Learning on All Hardware Platforms

Oct 3, 2018 • Lianmin Zheng, Eddie Yan, Tianqi Chen

Optimizing the performance of deep neural network on a diverse range of hardware platforms is still a hard problem for AI developers. In terms of system support, we are facing a many-to-many problem here: deploying trained models from multiple frontends (e.g. Tensorflow, ONNX, MXNet) to multiple hardware platforms (e.g. CPU, GPU, Accelerators). The most performance critical part of this problem is obtaining high performance kernel implementations for growing model architectures and hardware platforms.

To address this challenge, TVM takes a full stack compiler approach. TVM combines code generation and automatic program optimization to generate kernels that are comparable to heavily hand-optimized libraries, obtaining state-of-the-art inference performance on hardware platforms including ARM CPUs, Intel CPUs, Mali GPUs, NVIIDA GPUs and AMD GPUs.

In this blog post, we show the workflow of automatic kernel optimization in TVM compiler stack and benchmark results on several hardware platforms.

System Overview

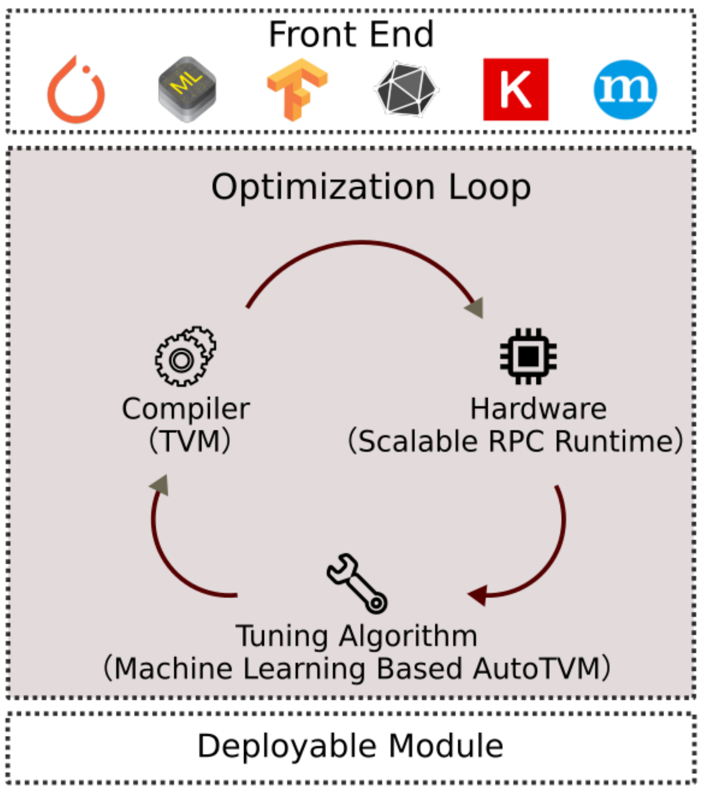


Figure 1. System Overview

Kernel optimization in TVM is done in an iterative loop fashion. As shown in Figure 1, the automatic kernel optimization takes a neural network (typically in computational graph representation) from frontend frameworks as input, and generates kernels for all operators in this network.

The inner loop uses a scalable RPC runtime, machine learning based tuners and a tensor compiler. In each round of the loop, the tuner picks a batch of promising candidate kernel implementations from a large search space, and profile them on real hardware. Then the tuner gets the profiling results. These profiling results are used as training data to fit a prediction model. After fitting the prediction model, the tuner picks the next promising candidates according to the predictions, and the loop continues. This way, we search for fast kernels iteratively.

The below figure compares traditional auto-tuning and AutoTVM. The major difference is that AutoTVM is

Scalable to heterogenous cluster of devices
Learning to optimize tensor programs with a transferable machine learning cost model
You can refer to our paper[1] for more details.

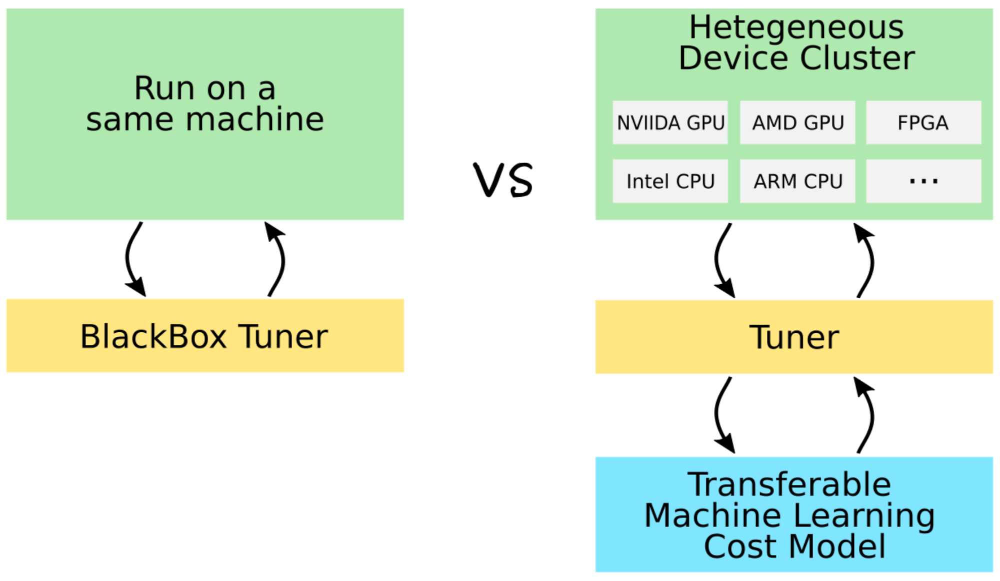


Figure 2. Comparison of Traditional Auto-tuning and AutoTVM

Begin Tuning

For demonstration, we run our optimization for resnet-18 on RK3399, an ARM development board. The detailed instructions are omitted due to the space limit of a blog post. Links to tutorials for ARM CPU, Mali GPU, NVIDIA GPU, AMD GPU are all available at the end of this blog.

First we get a pre-trained model from MXNet model zoo, and extract tuning tasks from it.

```
from mxnet.gluon.model_zoo.vision import get_model

block = get_model('resnet18_v1', pretrained=True)
net, params = nnvm.frontend.from_mxnet(block)

tasks = autotvm.extract_from_graph(net)
tune_tasks(tasks, **tuning_option)
```

There are 12 different conv2d layers in resnet-18, so we launch 12 tuning tasks. For each of them, the tuner makes several hundreds of trials and picks the best one. After finishing all tuning tasks, we compile the whole network and generate a single deployable minimal library. One sample output is

```
Extract tasks...
Tuning...
[Task  1/12]  Current/Best:   22.37/  52.19 GFLOPS | Progress: (544/1000) | 406.59 s Done.
[Task  2/12]  Current/Best:    6.51/  18.77 GFLOPS | Progress: (608/1000) | 325.05 s Done.
[Task  3/12]  Current/Best:    4.67/  24.87 GFLOPS | Progress: (480/1000) | 372.31 s Done.
[Task  4/12]  Current/Best:   11.35/  46.83 GFLOPS | Progress: (736/1000) | 602.39 s Done.
[Task  5/12]  Current/Best:    1.01/  19.80 GFLOPS | Progress: (448/1000) | 262.16 s Done.
[Task  6/12]  Current/Best:    2.47/  23.76 GFLOPS | Progress: (672/1000) | 563.85 s Done.
[Task  7/12]  Current/Best:   14.57/  33.97 GFLOPS | Progress: (544/1000) | 465.15 s Done.
[Task  8/12]  Current/Best:    1.13/  17.65 GFLOPS | Progress: (576/1000) | 365.08 s Done.
[Task  9/12]  Current/Best:   14.45/  22.66 GFLOPS | Progress: (928/1000) | 724.25 s Done.
[Task 10/12]  Current/Best:    3.22/  15.36 GFLOPS | Progress: (864/1000) | 564.27 s Done.
[Task 11/12]  Current/Best:   11.03/  32.23 GFLOPS | Progress: (736/1000) | 635.15 s Done.
[Task 12/12]  Current/Best:    8.00/  21.65 GFLOPS | Progress: (1000/1000) | 1111.81 s Done.
Compile...
Upload...
Evaluate inference time cost...
Mean inference time (std dev): 162.59 ms (0.06 ms)
```

The tuning is especially helpful and worth a try if your model has some strange shapes or your hardware is customized, as hand-optimized static libraries cannot consider all situations.

Benchmark Results

We pre-tuned some popular networks on our device cluster and released the following benchmark. Instructions for reproduction are at the end of this blog.

numbers in a table, and then provide an incomplete comparison with some other libraries.

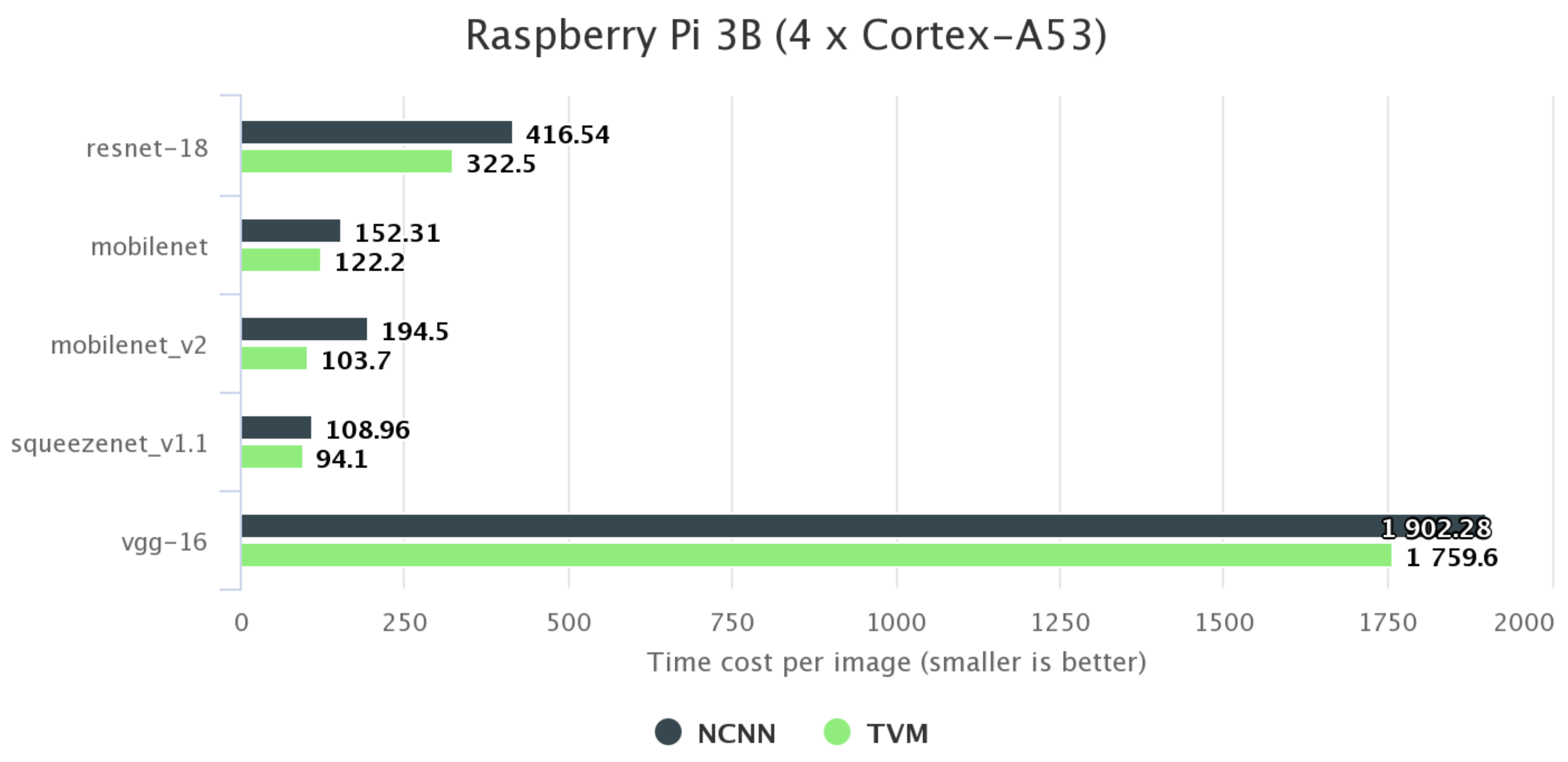
Comparison

We validate the effectiveness of our automatic optimization stack by comparing with heavily optimized traditional libraries on each platform.

We tested popular image classification networks on ImageNet (3x224x224) dataset with batch size = 1 and data type = float32. The reported numbers are time costs per image in milliseconds.

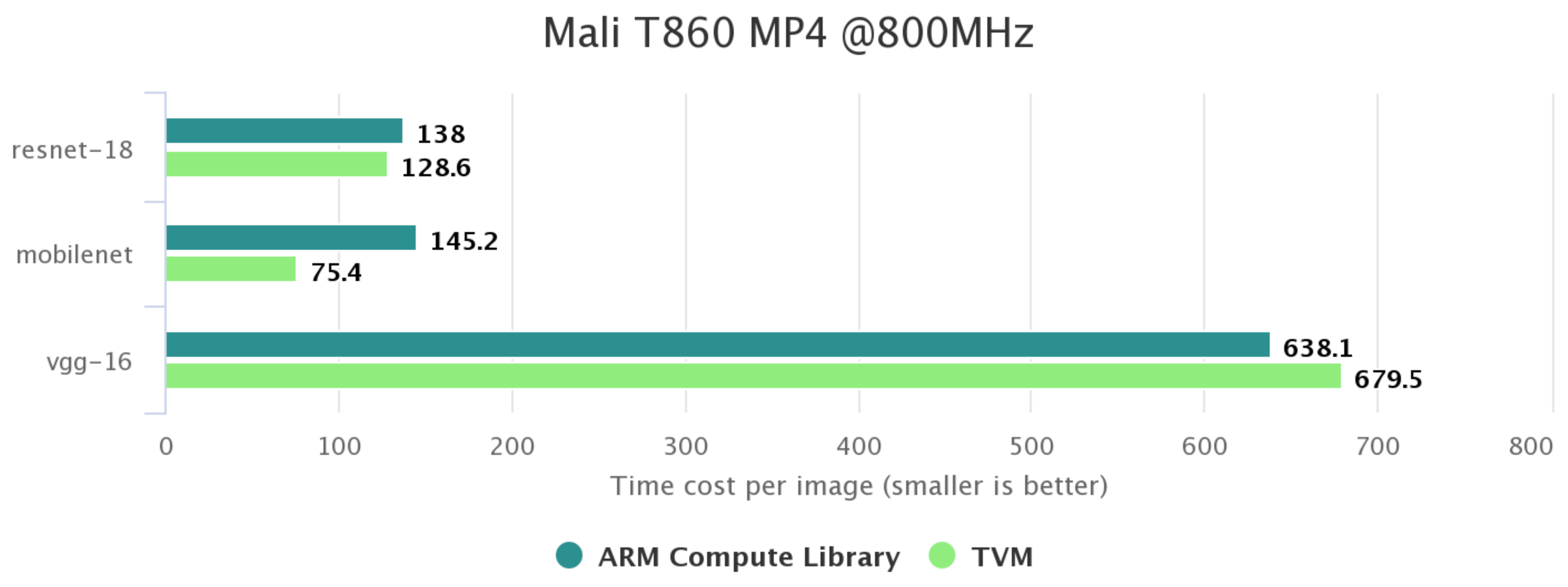
ARM CPU

We choose NCNN, a widely used, hand-optimized kernel library as baseline. It makes extensive use of NEON assembly instructions. For example, the code base contains 13k lines of code for only 3x3 convolution layers. We reference the benchmark numbers in their project repository. As shown in the figure below, TVM outperforms it for all networks on Rasbperry Pi 3B.



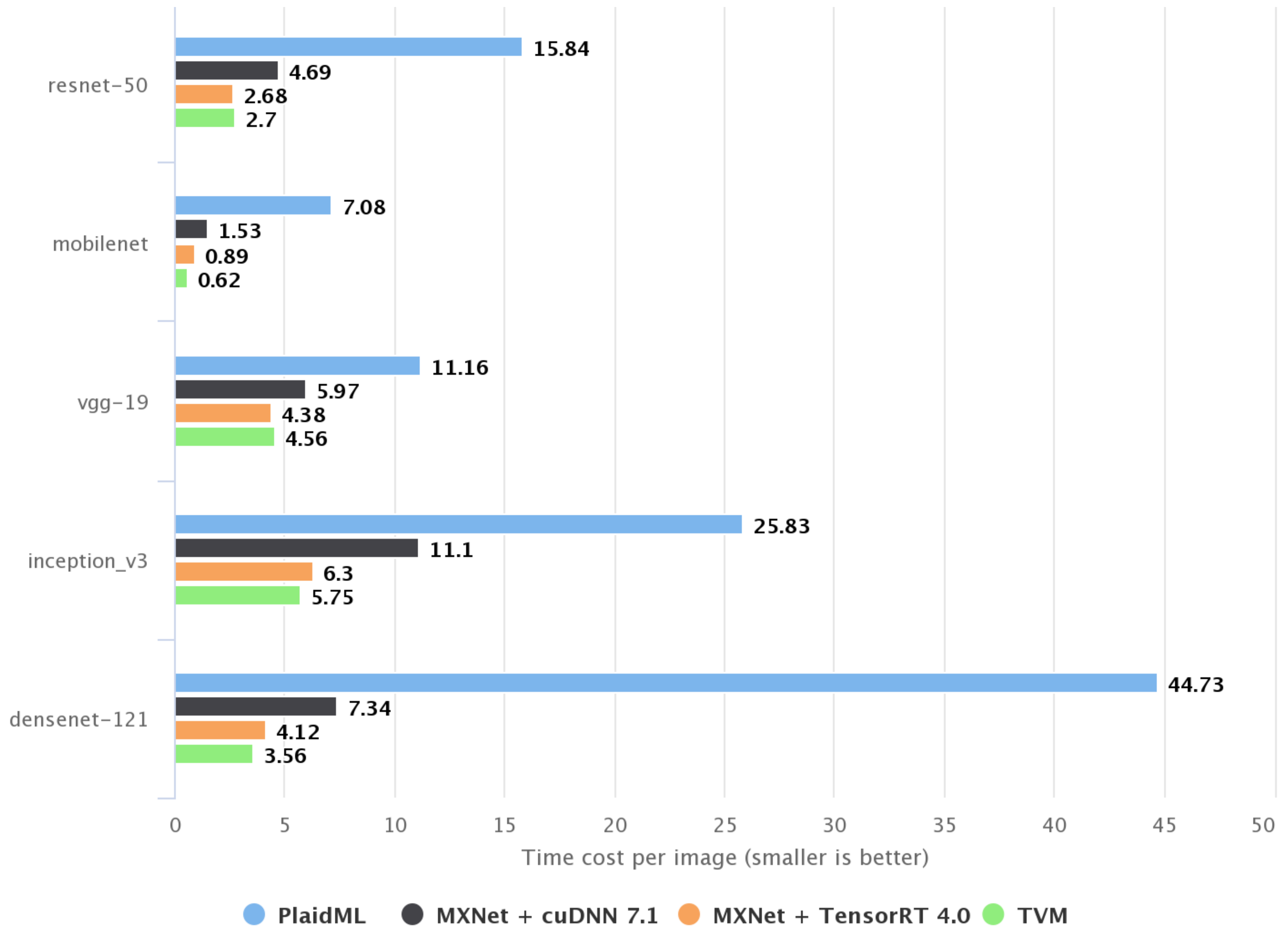
Mali GPU

ARM Compute Library is a vendor provided library that supports Mali GPU (OpenCL) well. According to the results, TVM provides stronger performance in ResNet and MobileNet due to advantages in convolutional layers. TVM lags behind a bit on vgg-16 because vgg-16 is an old and huge network and has several large dense layers.



NVIDIA GPU

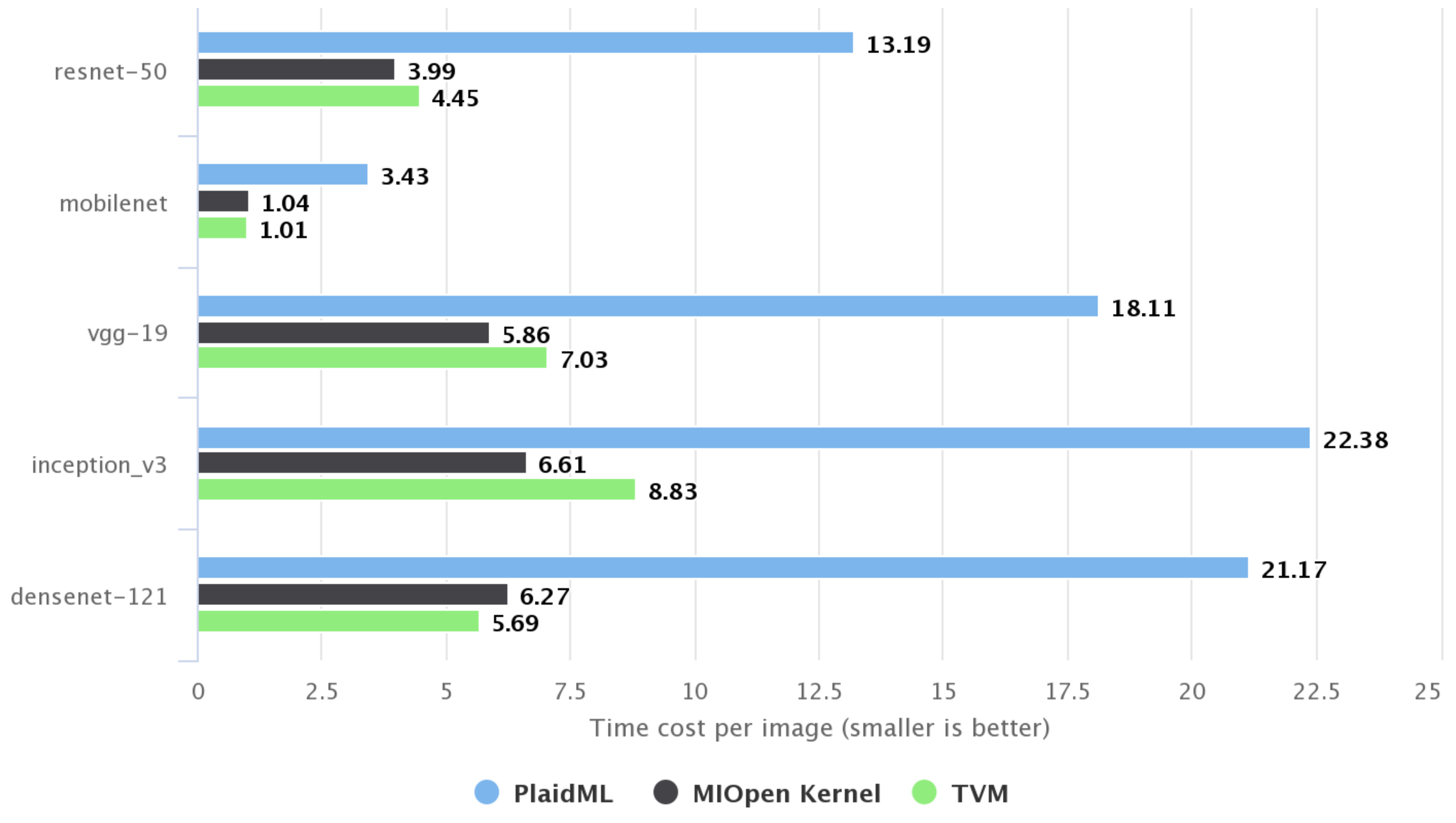
On NVIDIA GPU, CuDNN and TensorRT are two vendor-provided libraries for training and inference respectively. Since we focus on inference, we run our benchmark in the unbatched setting. Another tensor compiler PlaidML is also reported as baseline as there is a previous benchmark of it compared against a pre-AutoTVM version of TVM. We reference its benchmark results from PlaidBench. According to the results below, TVM achieves parity with TensorRT performance.



AMD GPU

We also take a quick look at a AMD GPU. TVM supports OpenCL and ROCm backend. We found ROCm is better since it is more specialized for AMD GPUs. MIOpen is a vendor provided kernel library. TVM's graph runtime can call MIOpen's kernel implementations directly, so we report the baseline performance by using this integration.

We didn't do any specific optimization for AMD GPU. All computation definition and schedule code for NVIDIA GPU is directly reused. As a result, TVM is a little bit slower then MIOpen in most cases. We believe there is still room for improvement.



All Our Results

We tested the following networks on ImageNet (3x224x224) dataset with batch size = 1 and data type = float32. The reported numbers are time costs per image in milliseconds.

	densenet121	inception v3	mobilenet	mobilenet v2	resnet18	resnet50	squeezenet v1.0	squeezenet v1.1	vgg16	vgg19
ARM CPU										
Huawei P20 Pro	181.4	439.9	41.1	34.5	76.5	208.2	51.8	25.7	480.6	627.0
Google Pixel2	162.2	433.5	39.5	30.1	61.1	181.3	47.3	23.2	391.1	487.7
Firefly RK3399	335.9	1285.9	78.6	66.7	161.2	403.8	94.6	48.5	902.9	1090.1
Raspberry Pi 3B	609.5	2070.4	122.2	103.7	322.5	725.8	185.1	94.1	1759.6	2118.6
Xilinx PYNQ	2888.3	9709.1	723.5	514.3	1234.6	3580.5	909.9	477.3	-(Note 1)	-
Mali GPU										
Mali-T860	410.9	783.1	75.4	70.8	128.6	352.9	106.2	58.0	679.5	805.3
NVIDIA GPU										
GTX 1080 Ti	3.6	5.8	0.6	- (Note 2)	-	2.7	-	-	4.0	4.6
GTX TITAN X	5.8	9.7	1.0	-	-	4.3	-	-	6.4	7.5
AMD GPU										
AMD Vega FE	5.7	8.8	1.0	-	-	4.5	-	-	5.9	7.0

Note 1: Out of memory on this board.
Note 2: We didn't tune some small networks on GPU due to time constraints. When profiling data is not available, TVM can use fallback code generation. But competitive performance is not guaranteed in this scenario.

With an expressive code generator and an efficient search algorithm, we are able to generate kernels that are comparable to heavily hand-optimized ones. Since programmer time is expensive and machine time is getting cheaper, we believe automatic optimization with real hardware and data in the loop will be the standard workflow for inference deployment. TVM just provides such a solution.

Links

[1] benchmark: https://github.com/dmlc/tvm/tree/master/apps/benchmark
[2] Tutorial on tuning for ARM CPU: https://tvm.apache.org/docs//tutorials/autotvm/tune_nnvm_arm.html
[3] Tutorial on tuning for Mobile GPU: https://tvm.apache.org/docs//tutorials/autotvm/tune_nnvm_mobile_gpu.html
[4] Tutorial on tuning for NVIDIA/AMD GPU: https://tvm.apache.org/docs//tutorials/autotvm/tune_nnvm_cuda.html
[5] Paper about AutoTVM: Learning to Optimize Tensor Program
[6] Paper about Intel CPU (by AWS contributors) : Optimizing CNN Model Inference on CPUs