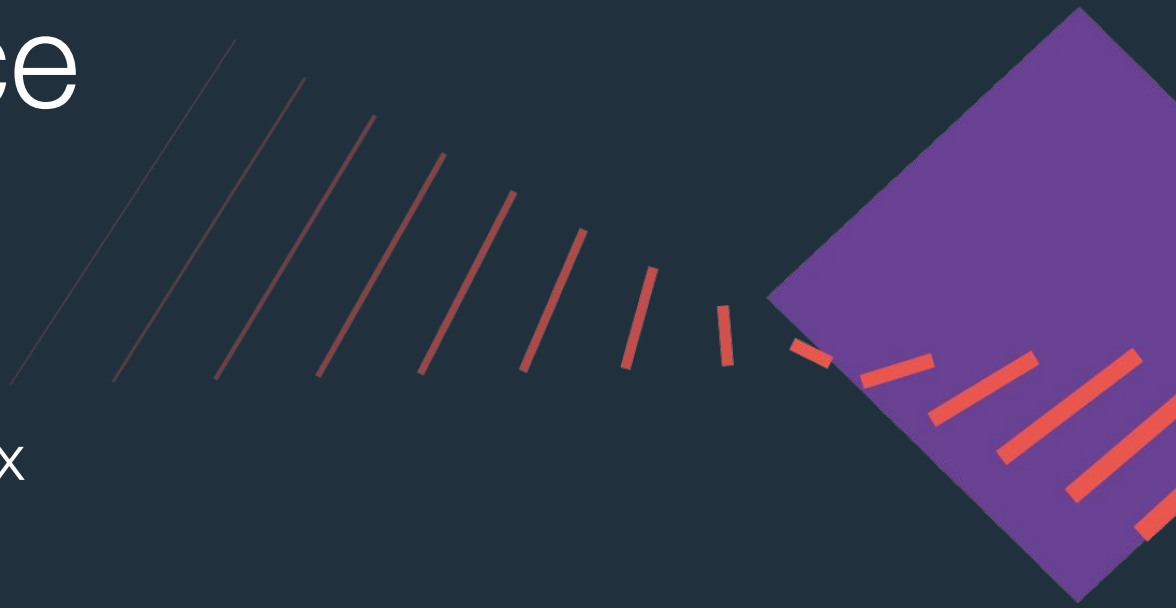
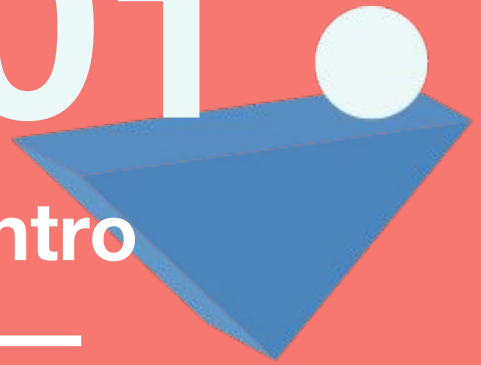
WiXEngineering

Speeding-up DL inference on CPU

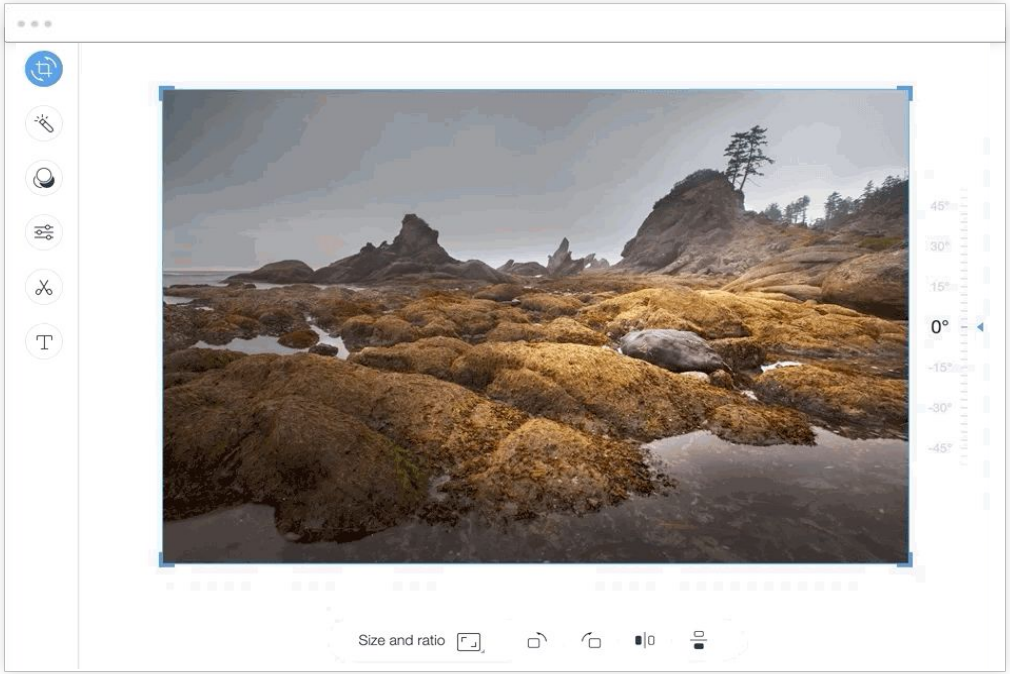
Oleksii Moskalenko, Wix

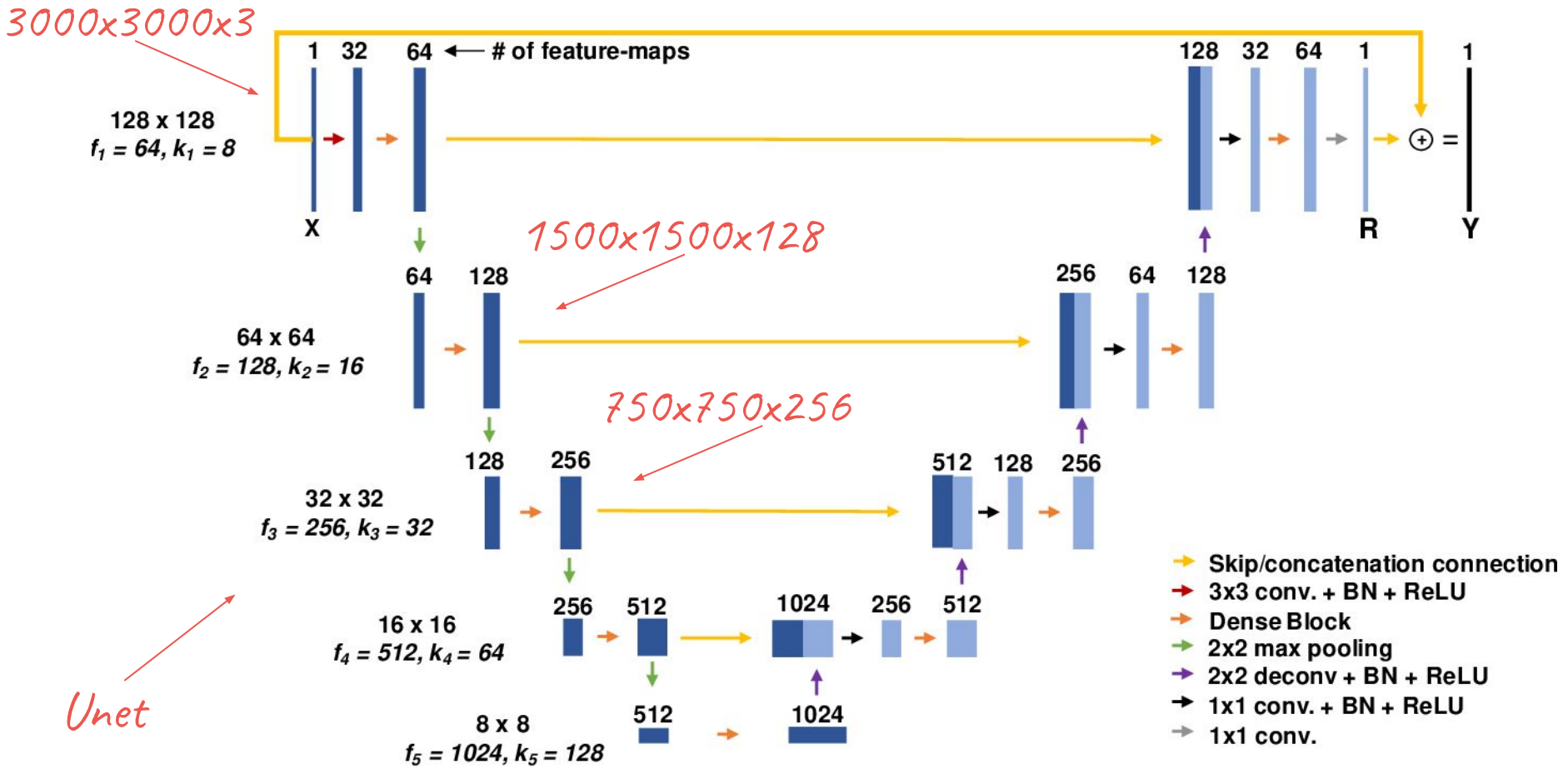


01



Intro


45°
30°
15°
0°
-15°
-30°
-45°
Size and ratio

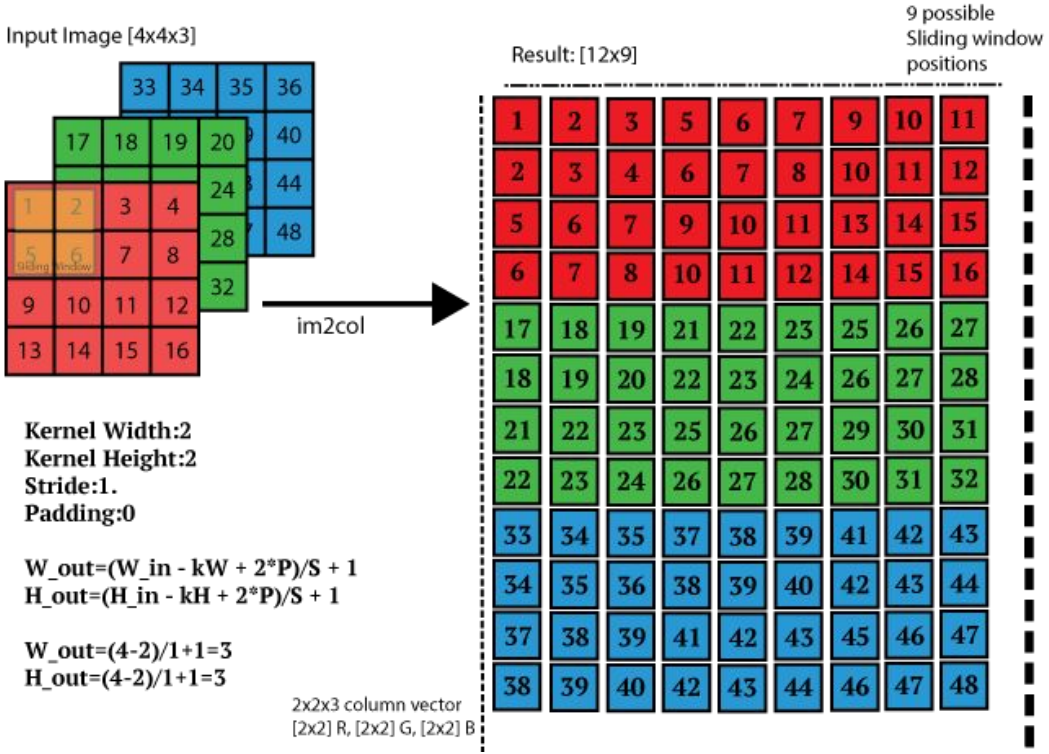

3000x3000x3
1 32 64 # of feature-maps
128 x 128
$f_1 = 64, k_1 = 8$
X
128 32 64 1
1
R
Y
1500x1500x128
64 128
64 x 64
$f_2 = 128, k_2 = 16$
256 64 128
750x750x256
128 256
32 x 32
$f_3 = 256, k_3 = 32$
512 128 256
256 512
16 x 16
$f_4 = 512, k_4 = 64$
1024 256 512
512
8 x 8
$f_5 = 1024, k_5 = 128$
1024
Unet
Skip/concatenation connection
3x3 conv. + BN + ReLU
Dense Block
2x2 max pooling
2x2 deconv + BN + ReLU
1x1 conv. + BN + ReLU
1x1 conv.

02

SIMD

```
[tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU
supports instructions that this TensorFlow binary was not
compiled to use: AVX2 FMA
```

Im2Col
Converting Conv to Matrix-Matrix multiplication



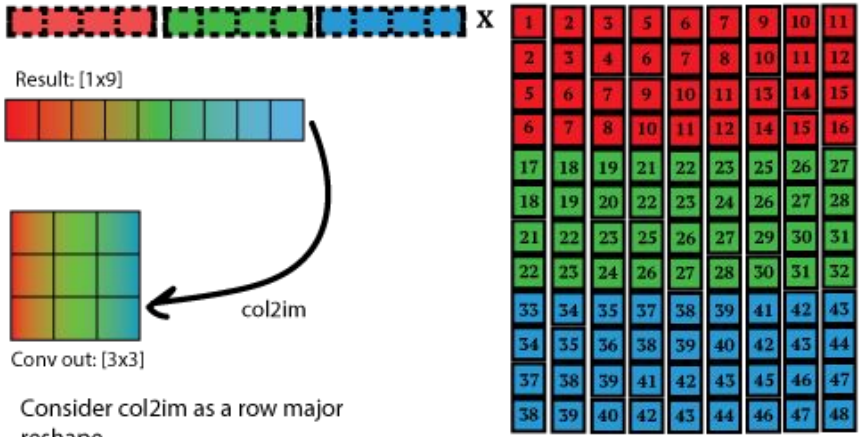
- M@M can be optimized with BLAS (Basic Linear Algebra Subprograms)
- But it will take $O(K^2 HWC)$ of additional memory

Tensor 1000x1000 with 128 channels, kernel 3x3 will need 4.5 Gb of memory

Image source: https://leonardoaraujosantos.gitbooks.io/

Im2Col
Converting Conv to Matrix-Matrix multiplication

We can multiply this result matrix [12x9]
with a kernel [1x12].
result = kernel x matrix
The result would be a row vector [1x9].
We need another operation that will convert
this row vector into a image [3x3].



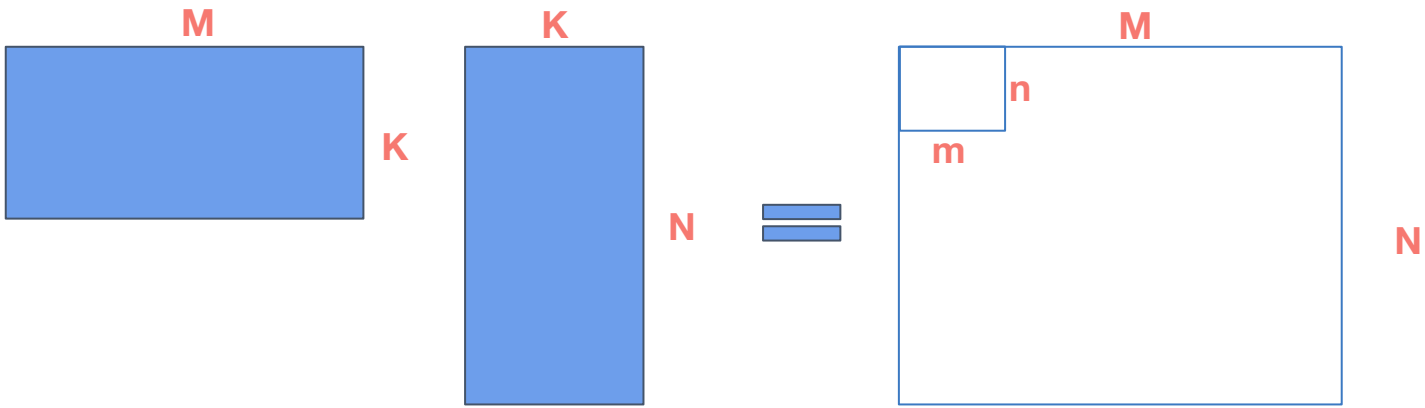
- M@M can be optimized with BLAS (Basic Linear Algebra Subprograms)
- But it will take $O(K^2 HWC)$ of additional memory

Tensor 1000x1000 with 128 channels, kernel 3x3 will need 4.5 Gb of memory

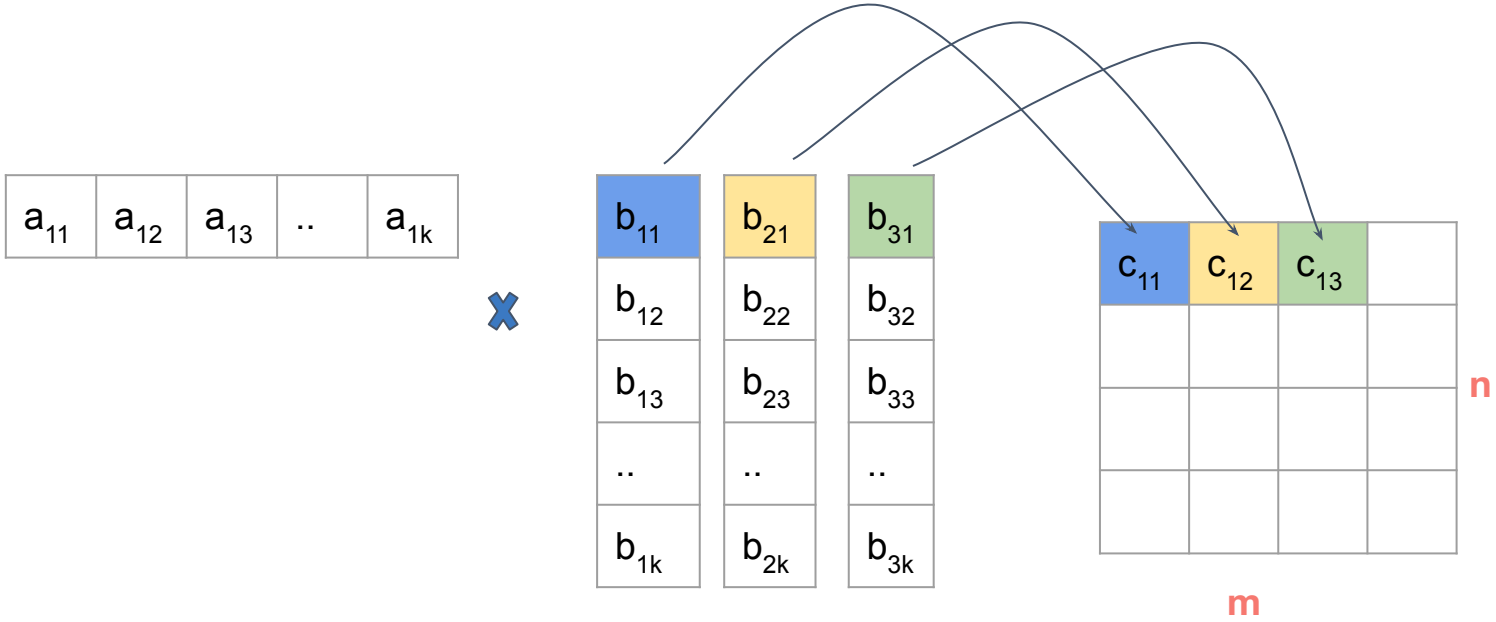
Image source: https://leonardoaraujosantos.gitbooks.io/

Matrix-matrix multiplication
Splitting loop in blocks

```
Loop over N:
   Loop over M:
      Loop over K:
          C[n, m] += A[n, k] * B[k, m]
```



Matrix-matrix multiplication
Parallelizing



Matrix-matrix multiplication
Parallelizing

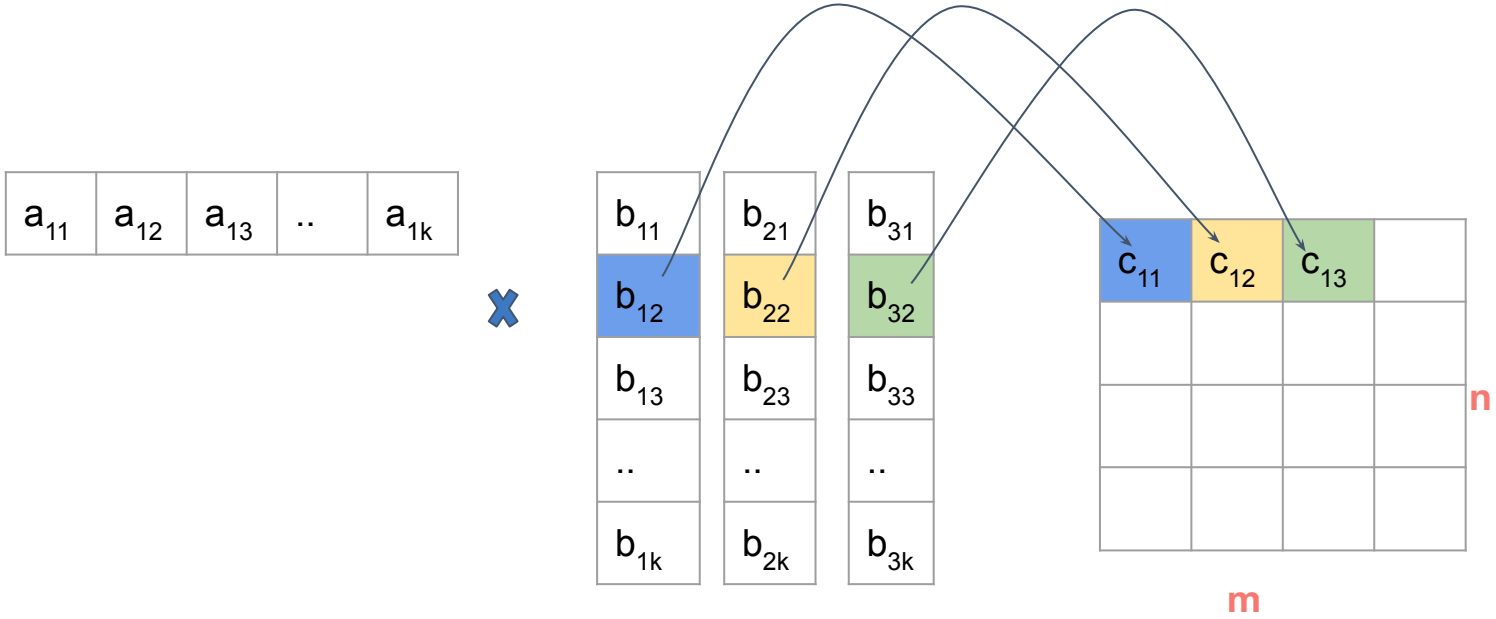


Image source: https://smileipic.github.io

Matrix-matrix multiplication
Parallelizing

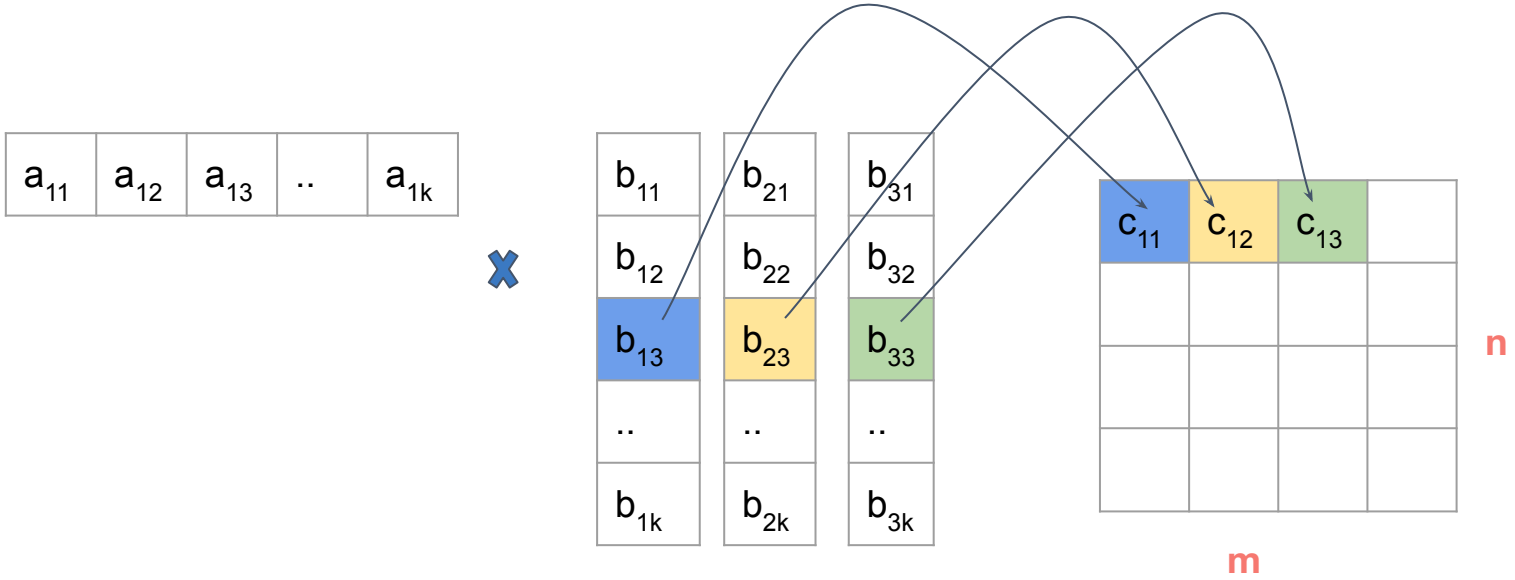
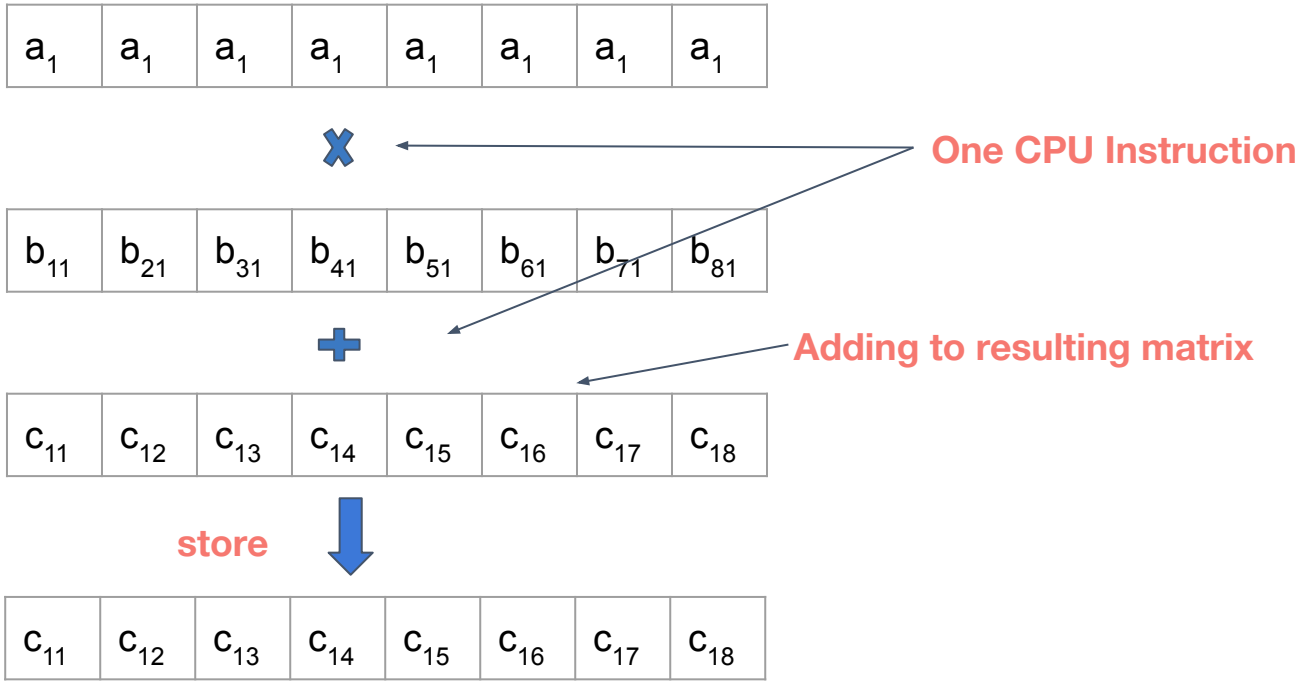


Image source: https://smileipic.github.io

Single Input Multiple Data

Vectorized Fused-Multiply-Add



SIMD Instructions
For M@M

```
Loop over N:
   Loop over M:
      Loop over K:
          C[n, m] += A[n, k] * B[k, m]
```

One CPU Instruction

```
Loop over N:
   Loop over blocks of M:
      Loop over K:
         C[n, m_block_start : m_block_end] += A[n, k] * B[k, m_block_start : m_block_end]
```

Vector Instructions Availability

AVX-512 (since Skylake or Knights Landing):

- 16 float32
- 64 int8
- 32 bfloat16

  (announced in Cooper Lake (Late 2019))

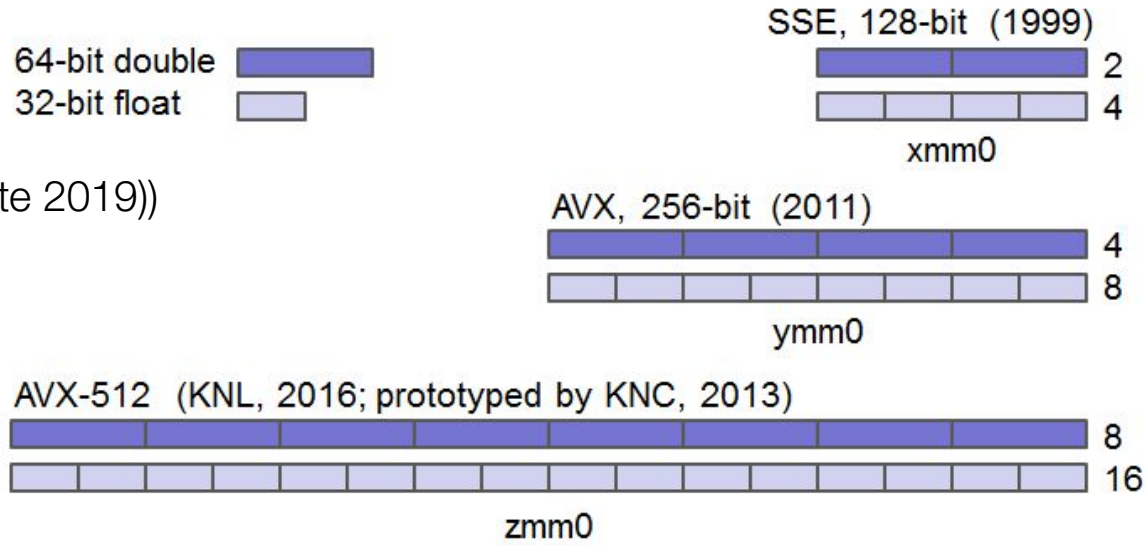


Image sources: Intel, Google

Benchmarking

Conv2D (One Layer)

```
# [1, 3000, 3000, 128] x [3, 3, 128, 128]
Conv2D (4608.00MB/13825.18MB, 4.53sec/9.03sec, 0us/0us, 4.53sec/9.03sec, 1/6|1/8)
```

```
# AVX512F (48 Threads/Cores)
Conv2D (4608.00MB/13825.18MB, 1.38sec/3.06sec, 0us/0us, 1.38sec/3.06sec, 1/6|1/8)
```

Full Model

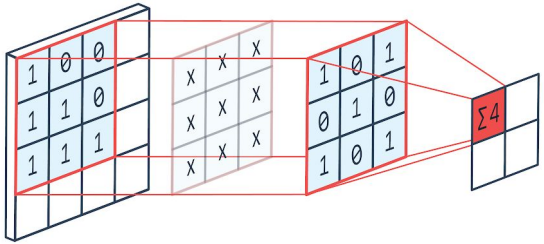
```
Mean 27.42sec (std 0.37sec)
```

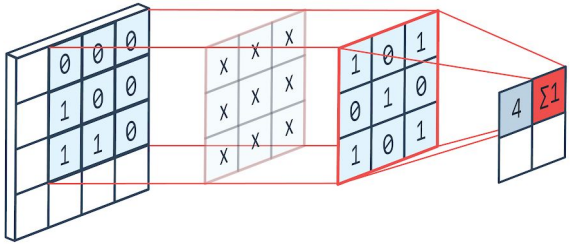
```
Mean 11.33sec (std 0.73sec)
```

03

Cache Locality

Looping over output tensor



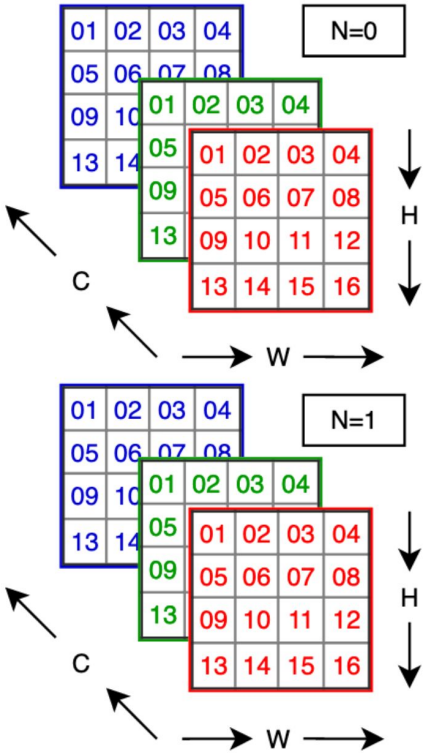


$$o[n, oh, ow, oc] = \sum_{kh}^{KH} \sum_{kw}^{KW} \sum_{ic}^{IC} i[n, oh * SH + kh, ow * SW + kw, ic] * w[kh, kw, ic, oc]$$

```
SH, SW <- strides

Loop over Batch (n)
    Loop over Output Height (oh)
        Loop over Output Width (ow)
            Loop over Output Channels (oc)
                Loop over Kernel Height (kh)
                    Loop over Kernel Width (kw)
                        Loop over Input Channels (ic)
                            o[n, oh, ow, oc] += i[n, oh * SH + kh, ow * SW + kw, ic] * w[kh, kw, ic, oc]
```

Memory Layout



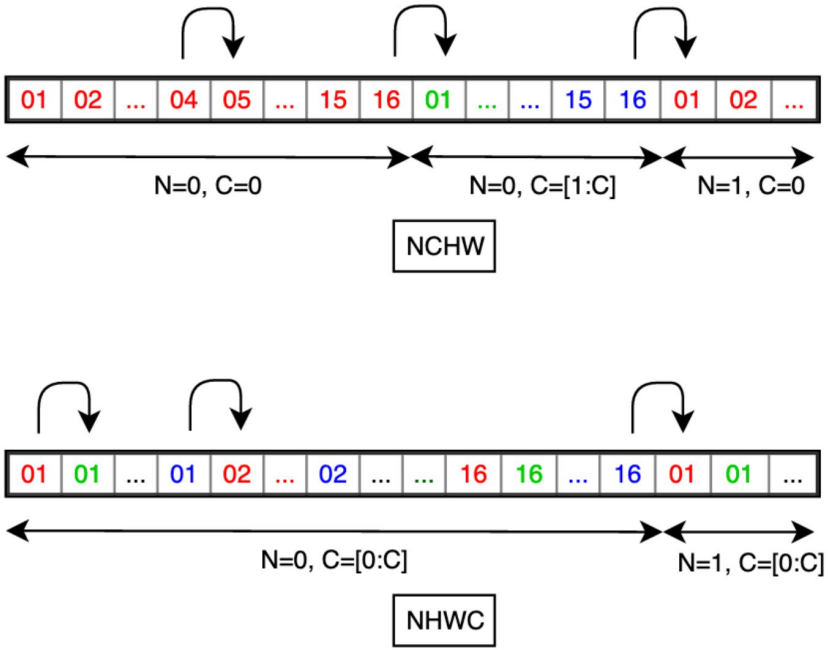
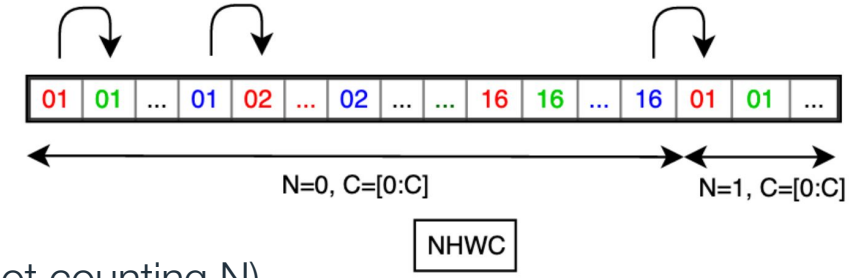


Image source: Efficient Winograd or Cook-Toom Convolution Kernel Implementation on Widely Used Mobile CPUs

Memory Layout

With NHWC layout to calculate 1 output cell

And Kernel 3x3



- You need to load 2 full rows + 3 cells

  since rows (H) is the outermost dimension (not counting N)

- E.g. for tensor 1000 x 1000 and 128 input channels you'll read

  (2 x 1000 + 3) x 128 x *sizeof (float)* ~ 1 Mb of memory

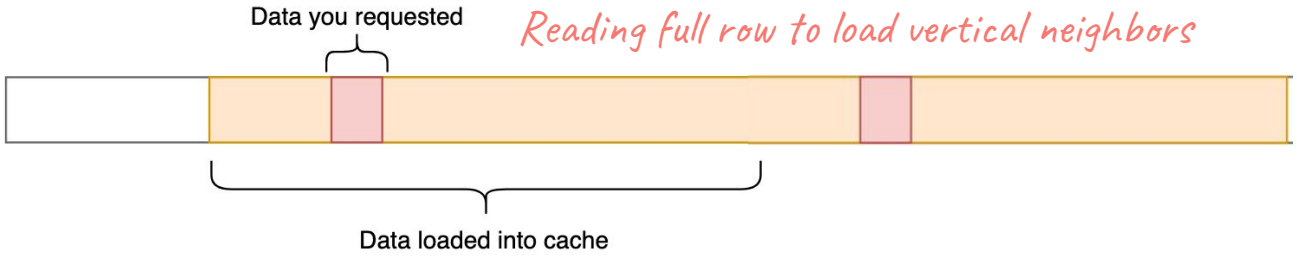
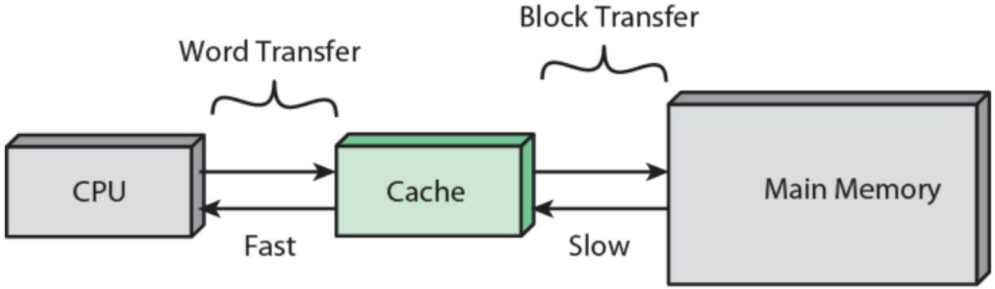
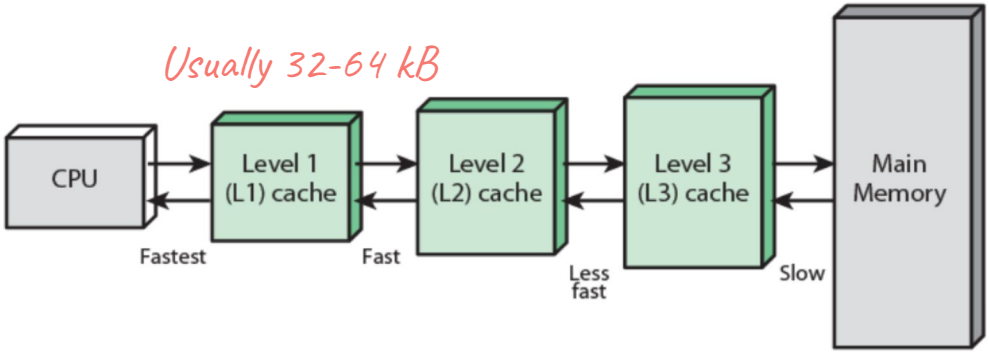
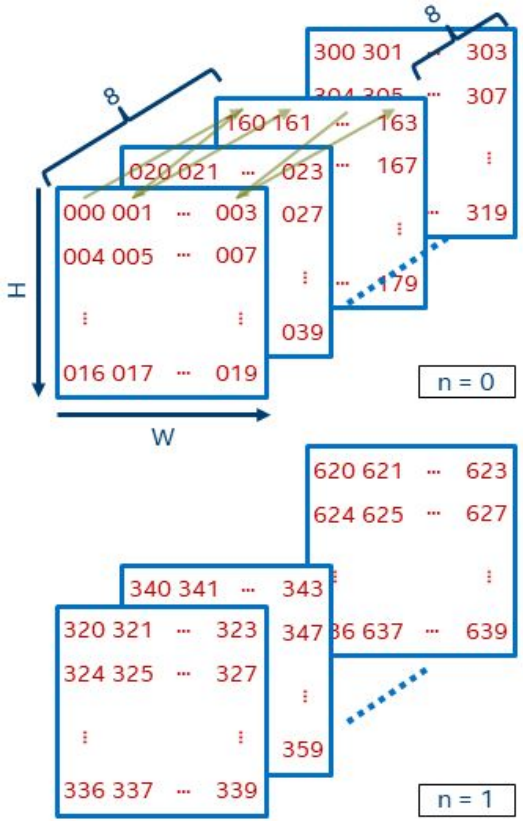


Image source: https://sahnimanas.github.io

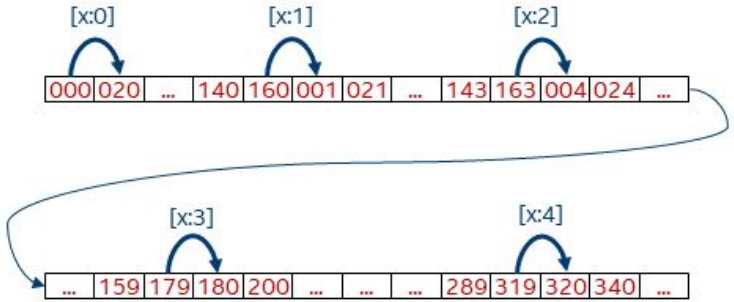
CPU Cache Levels







Physical data layout
nChw8c layout



Input tensor 16 x 1000 x 1000 x 8 and kernel 3x3
*To Load (2 full rows + 3) * 8 * size of(float) ~ 64kB*

Image source: https://intel.github.io

Vectorizing Loop

```
Loop over Output Channels (oc)
    Loop over Kernel Height (kh)
        Loop over Kernel Width (kw)
            Loop over Input Channels (ic)
                o[n, oh, ow, oc] += i[n, oh * SH + kh, ow * SW + kw, ic] * w[kh, kw, ic, oc]
```

```
Loop over Blocks of Output Channels (oc_block)
    ....
    o[..., oc_block] += i[n, oh * SH + kh, ow * SW + kw, ic] * w[..., oc_block]
```

Input Loaded Once

Vectorizing by Output Channels

```
o[n, OC, oh, ow, oc_block] += i[n, IC, oh * SH + kh, ow * SW + kw, ic] * w[kh, kw, ic, oc_block]
```

NCHWc Layout

- Implemented in Intel MKL-DNN (but not only there)
- Available as Tensorflow extension (tensorflow-MKL)
- Automatically converts NHWC & NCHW (before & after)

  (no conversion between MKL operators)

```
# [1, 3000, 3000, 128] x [3, 3, 128, 128]
# AVX512F (48 Threads/Cores)
Conv2D (4608.00MB/13825.18MB, 1.38sec/3.06sec, 0us/0us, 1.38sec/3.06sec, 1/6|1/8)
```

```
# MKL-DNN
Conv2D (4608.59MB/13826.36MB, 607.44ms/1.97sec, 0us/0us, 607.44ms/1.97sec, 1/6|1/8)
```

MKL-DNN: Benchmarking

Full Model

```
# [1, 3000, 3000, 3] -> [1, 6000, 6000, 3]
Mean 11.33sec (std 0.73sec)
```

```
# MKL-DNN: direct
Mean 4.73sec (std 0.32sec)
```

MKL Drawbacks

- Not all operators are supported
- TF decides to fallback to common operators
- Additional conversion layers between TF and MKL are being added


```
mkldnn_verbose,exec,reorder,jit:uni,undef,in:f32_nhwc out:f32_nChw16c,num:1,1x128x3000x3000,311.929
mkldnn_verbose,exec,convolution,jit:avx512_common,forward,fsrc:nChw16c fwei:OIhw16i16o fdst:nChw16c,
alg:convolution_direct,mb1_ic128oc128_ih3000oh3000kh3sh1dh0ph1_iw3000ow3000kw3sw1dw0pw1,618.786
```

```
# MKL-DNN
 Conv2D (4608.59MB/9217.18MB, 1.20sec/1.37sec, 0us/0us, 1.20sec/1.37sec, 1/4|1/6)
```

04

Winograd

Winograd

- Approximation of Convolution (like FFT but for small kernels)
- Based on Chinese remainder theorem
- (Unexpectedly) implemented in many libraries (long ago):

  MKL-DNN, cuDNN, TF (since 1.0)
- But limited to only 3x3 kernels with stride 1

  (and sometimes not parallelized -> useless)

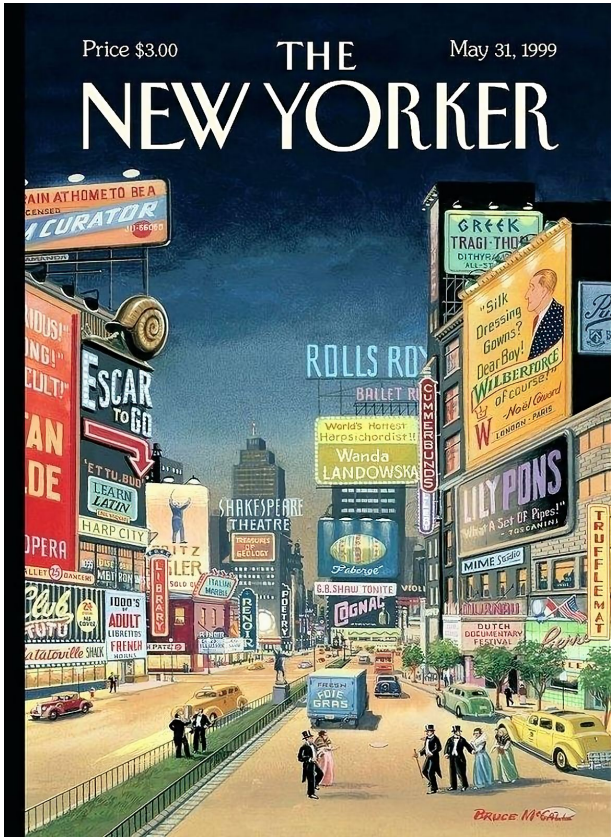
Benchmarking

```
# [16, 1000, 1000, 128] x [3, 3, 128, 128]
# MKL-DNN: direct
Conv2D (8192.59MB/24578.36MB, 1.65sec/4.71sec, 0us/0us, 1.65sec/4.71sec, 1/6|1/8)
```

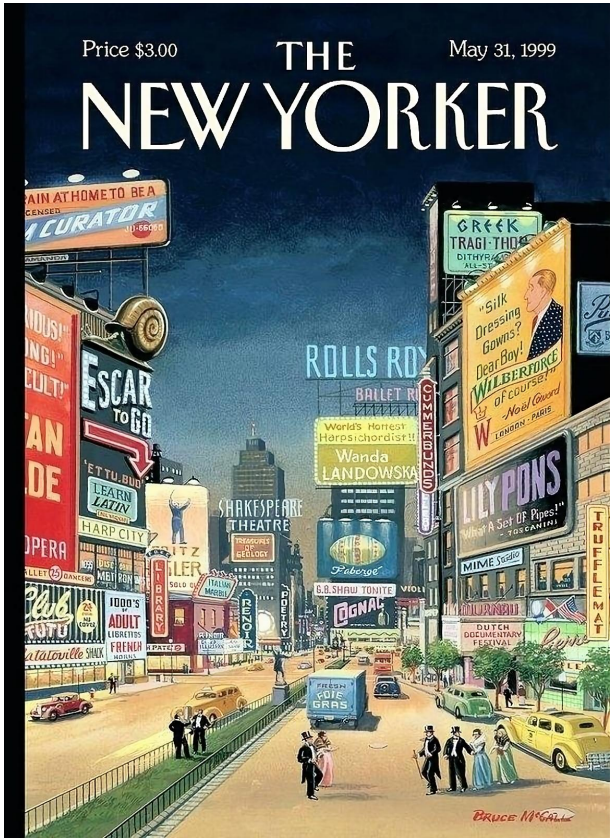
```
# MKL-DNN: winograd
# 30 Threads/Cores
Conv2D_1 (8192.59MB/24578.36MB, 1.09sec/5.36sec, 0us/0us, 1.09sec/5.36sec, 1/6|1/8)
```

Approximation error

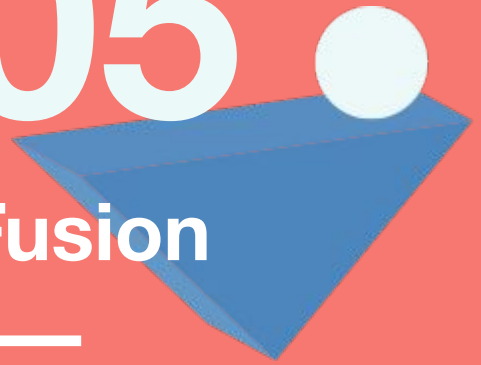
Regular



Winograd



05



Fusion

Memory Management

- Tensorflow dynamically allocates memory (for output tensor) before running operation and deallocates (for input) after it (every time)
- That could lead to huge allocation/deallocation overhead (if you have many simple operation working with big tensors in a row)
- We can get rid of overhead by fusing several operations into one

Solutions

- Fuse operations - is the main selling feature of TF XLA

  (however it really works only for GPU)
- Alternatively there are many external tools (similar to XLA) that convert

  graph to IR that is being compiled on LLVM.

  IR operations could be fused on compile time.

  (IR: Relay/Diesel/Tiramisu/Glow)

All-in-One: TVM

- TVM is compiler for DL graphs
- Takes graph (TF, PyTorch, ONNX, etc). Produces byte code.
- It uses low-level IR (Relay) and gives ability to schedule low-level primitives (memory layout, parallelization pattern, etc)
- Includes auto-tuning for choosing best scheduling for target hardware
- Many targets are supported: CPU, ARM, GPU

All-in-One: TVM

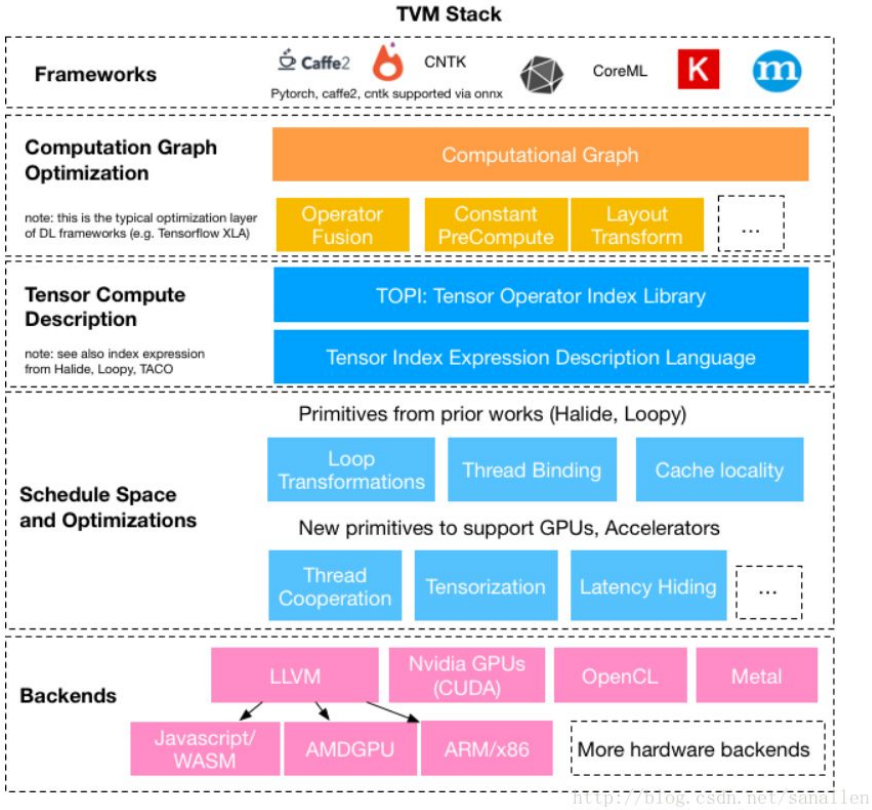


Image source: https://tvm.ai

TVM
Scheduling example

```
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute((M, N),
            lambda x, y: tvm.sum(A[x, k] * B[k, y], axis=k),
            name='C')
s = tvm.create_schedule(C.op)
```

Vectorization

```
bn = 16

# Creating inner blocks
xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)

s[C].vectorize(yi)
```

TVM: Relay

```
def @main(%input: Tensor[(1, 3, 2800, 3000), float32], %conv1_1/kernel: Tensor[(3, 3, 3, 32), float32], ...)
-> Tensor[(1, 3, 2800, 3000), float32] {

  %0 = nn.pad(%input, pad_width=[[0, 0], [0, 0], [1, 1], [1, 1]]) /*ty=Tensor[(1, 3, 2802, 3002), float32]*/;
  %1 = transpose(%conv1_1/kernel, axes=[3, 2, 0, 1])            /* ty=Tensor[(32, 3, 3, 3), float32] */;
  %2 = nn.conv2d(%0, %1, channels=32, kernel_size=[3, 3]) /* ty=Tensor[(1, 32, 2800, 3000), float32] */;
  %3 = reshape(%conv1_1/bias, newshape=[1, -1, 1, 1])           /* ty=Tensor[(1, 32, 1, 1), float32] */;
  %4 = add(%2, %3)                                          /* ty=Tensor[(1, 32, 2800, 3000), float32] */;
  %5 = nn.batch_norm(%4, %batch_normalization/gamma, %batch_normalization/beta,
%batch_normalization/moving_mean, %batch_normalization/moving_variance, epsilon=0.001f)
  %6 = %5.0;
  %7 = nn.leaky_relu(%6, alpha=0.2f)                          /* ty=Tensor[(1, 32, 2800, 3000), float32] */;
  %8 = nn.pad(%7, pad_width=[[0, 0], [0, 0], [1, 1], [1, 1]]) /* ty=Tensor[(1, 32, 2802, 3002), float32] */;
  %9 = transpose(%conv1_2/kernel, axes=[3, 2, 0, 1])               /* ty=Tensor[(32, 32, 3, 3), float32] */;
  %10 = nn.conv2d(%8, %9, channels=32, kernel_size=[3, 3])    /* ty=Tensor[(1, 32, 2800, 3000), float32] */;
  %11 = reshape(%conv1_2/bias, newshape=[1, -1, 1, 1])             /* ty=Tensor[(1, 32, 1, 1), float32] */;
  %12 = add(%10, %11)                                         /* ty=Tensor[(1, 32, 2800, 3000), float32] */;
...
```

TVM: Fusion Example

```
Op #0 fused_nn_pad_layout_transform: 1910.83 us/iter
Op #1 fused_nn_contrib_conv2d_NCHWc_add_add_nn_leaky_relu_7: 19310.8 us/iter
Op #2 fused_layout_transform_nn_pad_layout_transform_3: 25283.4 us/iter
Op #3 fused_nn_contrib_conv2d_NCHWc_add_add_nn_leaky_relu: 53126.1 us/iter
Op #4 fused_nn_max_pool2d_1: 13441.2 us/iter
Op #5 fused_layout_transform_nn_pad_layout_transform_7: 6772.94 us/iter
Op #6 fused_nn_contrib_conv2d_NCHWc_add_add_nn_leaky_relu_6: 16677.4 us/iter
Op #7 fused_layout_transform_nn_pad_layout_transform_4: 11776.5 us/iter
Op #8 fused_nn_contrib_conv2d_NCHWc_add_add_nn_leaky_relu_2: 24983.6 us/iter
```

TVM: Benchmarking

Full Model

```
# [1, 3000, 3000, 3] -> [1, 6000, 6000, 3]
# MKL-DNN: direct
Mean 4.73sec (std 0.32sec)
```

```
# TVM Compiled
# 48 Threads/Cores
Mean 2.48sec (std 0.184sec)
```

TVM: Auto-tuning

- Scheduling hyper parameters (block sizes, registers, unrolling, etc) can be optimized with ML
- Among available tuners: GridSearch, XGBoost, Genetic Algorithm
- Graph optimizations (eg choosing what to fuse / what to not) are also can be tuned

TVM: Drawbacks

- Fixed size Tensors
- RNN support is limited
- Compex graph support is limited

  Switch, Merge, etc create problems. Always prepare graph for inference (or better, have two different graphs)

Full Story

```
# [1, 3000, 3000, 3] -> [1, 6000, 6000, 3]
Mean 27.42sec (std 0.37sec)
```

```
# AVX512
Mean 11.33sec (std 0.73sec)
```

```
# MKL-DNN: direct
Mean 4.73sec (std 0.32sec)
```

```
# TVM Compiled
Mean 2.48sec (std 0.184sec)
```

Papers

1. Optimizing CNN Model Inference on CPUs, Liu et al., 2019
2. Fast Algorithms for Convolutional Neural Networks, Lavin & Gray, 2015
3. Distributed Deep Learning Using Synchronous Stochastic Gradient Descent, Das et al., 2016
4. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning, Chen et al., 2018

THANK YOU