

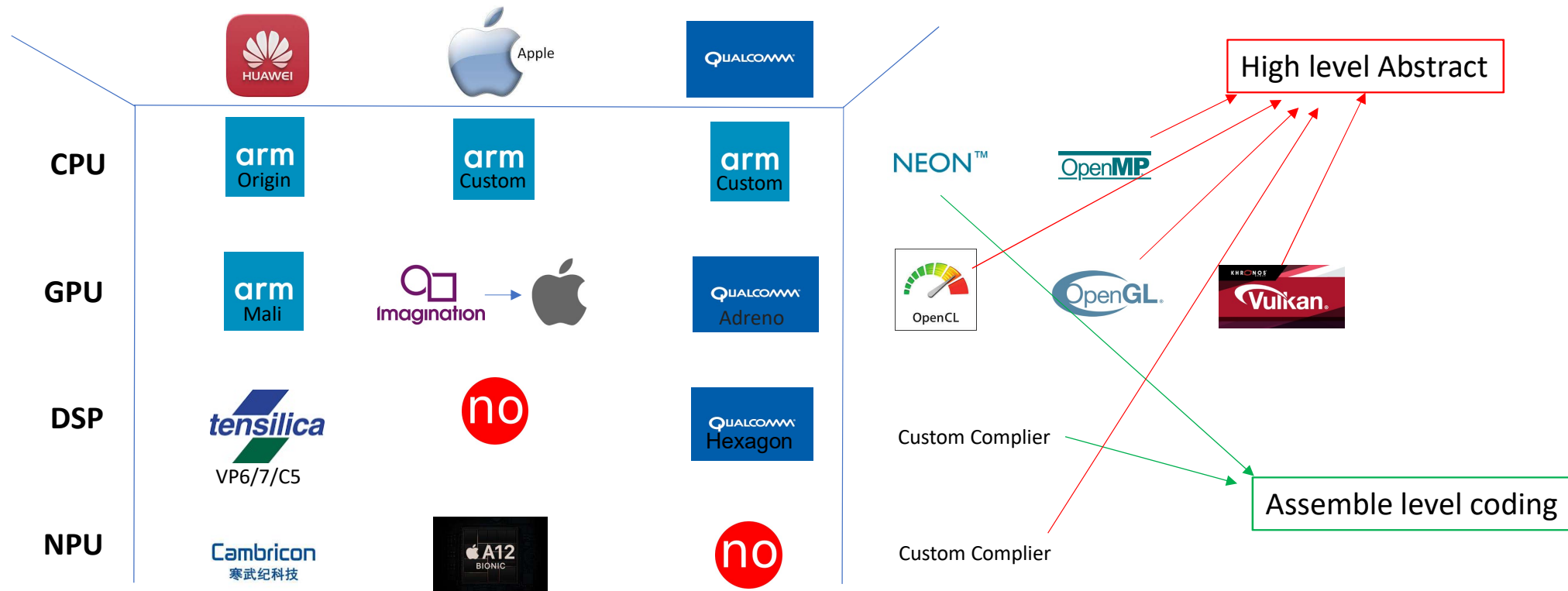
# Deep learning Computing on ARM devices

4/28/2019

# Introduction

- Hand-coding Computing libraries
- Generation optimization direction
- Auto-coding stack: Halide/TVM

# Embedded devices overview



Generally, the peak performance of CPU:GPU:DSP = 1:1:1; power consumption of CPU:GPU:DSP = 4:2:1  
 CPU is the most flexible (suit third party app dev), GPU and DSP is more efficient(suit for vendor app)

# DNN libraries(1): arm computing library

<https://github.com/ARM-software/ComputeLibrary>



## ➤ Function:

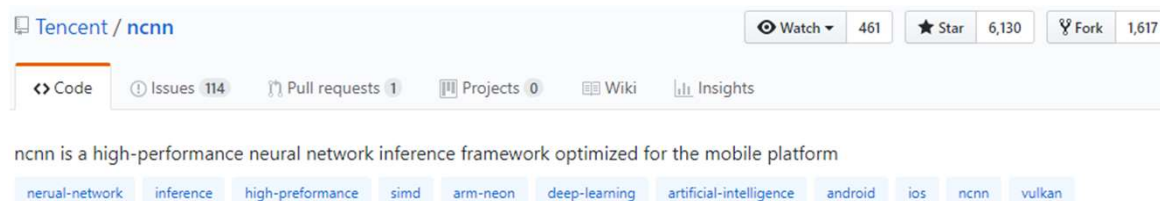
- Basic arithmetic, mathematical, and binary operator functions
- Color manipulation (conversion, channel extraction, and more)
- Convolution filters (Sobel, Gaussian, and more)
- Canny Edge, Harris corners, optical flow, and more
- Pyramids (such as Laplacians)
- HOG (Histogram of Oriented Gradients)
- SVM (Support Vector Machines)
- H/SGEMM (Half and Single precision General Matrix Multiply)
- Convolutional Neural Networks building blocks (Activation, Convolution, Fully connected, Locally connected, Normalization, Pooling, Soft-max)mathematical, and binary operator functions

## ➤ Comment:

- Most flexible computing library, support almost all known operators
- FP32/FP16 support for both CPU NEON and GPU OpenCL
- Three kinds of implementation for conv
  - Im2col + gemm
  - Imcol + Winograd gemm
  - Direct conv
- Well document/code style
- ❖ Performance seems not good, everyone claims beaten of it

# DNN libraries(2): Tencent NCNN

<https://github.com/Tencent/ncnn>



## ➤ Function:

- Classical CNN: VGG AlexNet GoogleNet Inception ...
- Practical CNN: ResNet DenseNet SENet FPN ...
- Light-weight CNN: SqueezeNet MobileNetV1/V2 ShuffleNetV1/V2 MNasNet ...
- Detection: MTCNN facedetection ...
- Detection: VGG-SSD MobileNet-SSD SqueezeNet-SSD MobileNetV2-SSDLite ...
- Detection: Faster-RCNN R-FCN ...
- Detection: YOLOV2 YOLOV3 MobileNet-YOLOV3 ...
- Segmentation: FCN PSPNet ...

## ➤ Comment:

- Computing library + framework
  - support most common used operators
  - FP32/FP16 support for both CPU NEON and GPU vulkan
  - Fix point computing / 8 bit quantization
- Multiple platforms: x86/arm/ios/android/linux/windows
- Memory-efficient / Production level code
  - QQ, Qzone, WeChat, Pitu and so on.
- Support to import models from caffe/pytorch/mxnet/onnx
- No third party libraries dependent
- Custom layer supported
- ❖ Good reputation in the beginning, but stuck in large amount of burden

# DNN libraries(3): Tencent FeatherCNN

<https://github.com/Tencent/FeatherCNN>



## ➤ Function:

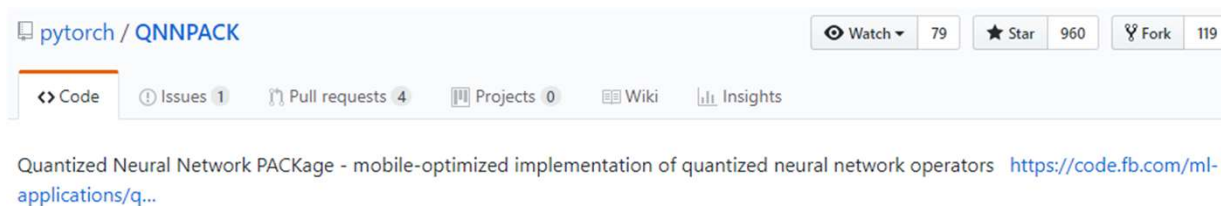
- Less support than NCNN
- Only CPU support
- Origins from our game AI project for King of Glory (Chinese: 王者荣耀)

## ➤ Comment:

- ❖ Seems to be a result of horse racing mechanism inner Tencent

# DNN libraries(4): Facebook QNNPACK

<https://github.com/pytorch/QNNPACK>



## ➤ Function:

- Mainly for computing on quantized 8-bit tensors.
- QNNPACK is integrated in pytorch1.0 with Caffe2 graph representation

- ☒ 2D Convolution
- ☒ 2D Deconvolution
- ☒ Channel Shuffle
- ☒ Fully Connected
- ☐ Locally Connected
- ☒ 2D Max Pooling
- ☒ 2D Average Pooling
- ☒ Global Average Pooling
- ☒ Sigmoid
- ☒ Leaky ReLU
- ☒ Clamp (can be used for ReLU, ReLU6 if it is not fused in another operator)
- ☒ SoftArgMax (aka SoftMax)
- ☐ Group Normalization

## ➤ Comment:

- Android/IOS
- CPU only
- Highly support with the integration of Pytorch
- 8-bit tensors oriented
- FP16/Fixed16 on the way

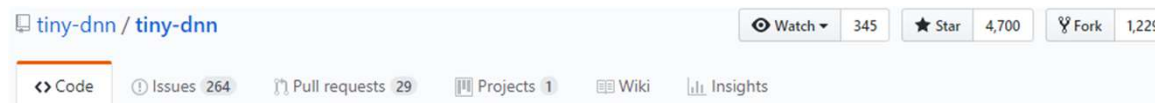
# DNN libraries(5): others

## 1. <https://github.com/NervanaSystems/neon>



Intel® Nervana™ reference deep learning framework committed to best performance on all hardware

## 2. <https://github.com/tiny-dnn/tiny-dnn>



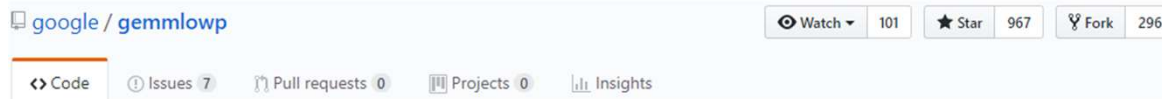
header only, dependency-free deep learning framework in C++14 <http://tiny-dnn.readthedocs.io>

## 3. <https://github.com/OAID/Tengine>



Tengine is a lite, high performance, modular inference engine for embedded device

## 4. <https://github.com/google/gemmlowp>



Low-precision matrix multiplication

## 5. OpenBlas / Eigen

A startup bought by Intel

Two authors totally, give up maintain

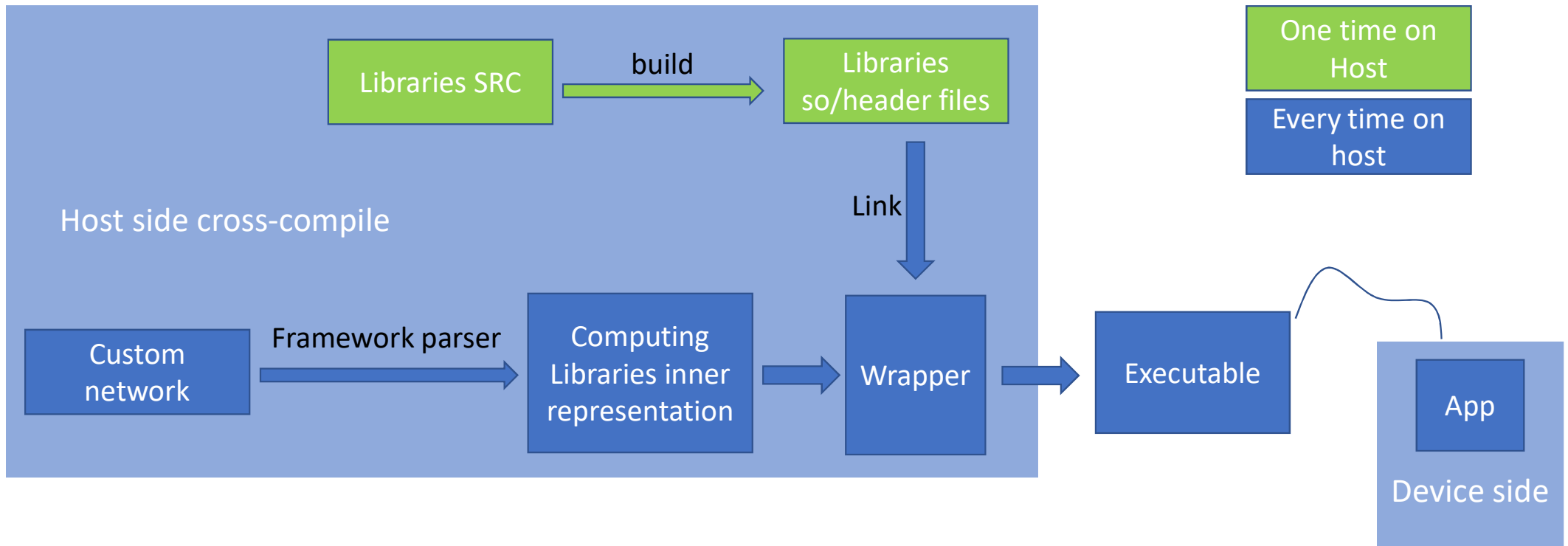
By Open AI lab

Only Gemm(INT8/Short16/FP16/FP32; CPU only)

Origin as PC libraries but also compatible with ARM platform (CPU only)

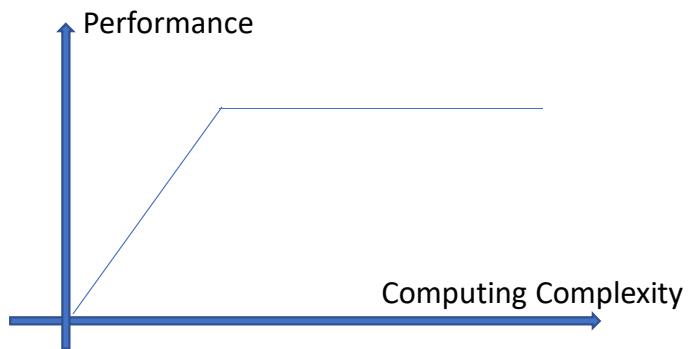


# How to use

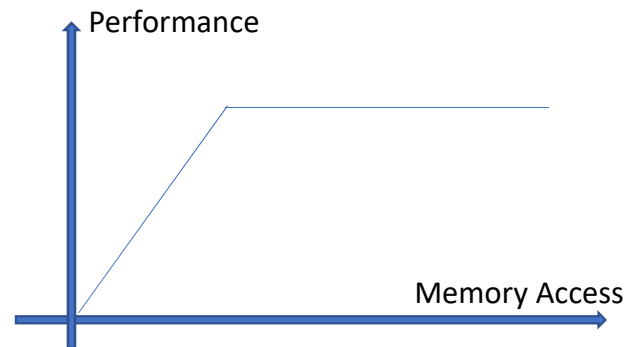


# Generation optimization direction

- Overall speaking, a program is bounded by
  - Memory
  - Computing
- Trick to estimate the gap
  - memcpy benchmark to get insight of memory bandwidth
  - math function benchmark to get computing ability



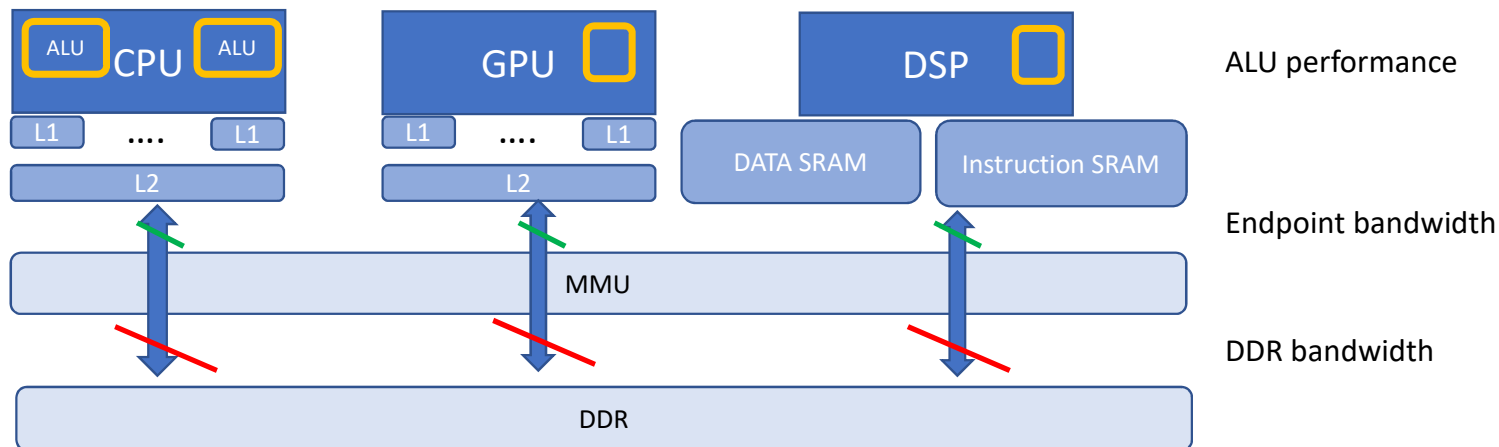
(1) Computing Bound Program



(2) Memory Bound Program

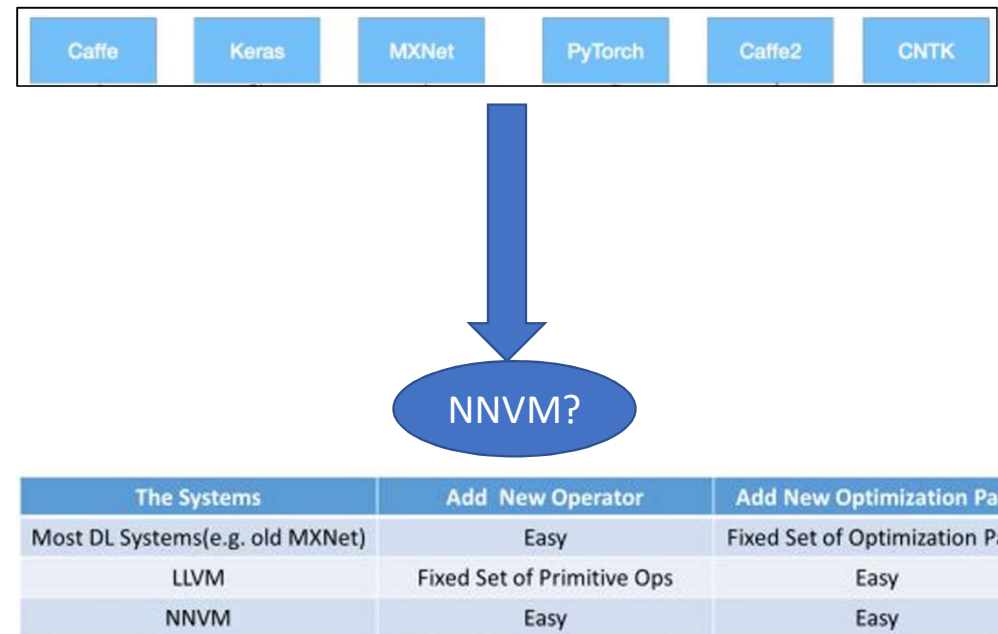
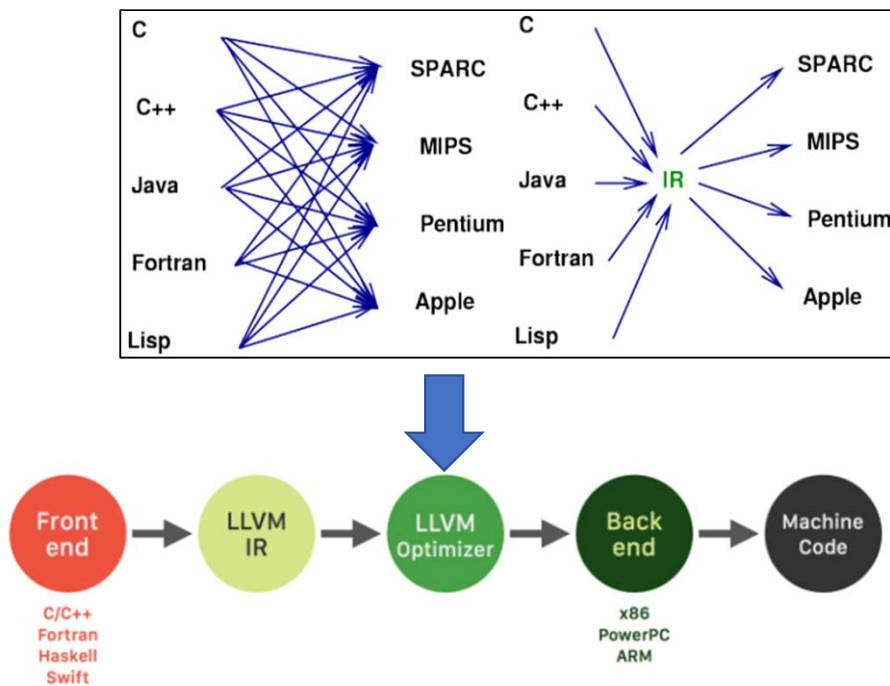
# Kernel optimization

- Tricks:
  - Performance comparison(Apple2Apple)
    - Clean system, same frequency, result accurate
  - Get basic understand of the system
  - If memory bound, consider:
    - Layer fusion / quantization / weight share / data reuse
    - Cache hint rate
  - If computing bound:
    - Reduce algorithms complexity
    - ALU frequency/ ALU pipeline



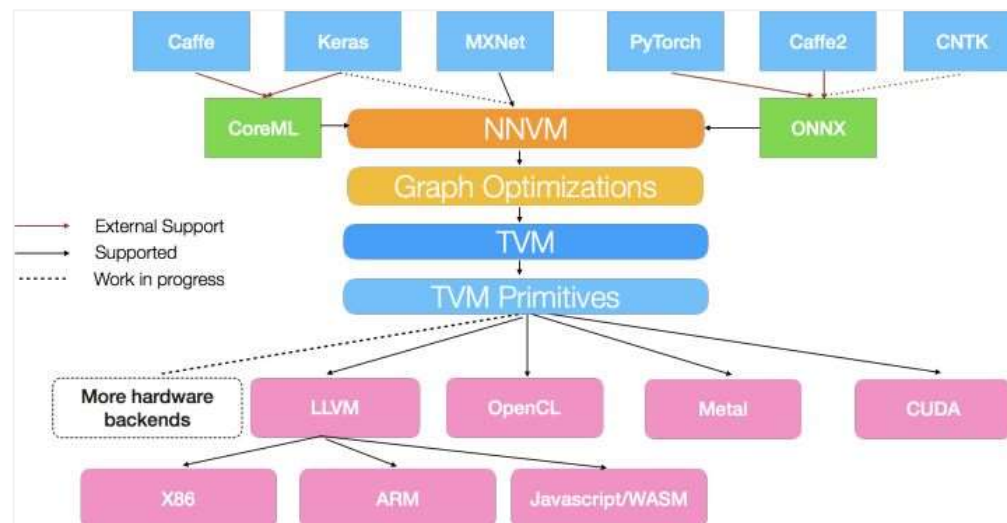
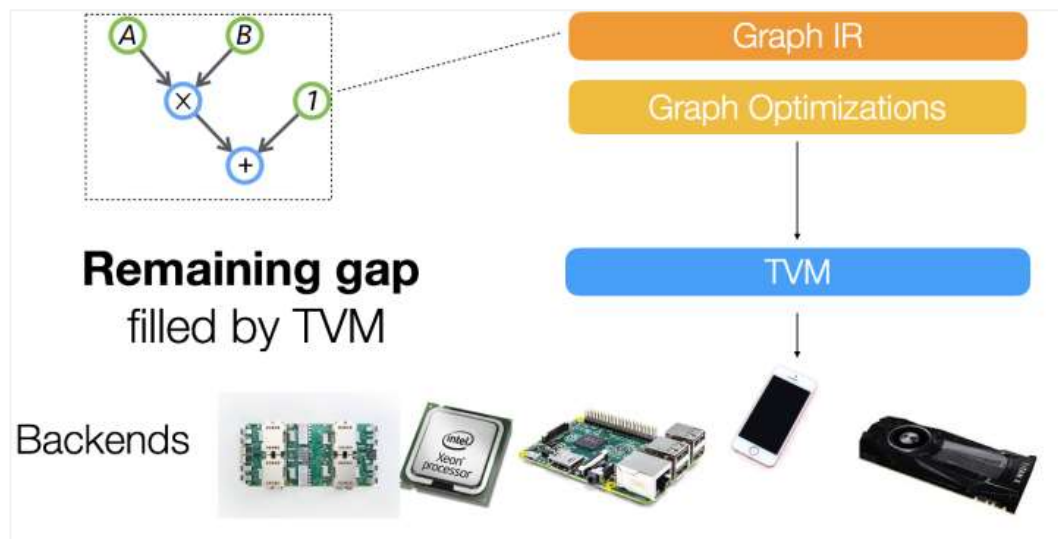
# Auto-coding stack

- Terminology:
  - LLVM
  - NNVM / TVM
  - IR
  - Halide



# TVM/LLVM

- NNVM:
  - Just another framework?
- TVM:
  - Second level IR
  - Code Generator & Feedback



# TVM

- Code Generator:
  - Separation of Data flow and Execution flow

```
for (int i = 0; i < n; ++i) {  
    C[i] = A[i] + B[i];  
}
```

```
n = tvm.var("n")  
A = tvm.placeholder((n,), name='A')  
B = tvm.placeholder((n,), name='B')  
C = tvm.compute(A.shape, lambda i: A[i] + B[i], name="C")
```

```
if tgt == "cuda" or tgt.startswith('opencl'):  
    s[C].bind(bx, tvm.thread_axis("blockIdx.x"))  
    s[C].bind(tx, tvm.thread_axis("threadIdx.x"))
```

```
fadd = tvm.build(s, [A, B, C], tgt, target_host=tgt_host, name="myadd")
```

```
ctx = tvm.context(tgt, 0)  
  
n = 1024  
a = tvm.nd.array(np.random.uniform(size=n).astype(A.dtype), ctx)  
b = tvm.nd.array(np.random.uniform(size=n).astype(B.dtype), ctx)  
c = tvm.nd.array(np.zeros(n, dtype=C.dtype), ctx)  
fadd(a, b, c)  
tvm.testing.assert_allclose(c.asnumpy(), a.asnumpy() + b.asnumpy())
```

```
if tgt == "cuda" or tgt.startswith('opencl'):  
    dev_module = fadd.imported_modules[0]  
    print("-----GPU code-----")  
    print(dev_module.get_source())  
else:  
    print(fadd.get_source())
```

```
for (int bx = 0; bx < ceil(n / 64); ++bx) {  
    for (int tx = 0; tx < 64; ++tx) {  
        int i = bx * 64 + tx;  
        if (i < n) {  
            C[i] = A[i] + B[i];  
        }  
    }  
}
```

```
bx, tx = s[C].split(C.op.axis[0], factor=64)
```

# TVM

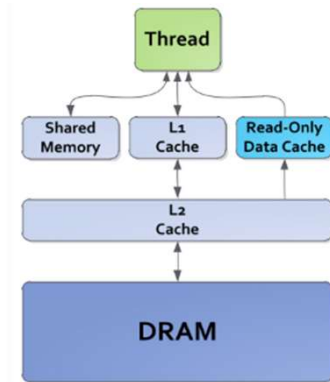
## • Code Generator (GEMM)

```
# The sizes of inputs and filters
batch = 256
in_channel = 256
out_channel = 512
in_size = 14
kernel = 3
pad = 1
stride = 1

# Algorithm
A = tvm.placeholder((in_size, in_size, in_channel, batch), name='A')
W = tvm.placeholder((kernel, kernel, in_channel, out_channel), name='W')
out_size = (in_size - kernel + 2*pad) // stride + 1
# Pad input
Apad = tvm.compute(
    (in_size + 2*pad, in_size + 2*pad, in_channel, batch),
    lambda yy, xx, cc, nn: tvm.if_then_else(
        tvm.all(yy >= pad, yy - pad < in_size,
                xx >= pad, xx - pad < in_size),
        A[yy - pad, xx - pad, cc, nn], tvm.const(0., "float32")),
    name='Apad')

# Create reduction variables
rc = tvm.reduce_axis((0, in_channel), name='rc')
ry = tvm.reduce_axis((0, kernel), name='ry')
rx = tvm.reduce_axis((0, kernel), name='rx')

# Compute the convolution
B = tvm.compute(
    (out_size, out_size, out_channel, batch),
    lambda yy, xx, ff, nn: tvm.sum(
        Apad[yy * stride + ry, xx * stride + rx, rc, nn] * W[ry, rx, rc, ff],
        axis=[ry, rx, rc]),
    name='B')
```



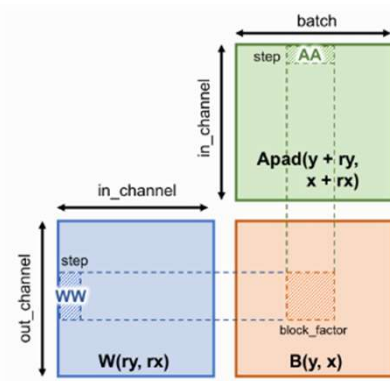
```
# Designate the memory hierarchy
s = tvm.create_schedule(B.op)
s[Apad].compute_inline() # compute Apad inline
AA = s.cache_read(Apad, 'shared', [B])
WW = s.cache_read(W, "shared", [B])
AL = s.cache_read(AA, "local", [B])
WL = s.cache_read(WW, "local", [B])
BL = s.cache_write(B, "local")
```

```
# tile consts
tile = 8
num_thread = 8
block_factor = tile * num_thread
step = 8
vthread = 2
```

```
# Get the GPU thread indices
block_x = tvm.thread_axis("blockIdx.x")
block_y = tvm.thread_axis("blockIdx.y")
block_z = tvm.thread_axis("blockIdx.z")
thread_x = tvm.thread_axis((0, num_thread), "threadIdx.x")
thread_y = tvm.thread_axis((0, num_thread), "threadIdx.y")
thread_xz = tvm.thread_axis((0, vthread), "vthread", name="vx")
thread_yz = tvm.thread_axis((0, vthread), "vthread", name="vy")
```

```
# Split the workloads
hi, wi, fi, ni = s[B].op.axis
bz = s[B].fuse(hi, wi)
by, fi = s[B].split(fi, factor=block_factor)
bx, ni = s[B].split(ni, factor=block_factor)
```

```
# Bind the iteration variables to GPU thread indices
s[B].bind(bz, block_z)
s[B].bind(by, block_y)
s[B].bind(bx, block_x)
```



Q & A