doi:10.3772/j.issn.1006-6748.2019.04.010

Optimizing deep learning inference on mobile devices with neural network accelerators①

Zeng Xi(曾 惜)②* ** ***, Xu Yunlong***, Zhi Tian* ***
(* Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, P. R. China)
(** University of Chinese Academy of Sciences, Beijing 100049, P. R. China)
(*** Cambricon Technologies Corporation Limited, Beijing 100191, P. R. China)

Abstract

Deep learning has now been widely used in intelligent apps of mobile devices. In pursuit of ultra-low power and latency, integrating neural network accelerators (NNA) to mobile phones has become a trend. However, conventional deep learning programming frameworks are not well-developed to support such devices, leading to low computing efficiency and high memory-occupation. To address this problem, a 2-stage pipeline is proposed for optimizing deep learning model inference on mobile devices with NNAs in terms of both speed and memory-footprint. The 1st stage reduces computation workload via graph optimization, including splitting and merging nodes. The 2nd stage goes further by optimizing at compilation level, including kernel fusion and in-advance compilation. The proposed optimizations on a commercial mobile phone with an NNA is evaluated. The experimental results show that the proposed approaches achieve 2.8 × to 26 × speed up, and reduce the memory-footprint by up to 75%.

Key words: machine learning inference, neural network accelerator (NNA), low latency, kernel fusion, in-advance compilation

0 Introduction

In recent years, more and more deep learning (DL) models have delivered outstanding performance in varieties of fields, including image classification[1,2] andnatural language processing[3,4]. DL models often have heavy load of computing and large size of model files, making it hard to deploy these models on edge devices like mobile phones, whose performance are still far behind the normal desktops and remote servers. However, there are more and more demands for applying DL models on mobiles, as it helps to protect privacy and saves time of transporting streaming data generated by edge users from local device to cloud centers. To launch DL models on mobile devices, there are mainly 2 challenges.

1) Latency. Mobile users are often sensitive to the response time of apps. Besides, DL models are evolving deeper and deeper. In the filed of image classification, AlexNet from Ref. [5] has only 8 layers and nearly 60 million parameters approximately, while Inception-V3 from Ref. [6] has 47 layers and nearly 5 billion parameters.

2) Memory-footprint. Compared to server processors, memory size of mobile devices is usually small. When doing DL inference, we need to minimize the occupation of memory, especially when mobile devices are doing multiple tasks.

Thanks to the development of mobile's neural network accelerators (NNAs), latency and energy consumed when performing inference has decremented drastically. However, there also exist limitations of NNAs. Firstly, as the special-purpose processors, NNAs often have different instructions from general processors. Common deep learning frameworks are not able to fully support all these different architectures. Secondly, NNAs perform well on computing, while they lack enough optimization for I/Os. Thirdly, their memory capacities, similar to mobile devices', are the bottleneck when processing large scale data.

In this work, a pipeline for optimizing inference

① Supported by the National Key Research and Development Program of China (No. 2017YFB1003101, 2018AAA0103300, 2017YFA0700900), the National Natural Science Foundation of China (No. 61702478, 61732007, 61906179), the Beijing Natural Science Foundation (No. JQ18013), the National Science and Technology Major Project (No. 2018ZX01031102) and the Beijing Academy of Artificial Intelligence.
② To whom correspondence should be addressed. E-mail: zengxi15@mails.ucas.ac.cn
Received on Oct, 11, 2018

on mobile devices with NNAs is proposed. Given a pre-trained DL model, graph optimization like splitting or merging nodes to reduce workload of computation is firstly done, and then the model is transformed into a special binary model through in-advance compilation. Meantime, a memory-reuse assignment is also performed to save memory cost.

The paper's main contributions contain:

1) A 2-stage optimizing pipeline for accelerating inference on mobile devices with NNAs is introduced which is also memory-efficient.

2) The proposed optimizations are implemented on concrete deep learning models running on a real commercial mobile device with an NNA, and results show that our method is capable of improving inference performance.

1 Background

In this section, background information of deep learning and common DL programing frameworks is firstly introduced. Then neural network accelerators in recent days are discussed.

1.1 DL graph architecture

Deep learning, consisting of at least one deep neural network (DNN)[7], is inspired by the connection of neurons of human brain. Common operations in a DNN include convolution, activation and so on. To facilitate users' programming, DL frameworks like Tensorflow[8] and MxNet[9] are using computation graph to represent the model. Data flows along with the connection in the graph. When they arrive a graph node, a particular computation like addition or convolution will launch. This is called layer-by-layer execution.

In a heterogeneous processor architecture, each node can be assigned to a particular device. For example, all addition nodes can be placed to a GPU and all convolution operation nodes to an NNA. According to the varied status (memory size, number of cores, etc.) of these processors, device placement should be adjusted to maximize utilization of their computation and memory resources.

1.2 Neural network accelerators

In 2012, Google used 16 000 CPU cores to train a neural network (NN) for teaching computer to recognize cat[10], which costs 3 days to finish the training. A single NN operation may trigger thousands of instructions in a CPU, which needs too much time to load, decode and manage them. GPUs are processors specialized for complicated mathematics and geometry. They accelerate deep learning significantly. But they focus more on parallel and cannot fully meet requirements of DL models that may have many branches and cannot be efficiently parallelized.

Application specific integrated circuit (ASIC) has now become a popular way to accelerate deep learning[11]. In 2016, Google launched TPU (Tensor Processing Unit)[12], an ASIC processor for its open-sourced framework Tensorflow. Chen et al[13] designed DianNao that could have the same performance as the main stream GPUs while using much smaller chip size and less energy. DaDianNao, based on DianNao, even outperformed NVDIA K20M to a large extent[14-16]. There are also many other ASIC accelerators that have been designed to meet different demands of deep learning[17]. These ASIC neural network accelerators have been proved to significantly speed deep learning together with general purpose processors.

Though ASIC NNAs are helpful, they are not perfect. They are extremely good at computing but they may lack in some other aspects as below.

1) As the coprocessors, NNAs need CPU cores to begin launch, which can be costly since launching time is the same as that of computing kernels.

2) NNAs are not quite suitable for I/Os. So it is time-consuming to do operations like split, concatenate and so on.

3) Like mobile devices, they also have strictly limited on-chip memory.

2 Methodology

In this section, techniques of 2-stage optimization pipeline are presented in detail as shown in Fig. 1. Several graph optimization methods for the given model are discussed. Next, kernel fusion and in-advance compilation of the model are dipicted.

2.1 Graph optimization

DL models after training often remain many redundant nodes. Based on this fact, we implement several ways to release computing workload as much as we can.

Splitting nodes Sometimes there are operations that NNAs cannot compute efficiently. Taking LSTM-BlockCell operation in Tensorflow as an example, part of its computation is shown as the following.

$$xh = [x, h]$$

$$[i, c, f, o] = xh \times \boldsymbol{w} + b$$

where x has the shape $[bs, is]$, h has the shape $[bs, cs]$, and $\boldsymbol{w}$ has the shape $[is + cs, cs \times 4]$. It first concatenates input x and h to a new tensor, and then it does Matmul with matrix ***w*** and BiasAdd with a bias b to get result which has eventually been split into 4 parts along the 2nd axis.

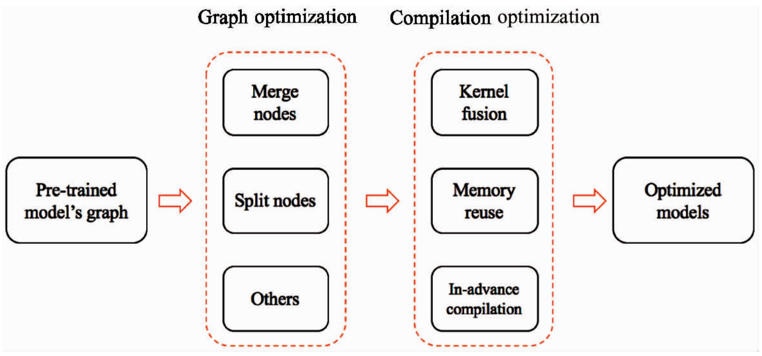
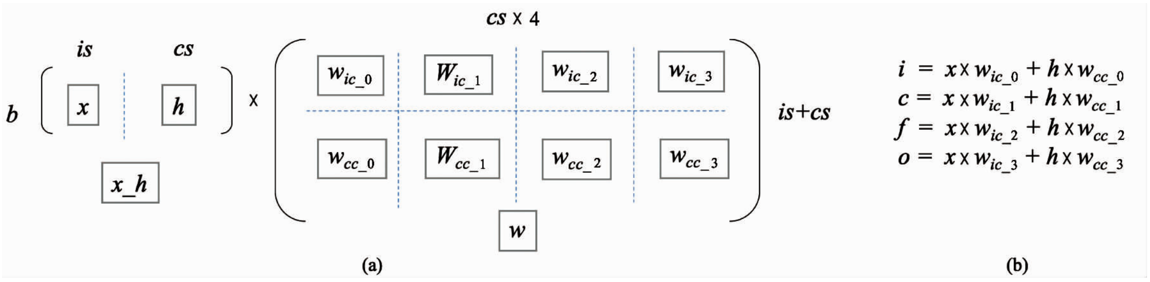


Fig. 1 Illustration of the complete process of our optimization pipeline

As mentioned on Section 1.2, performing splitting or concatenating operation on an NNA is not a wise choice. So that here we change the computation to avoid this situation. Rather than concatenating inputs, firstly the weight matrix ***w*** is split into 8 parts, as can be seen in Fig. 2. Each part does a MatMulandBiasAdd with input x or h. Then additions are conducted to get the final output. Since the weight matrix ***M*** is constant, we are able to do the split before launching this model, which saves time greatly.


cs x 4
is
cs
b
x
h
x_h
x
w_{ic_0} W_{ic_1} w_{ic_2} w_{ic_3}
w_{cc_0} W_{cc_1} w_{cc_2} w_{cc_3}
w
is+cs
$i = x \times w_{ic\_0} + h \times w_{cc\_0}$
$c = x \times w_{ic\_1} + h \times w_{cc\_1}$
$f = x \times w_{ic\_2} + h \times w_{cc\_2}$
$o = x \times w_{ic\_3} + h \times w_{cc\_3}$
(a)
(b)

(a) How we split weight ***w*** in to 8 parts; (b) How we do matrix multiply and addition to get the right result

Fig. 2 Illustration of our method of doing LSTMBlockCell

Merging nodes When a node or a kernel has been computed by an NNA, it experiences 4 stages showed in Fig. 3. It firstly needs a CPU core to begin the launch, and then loads data from DDR. Next it does computation and returns results back to the CPU. Kernel launch can be costly. It may take tens of microseconds to several milliseconds, which is the same as kernel computation on NNAs. Especially when the CPU is busy, launching costs more.

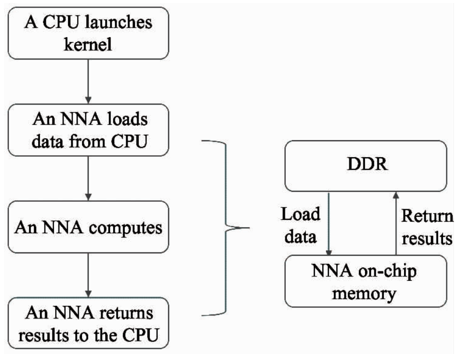


Fig. 3 Workflow of an NNA's computing an operation/kernel

Since each operation needs launching kernel once a time, several nodes are chosen to merge into a single one, which brings us 3 aspects of benefits as follows.

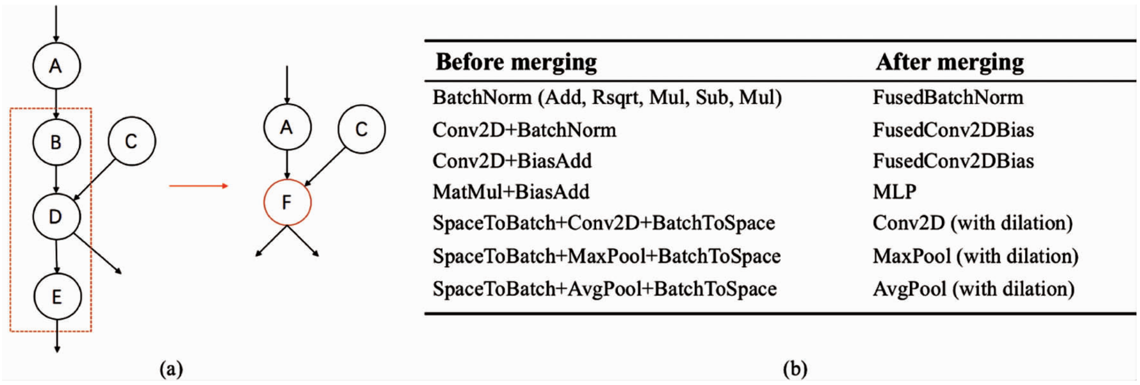
1) The number of kernel launches could be reduced, saving time and energy.

2) The kernel's internal computations can be optimized can be optimized by eliminating unnecessary operations or change their order to gain better memory locality.

3) Since outputs generated by former kernels are consumed by latter kernels immediately, there is no need to transport temporary data from on-chip memory to off-chip DDR, which saves time.

We have concluded several patterns of kernel combinations that can significantly help to improve performance, as shown in Fig. 4(b). Go through the whole graph, merge nodes when encountering these patterns.

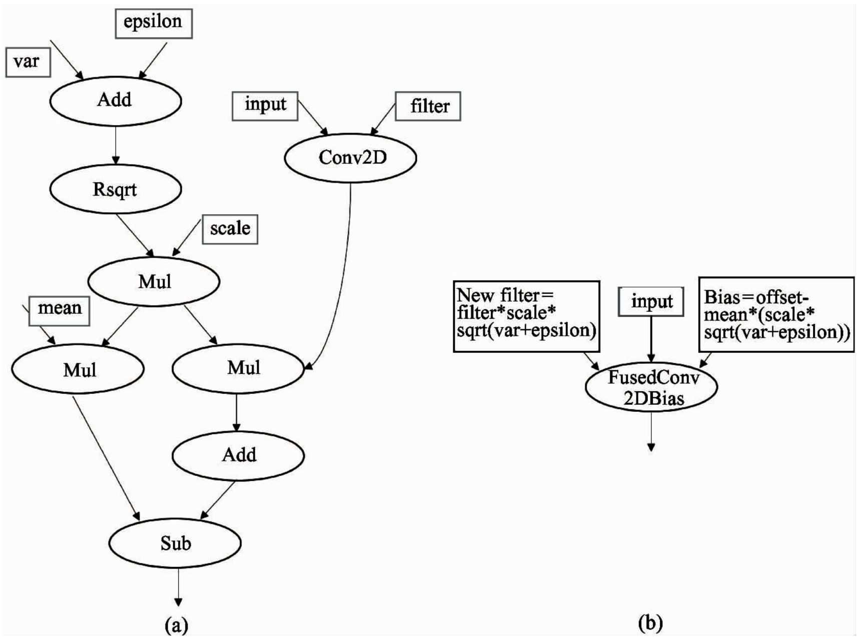
When merging nodes, we should pay attention to inputs and outputs. As can be seen in Fig. 4(a), both node B and D have an input from outside. So the merged node F's input should be A and C. Similarly, its outputs are the outputs of all its internal nodes.



Before merging	After merging
BatchNorm (Add, Rsqrt, Mul, Sub, Mul)	FusedBatchNorm
Conv2D+BatchNorm	FusedConv2DBias
Conv2D+BiasAdd	FusedConv2DBias
MatMul+BiasAdd	MLP
SpaceToBatch+Conv2D+BatchToSpace	Conv2D (with dilation)
SpaceToBatch+MaxPool+BatchToSpace	MaxPool (with dilation)
SpaceToBatch+AvgPool+BatchToSpace	AvgPool (with dilation)

Fig. 4 (a) Graph of how we merge nodes B, D, E (in dashed rectangular) are merged to a single node F; (b) Table of patterns which are useful for improving performance when merging them together

To illustrate how can we benefit from merging nodes, we take combining Conv2D and BatchNorm[18] as an example. BatchNorm means doing normalization in batch, aiming to solve problem of vanishing gradient[19] which is a common but unneglectable situation in deep learning. It consists of 7 nodes, as being visualized in Fig. 5(a). When the training process has been accomplished, weights of BatchNorm have been fixed, so that we can pre-compute these constants rather than calculate all 7 nodes every time. Jiang et al[20] also reduces the computation of BatchNorm while we do more than his strategy. When we merge BatchNorm and Conv2D, all data are constant except the original input. We firstly multiply Conv2D's filter with BatchNorm's constant input to form a new filter. Then only an Add operation after convolution is needed. As listed in Fig. 4(b), we have already merged Conv2D and BiasAddor Add to a new operation named FusedConv2DBias. So we are able to incorporate BatchNorm into a single convolution operation which has a new filter and a following BiasAdd operation. There is no need to move data out since we do BiasAdd immediately after convolution. So far, we have managed to reduce totally 8 independent operations into a single one, as can be seen in Fig. 5(b).



(a) Original graph of Conv2D and BatchNorm that consists of 7 operations;
(b) New graph after Conv2D and BatchNorm are merged into a new node named FusedConv2DBias

Fig. 5 Example process of merging nodes

Miscellany Not all operations contained in the graph are necessary for inference. Those redundant nodes could be safely removed if they are in one of the following types.

1) Nodes' input are only weight data or other constant data. These nodes can be pre-computed since they don't have to wait for input data from other nodes.

2) Nodes only need intraining process. For example, nodes for loading datasets, saving models or updating weights, are useless during inference. Another example is Dropout, which will not be calculated too.

2.2 Compiling optimization

In the 2nd stage, we do optimization on the compilation levelwith the aim of accelerating computing and saving memory-footprint.

Fusing kernels as many as possible We use fusion kernel to represent a set of continuous operations on the same device. These kernels have been fused together. A single NNA's launch processes a single fusion kernel. This fusion is not the same as the method mentioned in Section 2.1, which changes the graph, explicitly replacing multiple nodes with a single merged one. Fusion kernel is a compiling notion and its internal kernels' computation has been delayed as much as possible, which is the most significant difference compared to the normal way of layer-by-layer graph execution.

Our optimization goal is based on the following principle. Since NNAs give high performance on NN computation, we want to assign operations to it as many as possible, which means the larger the fusion kernel is, the better performance it will have. This is because:

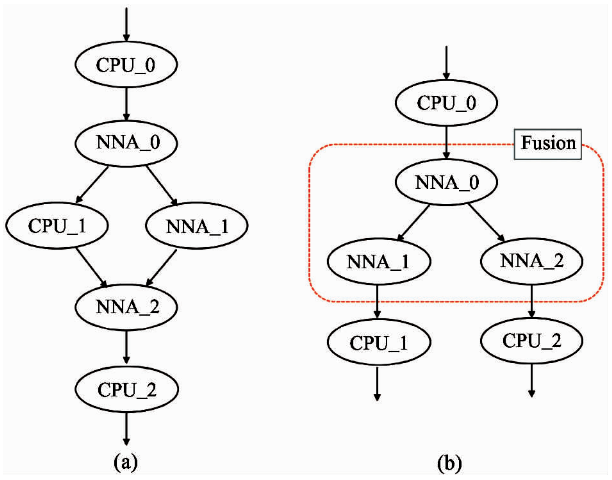
1) Number of kernel launches can dramatically be reduced, even for some networks only one time launch is needed.

2) Number of instructions of loading DDR can be greatly reduced, since temporary data will not be transported from the NNA'son-chip memory to outside until the fusion kernel has finished computation.

Kernel fusion is implemented on the following way. When a kernel has been loaded, we obtain its operation's description (including operation type, source nodes and input data shape, etc.) and not actually do the computation but caching them into a fusion kernel list. Then the next kernel is processed, and next. When we find an operation outputs for a kernel on a different device or there is no enough space for the current kernel to do the computation (we pre-allocate memory for every cached operation), we then stop and actually launch those kernels we cached before.

Unexpectedly, not all continuous kernels on the same device can be compiled into a single fusion kernel. As shown in Fig. 6(a), the kernel NNA _ 2 cannot be merged into fusion consisting of NNA _ 0 and NNA _ 1. Because NNA _ 2 needs inputs from both CPU _ 1 and NNA _ 1, while CPU _ 1 needs inputs from NNA _ 0. If we fused NNA _ 2 with NNA _ 0, NNA _ 2 can no longer wait out its inputs' coming.

Another special situation is shown as Fig. 6(b). Both NNA _ 1 and NNA _ 2 have outputs for next CPU _ 1 and CPU _ 2. According to the algorithm mentioned above, NNA _ 1 finds its output will be consumed by CPU _ 1, a node on other device. So it begins launching current cached kernels. But at this moment NNA _ 2 hasn't been added to this fusion yet while it should be. To solve this problem, we have to delay sending outputs from NNA _ 1 to CPU _ 1, waiting for NNA _ 2 being added and launched. In another word, we must guarantee that all data transportation of fusion kernels on different devices takes place after all kernels in this fusion finish their computation.



(a) One situationthat two continuous nodes on the same device can not be fused; (b) One situation that control dependency is added to some nodes to fuse them

Fig. 6 Special case when performing kernel fusion

Reusing memory inside a fusion kernel Inside a fusion kernel, large amount of temporary outputs will be generated and it is necessary to reuse buffer space. A simple reusing-memory assignment method is implemented to reduce the memory footprint during inference.

As shown in the following Pseudocode, a buffer list is used to record and manage allocated buffer space. When a new operation needs buffer, we firstly estimate memory size it requires according to its metadata including inputs/outputs' shape and operation type. Then we ask buffer list if there is such a large space available now. If so, buffer list will return the address of this already-assigned space. While if not, we need to allocate new buffer space and record this buffer into the buffer list, setting its status to be busy. When the operation finishes computation, we only set its buffer to available status rather than actually free it.

- Pseudocode of memory-reuse assignment algorithm

```
class Buffer {
buffer _ addr
buffer _ size
  status
}
buffer _ list = [ ]
for op in fusion _ kernel:
    reuse = false
required _ size = estimate( op. input _ shape, op.
output _ shape, op. type)
    for buffer in buffer _ list:
        if required _ size < buffer. buffer _ size
        and buffer. statu = = available:
            #reuse this buffer
addr = buffer. buffer _ addr
buffer. status = busy
            . . . #do operation
buffer. status = available
            reuse = true
            break
    if not reuse: #cannot reuse buffer
        buffer = allocate( required _ size)
buffer _ list. append( buffer)
buffer. status = busy
        . . . #do operation
    buffer. status = available
```

Compile the model in advance Usually deep learning models are compiled just-in-time (JIT). This is because through compilation frameworks could know metadata of this model, like kernel type, input size or data type. With these information, they could perform some optimizations. But compilation takes a long time.

In-advance compilation is an effective way to improve inference performance on edge device. By saving model into instructions ahead-of-time, on the one hand, we can get rid of redundant DL frameworks but only remain kernels that we need, which helps to reduce model size. While on the other hand, there is no need to compile when doing inference so the response time has been greatly reduced.

Now we explain how to do in-advance compilation to a model with a multiple-devices assignment. We take a model ofTensorflow as an example. As for those fusion kernels assigned to the NNA, a platform-cross compiler is applied to save its instructions and model data. While for those CPU nodes, we seek help to AOT (ahead of time) offered by Tensorflow, which helps to pre-compile a sub graph of the model into a dynamic-linked shared library (. so file) which we can call later during inference. All we need to do is to set the inputs and outputs of this subgraph. If there is no dependency of two fusion kernels on different devices, parallelism between them can be applied to accelerate computation. Fig. 7 is the illustration of this combined pre-compiling method.

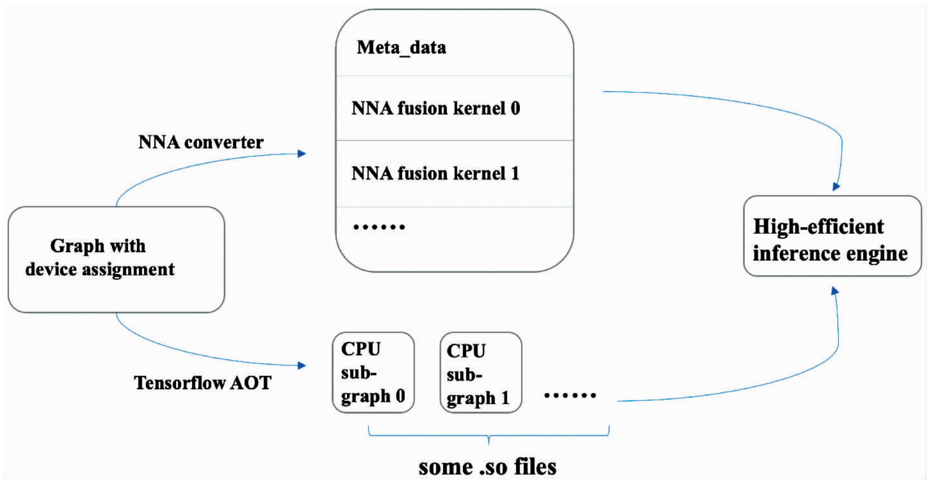


Fig. 7 Illustration of compiling methods in advance

Compiling in advance helps to reduce preparation time when performing inference, but in order to do that, we need to prepare some extra stuff. Firstly, more attention should be paid to the model's detail information. For subgraph consisting of CPU nodes, its input nodes and output nodes should be specified carefully. High-efficient inference engine is another work we need to do to execute the whole inference. It defines how we manage all these small files, and this is also a challenge.

3 Evaluation

We did our experiments of graph and compiling optimization methods based on Tensorflow, Google's open source deep learning framework. The mobile device used for evaluation is acommercial mobile phone with a special AI accelerator. We benchmarked several classical convolution neural networks to see if our methods really work.

3.1 Speed up of kernels

For the method of splitting nodes, LSTMBlockCell is tested with varied input size of *is*, *bs* and *cs*. As can be seen in Fig. 8, on some of the cases our methods could gain a little improvement. This is because we pre-compute the weights which help to reduce time.

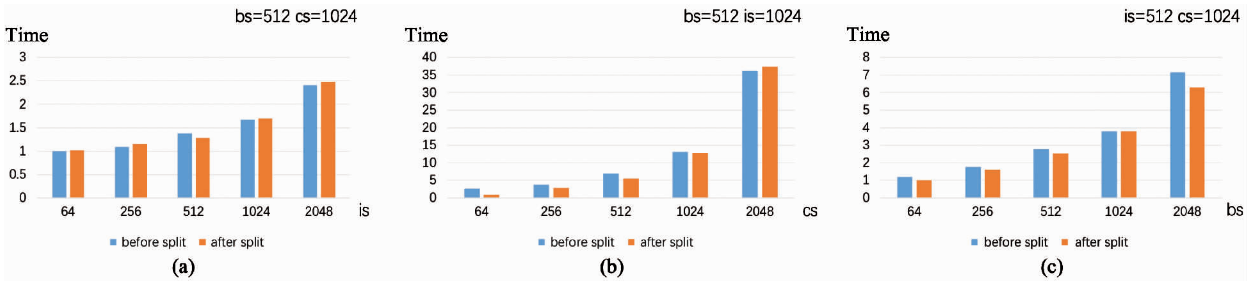


Fig. 8 Results of before and after split LSTMBlockCell node

VGG16 model is also tested before and after merging graph nodes of convolution and adding bias. Table 1 shows the result. We can see that merging nodes outperform original graph on all workloads. When the input channel and output channel are not much deep, more speed up can be gained. This is because convolution's executing time is more allergic to the computation scales while BiasAdd remains nearly unchanged. Despite that, by merging these 2 nodes, we can reduce the time which is occupied by BiasAdd, which still make contributions to overall time-saving.

Table 1 Comparison of before and after merging Conv2D + BiasAdd of VGG16. H/W is the image scale. IC and OC stand for number of input channel and output channel respectively

H/W	IC	OC	conv + bias	fused _ conv _ bias	speed up
224	3	64	1.37	1.00	1.37
224	64	64	1.47	1.05	1.39
112	64	128	1.40	1.03	1.37
112	128	128	1.51	1.14	1.32
56	128	256	2.87	2.52	1.14
56	256	256	2.95	2.64	1.12
56	256	256	2.95	2.68	1.10
28	512	512	4.32	3.99	1.08
28	512	512	8.38	5.90	1.42
28	512	512	8.29	7.93	1.04
14	512	512	5.74	5.44	1.06
14	512	512	5.65	5.40	1.05
14	512	512	5.72	5.47	1.05

3.2 Speed up of networks

Fig. 9 is the end-to-end inference performance of several models. Pre-trained models are used to classify 1 000 images with batch size set to 1. Every model has been executed 100 times and each batch's average time are the final performance of this model.

Totally 3 steps are implemented to accelerate inference. Graph optimization can slightly speed up due to removing redundant computations which are still a tiny fractional part compared to large graph with thousands of nodes. But in Inception-V3, which repeatedly has pattern of convolution followed by batch normalization. Its computation time can be effectively reduced by merging this pattern into a single node.

Kernel fusion and in-advance compilation has been proven to be extremely effective to accelerate computing. The deeper the model is, the more speed-up it could have. Kernel fusion helps to dig potential computation ability of NNAs. It helps reducing kernel launches as well as eliminating data transportation between CPU and an NNA. So that with kernel fusion we can improvethe time of inference by 20% for AlexNet and 85% of Inception-V3.

As for in-advance compilation, though we need to manage more small files, it separates compiling from computing, which saves time dramatically. It can be seen from experiments that it brings 2.8 × speed up for VGG16 and 26 × for Inception-v3 compared to their baseline respectively.

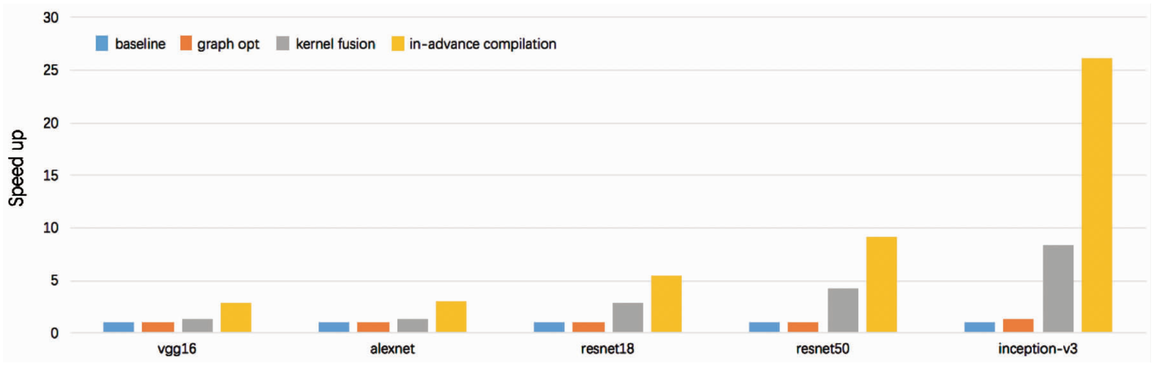


Fig. 9 End-to-end inference performance of five models

3.3 Memory-footprint.

The maximum buffer size for every model's inference is also counted, as can be seen in Table 2. The simple memory-reuse assignment algorithm is able to save at least 2.2% of AlexNet and at most 75.7% of ResNet152. The deeper the model is, the more space it can save. As larger batch size is used, memory saved also greatly increases. This is because buffers' size in the buffer list tends to be large enough, making it easier to be reused. Actually, there exists space for us to improve this simple memory-reuse assignment. Since buffer list chooses to give the first buffer space it finds suitable for the requirements, while in some situation, this buffer may be extremely larger than what we need.

Table 2 Comparison of memory occupation before and after conducting our memory-reuse assignment methods

Model	Batch size	# of layers	Save ratio
alexnet	1	37	2.2%
	10	37	12.8%
vgg16	1	49	12.3%
	5	49	31.6%
resnet152	1	722	55.0%
	5	722	75.7%
googlenet	1	174	38.9%
	10	174	55.3%

4 Related work

Many DL frameworks have been developed to help people both in academic and industry fields[21,22]. On nowadays' AI heterogeneous computing platform, many high performance algorithm libraries like cuDNN[23] and Eigen help frameworks to optimize kernel computation. To further accelerate computation, Tensorflow and MXNet has developed XLA and NNVM/TVM[24] compiler, which can speed up high-frequently-used linear algebra. Both of them do graph optimization to accelerate computation on CPUs and GPUs. Compared to them, NVDIA's NN optimizing tool named TensorRT are targeting runtime rather than compilation. Based on GPUs, TensorRT are able to process DL models pre-trained by many kinds of frameworks.

With the increasing demand of deploying DL models on mobile devices, many tools emerged to accelerate inference. Google launched Tensorflow Lite and AndroidNN. Tensorflow models can be easily converted to a lite model and perform efficient inference by calling AndroidNN on android devices. While on iOS devices, Apple also released a machine learning framework named Core ML.

But all these related work targets on general processor like CPU or widely-used GPU, and there is little work on ASIC NN accelerators that are totally different from CPU in terms of instruction set and computation pattern. What we do in this paper aims to bridge this gap.

5 Conclusion

A 2-stage optimization pipeline for inference on mobile devices with neural network accelerators is performed. The 1st stage uses graph optimization to reduce workload of computation while the 2nd stage optimizes compilation to gain better execution mode as well as memory allocating. After we perform optimization, all DL models we tested are able to gain 2.8 × to 26 × speed up during inference and memory cost can be reduced up to 75.7%.

References

[1] Claudiu C D, Meier U, Gambardella L M, et al. Deep big simple neural nets excel on handwritten digit recognition[J]. *Computing Research Repository*, 2010,22(12): 3207-3220

[2] He K M, Zhan X Y, Ren S Q, et al. Deep residual learning for image recognition [C]//Proceeding of the 29th IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, USA, 2016:770-778

[3] Ankit K, Ozan I, Peter O, et al. Ask me anything: dynamic memory networks for Natural Language Processing [C]//Proceedings of the 33rd International Conference on Machine Learning, New York, USA, 2016:1378-1387

[4] Bengio Y, Ducharme R, Vincent P, et al. A neural probabilistic language model[J]. *Journal of machine learning research*, 2003, 3(2): 1137-1155

[5] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks[C]//International Conference on Neural Information Processing Systems, Nevada, USA, 2012:1097-1105

[6] Szegedy C, Vanhoucke V, Ioffe S, et al. Rethinking the inception architecture for computer vision[C]//Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, USA, 2016:2818-2826

[7] Epelbaum T. Deep learning: technical introduction[J]. *arXiv*: 1709.01412, 2017

[8] Girija S S. Tensorflow: Large-scale machine learning on heterogeneous distributed systems [J]. *arXiv*: 1603.04467, 2016

[9] Chen T Q, Li M, Li Y, et al. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems[J]. *Statistics*, *arXiv*: 1512.01274, 2015

[10] Le Q V. Building high-level features using large scale unsupervised learning [C]//Proceeding of Acoustics, Speech and Signal Processing, Vancouver, Canada, 2013: 8595-8598

[11] Shen Y, Harris N C, Skirlo S, et al. Deep learning with coherent nanophotonic circuits [J]. *Nature Photonics*, 2017, 11(7): 441-441

[12] Jouppi N P, Young C S, Patil N, et al. In-datacenter performance analysis of a fensorprocessing unit[J]. *International Symposium on Computer Architecture*, 2017, 45(2): 1-12

[13] Chen T, Du Z, Sun N, et al. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning[J]. *AcmSigplan Notices*, 2014, 49(4):269-284

[14] Chen Y, Luo T, Liu S, et al. Dadiannao: a machine-learning supercomputer[C]//Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 2014: 609-622

[15] Chen Y, Chen T, Xu Z, et al. DianNao family: energy-efficient hardware accelerators for machine learning[J]. *Communications of the ACM*, 2016, 59(11):105-112

[16] Chen Y, Luo T, Liu S, et al. DaDianNao: aneural network supercomputer[J]. *IEEE Transactions on Computers*, 2016, 66(1):73-88

[17] Han S, Liu X, Mao H, et al. EIE: efficient inference engine on compressed deep neural network[J]. *ACM Sigarch Computer Architecture News*, 2016, 44(3):243-254

[18] Ioffe S, Szegedy C. Batch normalization: accelerating deep network training by reducing internal covariate shift [C]//Proceedings of the 32nd International Conference on International Conference on Machine Learning, Lille, France, 2015: 448-456

[19] Pascanu R, Mikolov T, Bengio Y. On the difficulty of training recurrent neural networks [C]//Proceedings of the 30th International Conference on Machine Learning, Atlanta, USA, 2013:1310-1318

[20] Jiang Z, Chen T, Li M. Efficient deep learning inference on edge devices [C]//the Conference on Systems and Machine Learning, California, USA, 2018

[21] Bastien F, Lamblin P, Pascanu R, et al. Theano: new features and speed improvements[J]. *arXiv*:1211.5590, 2012

[22] Tokui S, Oono K, Hido S, et al. Chainer: a next-generation open source framework for deep learning[C]//Proceedings of Workshop on Machine Learning Systems in the 29th Conference on Neural Information Processing Systems, Montréal, Canada, 2015: 1-6

[23] Chetlur S, Woolley C, Vandermersch P, et al. cuDNN: Efficient primitives for deep learning[J]. *arXiv* :1410.0759, 2014

[24] Chen T, Moreau T, Jiang Z, et al. TVM: end-to-end optimization stack for deep learning[C]// the Conference on Systems and Machine Learning, California, USA, 2018

Zeng Xi, born in 1993. Now, she is a Ph. D candidate in Institute of Computing Technology, Chinese Academy of Sciences. Her research interests are machine learning inference, low latency technology and neural network accelerator.