Efficient Deep Learning Inference on Edge Devices

Ziheng Jiang*
Fudan Unversity
ziheng@apache.org

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

Mu Li
Amazon AI
mli@amazon.com

ABSTRACT

Deploying deep learning (DL) models on edge devices is getting popular nowadays. The huge diversity of edge devices, with both computation and memory constraints, however, make efficient deployment challenging. In this paper, we propose a two-stage pipeline that optimizes DL models on target devices. The first stage optimizes the inference workloads, and the second stage searches optimal kernel implementations on the target device. We implemented this pipeline with the TVM stack. Our contributions include new algorithmic optimization that is crucial to edge devices, such as quantization and joint kernel turning. On Raspberry Pi, compared to manually optimized frameworks, we will demonstrate our pipeline improves inference latency by 3x for ResNet-18 and by 10x for MobileNet, and generates compact runtime library with size less than 1MB.

ACM Reference Format:
Ziheng Jiang, Tianqi Chen, and Mu Li. 2018. Efficient Deep Learning Inference on Edge Devices. In *Proceedings of ACM Conference on Systems and Machine Learning (SysML'18).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Thanks to the rapid performance increase of low-power processors and the huge demand of Internet of Things applications, there is a growing interest in deploying deep learning models on edge devices [12]. It not only reduces the latency of streaming data back to cloud but also saves the cost. These devices, however, are still an order of magnitude lower than desktop and server processors. The performance, together with memory consumption, therefore, is the key to success.

One major challenging for optimizing the performance is the huge diversity of edge devices. Even for the same ARM CPU architecture, each chip vendor may use its own unique specification, despite the more diverse low-power GPU, DSP and DL accelerators.

There exists libraries to simply this optimization. NNPack [5] provides manually optimized neural network operators on ARM CPUs. Android NN [6] targets for efficient DL inference on Android devices while Core ML [2] aims for the same thing but only for Apple devices. To our best knowledge, none of them is able to optimize models trained with different DL frameworks on a wide range of edge devices.

In this paper, we propose an end-to-end pipeline that optimizes DL inference on various edge devices. Given a pre-trained model and a target device, we first optimize its computation graph to simplify the inference workload, and then automatically search an efficient implementation on the target device. The outputs contain a minimal runtime for the target device, a set of optimized operator kernels, and compact model parameters.

2 GRAPH OPTIMIZATION

DL workloads can be represented as computation graphs, whose vertices are abstract operators, such as `conv2d`, which are then bound with kernels that describe the concrete computational procedures. The edges present the data, which are multi-dimensional arrays with a particular data type, passed between operators.

The goal of graph optimization is transforming a computation graph into another form to reduce execution time and memory consumption. We list several optimization procedures as the following, which share a similarity to compiler optimization.

Constant Folding. If an operator only has constant inputs, then we can replace it by pre-computing its results.

Graph Simplification. Some operators can be simplified during inference. For example, dropout [14] layers can be removed because it is an identity function during inference. The batch-norm layer [10] normalizes input x by

$$y = \gamma \frac{x - mean}{\sqrt{var + \epsilon}} + \beta = \left(\frac{\gamma}{\sqrt{var + \epsilon}}\right) x + \left(\beta - \frac{mean}{\sqrt{var + \epsilon}}\right)$$

All inputs except for x are constants and therefore it can be folded to a multiplication followed by a plus.

Kernel Fusion. Multiple operators can be grouped together to be bound with a single fused kernel. A fuse kernel is often more efficient due to better memory locality and consumes less memory. One example rule is fusing an operator with its subsequent elementwise operators, such as `conv2d` with all following activation and batch-norm layers.

Pre-computing Layout Transformation. A kernel may use different data layouts internally to improve the memory locality. If inputs are constant, e.g. model weights, then we can pre-compute the layout transformation by inserting proper transformation operators into the graph and then performing constant folding.

Quantization. Use a low-precision data type than the commonly used 320bit floating-point reduces not only memory usage but may also accelerate the computation and save the power [7].

We followed [7] to use the fixed-point data type with randomized rounding. The quantization operator may lead to enormous overhead on edge devices. We pre-compute both the number of

*This work is done by Ziheng Jiang during his internship at Amazon AI

SysML'18, April 2018, Stanford, CA USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

fractional bits with a validation dataset and a random number bank to accelerate the quantization.

3 KERNEL OPTIMIZATION

A kernel can be partitioned into two parts: one is the algorithm that specifies how to compute outputs, the other one is the schedule that defines the execution of the algorithm. The kernel efficiency, therefore, depends on both the arithmetic complexity of the algorithm and hardware resource utilization of the schedule.

Take conv2d as an example, the plain algorithm computes 2D convolution straightforward by definition. A fast algorithm may use either FFT or Winograd to accelerate the convolution [11]. A schedule explores multiple aspects to improve the execution efficiency on hardware, such as:

Tiling. Partition each input into blocks to fit into cache, then compute block by block.

Reordering. Arrange the order of for loops to improve memory locality. .

Unrolling. Replace a for loop into repeated code sentences.

Vertorization: Replace a for loop with vector instructions.

Parallelization. Execute a for loop in parallel.

The best schedule depends on the hardware specification, such as the cache size, the number of cores, and the instruction set. Manually optimizing schedule for edge devices is difficult due to the diversity. Instead, for each operator with particular input shapes, we generate a large number of configurations with various tile sizes, loop orders, parallelization schemes, and unrolling and vectorization lengths. Different to [13] by using a cost model, which is also hard for edge devices, we benchmark each configuration on the target device directly and then pick the fastest schedule.

4 EVALUATION

We implemented the graph optimization in NNVM [4], a computation graph manipulation library, which supports multiple DL frameworks as the frontend. Kernels are defined by using the TVM IR [1], which is Tensor IR stack supporting multiple backend devices. We used straightforward algorithms for simplicity.

We benchmarked two convolutional neural networks, ResNet-18 [8] and MobileNet [9]. We report the inference latency with batch size 1. The edge device used for evaluation is Raspberry Pi 3B, which ships a quad-core 1.2GHz ARM Cortex A52 CPU. The schedules are searched on 10 Raspberry Pi in parallel. For each model, it took around an hour to find the best schedules.

We first compare the performance of the searched kernels versus NNPack. As can be seen in Figure 1, the searched kernels outperform NNPACK kernels by 2x in total. Note that NNPack uses Winograd and FFT to accelerate conv2d. Even with a plain algorithm, searching best schedules in a large space on the target device significantly outperforms fast algorithms with sub-optimal schedules.

The performance of the quantized conv2d is shown in Figure 1 as well. The version uses INT8 inputs and accumulates the results with INT32 outperforms the FP32 version by 22%. The A52 CPU provides a more efficient INT8 FMA instruction that accumulates results with INT16, which improves the FP32 version by 1.3x.

The end-to-end inference performance results are shown in Figure 2. We used pre-trained model parameters from MXNet [3], and

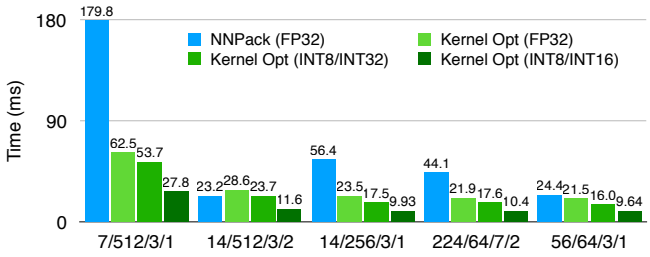

Figure 1: Inference time of the top 5 time consuming convolution layers in ResNet-18. The name convention is *input-shape/output-channel/kernel-size/stride.*

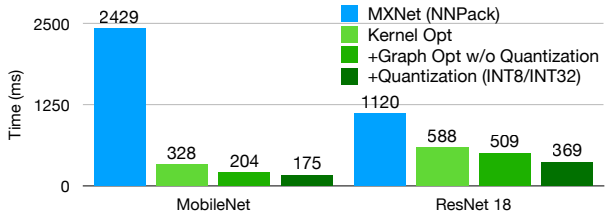

Figure 2: End-to-end inference time.

MXNet with NNPack backend is used as the baseline. As can be seen, even just executing searched kernels sequentially, the proposed pipeline outperforms the baseline by 1.9x for ResNet-18 and 7.4x for MobileNet. The reason that the improvement for MobileNet is more significant than ResNet is because that neither MXNet nor NNPack provide well-optimized depth-wise convolution layers on ARM CPUs. Adding graph optimization except for quantization, we can improve the inference time further by 15% for ResNet-18 and 60% for MobileNet.

For the quantization optimization, we quantized both model weights and inputs into INT8. Despite accumulating results with INT16 is 2x faster than INT32, there is around 5% loss on inference accuracy due to the insufficient numerical precision of INT16. We, therefore, let cond2d output INT32 results. As a result, INT8 quantization reduces 40% latency compared to FP32, with the cost that decreasing the top-1 accuracy from 68.4% to 67.8%. The improvement for MobileNet, 17%, is less significant than ResNet, mainly becuase the quantization overhead takes a larger portion of the total time.

Finally, our pipeline generates a 476KB size runtime for Raspberry Pi. The size of the searched kernels is 179KB for ResNet-18 and 145KB for MobileNet. Graph simplification reduces the size of the batch-norm layer parameters. With INT8 quantization, we can decrease the model size by another 4x.

5 CONCLUSION

In this paper, we proposed a two-stage pipeline to optimize deep learning inference on edge devices. Inference workloads are first optimized through graph transformation, and then optimized kernel implementations are searched on the target device. We demonstrated that the proposed pipeline significantly reduces both runtime library size and inference latency on Raspberry Pi.

REFERENCES

[1] 2017. Tensor Virtual Machine. (2017). https://github.com/dmlc/tvm

[2] Apple. 2017. Core ML: Integrate machine learning models into your app. (2017). https://developer.apple.com/documentation/coreml

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[4] DMLC. 2016. Neural Network Virtual Machine. (2016). https://github.com/dmlc/nnvm

[5] Marat Dukhan. 2016. NNPACK: an acceleration package for neural network computations. (2016). https://github.com/Maratyszcza/NNPACK

[6] Google. 2017. Android NDK: Neural Networks API. (2017). https://developer.android.com/ndk/guides/neuralnetworks/index.html

[7] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 1737–1746.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[9] 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. *arXiv preprint arXiv:1704.04861* (2017).

[10] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*. 448–456.

[11] 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.

[12] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. 2017. Deep Learning for IoT Big Data and Streaming Analytics: A Survey. *arXiv preprint arXiv:1712.04301* (2017).

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

[14] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research* 15, 1 (2014), 1929–1958.