

Compute Primitives

scalar · vector

Memory Subsystem

L2	
SM	SM
TX/L1	TX/L1
RF	RF	RF	RF

mixed

Shared memory among compute cores

Use of Shared Memory	Thread Cooperation

Hardware-aware Search Space

TPU-like Specialized Accelerators

Compute Primitives

tensor

Memory Subsystem

Unified Buffer

FIFO

Acc

explicitly managed

Hardware-aware Search Space

TPU-like Specialized Accelerators

Compute Primitives

tensor

Memory Subsystem

FIFO

Unified Buffer

Acc

explicitly managed
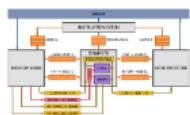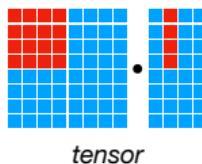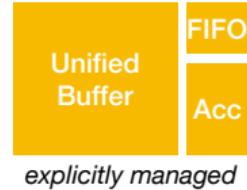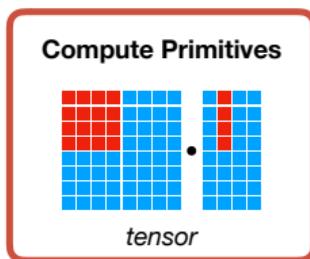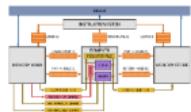
Tensorization Challenge

Compute primitives

Tensorization Challenge
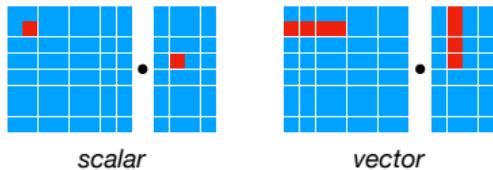
Compute primitives

scalar

Tensorization Challenge

Compute primitives
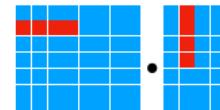
scalar   *vector*

Tensorization Challenge
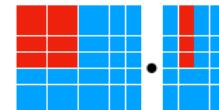
Compute primitives
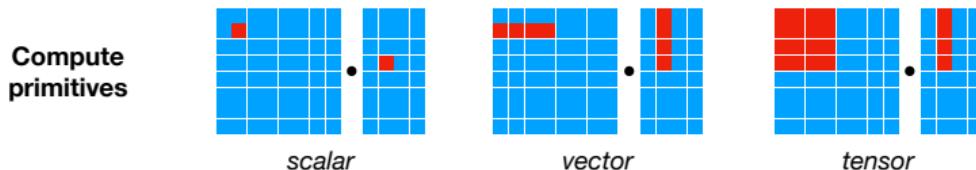
scalar *vector* *tensor*

Tensorization Challenge

Compute primitives

scalar　　　　　*vector*　　　　　*tensor*

Hardware designer:
declare tensor instruction interface
with Tensor Expression

```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
                t.sum(w[i, k] * x[j, k], axis=k))
```

declare behavior

```
def gemm_intrin_lower(inputs, outputs):
    ww_ptr = inputs[0].access_ptr("r")
    xx_ptr = inputs[1].access_ptr("r")
    zz_ptr = outputs[0].access_ptr("w")
    compute = t.hardware_intrin("gemm8x8", ww_ptr, xx_ptr, zz_ptr)
    reset = t.hardware_intrin("fill_zero", zz_ptr)
    update = t.hardware_intrin("fuse_gemm8x8_add", ww_ptr, xx_ptr, zz_ptr)
    return compute, reset, update

gemm8x8 = t.decl_tensor_intrin(y.op, gemm_intrin_lower)
```

lowering rule to generate hardware intrinsics to carry out the computation

Tensorization Challenge

Compute primitives

scalar   *vector*   *tensor*

Hardware designer:
declare tensor instruction interface
with Tensor Expression

Tensorize:
transform program
to use tensor instructions

```
w, x = t.placeholder((8, 8)), t.placeholder((8, 8))
k = t.reduce_axis((0, 8))
y = t.compute((8, 8), lambda i, j:
              t.sum(w[i, k] * x[j, k], axis=k))
```
declare behavior

```
def gemm_intrin_lower(inputs, outputs):
    ww_ptr = inputs[0].access_ptr("r")
    xx_ptr = inputs[1].access_ptr("r")
    zz_ptr = outputs[0].access_ptr("w")
    compute = t.hardware_intrin("gemm8x8", ww_ptr, xx_ptr, zz_ptr)
    reset = t.hardware_intrin("fill_zero", zz_ptr)
    update = t.hardware_intrin("fuse_gemm8x8_add", ww_ptr, xx_ptr, zz_ptr)
    return compute, reset, update

gemm8x8 = t.decl_tensor_intrin(y.op, gemm_intrin_lower)
```
lowering rule to generate
hardware intrinsics to carry
out the computation

scalar   *tensor*

Hardware-aware Search Space

TPU-like Specialized Accelerators

Compute Primitives

tensor

Memory Subsystem

FIFO

Unified Buffer

Acc

explicitly managed

Hardware-aware Search Space

TPU-like Specialized Accelerators

Compute Primitives

tensor

Memory Subsystem

Unified Buffer

FIFO

Acc

explicitly managed

Software Support for Latency Hiding

Single Module
No Task-Pipelining

| load inputs | load weights | matrix multiplication | store outputs | load inputs | load weights | matrix multiplication | store outputs |

Software Support for Latency Hiding

**Single Module
No Task-Pipelining**

| load inputs | load weights | matrix multiplication | store outputs | load inputs | load weights | matrix multiplication | store outputs |

**Multiple-Module
Task-Level Pipelining**

| load inputs | load weights | load inputs | load weights |

| matrix multiplication | matrix multiplication |

| store outputs | store outputs |

Software Support for Latency Hiding

Single Module
No Task-Pipelining

| load inputs | load weights | matrix multiplication | store outputs | load inputs | load weights | matrix multiplication | store outputs |

Multiple-Module
Task-Level Pipelining

Explicit dependency tracking
managed by software to hide memory latency
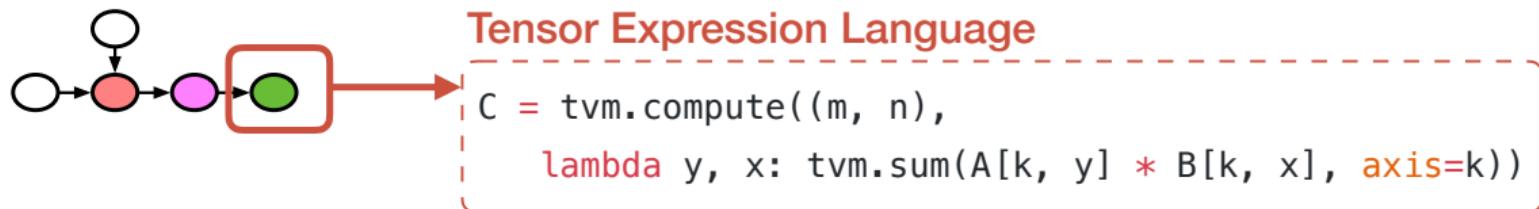
Hardware-aware Search Space

Tensor Expression Language

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Hardware

Hardware-aware Search Space

Tensor Expression Language

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Primitives in prior work: Halide, Loopy
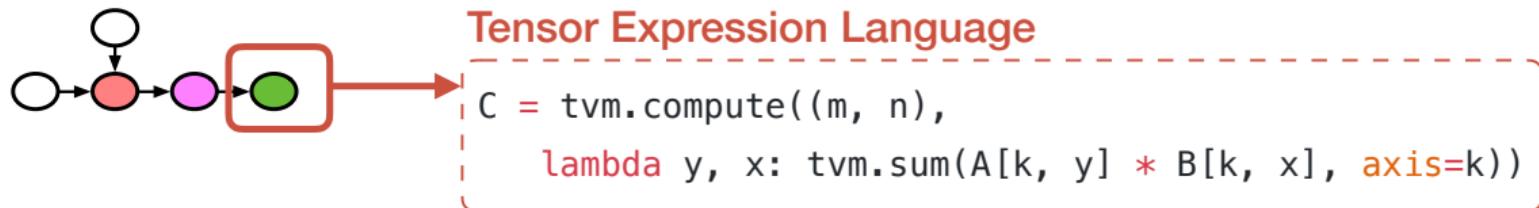
| Loop Transformations | Thread Bindings | Cache Locality |

Hardware

Hardware-aware Search Space

Tensor Expression Language

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Primitives in prior work: Halide, Loopy

| Loop Transformations | Thread Bindings | Cache Locality |

New primitives for GPUs, and enable TPU-like Accelerators

| Thread Cooperation | Tensorization | Latency Hiding | ... |
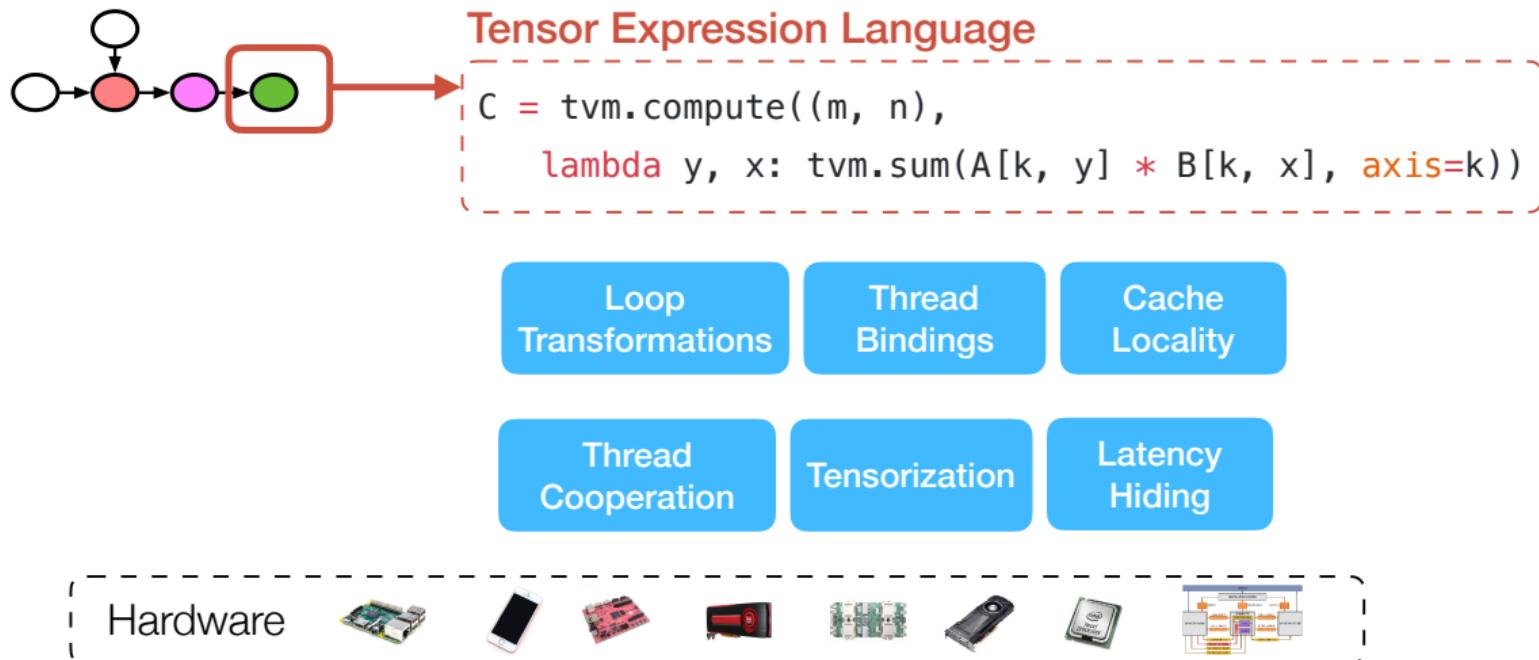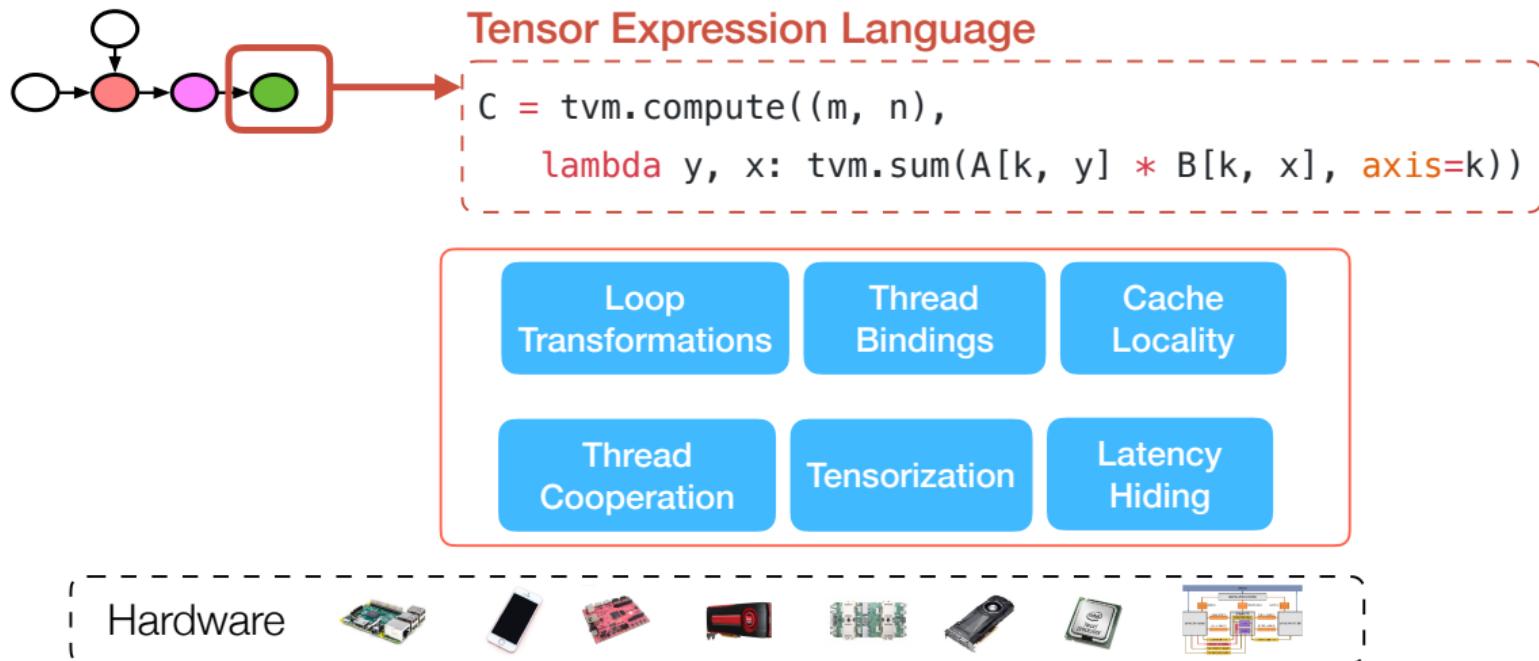
Hardware

Hardware-aware Search Space

Tensor Expression Language

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Loop Transformations	Thread Bindings	Cache Locality

Thread Cooperation	Tensorization	Latency Hiding

Hardware

Hardware-aware Search Space

Tensor Expression Language

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```

Loop Transformations	Thread Bindings	Cache Locality
Thread Cooperation	Tensorization	Latency Hiding

Hardware

Hardware-aware Search Space

Tensor Expression Language

```
C = tvm.compute((m, n),
    lambda y, x: tvm.sum(A[k, y] * B[k, x], axis=k))
```
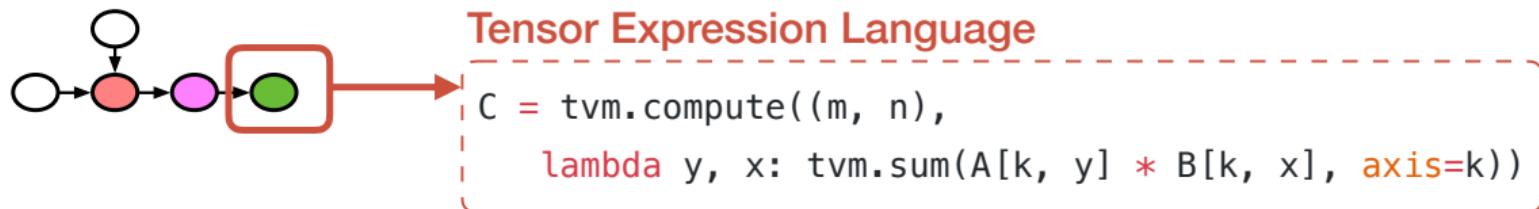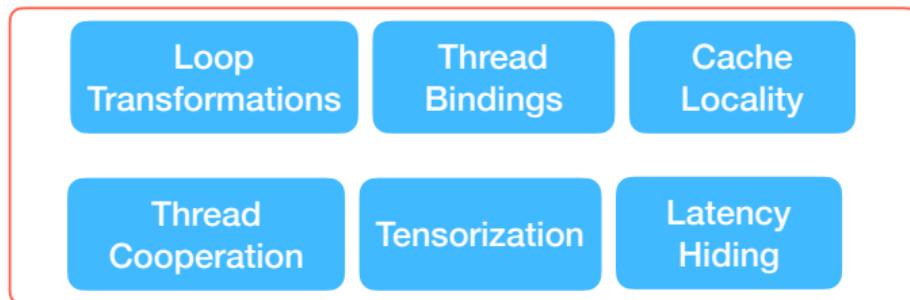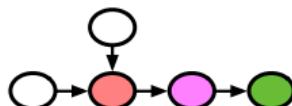
Billions of possible optimization choices

- Loop Transformations
- Thread Bindings
- Cache Locality
- Thread Cooperation
- Tensorization
- Latency Hiding
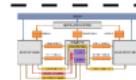
Hardware

Learning-based Learning System

Frameworks

High-level data flow graph and optimizations

Hardware aware Search Space of Optimized Tensor Programs

Machine Learning based Program Optimizer
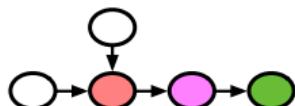
Hardware

Learning-based Learning System

Frameworks
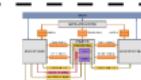
High-level data flow graph and optimizations

Hardware aware Search Space of Optimized Tensor Programs
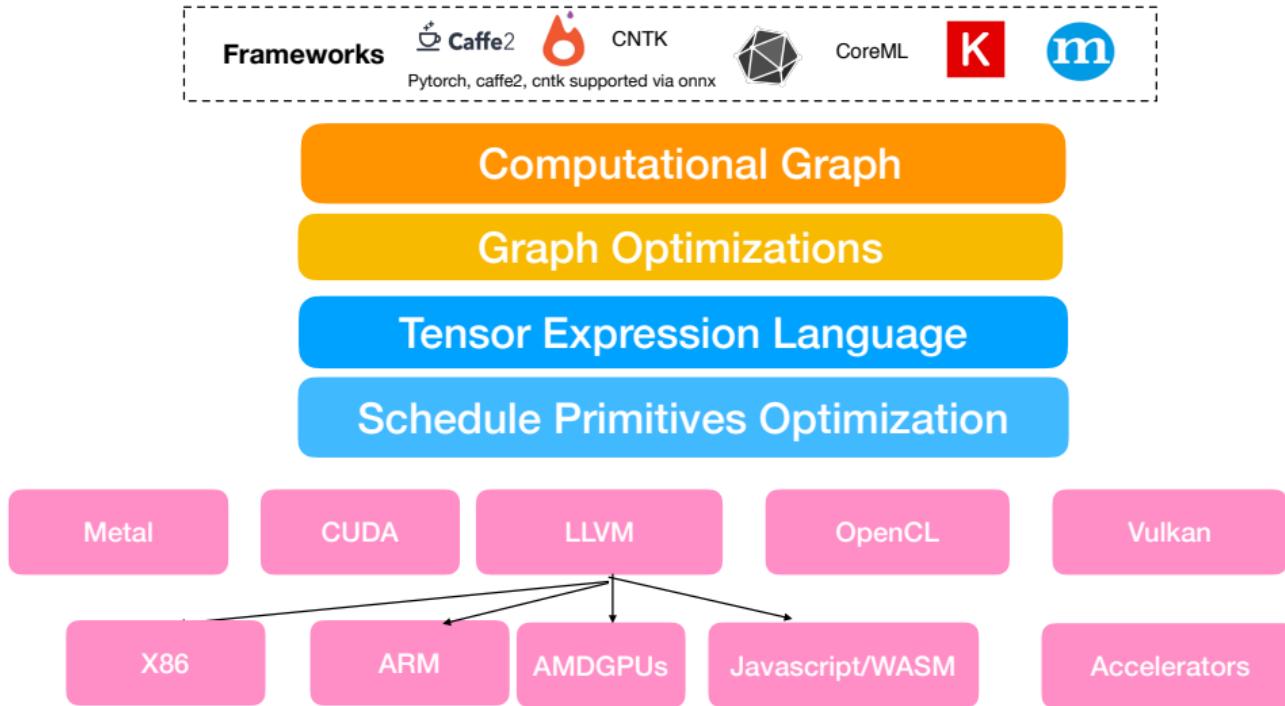
Machine Learning based Program Optimizer

Hardware

Global View of TVM Stack
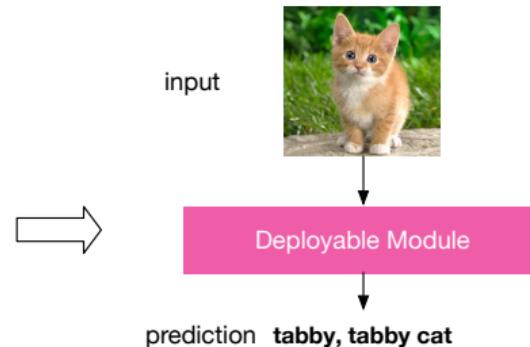
High Level Compilation Frontend

```python
module = runtime.create(graph, lib, tvm.gpu(0))
module.set_input(**params)
module.run(data=data_array)
output = tvm.nd.empty(out_shape, ctx=tvm.gpu(0))
module.get_output(0, output)
```

```python
import tvm
import nnvm.frontend
import nnvm.compiler

graph, params =
nnvm.frontend.from_keras(keras_resnet50)
graph, lib, params =
    nnvm.compiler.build(graph, target)
```

input

Deployable Module

prediction **tabby, tabby cat**

On languages and platforms you choose

Program Your Phone with Python from Your Laptop

RPC Server on
Embedded Device

Compiler Stack

upload module to remote
get remote function

copy data to remote
get remote array handle

run function on remote
get profile statistics back
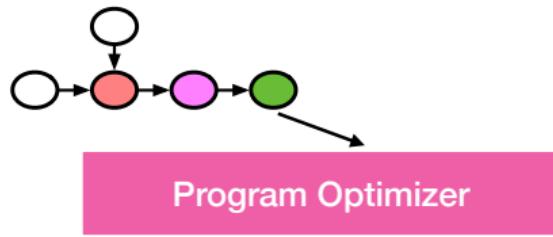
copy data back to host
for correctness verification

```python
lib = t.build(s, [A, B],
    'llvm –target=armv7l–none–linux–gnueabihf',
     name='myfunc')
remote = t.rpc.connect(host, port)
lib.save('myfunc.o')
remote.upload('myfunc.o')
f = remote.load_module('myfunc.o')
ctx = remote.cpu(0)
a = t.nd.array(np.random.uniform(size=1024), ctx)
b = t.nd.array(np.zeros(1024), ctx)
remote_timer = f.time_evaluator('myfunc', ctx, number=10)
time_cost = remote_timer(a, b)

np.testing.assert_equal(b.asnumpy(), expected)
```
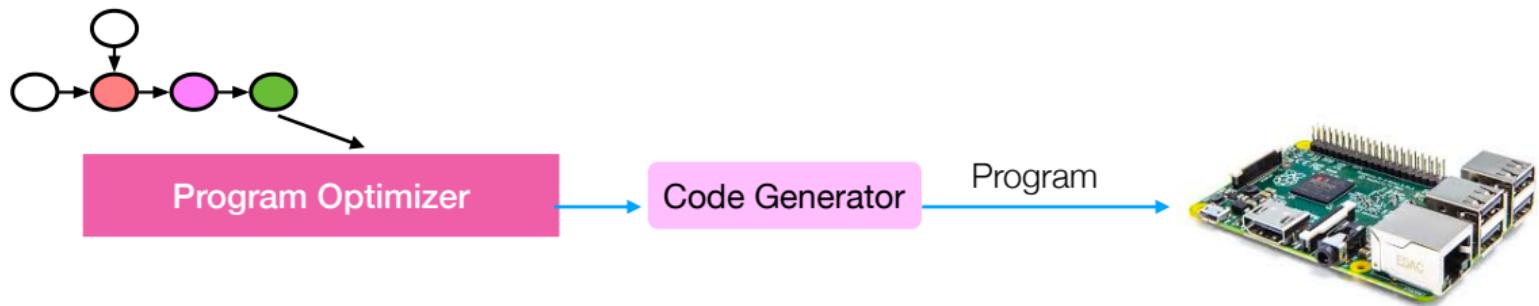
Learning-based Program Optimizer
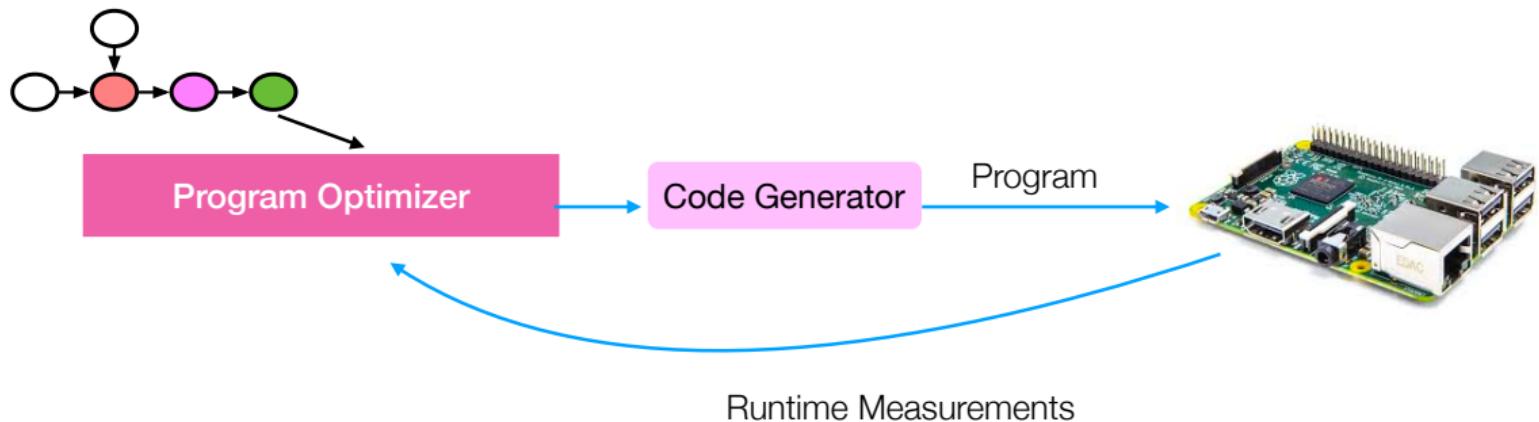
Program Optimizer

Learning-based Program Optimizer
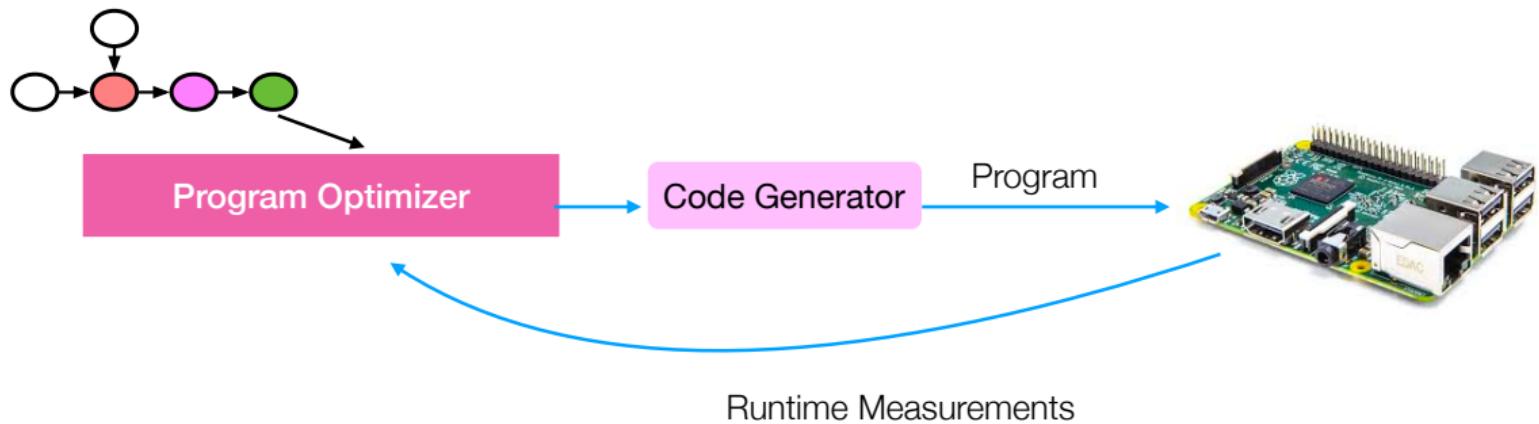
Learning-based Program Optimizer
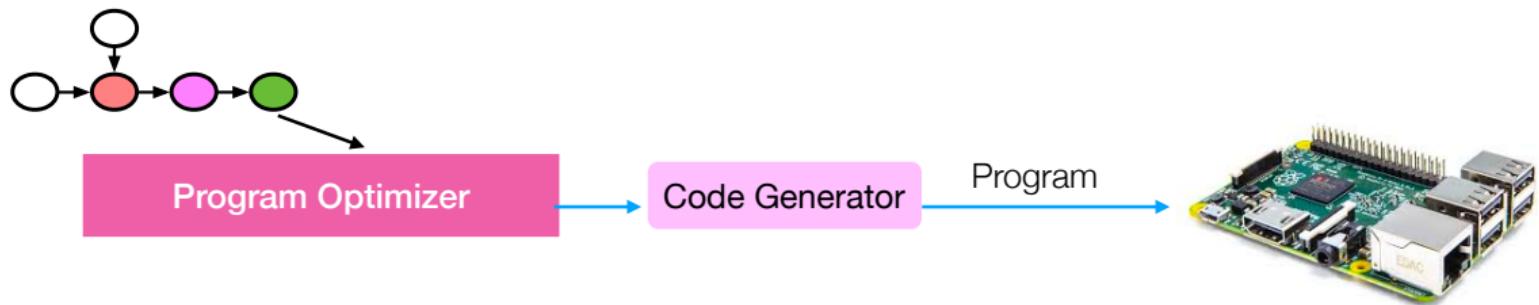
Learning-based Program Optimizer

High experiment cost, each trial costs ~1second
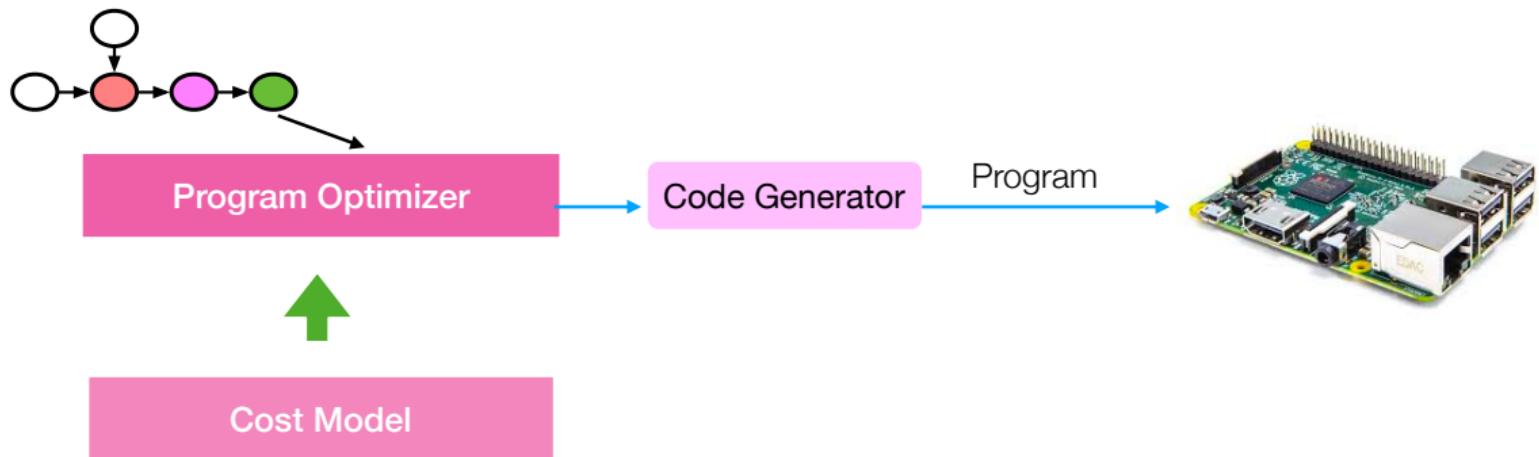
Learning-based Program Optimizer
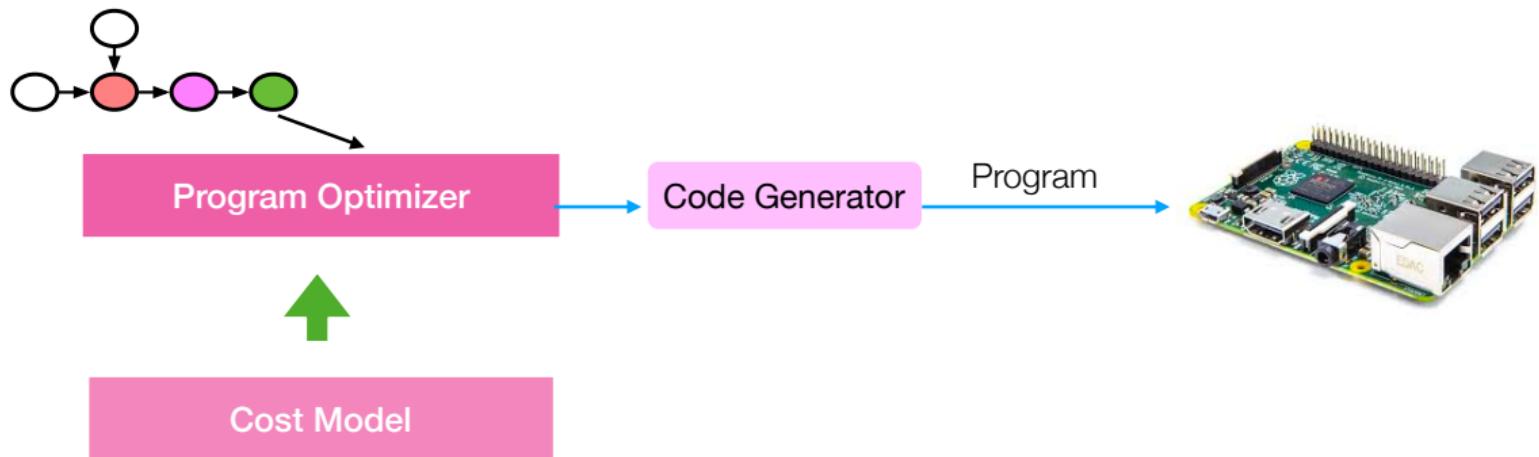
Learning-based Program Optimizer
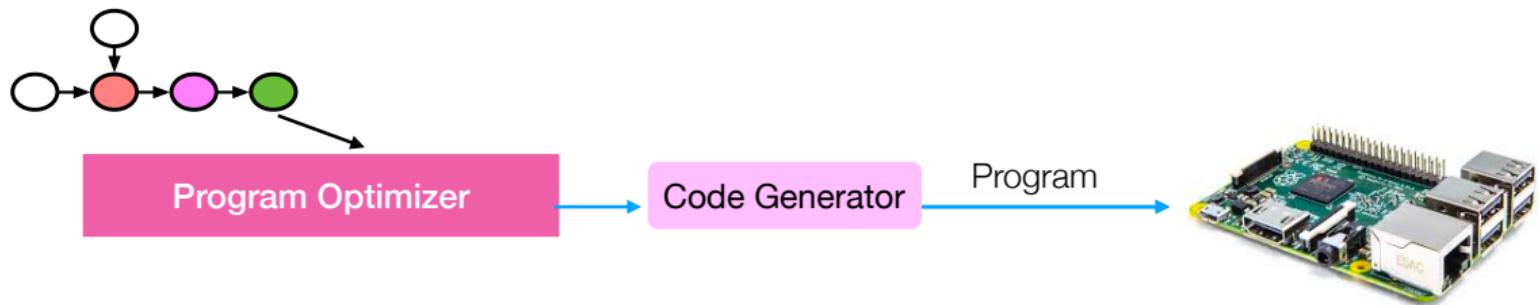
Learning-based Program Optimizer

Need reliable cost model per hardware
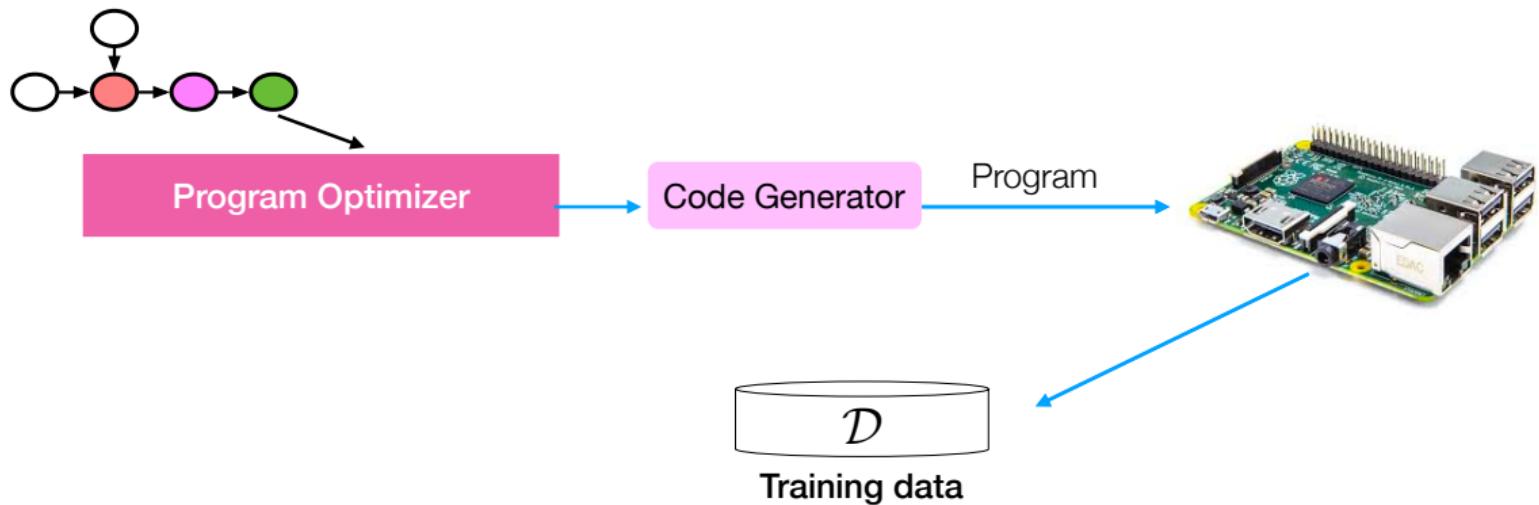
Learning-based Program Optimizer
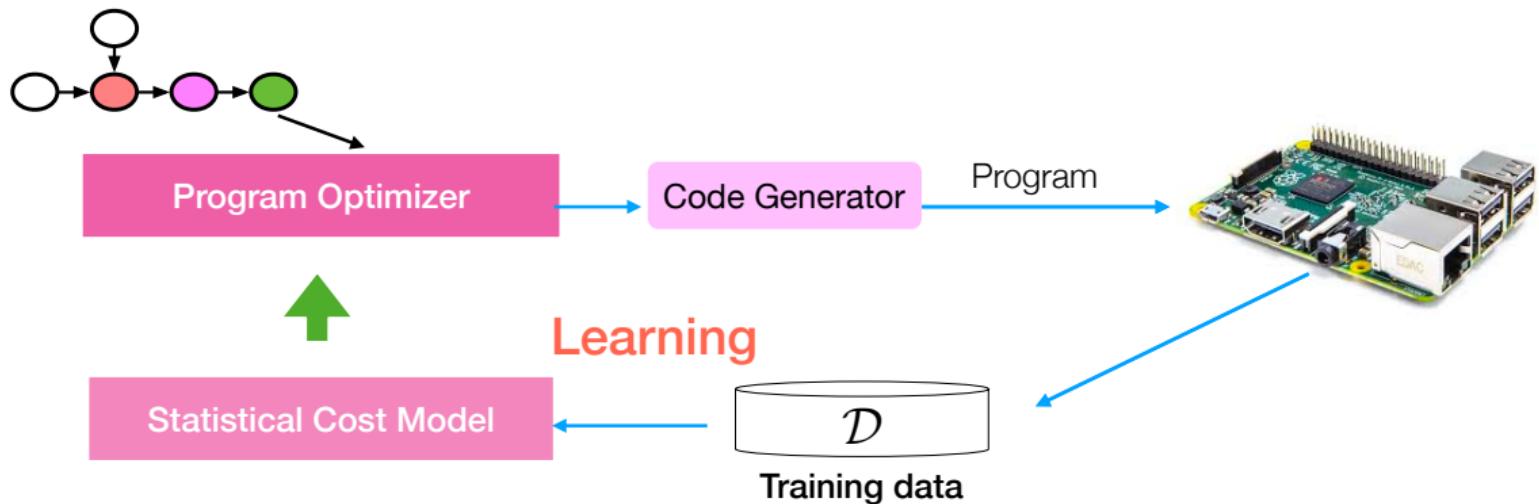
Learning-based Program Optimizer

Learning-based Program Optimizer

Learning-based Program Optimizer

Program Optimizer → Code Generator → Program

Learning

Statistical Cost Model ← $\mathcal{D}$ Training data
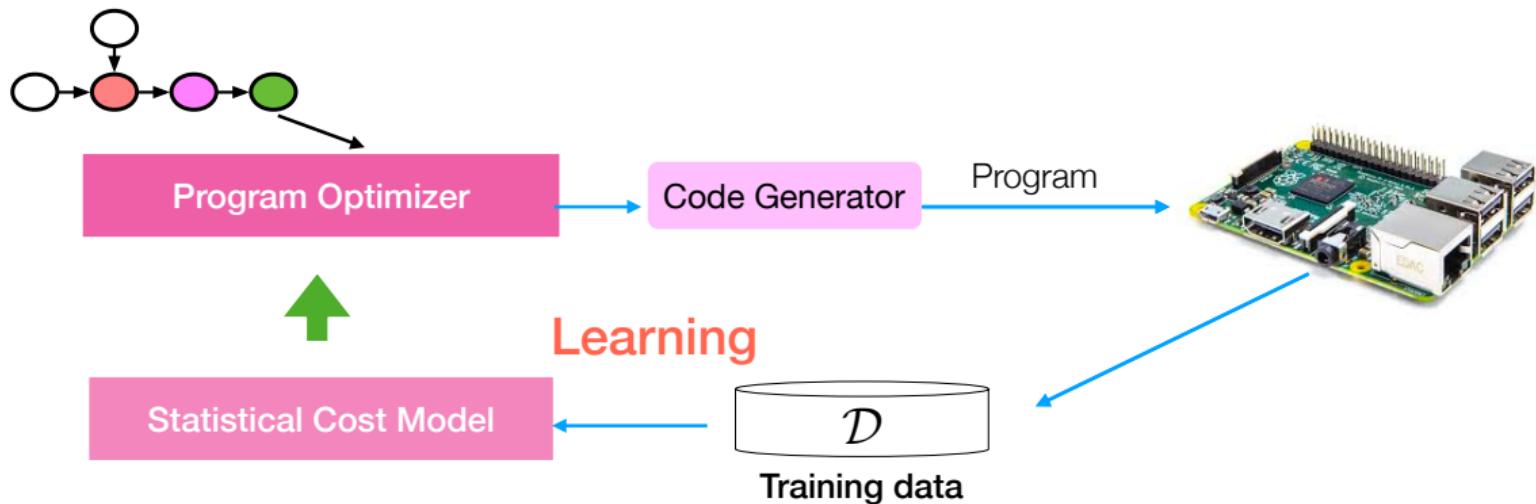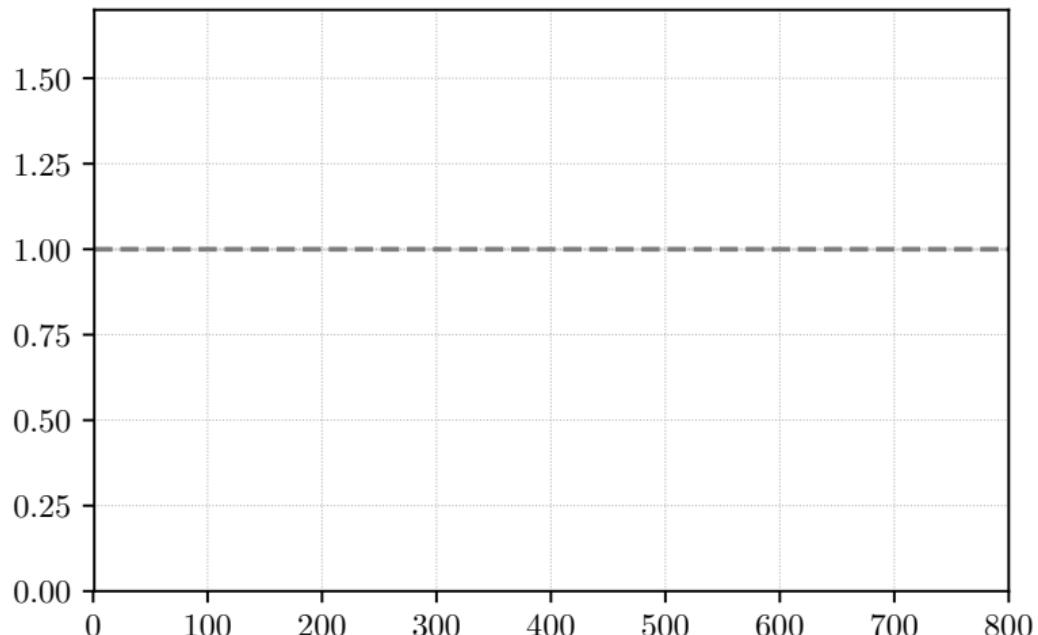
Adapt to hardware type by learning
Make prediction in 1ms level
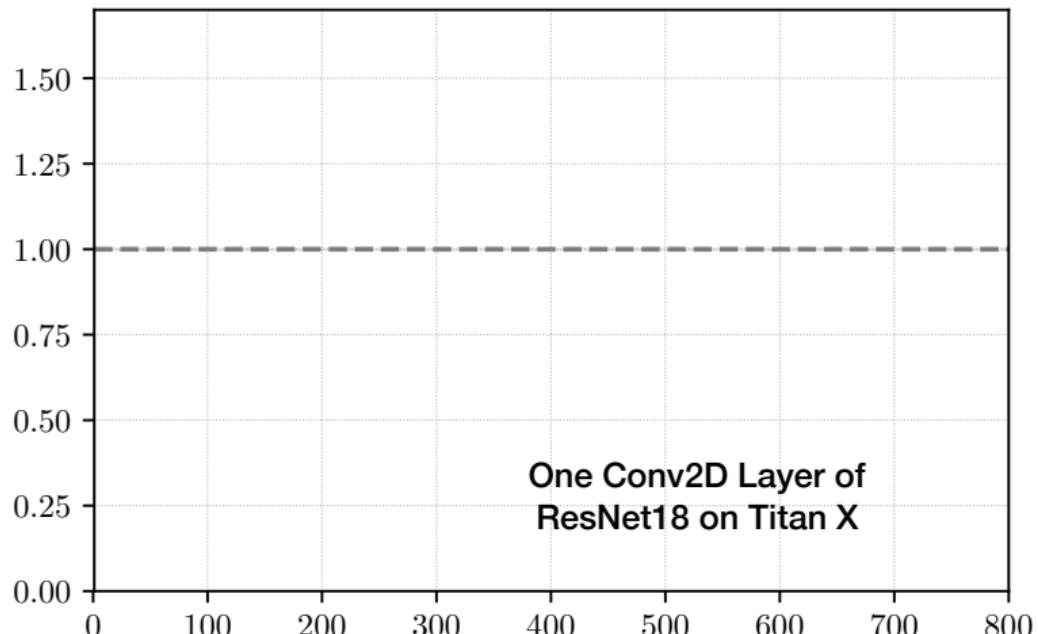
Effectiveness of ML based Model
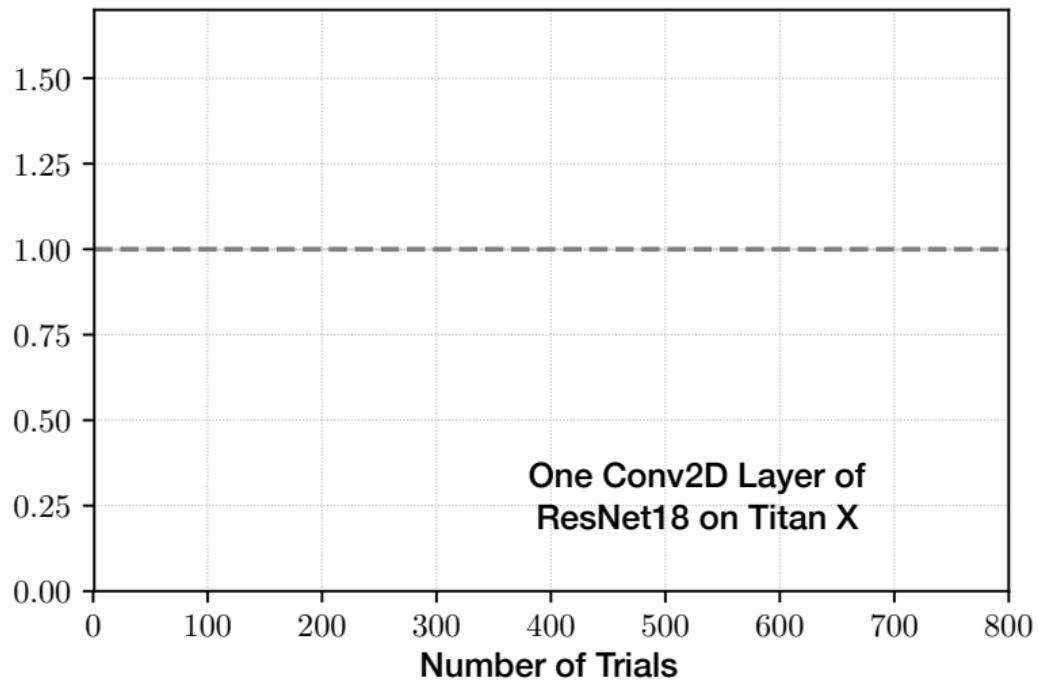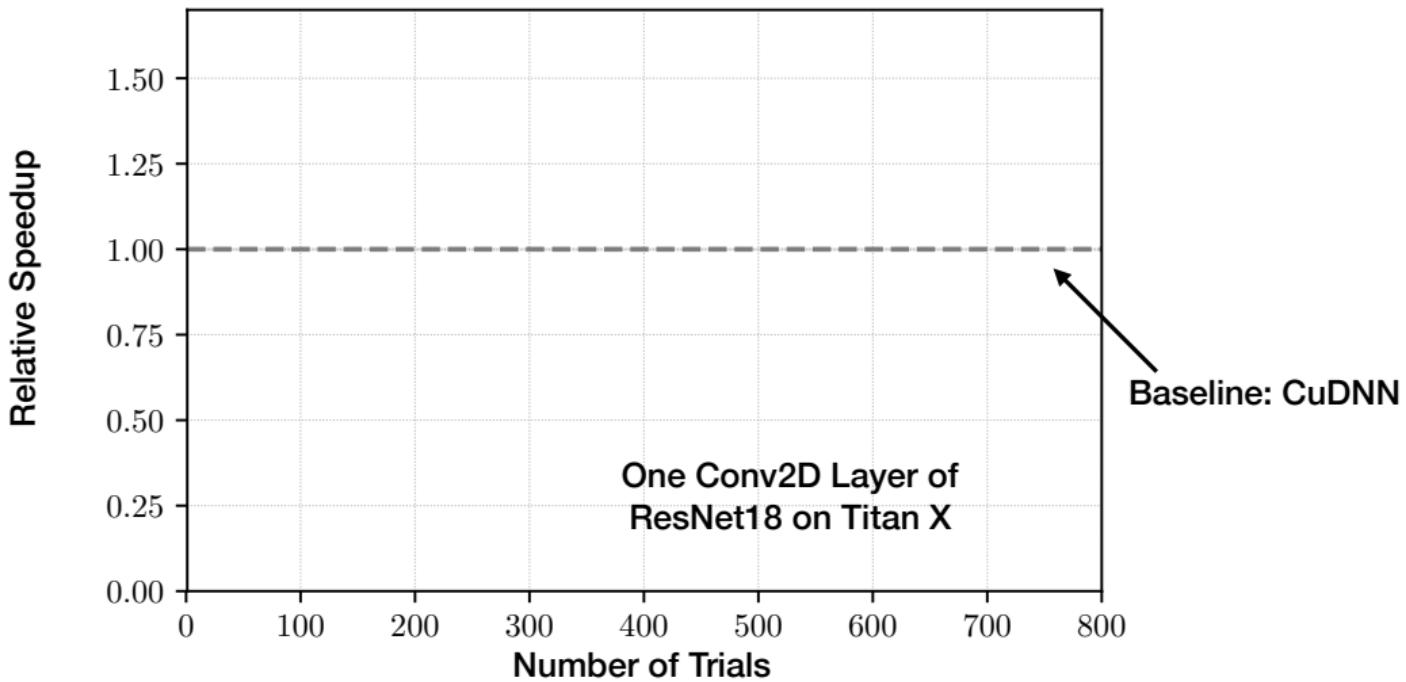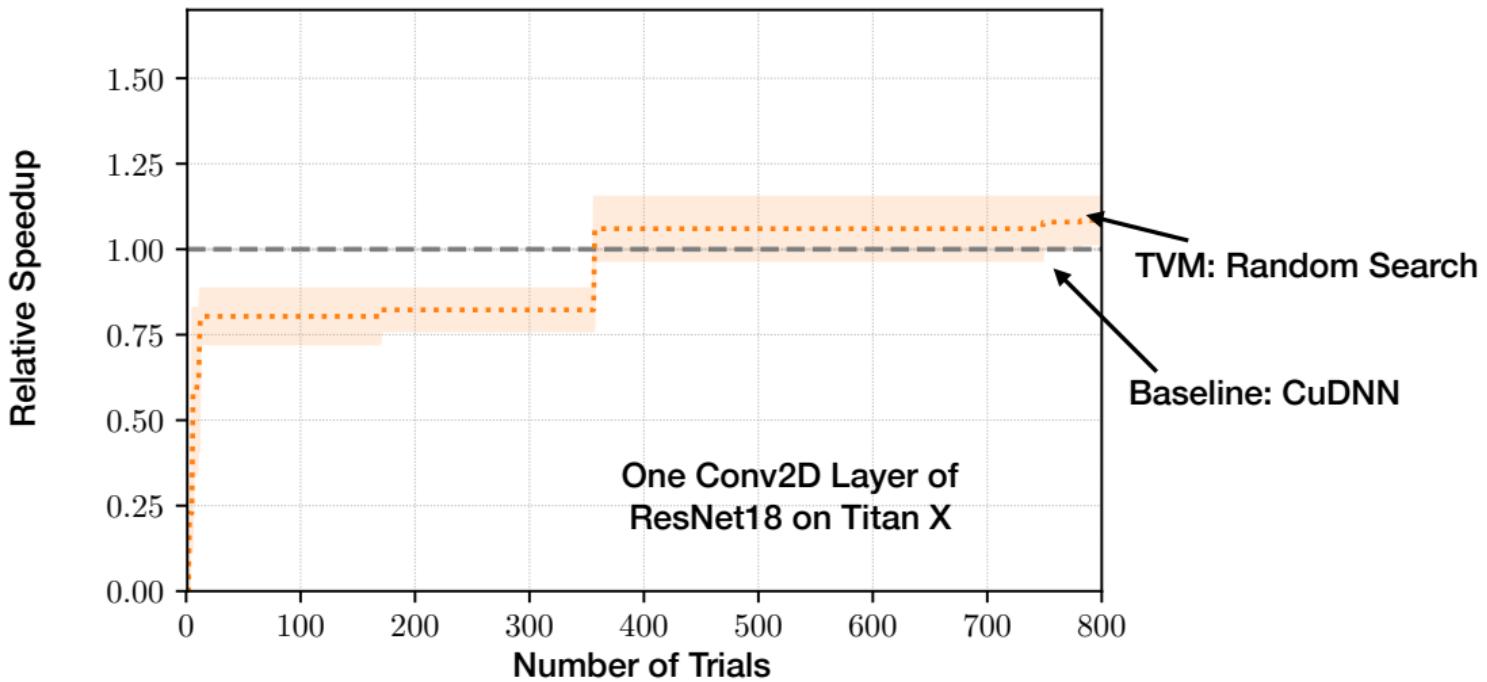
Effectiveness of ML based Model

One Conv2D Layer of ResNet18 on Titan X

Effectiveness of ML based Model

One Conv2D Layer of
ResNet18 on Titan X

Number of Trials

Effectiveness of ML based Model

Effectiveness of ML based Model

One Conv2D Layer of
ResNet18 on Titan X

Effectiveness of ML based Model

Relative Speedup vs Number of Trials

- TVM: ML-based Model
- TVM: Random Search
- Baseline: CuDNN

One Conv2D Layer of
ResNet18 on Titan X
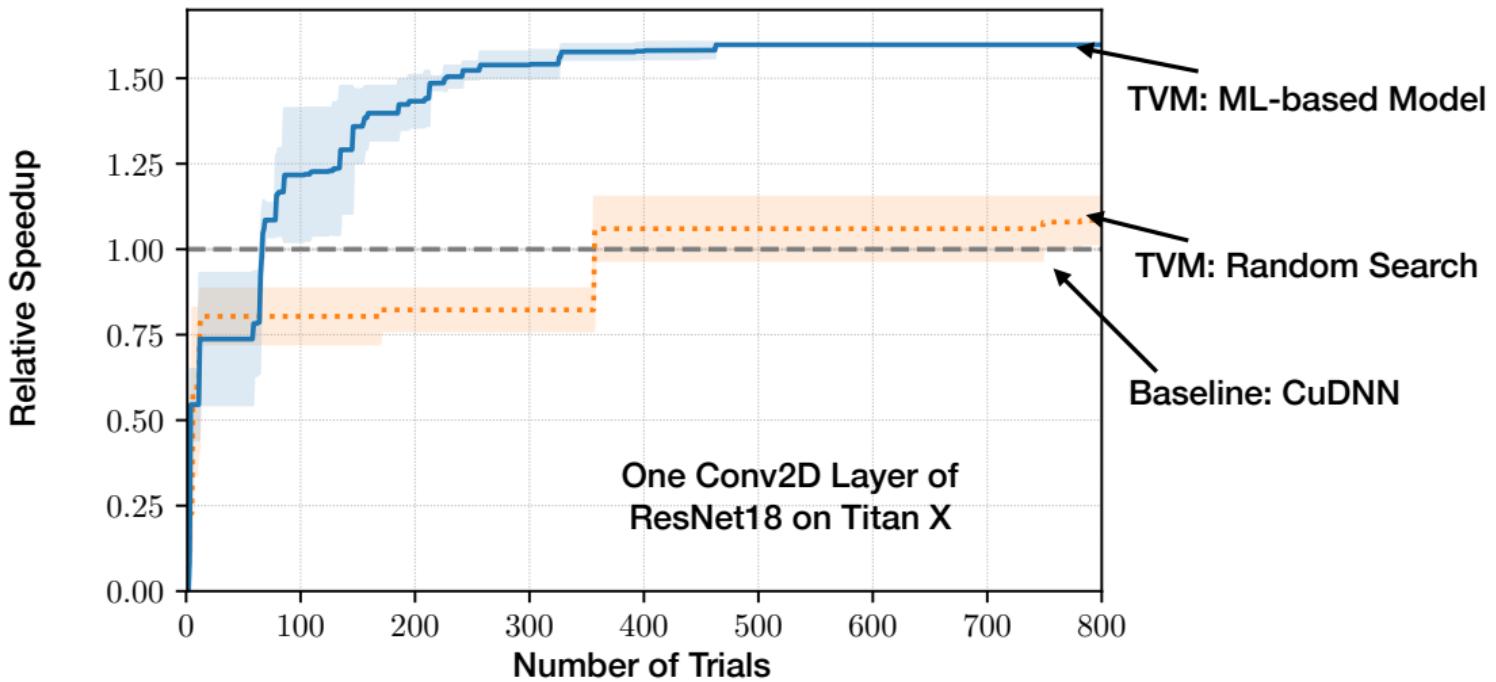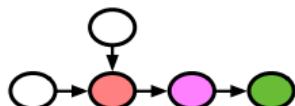
Learning-based Learning System

Frameworks
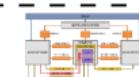
High-level data flow graph and optimizations

Hardware aware Search Space of Optimized Tensor Programs

Machine Learning based Program Optimizer
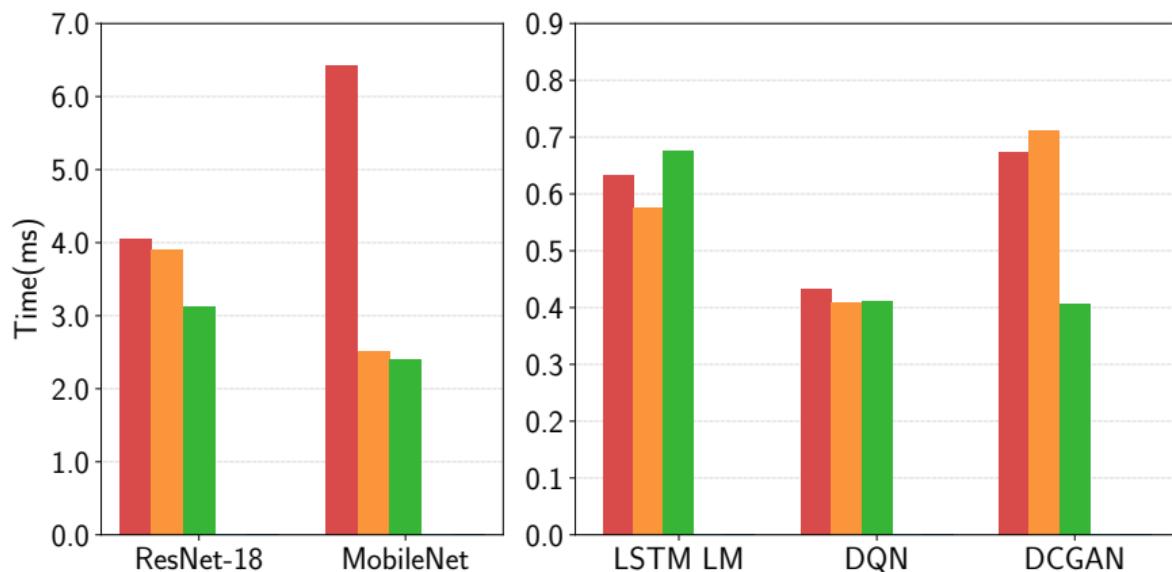
Hardware

End to End Inference Performance (Nvidia Titan X)
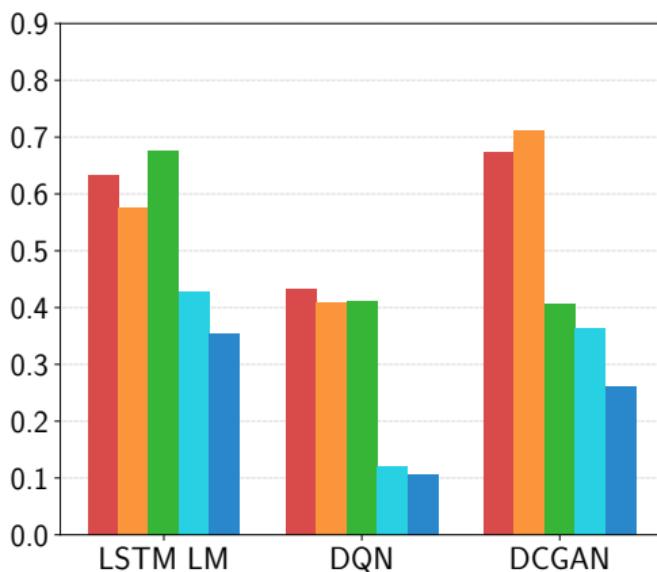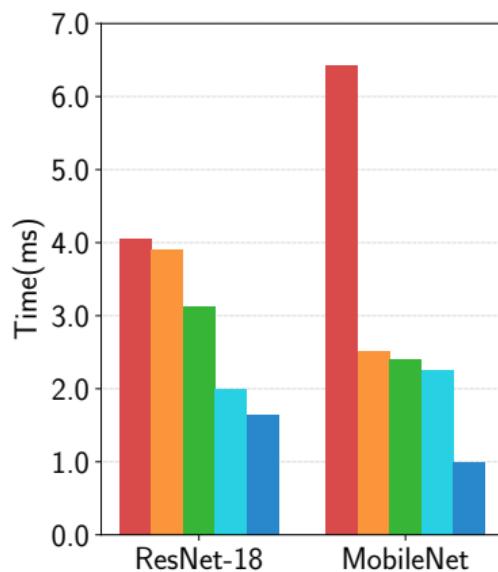
End to End Inference Performance (Nvidia Titan X)

End to End Inference Performance (Nvidia Titan X)

End to End Inference Performance (Nvidia Titan X)

End to End Inference Performance (Nvidia Titan X)

Legend: Tensorflow, Tensorflow-XLA, Apache MXNet, TVM: without graph optimizations, TVM: all optimizations

Competitive on standard models

End to End Inference Performance (Nvidia Titan X)
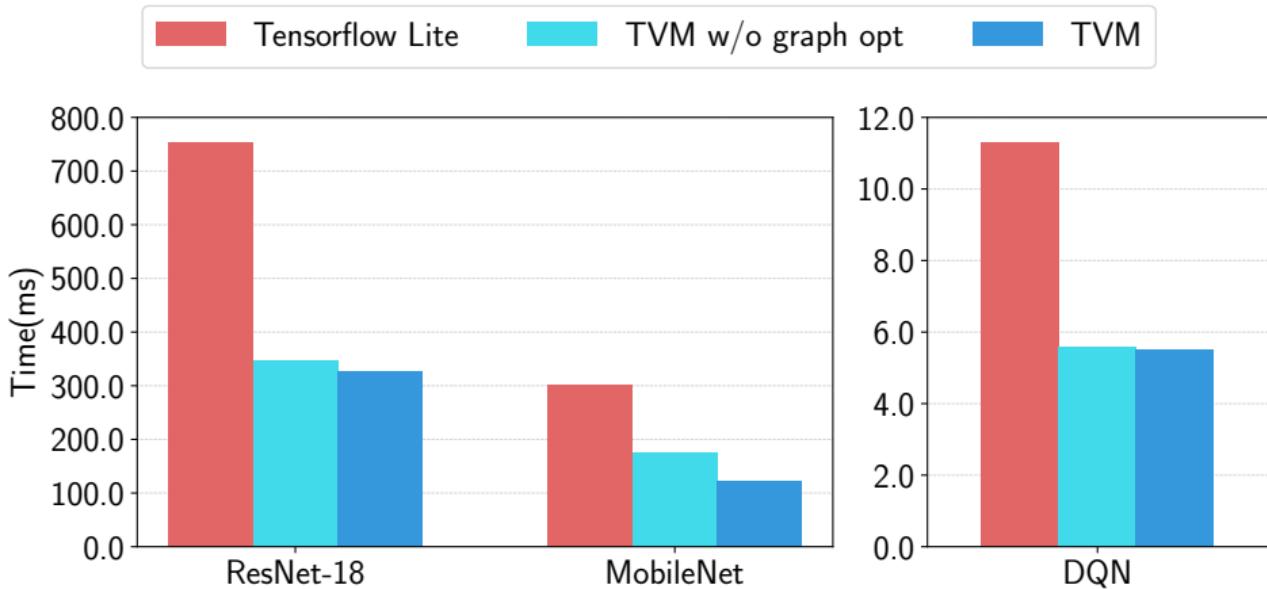
Legend: Tensorflow, Tensorflow-XLA, Apache MXNet, TVM: without graph optimizations, TVM: all optimizations

Models: ResNet-18, MobileNet, LSTM LM, DQN, DCGAN

Competitive on standard models

Bigger gap on less conventional models
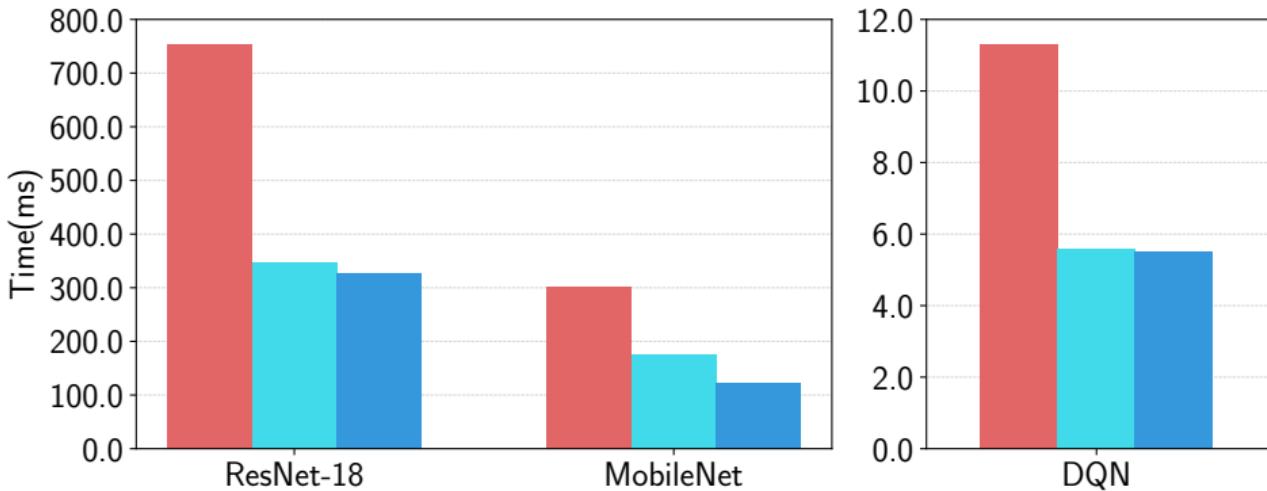
End to End Performance(ARM Cortex-A53)
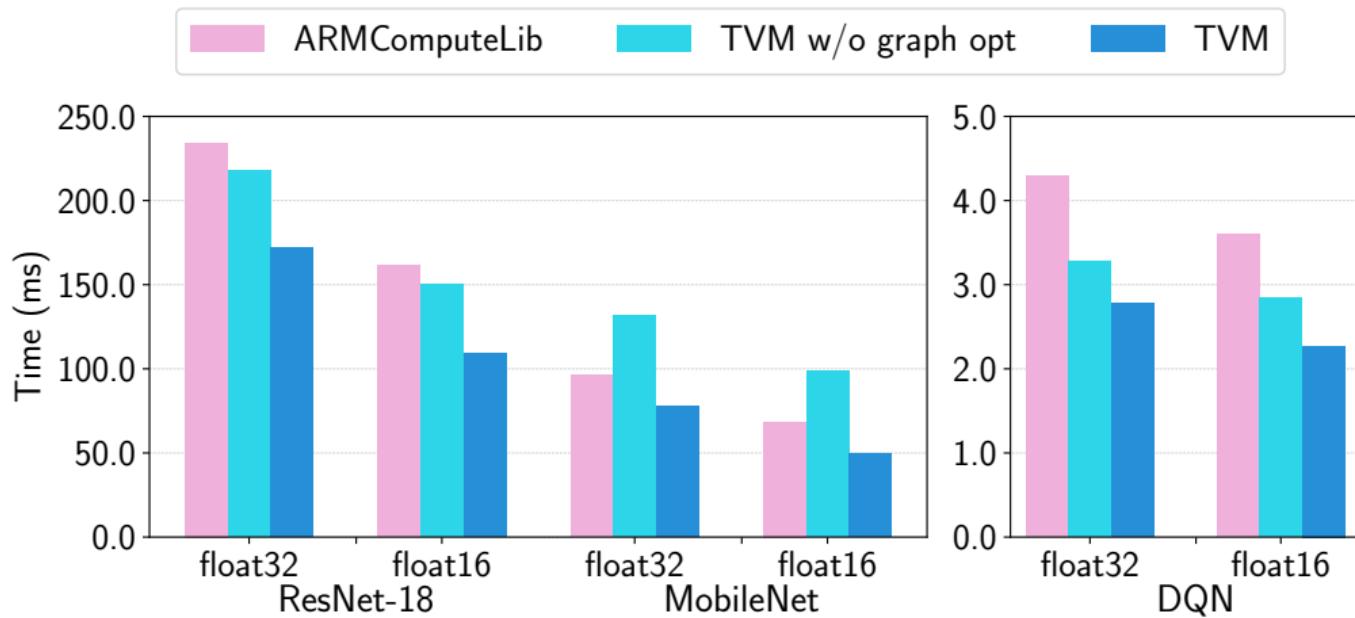
End to End Performance(ARM Cortex-A53)

Specially optimized for Embedded system(ARM)

Legend: Tensorflow Lite, TVM w/o graph opt, TVM
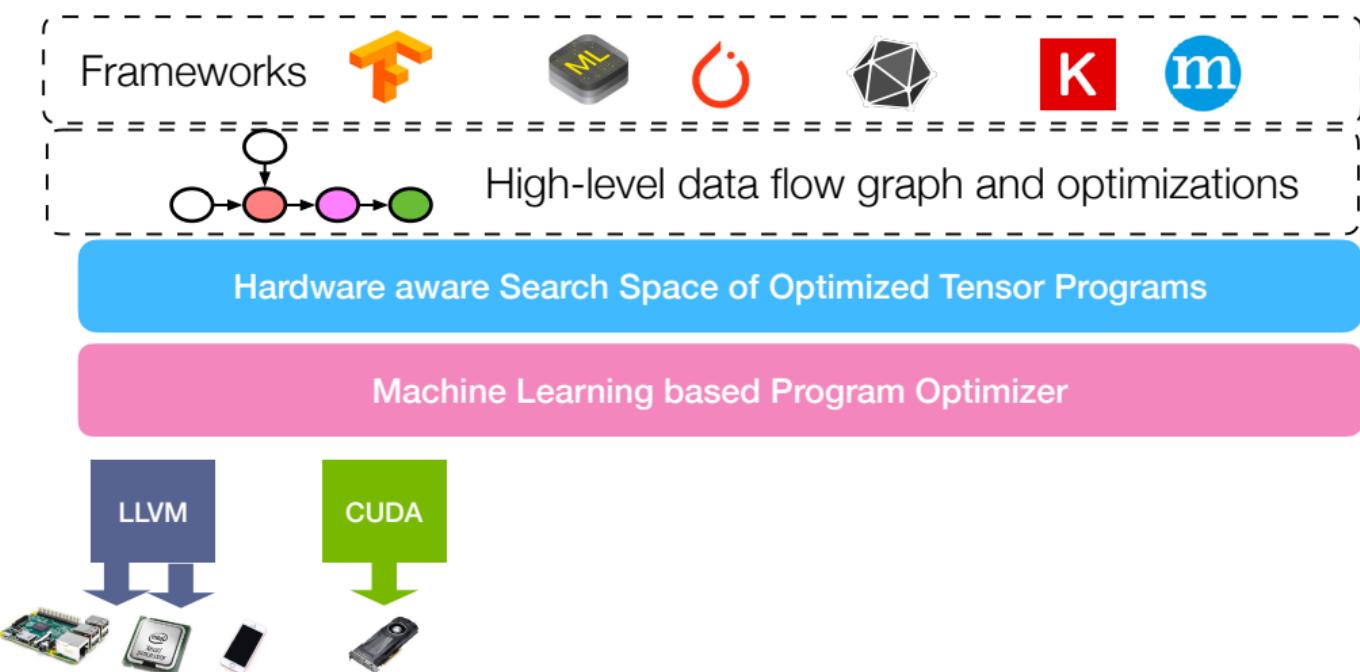
End to End Performance(ARM GPU)
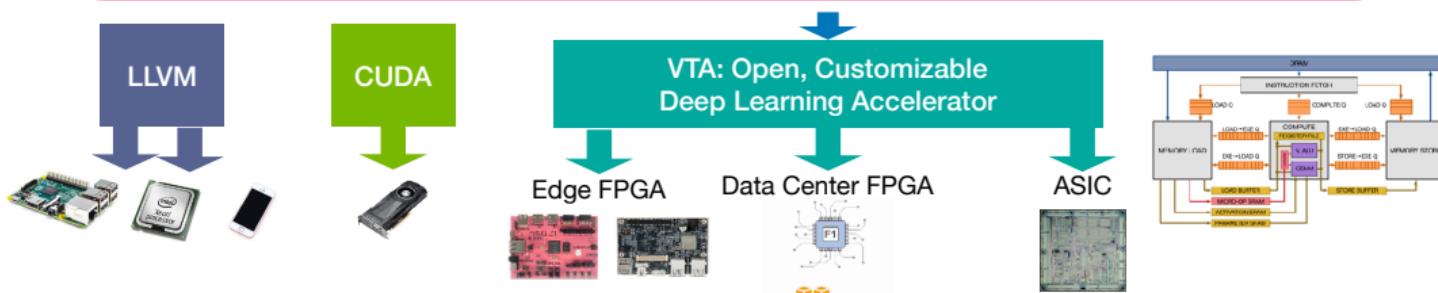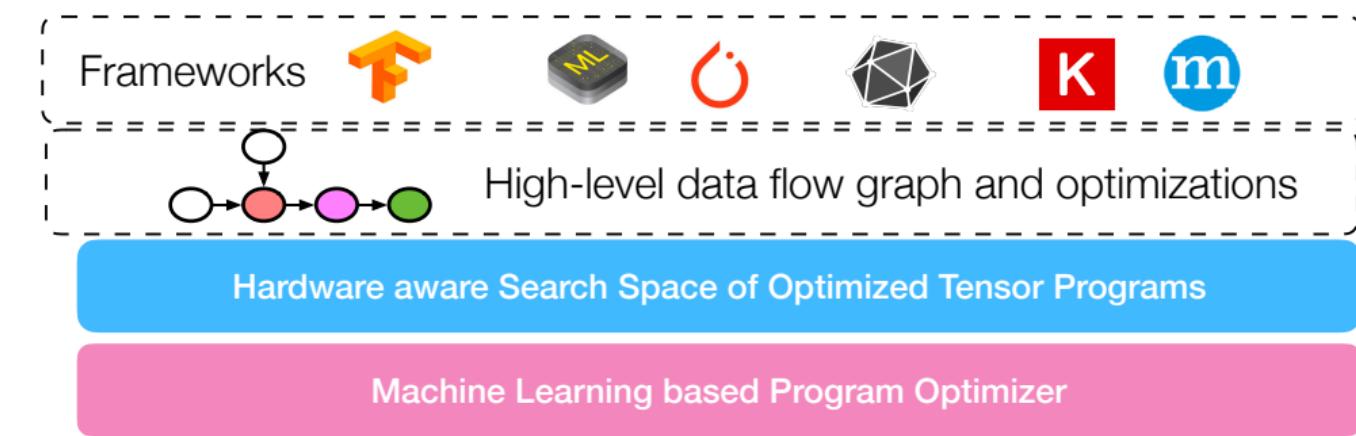
Supporting New Specialized Accelerators

Supporting New Specialized Accelerators

Frameworks

High-level data flow graph and optimizations

Hardware aware Search Space of Optimized Tensor Programs

Machine Learning based Program Optimizer

LLVM

CUDA

VTA: Open, Customizable Deep Learning Accelerator

Edge FPGA

Data Center FPGA

ASIC

36

TVM/VTA: Full Stack Open Source System

High-level Optimizations

Tensor Program Search Space

ML-based Optimizer

TVM/VTA: Full Stack Open Source System

TVM/VTA: Full Stack Open Source System

High-level Optimizations

Tensor Program Search Space

ML-based Optimizer

VTA Hardware/Software Interface (ISA)

VTA MicroArchitecture

TVM/VTA: Full Stack Open Source System

High-level Optimizations

Tensor Program Search Space

ML-based Optimizer

VTA Runtime & JIT Compiler

VTA Hardware/Software Interface (ISA)

VTA MicroArchitecture

TVM/VTA: Full Stack Open Source System

TVM/VTA: Full Stack Open Source System

High-level Optimizations

Tensor Program Search Space

ML-based Optimizer

VTA Runtime & JIT Compiler

VTA Hardware/Software Interface (ISA)

VTA MicroArchitecture

VTA Simulator

- JIT compile accelerator micro code

- Support heterogenous devices, 10x better than CPU on the same board.

- Move hardware complexity to software

TVM/VTA: Full Stack Open Source System
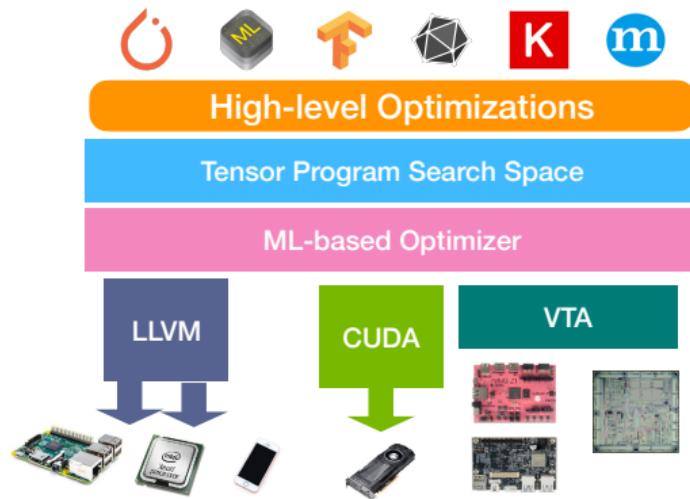
High-level Optimizations

Tensor Program Search Space

ML-based Optimizer

VTA Runtime & JIT Compiler

VTA Hardware/Software Interface (ISA)

VTA MicroArchitecture | VTA Simulator

- JIT compile accelerator micro code

- Support heterogenous devices, 10x better than CPU on the same board.

- Move hardware complexity to software

compiler, driver, hardware design full stack open source

TVM: Learning-based Learning System

Check it out!

tvm.ai
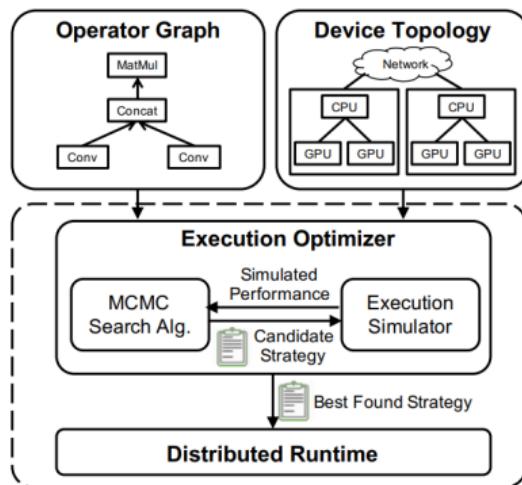
SOTA Solutions

- "The optimizer uses a MCMC search algorithm to explore the space of possible parallelization strategies and iteratively proposes candidate strategies that are evaluated by a execution simulator."

[1]
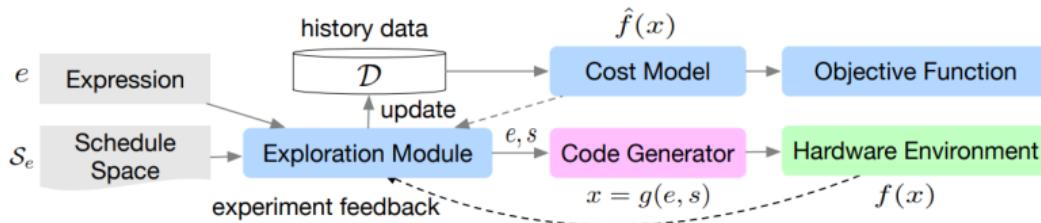
[1]Zhihao Jia, Matei Zaharia, and Alex Aiken (2019). "Beyond Data and Model Parallelism for Deep Neural Networks". In: *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019.* Ed. by Ameet Talwalkar, Virginia Smith, and Matei Zaharia. mlsys.org. URL: https://proceedings.mlsys.org/book/265.pdf.

- "We learn domain-specific statistical cost models to guide the search of tensor operator implementations over billions of possible program variants. We further accelerate the search using effective model transfer across workloads."

[2]
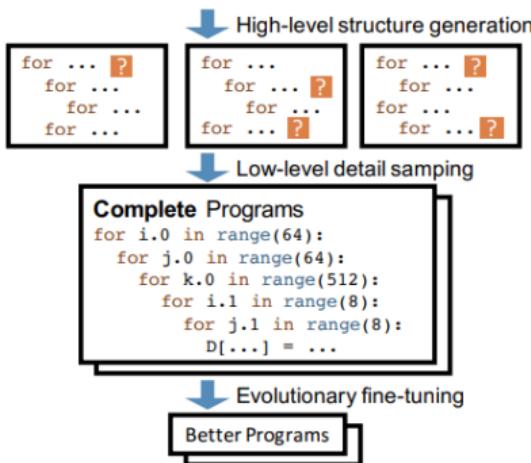
[2] Tianqi Chen et al. (2018). "Learning to Optimize Tensor Programs". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. Ed. by Samy Bengio et al., pp. 3393–3404. URL: http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs.

- "We present Ansor, a tensor program generation framework for deep learning applications. Compared with existing search strategies, Ansor explores much more optimization combinations by sampling programs from a hierarchical representation of the search space."

[3]
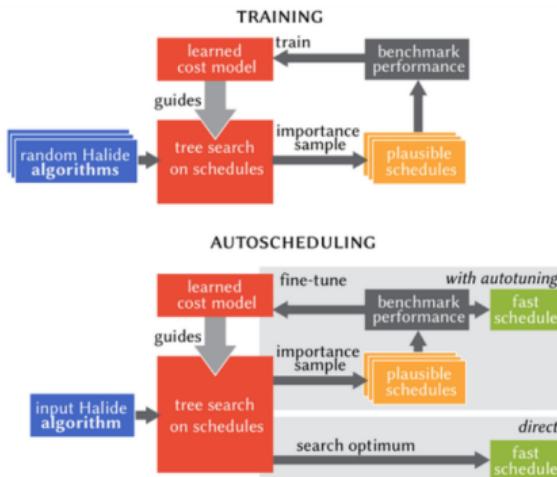
[3]Lianmin Zheng et al. (2020). "Ansor : Generating High-Performance Tensor Programs for Deep Learning". In: *CoRR* abs/2006.06762. arXiv: 2006.06762. URL: https://arxiv.org/abs/2006.06762.

- "We generate schedules for Halide programs using tree search over the space of schedules guided by a learned cost model and optional autotuning. The cost model is trained by benchmarking thousands of randomly-generated Halide programs and schedules. The resulting code significantly outperforms prior work and human experts."
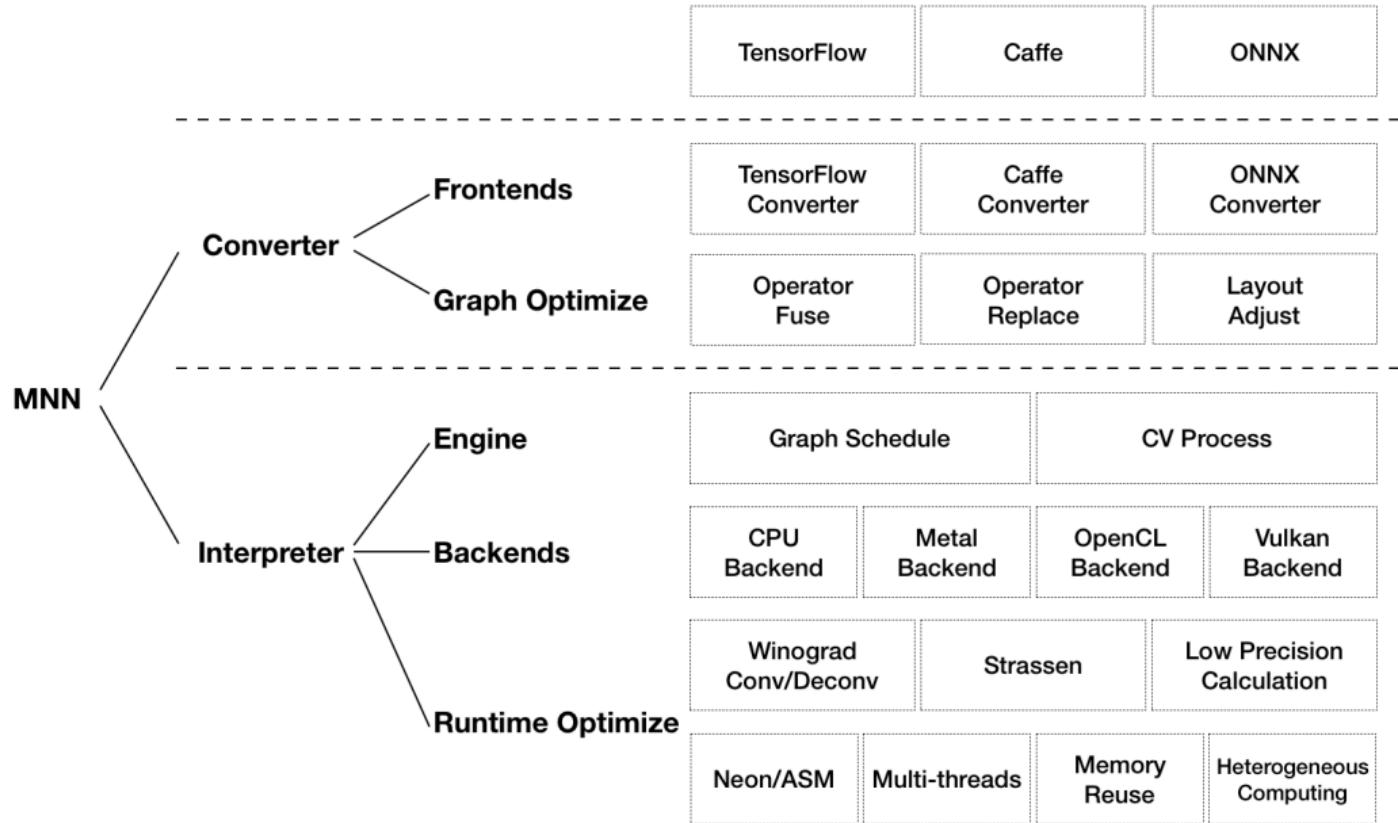
---
[4]

[4]Andrew Adams et al. (2019). "Learning to optimize halide with tree search and random programs". In: *ACM Trans. Graph.* 38.4, 121:1–121:12. DOI: 10.1145/3306346.3322967. URL: https://doi.org/10.1145/3306346.3322967.

MNN
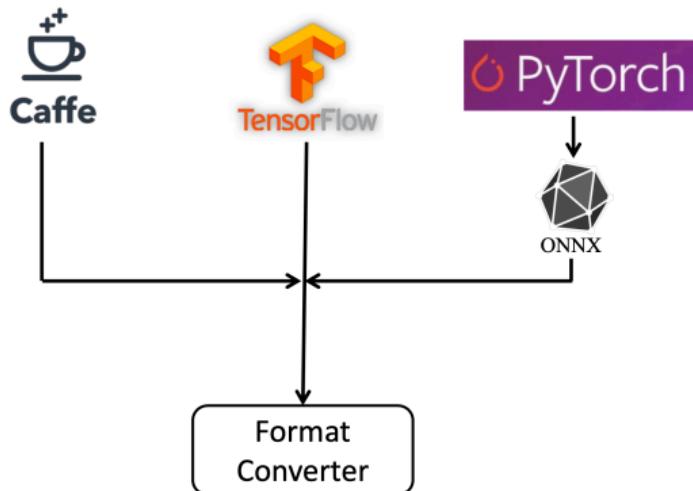
[5]Xiaotang Jiang et al. (2020). "MNN: A Universal and Efficient Inference Engine". In: *Proc. MLSys*.

- Caffe Deep Learning Framework
- TensorFlow Deep Learning Framework
- Pytorch Deep Learning Framework

- PyTorch is a python package that provides two high-level features:
  - Tensor computation (like numpy) with strong GPU acceleration
  - Deep Neural Networks built on a tape-based autograd system

- Model Deployment:
  - For high-performance inference deployment for trained models, export to ONNX format and optimize and deploy with NVIDIA TensorRT or MNN inference accelerator

```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

- TensorFlow is an open source software library for numerical computation using data flow graphs

- Model Deployment

  - For high-performance inference deployment for trained models, using TensorFlow-MNN integration to optimize models within TensorFlow and deploy with MNN inference accelerator

```python
import tensorflow as tf
from tensorflow.keras import Model, layers
import numpy as np

# Create TF Model.
class NeuralNet(Model):
    # Set layers.
    def __init__(self):
        super(NeuralNet, self).__init__()
        # First fully-connected hidden layer.
        self.fc1 = layers.Dense(n_hidden_1, activation=tf.nn.relu)
        # First fully-connected hidden layer.
        self.fc2 = layers.Dense(n_hidden_2, activation=tf.nn.relu)
        # Second fully-connecter hidden layer.
        self.out = layers.Dense(num_classes)

    # Set forward pass.
    def call(self, x, is_training=False):
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.out(x)
        if not is_training:
            # tf cross entropy expect logits without softmax, so only
            # apply softmax when not training.
            x = tf.nn.softmax(x)
        return x
```
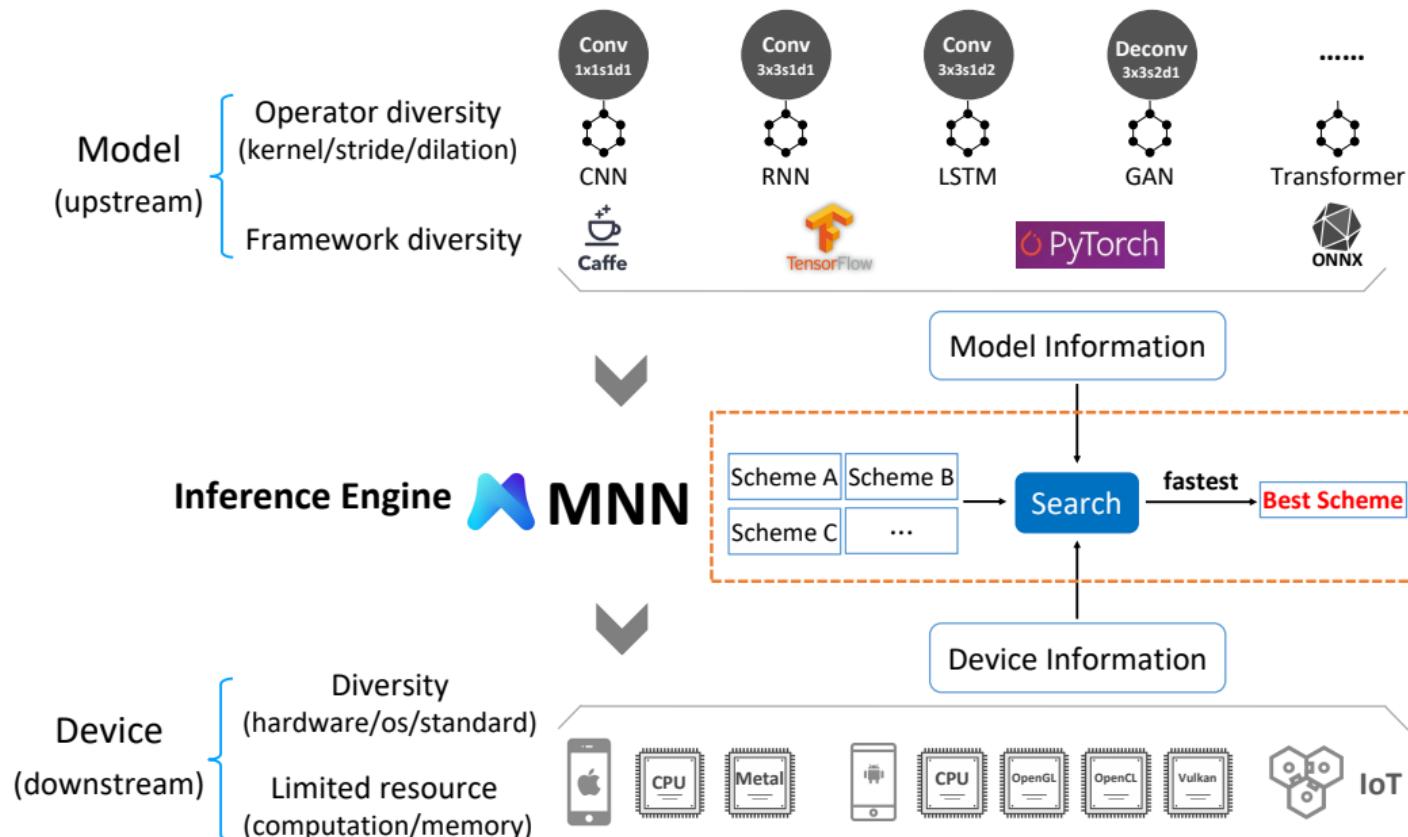
Caffe

- Caffe is a deep learning framework made with expression, speed, and modularity in mind:
  - Expressive architecture encourages application and innovation
  - Extensible code fosters active development.
  - Speed makes Caffe perfect for research experiments and industry deployment

- Model Deployment:
  - For high-performance inference deployment for trained models, using Caffe-MNN integration to optimize models within Caffe and MNN inference accelerator
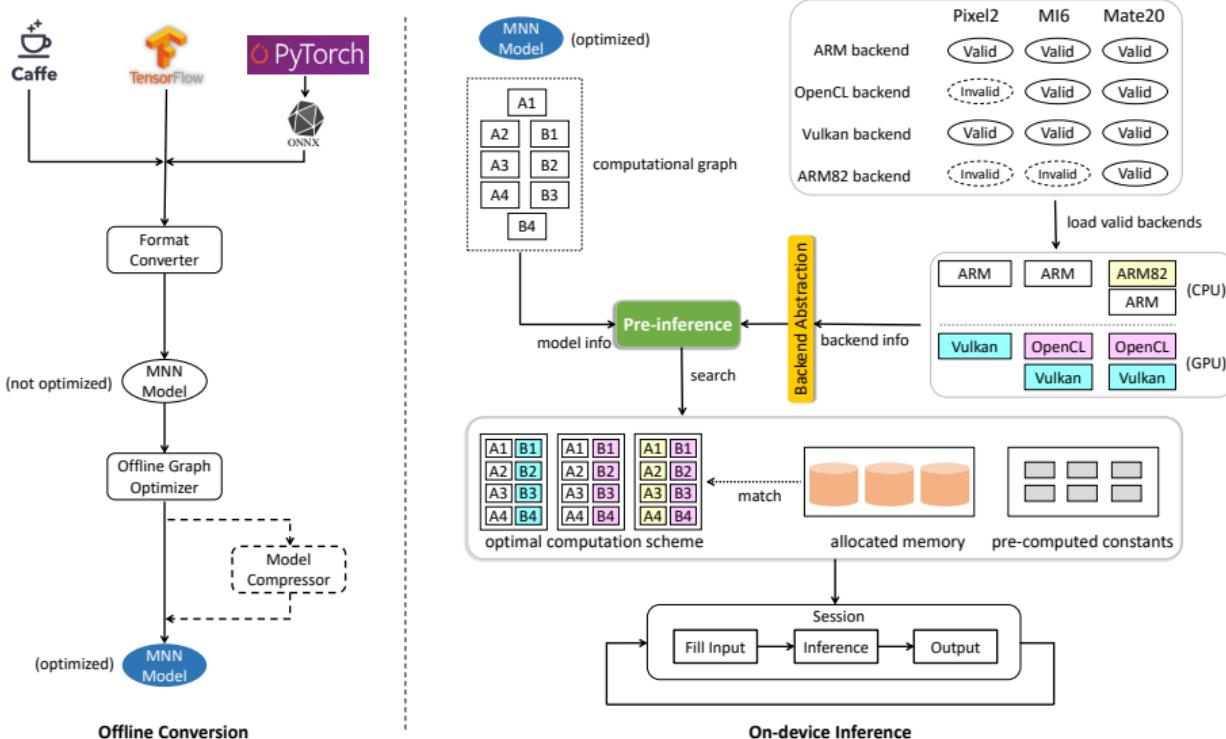
```python
caffe_root = '../'
import sys
sys.path.insert(0, caffe_root + 'python')
import caffe
# run scripts from caffe root
import os
os.chdir(caffe_root)
# Download data
!data/mnist/get_mnist.sh
# Prepare data
!examples/mnist/create_mnist.sh
# back to examples
os.chdir('examples')

from caffe import layers as L, params as P

def lenet(lmdb, batch_size):
    # our version of LeNet: a series of linear and simple nonlinear transformations
    n = caffe.NetSpec()

    n.data, n.label = L.Data(batch_size=batch_size, backend=P.Data.LMDB, source=lmdb,
                             transform_param=dict(scale=1./255), ntop=2)

    n.conv1 = L.Convolution(n.data, kernel_size=5, num_output=20, weight_filler=dict(type='xavier'))
    n.pool1 = L.Pooling(n.conv1, kernel_size=2, stride=2, pool=P.Pooling.MAX)
    n.conv2 = L.Convolution(n.pool1, kernel_size=5, num_output=50, weight_filler=dict(type='xavier'))
    n.pool2 = L.Pooling(n.conv2, kernel_size=2, stride=2, pool=P.Pooling.MAX)
    n.fc1 =   L.InnerProduct(n.pool2, num_output=500, weight_filler=dict(type='xavier'))
    n.relu1 = L.ReLU(n.fc1, in_place=True)
    n.score = L.InnerProduct(n.relu1, num_output=10, weight_filler=dict(type='xavier'))
    n.loss =  L.SoftmaxWithLoss(n.score, n.label)

    return n.to_proto()

with open('mnist/lenet_auto_train.prototxt', 'w') as f:
    f.write(str(lenet('mnist/mnist_train_lmdb', 64)))

with open('mnist/lenet_auto_test.prototxt', 'w') as f:
    f.write(str(lenet('mnist/mnist_test_lmdb', 100)))
```

Offline Conversion

On-device Inference

- Training in fp32 and inference in fp16 is expected to get same accuracy as in fp32 most of the time

- Add batch normalization to activation

- If it is integer RGB input (0 - 255), normalize it to be float (0 - 1)

- Advantages of FP16:
  - FP16 improves speed (TFLOPS) and performance
  - FP16 reduces memory usage of a neural network
  - FP16 data transfers are faster than FP32

- Disadvantages of FP16:
 - They must be converted to or from 32-bit floats before they are operated on
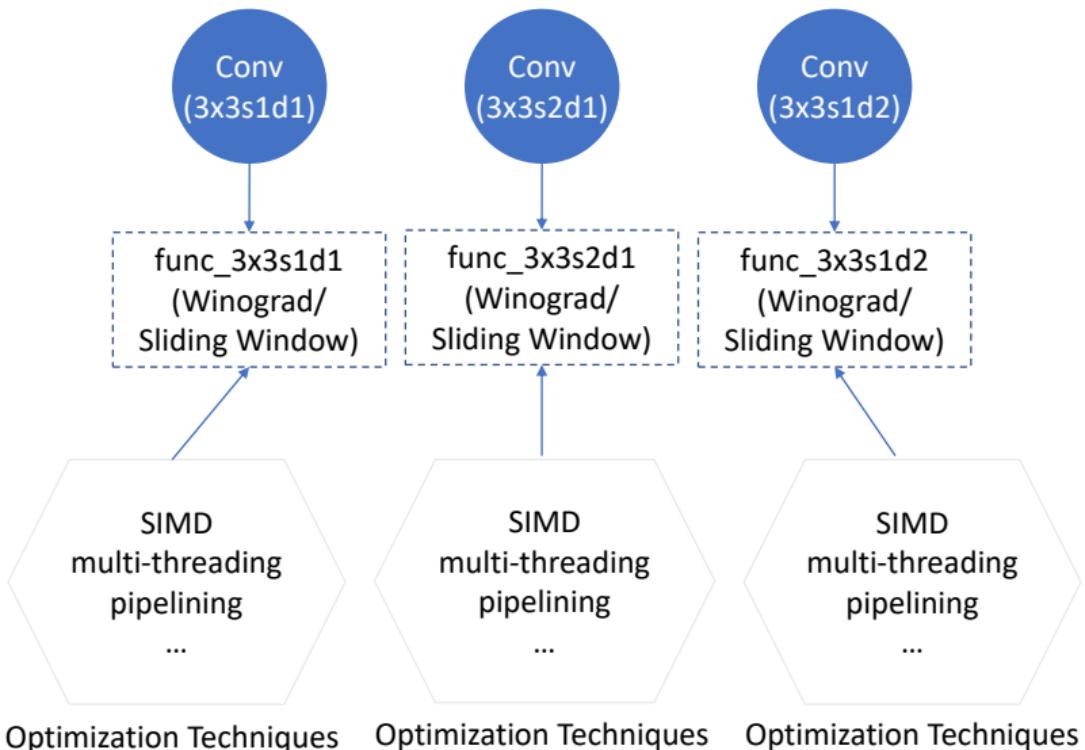
- As a programmer, there are several ways you can use Neon technology:
  - Neon intrinsics
  - Neon-enabled libraries
  - Auto-vectorization by your compiler
 - Hand-coded Neon assembler
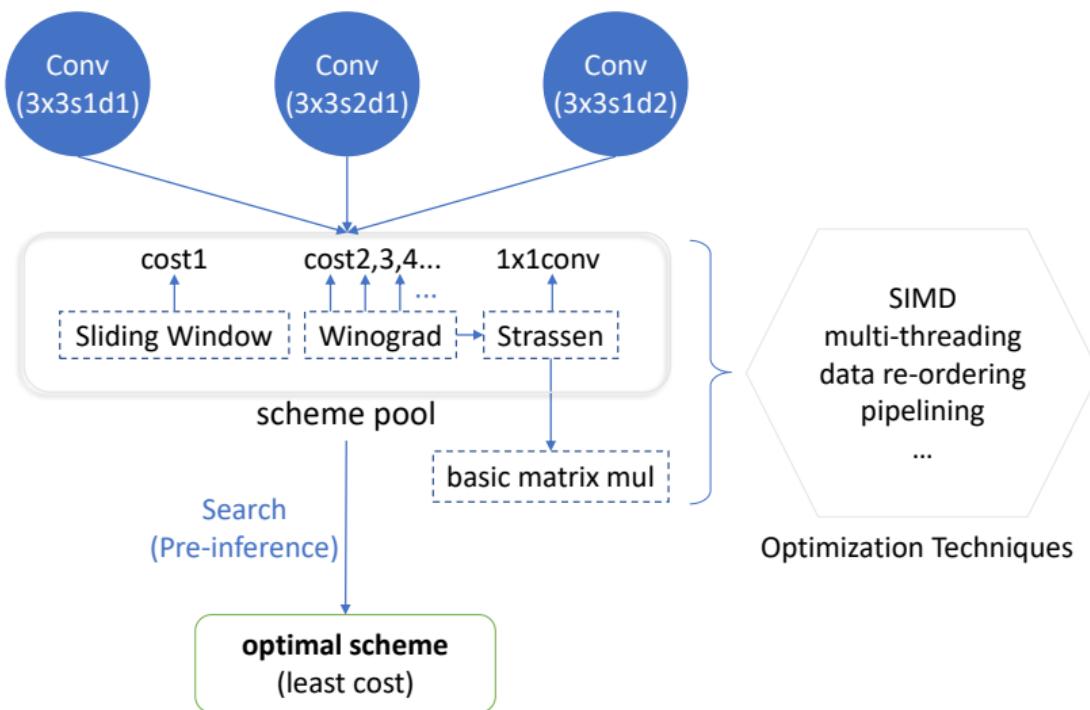
- Support for both integer and floating point operations ensures the adaptability of a broad range of applications, from codecs to High Performance Computing to 3D graphics.

- Tight coupling to the Arm processor provides a single instruction stream and a unified view of memory, presenting a single development platform target with a simpler tool flow
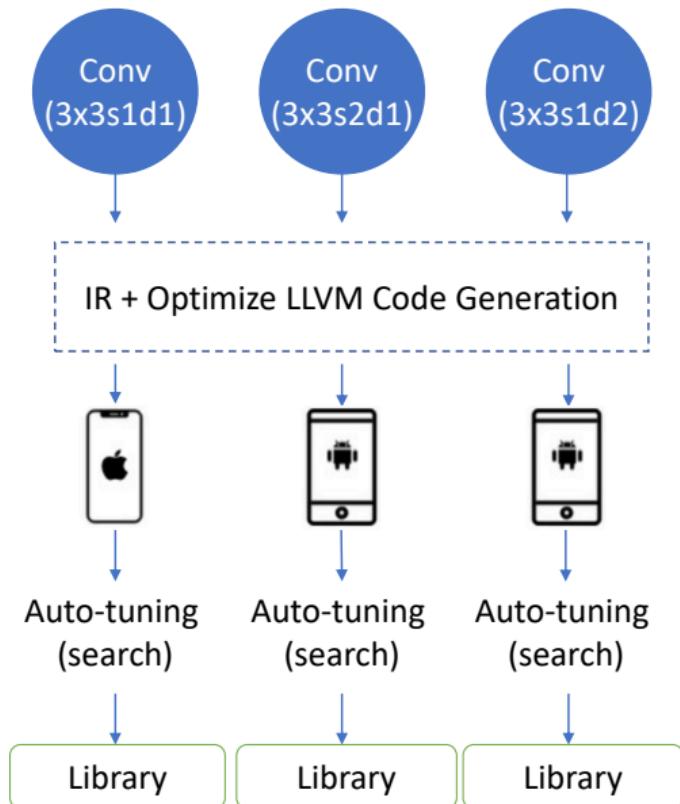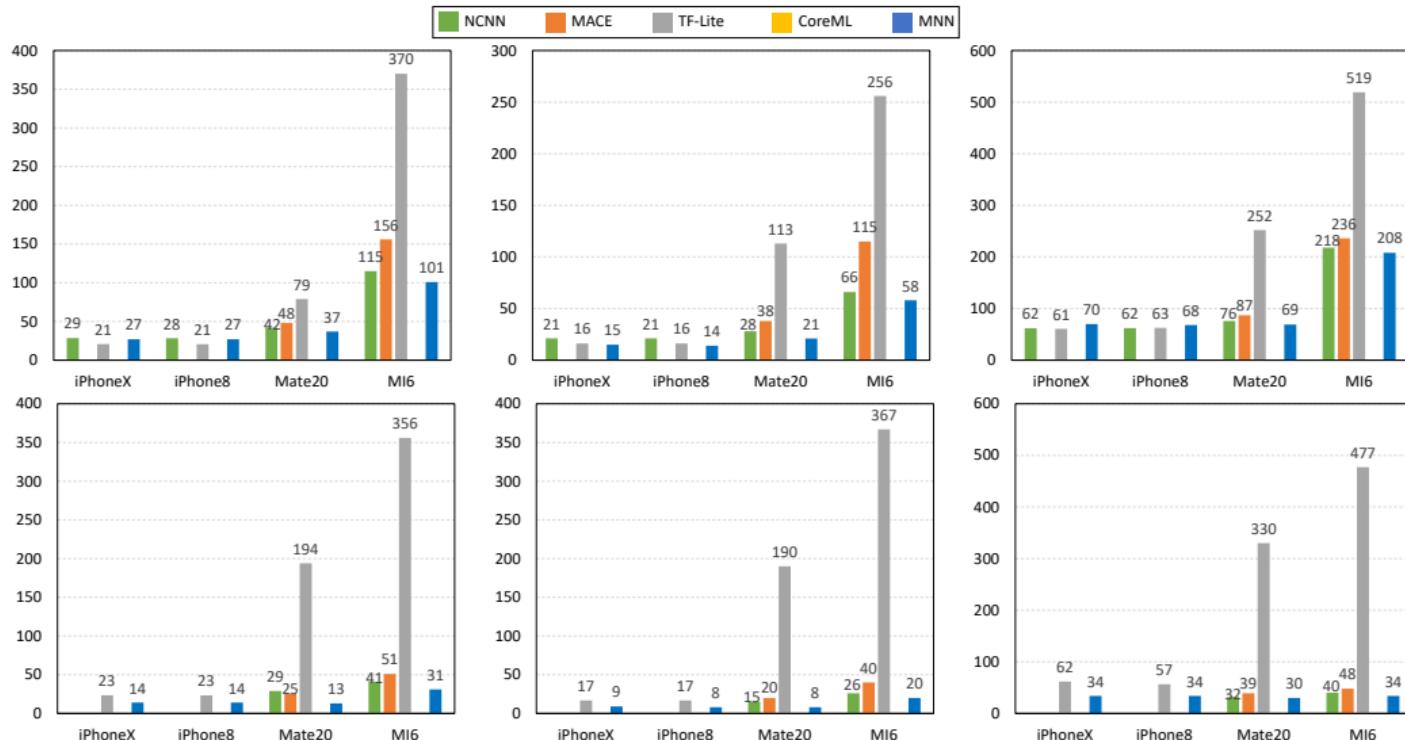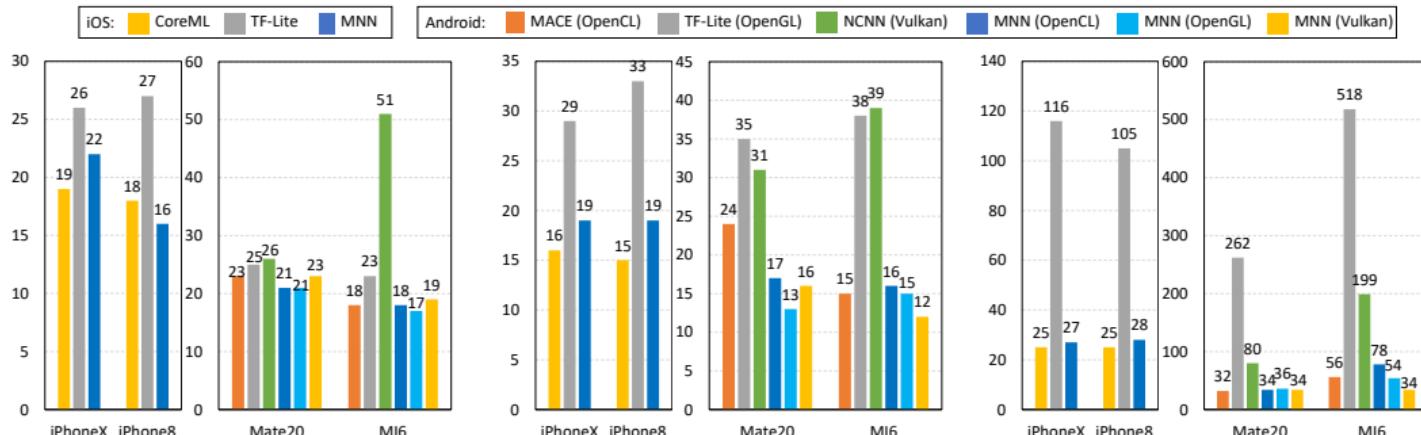
- Generally, MNN outperforms other inference engines under almost all settings by about $20\% - 40\%$, regardless of the smartphones, backends, and networks

- For CPU, on average, 4-thread inference with MNN is about 30% faster than others on iOS platforms, and about 34% faster on Android platforms

- For Metal GPU backend on iPhones, MNN is much faster than TF-Lite, a little slower than CoreML but still comparable

- **alibaba2020mnn**
- Christian Szegedy et al. (2015). "Going deeper with convolutions". In: *CVPR*