CENG 5030
Energy Efficient Computing

Lecture 08: Accurate Speedup II

Bei Yu

(Latest update: December 19, 2020)

Spring 2021

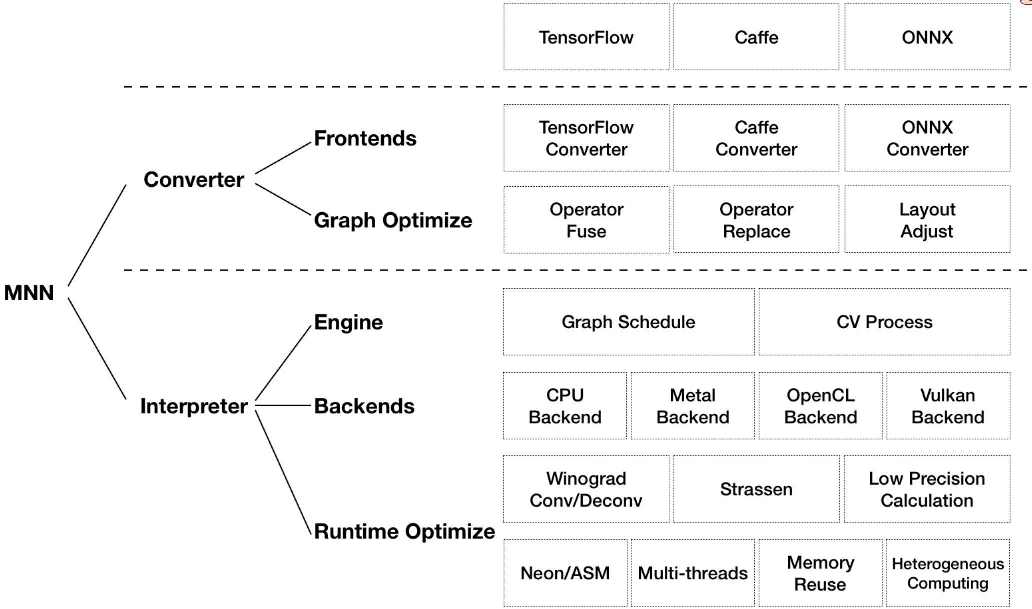
Overview

Overview

MNN Architecture

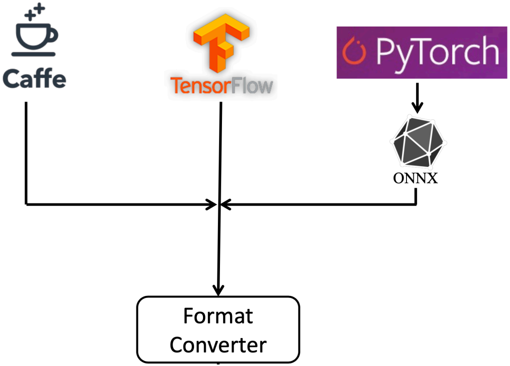
MNN Backend and Runtime

Architecture[1]



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

Frontends



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

PyTorch

- 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

PyTorch Code Sample

```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

TensorFlow

- 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

Tensorflow Code Sample

```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

- 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

Caffe Code Sample

```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)))
```

Data Layout Formats[2]

- N is the batch size
- C is the number of feature maps
- H is the image height
- W is the image width

EXAMPLE
N = 1
C = 64
H = 5
W = 4

c = 0

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

c = 1

20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39

c = 2

40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59

. . .

c = 30

600	601	602	603
604	605	606	607
608	609	610	611
612	613	614	615
616	617	618	619

c = 31

620	621	622	623
624	625	626	627
628	629	630	631
632	633	634	635
636	637	638	639

c = 32

640	641	642	643
644	645	646	647
648	649	650	651
652	653	654	655
656	657	658	659

. . .

. . .

c = 62

1240	1241	1242	1243
1244	1245	1246	1247
1248	1249	1250	1251
1252	1253	1254	1255
1256	1257	1258	1259

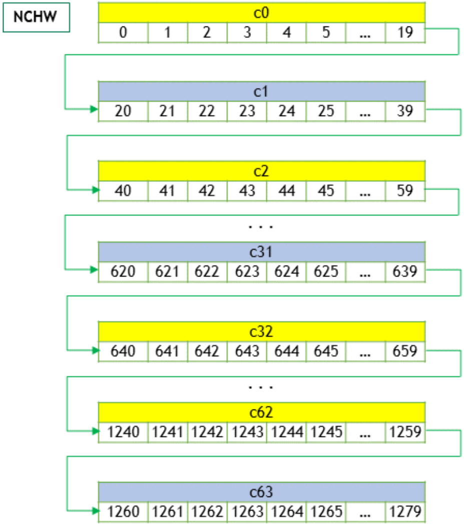
c = 63

1260	1261	1262	1263
1264	1265	1266	1267
1268	1269	1270	1271
1272	1273	1274	1275
1276	1277	1278	1279

[2] https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html

NCHW Memory Layout

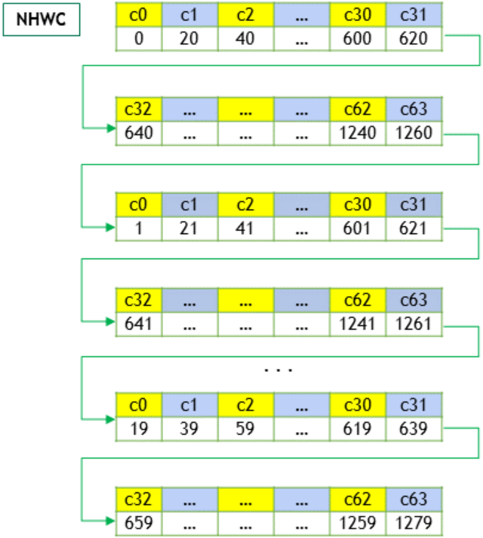
- Begin with first channel (c=0), elements arranged contiguously in row-major order
- Continue with second and subsequent channels until all channels are laid out



NHWC Memory Layout

- Begin with the first element of channel 0, then proceed to the first element of channel 1, and so on, until the first elements of all the C channels are laid out
- Next, select the second element of channel 0, then proceed to the second element of channel 1, and so on, until the second element of all the channels are laid out
- Follow the row-major order of channel 0 and complete all the elements
- Proceed to the next batch (if N is $>$ 1)

NHWC Memory Layout

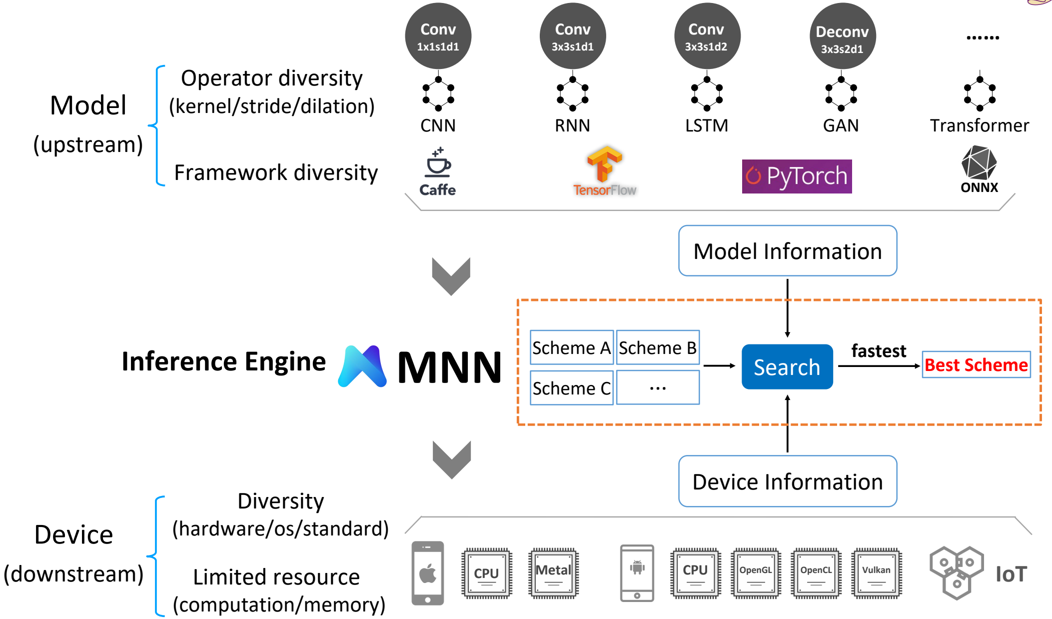


Overview

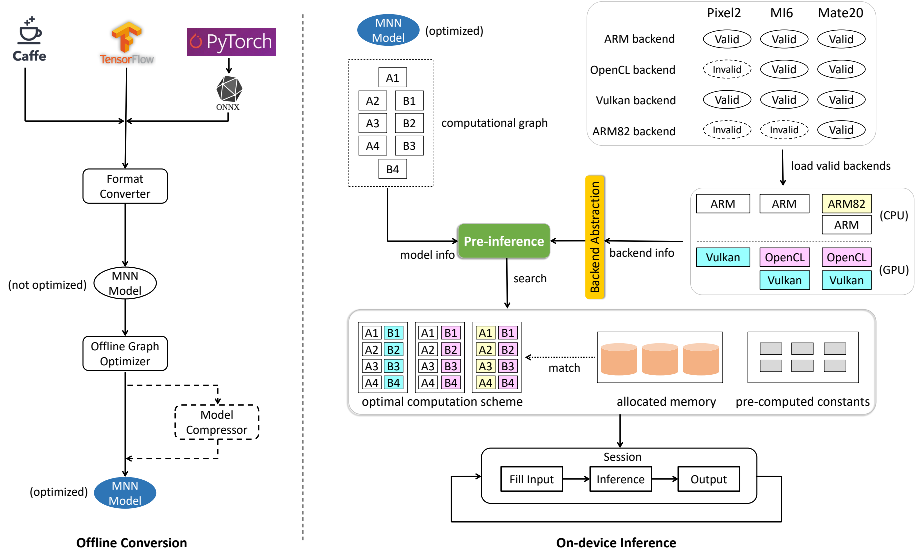
MNN Architecture

MNN Backend and Runtime

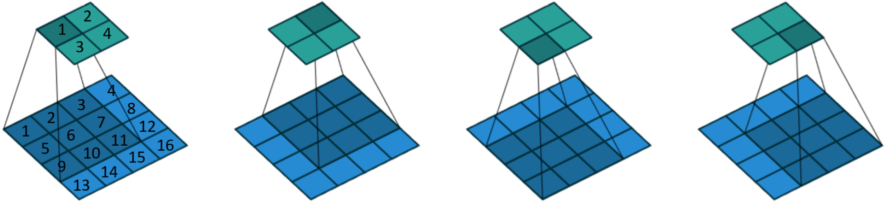
Overview of the proposed Mobile Neural Network



On-device inference



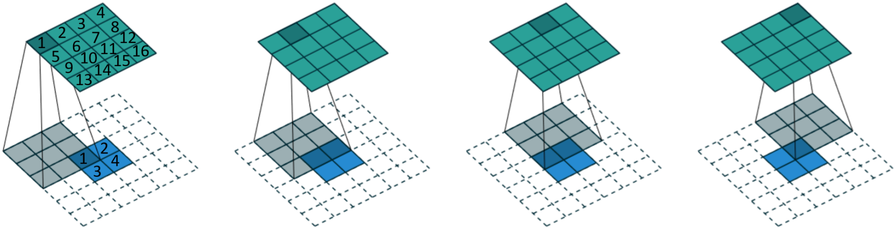
What is Convolution?



The calculation process of convolutional layer

- No padding
- Unit strides
- 3×3 kernel size
- 4×4 input feature map

What is Deconvolution (transposed convolution)?[3]



The calculation process of deconvolutional layer

- 2×2 padding with border of zeros
- Unit strides
- 3×3 kernel size
- 4×4 input feature map

[3] Vincent Dumoulin and Francesco Visin (2016). "A guide to convolution arithmetic for deep learning". In: *arXiv preprint arXiv:1603.07285*.

Strassen Algorithm[4]

[4] Jason Cong and Bingjun Xiao (2014). "Minimizing computation in convolutional neural networks". In: *Proc. ICANN*, pp. 281–290.

Strassen Algorithm

Matrix size	w/o Strassen	w/ Strassen
$(256, 256, 256)$	23	23
$(512, 512, 512)$	191	$\mathbf{176}$ $(\downarrow 7.9\%)$
$(512, 512, 1024)$	388	$\mathbf{359}$ $(\downarrow 7.5\%)$
$(1024, 1024, 1024)$	1501	$\mathbf{1299}$ $(\downarrow 13.5\%)$

```
class XPUBackend final : public Backend {
    XPUBackend(MNNForwardType type, MemoryMode mode);
    virtual ~XPUBackend();
    virtual Execution* onCreate(const vector<Tensor*>& inputs,
                                const vector<Tensor*>& outputs, const MNN::Op* op);
    virtual void onExecuteBegin() const;
    virtual void onExecuteEnd() const;
    virtual bool onAcquireBuffer(const Tensor* tensor, StorageType storageType);
    virtual bool onReleaseBuffer(const Tensor* tensor, StorageType storageType);
    virtual bool onClearBuffer();
    virtual void onCopyBuffer(const Tensor* srcTensor, const Tensor* dstTensor) const;
}
```

Winograd Algorithm[5]

4. Fast Algorithms

It has been known since at least 1980 that the minimal filtering algorithm for computing m outputs with an r-tap FIR filter, which we call $F(m,r)$, requires

$$\mu(F(m,r)) = m + r - 1 \tag{3}$$

multiplications [16, p. 39]. Also, we can nest minimal 1D algorithms $F(m,r)$ and $F(n,s)$ to form minimal 2D algorithms for computing $m \times n$ outputs with an $r \times s$ filter, which we call $F(m \times n, r \times s)$. These require

$$\begin{aligned} \mu(F(m \times n, r \times s)) &= \mu(F(m,r))\mu(F(n,s)) \\ &= (m + r - 1)(n + s - 1) \end{aligned} \tag{4}$$

[5] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *Proc. CVPR*, pp. 4013–4021.

Winograd Algorithm[5]

4.1. F(2x2,3x3)

The standard algorithm for $F(2,3)$ uses $2 \times 3 = 6$ multiplications. Winograd [16, p. 43] documented the following minimal algorithm:

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

[5] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *Proc. CVPR*, pp. 4013–4021.

Winograd Algorithm[5]

Fast filtering algorithms can be written in matrix form as:

$$Y = A^T\left[(Gg) \odot (B^T d)\right] \tag{6}$$

where $\odot$ indicates element-wise multiplication. For $F(2,3)$, the matrices are:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$
$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \tag{7}$$
$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$
$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$$
$$d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$$

[5] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *Proc. CVPR*, pp. 4013–4021.

Winograd Algorithm[5]

A minimal 1D algorithm $F(m, r)$ is nested with itself to obtain a minimal 2D algorithm, $F(m \times m, r \times r)$ like so:

$$Y = A^T \left[[GgG^T] \odot [B^T dB] \right] A \tag{8}$$

where now g is an $r \times r$ filter and d is an $(m + r - 1) \times (m+r-1)$ image tile. The nesting technique can be generalized for non-square filters and outputs, $F(m \times n, r \times s)$, by nesting an algorithm for $F(m, r)$ with an algorithm for $F(n, s)$.

$F(2\times 2, 3\times 3)$ uses $4\times 4 = 16$ multiplications, whereas the standard algorithm uses $2 \times 2 \times 3 \times 3 = 36$. This

[5] Andrew Lavin and Scott Gray (2016). “Fast Algorithms for Convolutional Neural Networks”. In: *Proc. CVPR*, pp. 4013–4021.

Winograd Algorithm[5]

The transforms for $F(3 \times 3, 2 \times 2)$ are given by:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}, G = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \\ 0 & 1 \end{bmatrix} \quad (14)$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

With $(3+2-1)^2 = 16$ multiplies versus direct convolution's $3 \times 3 \times 2 \times 2 = 36$ multiplies, it achieves the same $36/16 = 2.25$ arithmetic complexity reduction as the corresponding forward propagation algorithm.

[5] Andrew Lavin and Scott Gray (2016). "Fast Algorithms for Convolutional Neural Networks". In: *Proc. CVPR*, pp. 4013–4021.

Winograd Algorithm[5]

4.3. F(4x4,3x3)

A minimal algorithm for $F(4,3)$ has the form:

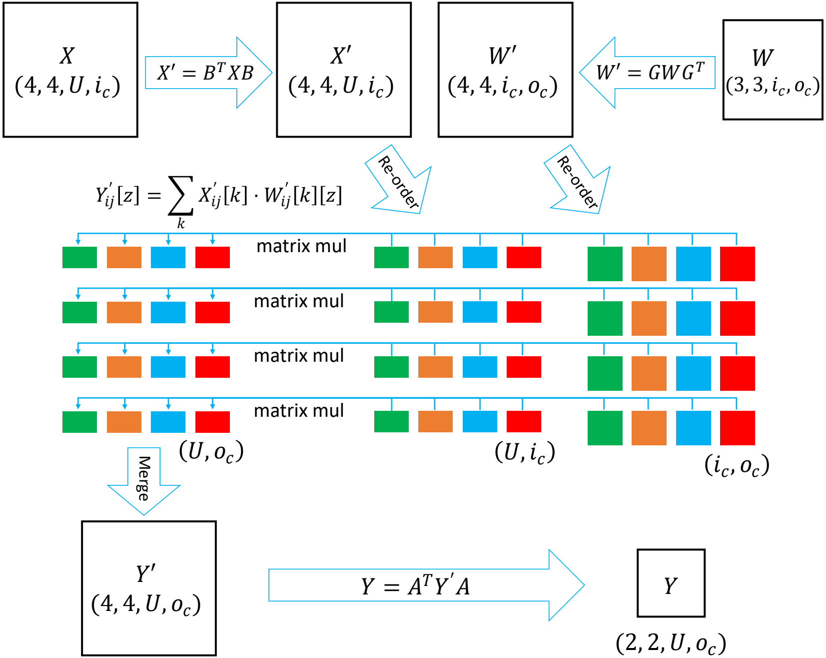
$$
\begin{aligned}
B^T &= \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix} \\
G &= \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix} \\
A^T &= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix}
\end{aligned}
\tag{15}
$$

The data transform uses 12 floating point instructions, the filter transform uses 8, and the inverse transform uses 10.

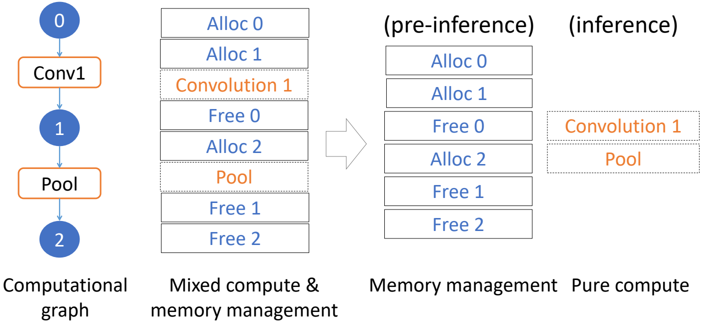
Applying the nesting formula yields a minimal algorithm for $F(4 \times 4, 3 \times 3)$ that uses $6 \times 6 = 36$ multiplies, while the standard algorithm uses $4 \times 4 \times 3 \times 3 = 144$. This is an arithmetic complexity reduction of 4.

[5] Andrew Lavin and Scott Gray (2016). “Fast Algorithms for Convolutional Neural Networks”. In: *Proc. CVPR*, pp. 4013–4021.

Optimized Winograd algorithm in MNN



Memory optimization of MNN



- MNN can infer the exact required memory for the entire graph:
  - virtually walking through all operations
  - summing up all allocation and freeing

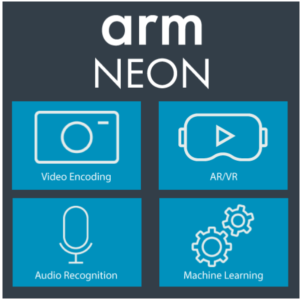
Inference in FP16

- 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)

Analysis of FP16 inference

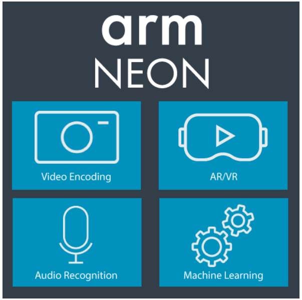
- Advantages of FP16:
  - FP16 improves speed (TFLOPS) and performance
  - FP16 reduces memory usage of a neural network
  - FP16 data transfers are faster than FP32
- Disadvantages of FP16:
  - They must be converted to or from 32-bit floats before they are operated on

Neon optimization



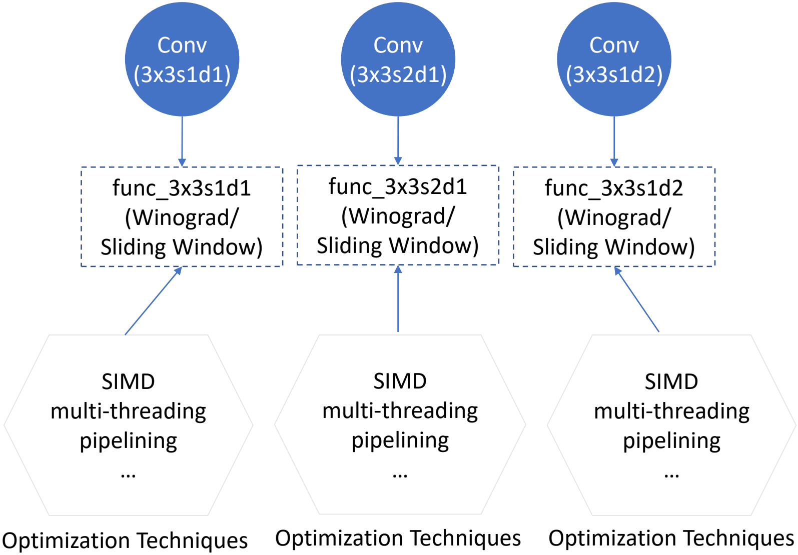
- As a programmer, there are several ways you can use Neon technology:
  - Neon intrinsics
  - Neon-enabled libraries
  - Auto-vectorization by your compiler
  - Hand-coded Neon assembler

Why use Neon

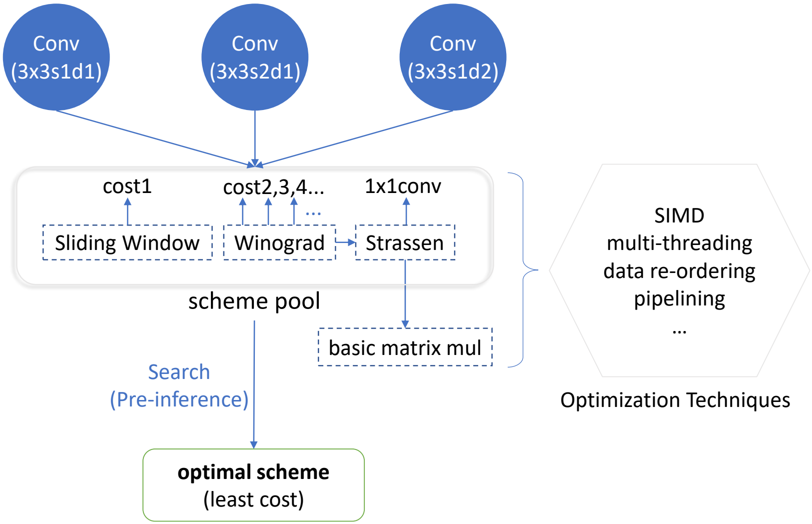


- Support for both integer and floating point operations ensures the adaptability of a broad range of applications, from codecs to High Performance Computing to 3D graphics.
- Tight coupling to the Arm processor provides a single instruction stream and a unified view of memory, presenting a single development platform target with a simpler tool flow

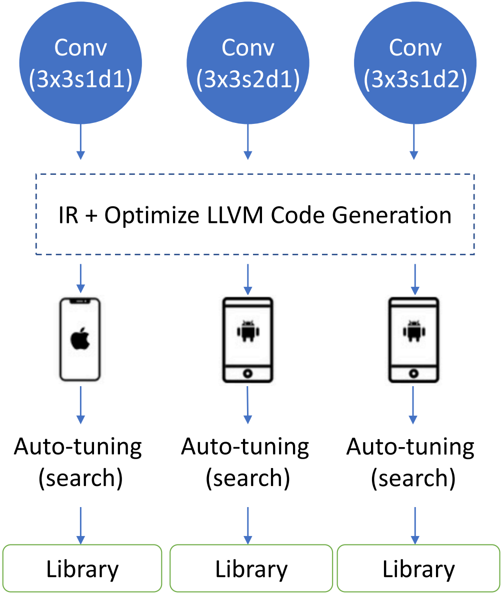
Manual Search



Semi-automated Search



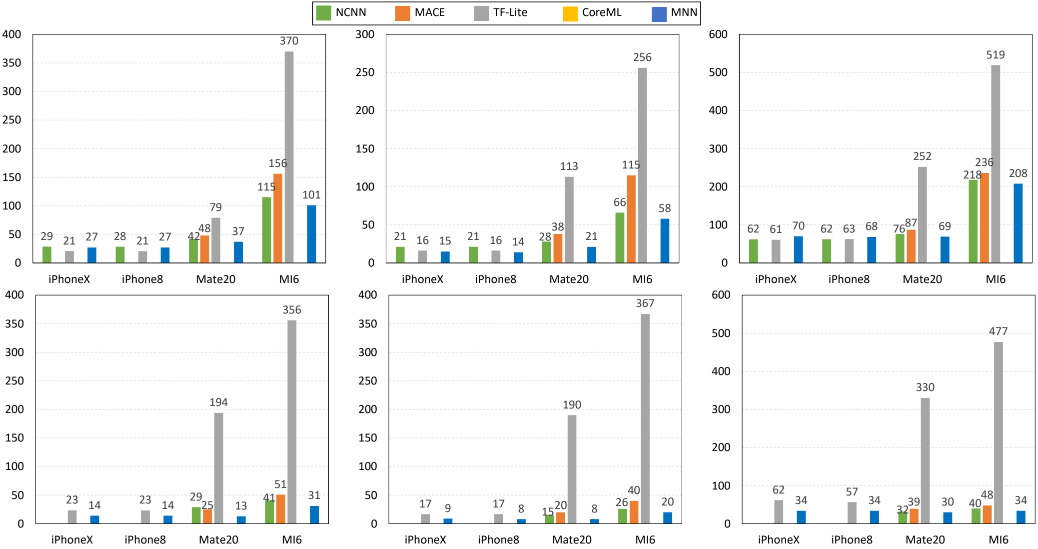
Automated Search



Performance on different smartphones and networks

- 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

Performance on different smartphones and networks



Performance on different smartphones and networks