Optimizing Deep Learning Workloads on ARM GPU with TVM

Lianmin Zheng
Shanghai Jiao Tong University
mercy_zheng@sjtu.edu.cn

Tianqi Chen
University of Washington
tqchen@cs.washington.edu

Abstract

With the great success of deep learning, the demand for deploying deep neural networks to mobile devices is growing rapidly. However, current popular deep learning frameworks are often poorly optimized for mobile devices, especially mobile GPU. In this paper, we follow the pipeline proposed by TVM/NNVM, and optimize both kernel implementations and dataflow graph for ARM Mali GPU. Compared with vendor-provided ARM Compute Library, our kernel implementations and end-to-end pipeline are 1.7x faster on VGG16 and 2.2x faster on mobilenet.

Keywords ARM GPU, GPU Kernel, Deep Learning, TVM

ACM Reference Format:
Lianmin Zheng and Tianqi Chen. 2018. Optimizing Deep Learning Workloads on ARM GPU with TVM. In *Proceedings of 1st ACM Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning (ReQuEST at ASPLOS'18)*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3229762.3229764

1 Technical Description

1.1 TVM IR Stack

TVM [4] is a framework for deploying deep learning frameworks on different hardware platforms. It includes a tensor IR stack TVM, a graph IR stack NNVM and a compiler. These components can offer optimization from different levels. We describe compute rules in TVM's domain specific language and use its schedules to optimize kernels. Then the compiler can generate the target code for us, which is OpenCL code on Mali GPU.

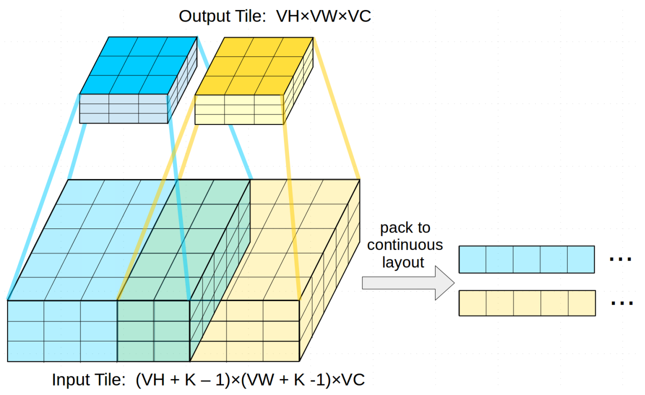
Output Tile: VH×VW×VC
pack to continuous layout
...
...
Input Tile: (VH + K – 1)×(VW + K -1)×VC

Figure 1. The Input/Output Tile of Spatial Packing

1.2 Optimizing Convolution Layer

1.2.1 Im2col with GEMM

A common implementation of convolution layer is transforming it to a GEMM by im2col, which is also adopted by ARM Compute Library[3]. The advantage of this method is the easy utilization of highly optimized BLAS library, while the disadvantage is the awful memory redundancy. For a 3x3 kernel, It needs 9x memory for packing the input data.

1.2.2 Spatial Packing

To reduce memory redundancy, we use another transformation called spatial packing. If the batch size for inference is 1, let the output shape of convolution be (CO, H, W). We do one-level tile on these three dimensions. Then the output shape of convolution can be $(CO/VC, H/VH, W/VW, VH, VW, VC)$. For an output tile $(VH \times VW \times VC)$, its input elements are not continuous in memory. So we pack the input elements for an output tile into a continuous input tile, as shown in Figure 1. We can see that im2col is a special case of spatial packing with $VH = 1$ and $VW = 1$.

The memory reduance of this method can be computed by

$$\textbf{memory redundancy} = \frac{(VH + K - 1) \times (VW + K - 1)}{VH \times VW}$$

Any configuration with $VH > 1$ or $VW > 1$ has smaller memory redundancy than im2col.

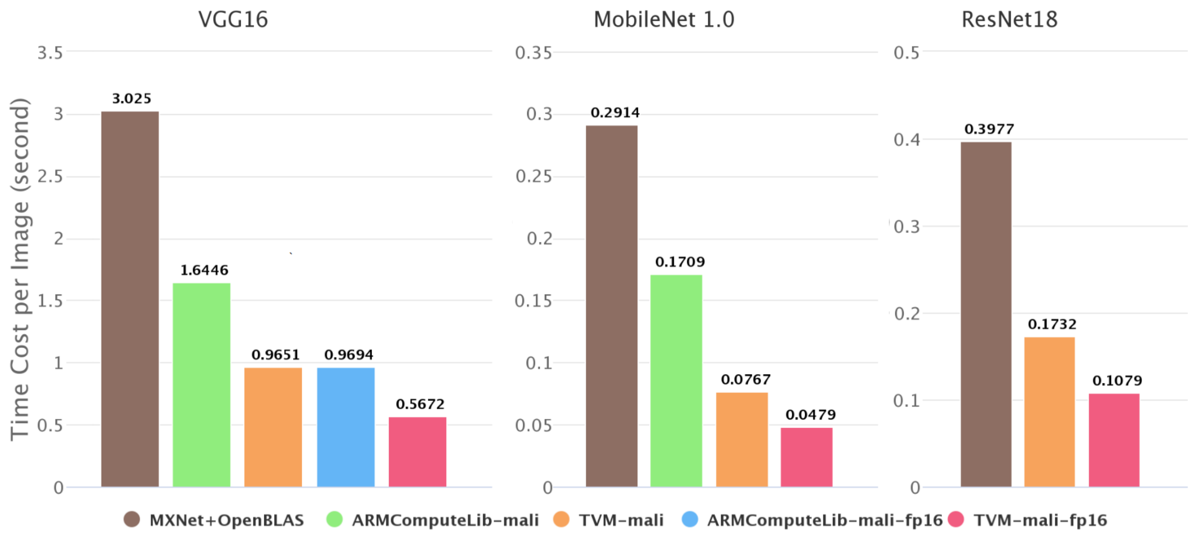

Figure 2. Inference Speed of Different Backends on ImageNet

1.2.3 Winograd Fast Convolution for 3x3 Kernel

Winograd's minimal filtering algorithm has been proven very suitable for convolution layers with small kernel[6, 8]. This algorithm can archive the minimal number of float point multiplication for convolution. For $r \times r$ input data tile d, and $m \times m$ kernel g, the convolution is computed by

$$Y = A^T[[GgG^T] \odot [B^T dB]]A$$

Where G is the transform matrix for data, B is the transform matrix for kernel and A is the inverse transform matrix. We implemented $F(2 \times 2, 3 \times 3)$ for convolution layers with 3x3 kernels. See [6] for more details of this algorithm.

1.2.4 Other Implementation Details

Because Mali Midgrad GPUs are based on SIMD architecture and need explicit vectorization [2]. We further try to vectorize and unroll some inner loops, which brings significant speedup. The shape of workgroup also matters a lot. It controls the memory reuse among threads. Since there are too many trade-offs in choosing the tile factor and workgroup shape, we use a tuner to search the optimal values for every layer.

1.3 Optimizing Depthwise Convolution Layer

Depthwise convolution layer is a variant of convolution layer. It can greatly reduce the computation complexity of convolution layer, which makes it very suitable for mobile applications [5]. We adopt the similar tiling strategies and get reasonable speedup.

1.4 Optimizing Dataflow Graph

This optimization is done by NNVM. The optimization passes include layout transformation, inference simplification, pre-compute and operator fusion.

2 Empirical Evaluation

Inference speed is the focus in this paper. So we choose time cost per image (batch size = 1) as the metric to evaluate in this section. We test both single layer performance under various optimization techniques, and end-to-end performance on VGG16, mobilenet and ResNet-18.

2.1 Test Platform

We use a commercial ARM board with Rockchip-RK3399 SoC as test environment. The detailed information of our test environment is listed below.

- Board: Firefly-RK3399 4G
- CPU: dual-core Cortex-A72 + quad-core Cortex-A53
- GPU: Mali-T860MP4@800Mhz
- Arm Compute Library : v17.12
- MXNet: v1.0.1
- Openblas: v0.2.18

To make the environment more stable, we close the GUI on board, and set the 'performance' power governor for GPU.

2.2 Single Layer Comparison

We take a layer in VGG16 as test case, and investigate the speed up bought by different optimization techniques. The input shape of this layer is $(C, H, W) = (56, 56, 256)$, and filter shape is $(OC, IC, K, K) = (256, 256, 3, 3)$. As shown in Table. 1, Winograd algorithm is the most efficient one.

For depthwise convolution, we measure the time cost of our kernel and ARM CL's kernel for all depthwise conv layers in mobilenet. As listed in Table 2, ours is faster for all layers.

2.3 End-to-End Benchmarking

As shown in Figure 2, we benchmark the end-to-end performance of different backends. Both float32 and float16 precision are measured. Some cases are missing in the plot, because the graph runtime in ARM Compute Library currently does not support skip connection. We try both GEMM and direct method of convolution layer in Arm Compute Library, and GEMM method is always faster than direct method in these test cases, so we only plot the result of GEMM method. The results indicate utilizing GPU in this commercial board can be than 2x ˜ 4x faster than 6-core big.Little CPU. Our end-to-end pipeline is 1.7x ˜ 2.2x faster then ARM Compute Library.

3 Conclusion

In this paper, we implemented several optimization techniques with TVM/NNVM stack to optimize the inference speed on ImageNet for ARM Mali GPU. Empirical evaluation results show that our method is faster than vendor-provided ARM Compute Library.

References

[1] 2018. ReQuEST at ASPLOS'18: 1st Reproducible Tournament on Pareto-efficient Image Classification. http://cknowledge.org/request-cfp-asplos2018.html.

[2] ARMDeveloper. 2016. ARM Mali GPU OpenCL Developer Guide. https://developer.arm.com/docs/100614/0302

[3] ARMDeveloper. 2017. ARM Compute Library. https://developer.arm.com/

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, and Haichen Shen. 2017. TVM: An End to End IR Stack for Deploying Deep Learning Workloads on Hardware Platforms. http://tvmlang.org/

[5] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861

[6] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 4013–4021.

[7] Thierry Moreau, Anton Lokhmotov, and Grigori Fursin. 2018. Introducing ReQuEST: an Open Platform for Reproducible and Quality-Efficient Systems-ML Tournaments. *ArXiv e-prints*. arXiv:stat.ML/1801.06378

[8] Shmuel Winograd. 1980. Arithmetic complexity of computations.

Kernel	Cost (second)	GFLOPS	speedup
Im2col in ARM CL	0.1821	20.3111	1.00x
Spatial Pack: simple bind	5.6154	0.6588	0.03x
Spatial Pack: + unrolling	0.3707	9.9796	0.49x
Spatial Pack: + vectorization	0.1304	28.3679	1.40x
Winograd:	0.0819	45.1546	2.22x

Table 1. Performance of a convolution layer in VGG16 (ARM CL stands for ARM ComputeLibrary, GFLOPS for Winograd is the effective GFLOPS)

(H, W, C, K)	ARM CL (ms)	Ours(ms)	speedup
(112, 112, 32, 3)	11.690	1.978	5.91
(112, 112, 64, 3)	6.666	2.412	2.76
(56, 56, 128, 3)	11.714	2.291	5.11
(56, 56, 128, 3)	3.343	1.460	2.29
(28, 28, 256, 3)	5.875	1.250	4.70
(28, 28, 256, 3)	1.768	1.105	1.60
(14, 14, 512, 3)	3.199	1.164	2.75
(14, 14, 512, 3)	2.223	0.731	3.04
(7, 7, 1024, 3)	3.736	0.863	4.33

Table 2. Performance of Depthwise Convolution Layers

A Artifact Appendix

Submission and reviewing methodology:
http://cTuning.org/ae/submission-20171101.html

A.1 Abstract

This Artifact Appendix describes experimental workflow, artifacts and results from this paper evaluated during the 1st reproducible ReQuEST tournament at the ACM ASPLOS'18:

- **Original artifact:** https://github.com/merrymercy/tvm-mali
- **Latest CK workflow:** https://github.com/ctuning/ck-request-asplos18-mobilenets-tvm-arm
- **CK results:** https://github.com/ctuning/ck-request-asplos18-results-mobilenets-tvm-arm
- **Artifact DOI:** https://doi.org/10.1145/3229770
- **ReQuEST submission and reviewing guidelines:** http://cknowledge.org/request-cfp-asplos2018.html ([1])
- **ReQuEST goals:** [7]
- **ReQuEST workflows:** https://github.com/ctuning/ck-request-asplos18-results
- **ReQuEST scoreboard:** http://cKnowledge.org/request-results

This artifact provides the scripts for the end-to-end benchmarking in the paper. It benchmarks the inference speed of TVM/NNVM, Arm Compute Library and MXNet + OpenBLAS on several popular deep neural networks. The image for test is 224x224x3 ImageNet format.

To validate the result, you can follow our instructions to build these platforms and run test scripts.

A.2 Artifact check-list

Details: http://cTuning.org/ae/submission_extra.html

- **Algorithm:** image classification
- **Program:** TVM/NNVM, ARM Compute Library, MXNet, OpenBLAS
- **Compilation:** g++
- **Binary:** will be compiled on a target platform
- **Data set:** ImageNet 2012 validation (50,000 images)
- **Run-time environment:** Linux with OpenCL
- **Hardware:** Firefly-RK3399 with ARM Mali-T860MP4 or other boards with ARM Mali GPUs
- **Run-time state:** set by our scripts
- **Metrics:** inference speed; accuracy
- **Output:** classification result; execution time; accuracy
- **Experiments:** benchmarking the inference speed of different backends on ImageNet (automated via CK command line)
- **How much disk space required (approximately)?:** 4GB
- **How much time is needed to prepare workflow (approximately)?:** several hours (mainly native compilation of packages)
- **How much time is needed to complete experiments (approximately)?:** hours for full ImageNet accuracy validation (50000 images)
- **Publicly available?:** Yes
- **Code license?:** MIT license
- **Collective Knowledge workflow framework used?:** Yes
- **CK workflow URL:** https://github.com/ctuning/ck-request-asplos18-mobilenets-tvm-arm
- **CK results URL:** https://github.com/ctuning/ck-request-asplos18-results-mobilenets-tvm-arm
- **Original artifact URL:** https://github.com/merrymercy/tvm-mali

A.3 Description

A.3.1 How to obtain

The artifact is publicly available on GitHub:
https://github.com/merrymercy/tvm-mali

A.3.2 Hardware dependencies

Our test hardware is Firefly-RK3399. Other similar hardwares with ARM Mali T8xx GPU should give comparable results.

A.3.3 Software dependencies

Tested platforms TVM/NNVM, Arm Compute Library and MXNet + OpenBLAS should be built on device. OpenCL Driver is also required.

A.4 Installation

Clone our repository and run the install script.

```
$ git clone https://github.com/merrymercy/tvm-mali.git
$ cd tvm-mali
$ bash install.sh
```

A.5 Experiment workflow

Run the test script and observe the output log in screen. (It needs root permission to set the power governor for GPU)

```
$ sudo bash run_test.sh
```

A.6 Evaluation and expected result

Our evaluation metric is the inference speed on ImageNet 224x224x3 images. The main result of this artifact is to reproduce the performance comparison in Figure 2. If the reviewer uses the same hardware, we expect the inference time costs in the output log file match the results in our plot. If other similar hardware is used, we expect the reviewer can observe a similar trend.

A.7 Unified installation and evaluation for the ReQuEST scoreboard using Collective Knowledge framework

A.7.1 Installation

NB: The '#' sign means 'sudo'.

Install global prerequisites (Ubuntu)

```
# sudo apt-get install libtinfo-dev
# pip install numpy scipy decorator matplotlib
 or
# pip3 install numpy scipy decorator matplotlib
```

Minimal CK installation

The minimal installation requires:

- Python 2.7 or 3.3+ (limitation is mainly due to unitests)
- Git command line client.

You can install CK in your local user space as follows:

```
$ git clone http://github.com/ctuning/ck
$ export PATH=$PWD/ck/bin:$PATH
$ export PYTHONPATH=$PWD/ck:$PYTHONPATH
```

You can also install CK via PIP with sudo to avoid setting up environment variables yourself:

```
$ sudo pip install ck
```

Install this CK repository with all dependencies (other CK repos to reuse artifacts)

```
$ ck pull repo:ck-request-asplos18-mobilenets-tvm-arm
```

A.7.2 Install this CK workflow from the ACM Digital Library snapshot

It is possible to install and test the snapshot of this workflow from the ACM Digital Library without interfering with your current CK installation. Download related file "request-asplos18-artifact-?-ck-workflow.zip" to a temporary directory, unzip it and then execute the following commands:

```
$ . ./prepare_virtual_ck.sh
$ . ./start_virtual_ck.sh
```

All CK repositories will be installed in your current directory. You can now proceed with further evaluation as described below.

Detect and test OpenCL driver

```
$ ck detect platform.gpgpu --opencl
```

Install libBLAS

```
$ sudo apt-get install libblas*
```

To detect and register in CK:

```
ck detect soft:lib.blas
```

To check the environment:

```
$ ck show env --tags=blas,no-openblas
```

A possible output:

```
cfe1e23a4472bb1d  linux-32  32 BLAS library api-3  32bits,blas,blas,cblas,host-os-linux-32,lib,
                                                     no-openblas,target-os-linux-32,v0,v0.3
```

Install OpenBLAS

```
$ ck install package:lib-openblas-0.2.18-universal
```

If you want to test other openblas version:

```
$ ck list package:lib-openblas*
```

Install LaPack

```
$ ck install package:lib-lapack-3.4.2
```

Install or detect llvm/clang compiler v4+

```
$ ck install package --tags=compiler,llvm
```

Though above is the suggested method, you can also install **llvm** via apt and the detect it via CK.

```
# apt-get install llvm-4.0 clang-4.0
$ ck detect soft:compiler.llvm
```

A.7.3 Packages installation

ARM Compute Library

```
$ ck install package:lib-armcl-opencl-17.12  --env.USE_GRAPH=ON --env.USE_NEON=ON \
    --env.USE_EMBEDDED_KERNELS=ON
```

To check/install other versions available via CK

```
$ ck list package:lib-armcl-opencl-*
$ ck install package --tags=lib,armcl env.USE_GRAPH=ON --env.USE_NEON=ON \
    --env.USE_EMBEDDED_KERNELS=ON
```

MXNet with OpenBLAS

```
$ ck install package:lib-mxnet-master-cpu --env.USE_F16C=0
```

NNVM / TVM

```
$ ck install package:lib-nnvm-tvm-master-opencl
```

A.7.4 Original benchmarking (no real classification)

ARM Compute Library client (OpenCL)

This program must be first compiled

```
$ ck compile program:request-armcl-inference
```

and then executed as follows:

```
$ ck run program:request-armcl-inference --cmd_key=all
```

You can also use "ck benchmark" command to automatically set CPU/GPU frequency to max, compile program, run it N times and perform statistical analysis on empirical characteristics:

```
$ ck benchmark program:request-armcl-inference --cmd_key=all
```

We validated results from the [authors](https://github.com/merrymercy/tvm-mali):

```
backend: ARMComputeLib-mali model: vgg16 conv_method: gemm dtype: float32 cost: 1.6511
backend: ARMComputeLib-mali model: vgg16 conv_method: gemm dtype: float16 cost: 0.976307
backend: ARMComputeLib-mali model: vgg16 conv_method: direct dtype: float32 cost: 3.99093
backend: ARMComputeLib-mali model: vgg16 conv_method: direct dtype: float16 cost: 1.61435
backend: ARMComputeLib-mali model: mobilenet conv_method: gemm dtype: float32 cost: 0.172009
backend: ARMComputeLib-mali model: mobilenet conv_method: direct dtype: float32 cost: 0.174635
```

MXNet with OpenBLAS client (CPU)

```
$ ck run program:request-mxnet-inference  --cmd_key=all
 or
$ ck benchmark program:request-mxnet-inference  --cmd_key=all
```

We validated results from the authors (https://github.com/merrymercy/tvm-mali):

```
backend: MXNet+OpenBLAS model: resnet18 dtype: float32 cost:0.4145
backend: MXNet+OpenBLAS model: mobilenet dtype: float32 cost:0.3408
backend: MXNet+OpenBLAS model: vgg16 dtype: float32 cost:3.1244
```

NNVM/TVM client (OpenCL)

```
$ ck run program:request-tvm-nnvm-inference  --cmd_key=all
 or
$ ck benchmark program:request-tvm-nnvm-inference  --cmd_key=all
```

We validated results from the [authors](https://github.com/merrymercy/tvm-mali):

```
backend: TVM-mali model: vgg16 dtype: float32 cost:0.9599
backend: TVM-mali model: vgg16 dtype: float16 cost:0.5688
backend: TVM-mali model: resnet18 dtype: float32 cost:0.1748
backend: TVM-mali model: resnet18 dtype: float16 cost:0.1122
backend: TVM-mali model: mobilenet dtype: float32 cost:0.0814
backend: TVM-mali model: mobilenet dtype: float16 cost:0.0525
```

A.7.5 Real classification (time and accuracy)

Original benchmarking clients did not include real classification in this ReQuEST submission. We therefore provided code for real image classification for each of the above CK programs. This is also required to calculate model accuracy on all (or a subset of) ImageNet data set.

MXNet with OpenBLAS client (CPU)

You can benchmark classification using MXNet with OpenBLAS as follows:

```
$ ck benchmark program:request-mxnet-inference --cmd_key=classify
```

You can also install ImageNet data sets for model accuracy validation as follows:

```
$ ck install package:imagenet-2012-val
or
$ ck install package:imagenet-2012-val-min-resized

$ ck install package:imagenet-2012-aux
```

You can then run accuracy test as follows:

```
$ ck run program:request-mxnet-inference --cmd_key=test --env.STAT_REPEAT=1
```

NNVM/TVM client (OpenCL)

You can benchmark classification and test accuracy using TVM/NNVM as follows:

```
$ ck benchmark program:request-tvm-nnvm-inference --cmd_key=classify
$ ck run program:request-tvm-nnvm-inference --cmd_key=test --env.STAT_REPEAT=1
```

ARM Compute Library client (OpenCL)

ReQuEST promotes reusability of AI/ML workflows, packages and artifacts using CK framework. Since image classification using ArmCL was already shared as CK workflows and packages at https://github.com/dividiti/

ck-request-asplos18-mobilenets-armcl-opencl and added to the ReQuEST scoreboard, we can simply reuse this workflow and compare against public results!

Please, follow this ReadME to reproduce ArmCL classification results on Firefly-RK3399: https://github.com/dividiti/ck-request-asplos18-mobilenets-armcl-opencl.

A.7.6 Validated results

Validated experimental results were recorded and processed using scripts in the following CK entry:

```
$ ck find script:benchmark-request-tvm-arm
```

We plan to automate them further for the future ReQuEST editions.