

ncnnAccel

Software 篇

参考资料

nihui 的知乎：

(32 封私信 / 82 条消息) nihui - 知乎 (zhihu.com)

demo 汇总：zchrissirhc/z/awesome-ncnn: □ A Collection of Awesome NCNN-based Projects
(github.com)

全志 D1 哪吒开发板运行 NCNN：

在 RISC-V 架构的全志 D1 「哪吒」开发板上，跑个 ncnn 神经网络推理框架的 demo_哔哩哔哩_bilibili

NCNN 安装和交叉编译

https://blog.csdn.net/weixin_41232202/article/details/109150047

模型导出与处理

<https://zhuanlan.zhihu.com/p/391788686>

NCNN+Int8+YOLOv4 量化模型和实时推理

NCNN+Int8+YOLOv4 量化模型和实时推理 - 知乎 (zhihu.com)

YOLOX-NCNN-INT8 量化实践记录

<https://zhuanlan.zhihu.com/p/399329939>

INT8 量化 wiki

<https://github.com/Tencent/ncnn/wiki/quantized-int8-inference>

ncnn model zoo

<https://github.com/nihui/ncnn-assets/tree/master/models>

非常好的一个 NCNN 资源仓库

<https://github.com/Ewenwan/MVision/tree/master/CNN/HighPerformanceComputing/example>

NCNN Breakdown

https://www.zhihu.com/column/c_1327298414569115648

编译和安装

<https://github.com/Ewenwan/MVision/blob/master/CNN/HighPerformanceComputing/example/readme.md>

Ubuntu

很简单 略

VS2019

https://blog.csdn.net/qq_40231159/article/details/111808792

<https://zhuanlan.zhihu.com/p/391609325>

https://blog.csdn.net/Skies_/article/details/109318667

基准测试

==> cd ncnn/build/benchmark ./benchncnn

```

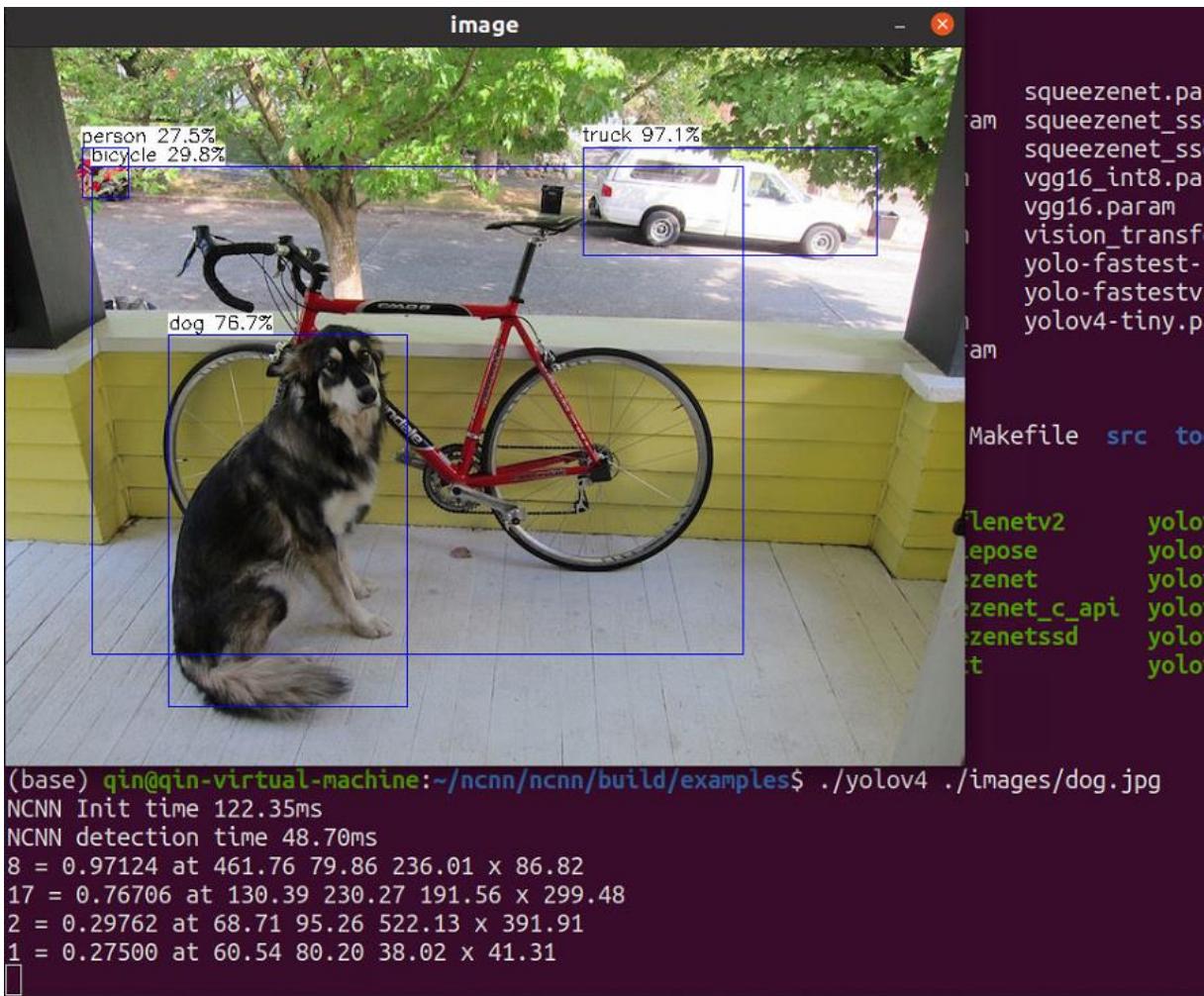
loop_count = 4
num_threads = 8
powersave = 0
gpu_device = -1
cooling_down = 1
    squeezeenet min = 3.72 max = 3.97 avg = 3.81
    squeezeenet_int8 min = 6.08 max = 6.27 avg = 6.16
        mobilenet min = 5.34 max = 5.75 avg = 5.53
        mobilenet_int8 min = 6.28 max = 7.11 avg = 6.54
        mobilenet_v2 min = 5.08 max = 5.77 avg = 5.37
        mobilenet_v3 min = 3.83 max = 4.04 avg = 3.92
        shufflenet min = 2.83 max = 3.16 avg = 3.00
        shufflenet_v2 min = 2.96 max = 3.19 avg = 3.08
        mnasnet min = 3.97 max = 4.05 avg = 4.02
    proxylessnasnet min = 4.34 max = 4.69 avg = 4.58
    efficientnet_b0 min = 8.08 max = 8.85 avg = 8.42
    efficientnetv2_b0 min = 9.06 max = 9.38 avg = 9.28
        regnet_y_400m min = 8.07 max = 8.67 avg = 8.43
        blazeface min = 0.75 max = 0.95 avg = 0.84
        googlenet min = 14.98 max = 15.78 avg = 15.35
        googlenet_int8 min = 22.07 max = 22.63 avg = 22.39
        resnet18 min = 14.09 max = 17.85 avg = 15.35
        resnet18_int8 min = 18.27 max = 19.43 avg = 18.65
        alexnet min = 11.25 max = 11.56 avg = 11.45
        vgg16 min = 68.50 max = 69.89 avg = 69.14
        vgg16_int8 min = 81.35 max = 83.31 avg = 82.14
        resnet50 min = 28.73 max = 29.39 avg = 29.04
        resnet50_int8 min = 38.49 max = 40.23 avg = 39.61
    squeezeenet_ssd min = 20.33 max = 25.18 avg = 21.90
    squeezeenet_ssd_int8 min = 16.90 max = 17.05 avg = 16.98
        mobilenet_ssd min = 11.41 max = 11.89 avg = 11.61
        mobilenet_ssd_int8 min = 13.52 max = 16.36 avg = 14.43
        mobilenet_yolo min = 41.54 max = 43.32 avg = 42.40
    mobilenetv2_yolov3 min = 17.79 max = 18.68 avg = 18.23
        yolov4-tiny min = 27.73 max = 32.61 avg = 29.09
        nanodet_m min = 6.87 max = 7.10 avg = 7.01
    yolo-fastest-1.1 min = 2.90 max = 3.62 avg = 3.22
    yolo-fastestv2 min = 2.93 max = 3.78 avg = 3.47
vision_transformer min = 315.51 max = 321.62 avg = 318.53
    FastestDet min = 3.39 max = 3.82 avg = 3.67
(base) qin@qin-virtual-machine:~/ncnn/ncnn/build/benchmark$ █

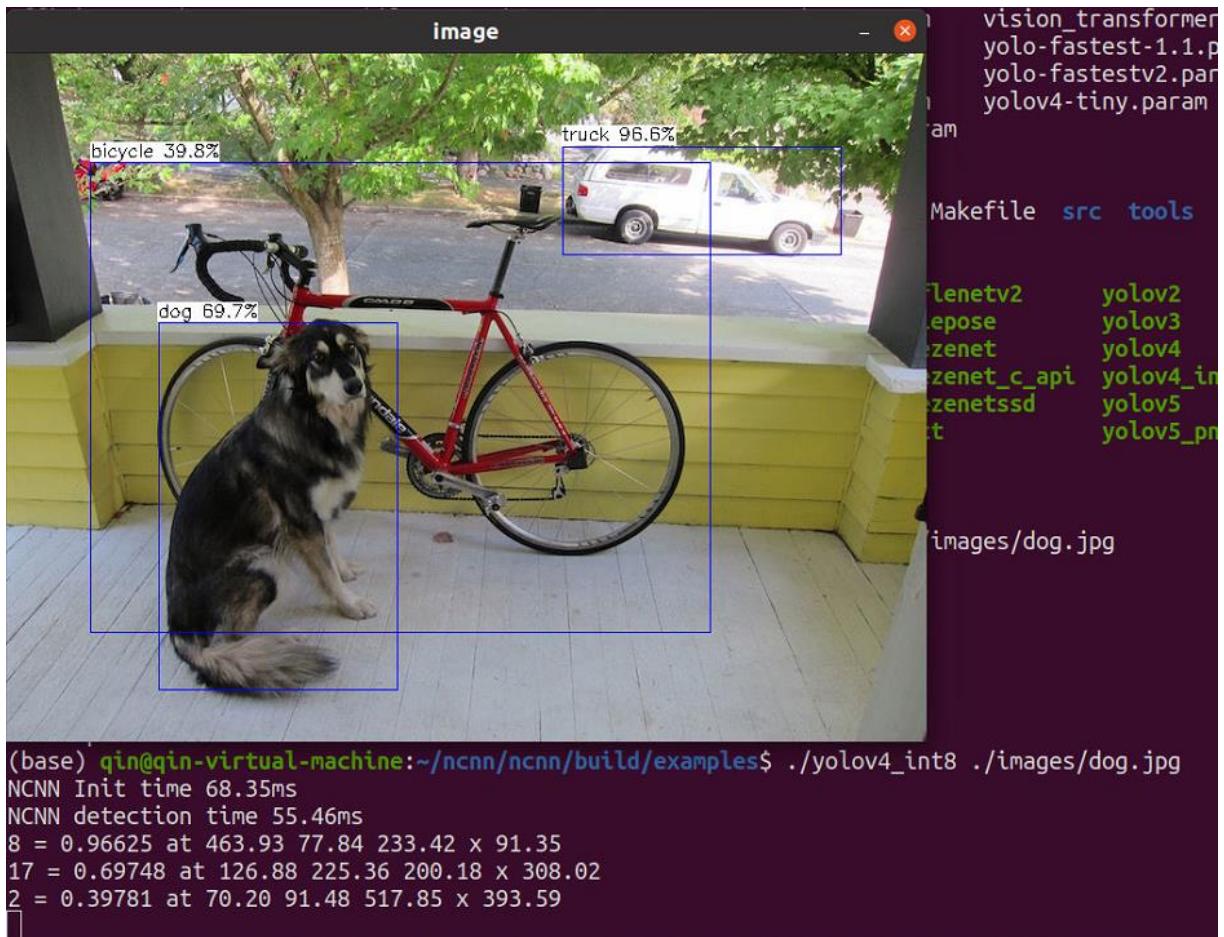
```

从基准测试的结果上看，int8量化之后反而推理更慢了，==>平台的问题~

Yolov4

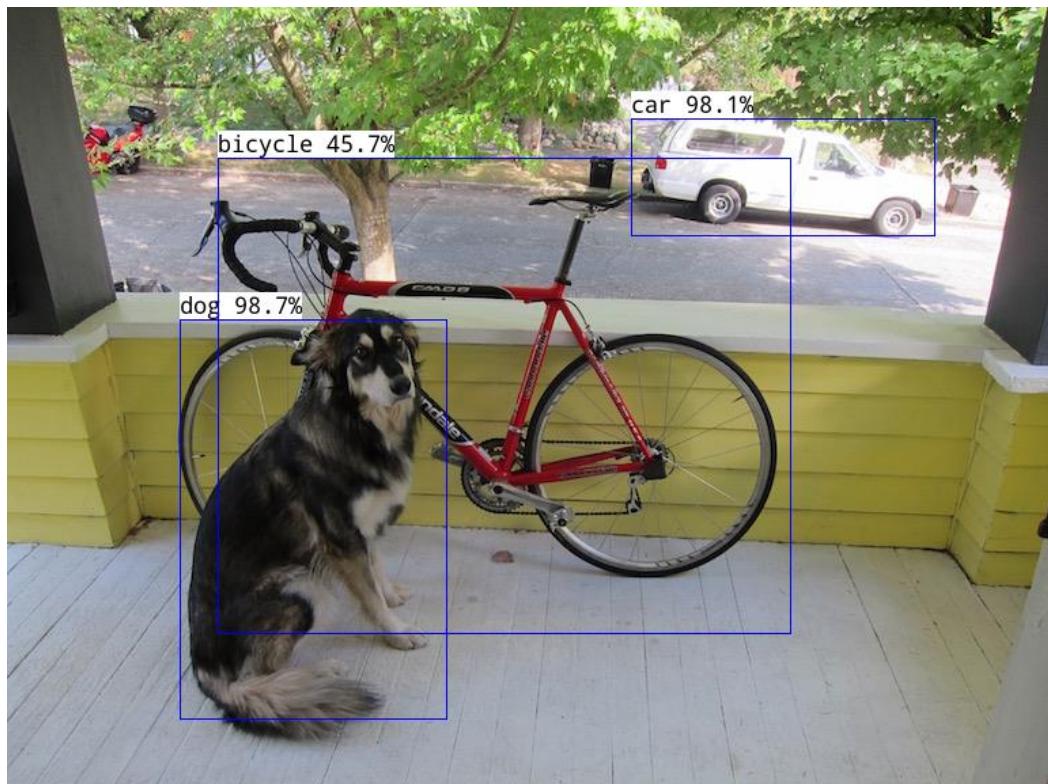
==> cd ncnn/build/examples ./yolov4 etc.





采用 int8 量化之后效果还不错~

mobilnetv2-yolov3



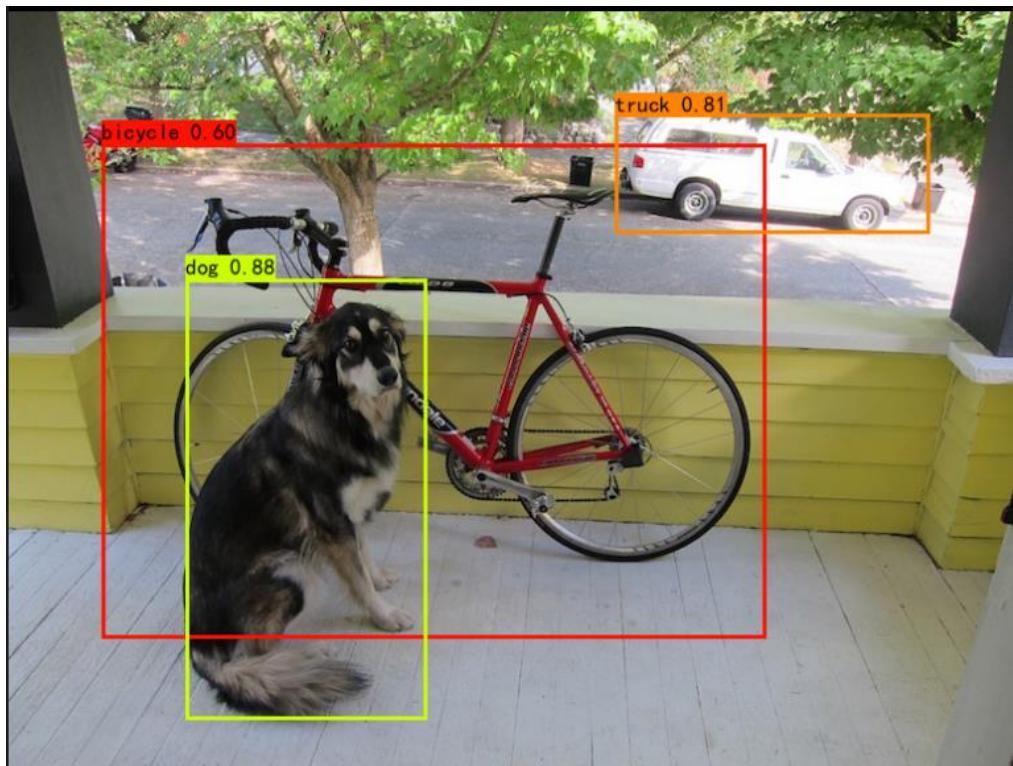
全流程 CPU 端 --- YOLOv4_TINY --COCO80

1、模型训练和检测：

良心 UP：Pytorch 搭建自己的 YoloV4-tiny 目标检测平台（Bubbliliing 深度学习 教程）_哔哩哔哩_bilibili

源码：

<https://github.com/bubbliliing/yolov4-tiny-pytorch>



2、在 pytorch 中转换模型到.onnx

```
torch.onnx.export(self.net,
                  im,
                  f           = model_path,
                  verbose     = False,
                  opset_version = 12,
                  training     = torch.onnx.TrainingMode.EVAL,
                  do_constant_folding = True,
                  input_names   = input_layer_names,
                  output_names  = output_layer_names,
                  dynamic_axes  = None)
```

3、simplify onnx model

```
(py38) qin@qin-virtual-machine:~/ncnn/ncnn/models/yolov4_tiny$ python -m onnxsim coco80.onnx coco80_sim.onnx
Simplifying...
Finish! Here is the difference:
```

	Original Model	Simplified Model
Concat	7	7
Constant	1	0
Conv	21	21
LeakyRelu	19	19
MaxPool	3	3
Resize	1	1
Split	3	3
Model Size	23.1MiB	23.1MiB

使用 netron *.onnx 查看网路结构

4、ONNX2NCNN

```
(py38) qin@qin-virtual-machine:~/ncnn/ncnn/build/tools/onnx$ ./onnx2ncnn ../../../../models/yolov4_tiny/coco80_sim.onnx ../../../../models/yolov4_tiny/coco80.param ../../../../models/yolov4_tiny/coco80.bin  
(py38) qin@qin-virtual-machine:~/ncnn/ncnn/build/tools/onnx$
```

5、ncnnoptimize

```
(py38) qin@qin-virtual-machine:~/ncnn/ncnn/build/tools$ ./ncnnoptimize ../../models/yolov4_tiny/coco80.param ../../models/yolov4_tiny/coco80.bin ../../models/yolov4_tiny/coco80_opt.param ../../models/yolov4_tiny/coco80_opt.bin 0  
fuse_convolution_activation Conv_0 LeakyRelu_1  
fuse_convolution_activation Conv_2 LeakyRelu_3  
fuse_convolution_activation Conv_4 LeakyRelu_5  
fuse_convolution_activation Conv_7 LeakyRelu_8  
fuse_convolution_activation Conv_9 LeakyRelu_10  
fuse_convolution_activation Conv_12 LeakyRelu_13  
fuse_convolution_activation Conv_16 LeakyRelu_17  
fuse_convolution_activation Conv_19 LeakyRelu_20  
fuse_convolution_activation Conv_21 LeakyRelu_22  
fuse_convolution_activation Conv_24 LeakyRelu_25  
fuse_convolution_activation Conv_28 LeakyRelu_29  
fuse_convolution_activation Conv_31 LeakyRelu_32  
fuse_convolution_activation Conv_33 LeakyRelu_34  
fuse_convolution_activation Conv_36 LeakyRelu_37  
fuse_convolution_activation Conv_40 LeakyRelu_41  
fuse_convolution_activation Conv_42 LeakyRelu_43  
fuse_convolution_activation Conv_44 LeakyRelu_45  
fuse_convolution_activation Conv_47 LeakyRelu_48  
fuse_convolution_activation Conv_52 LeakyRelu_53  
Input layer images without shape info, shape_inference skipped  
Input layer images without shape info, estimate_memory_footprint skipped  
(py38) qin@qin-virtual-machine:~/ncnn/ncnn/build/tools$
```

可以发现tools目录下存在ncnnoptimize的可执行文件

接着输入命令

```
1 | ./ncnnoptimize ncnn.param ncnn.bin new.param new.bin flag
```

注意这里的flag指的是fp32和fp16，其中0指的是fp32，1指的是fp16

可以利用大老师的转换工具 onnx 转 ncnn

<https://convertmodel.com/>

6、经过优化之后的网络结构总结

ERROR

坑1：NCNN 找不到 OPENCV 的话，设置一下 ON -> OFF

```
option(NCNN_SIMPLEOCV "minimal opencv structure emulation" OFF)
```

如何自定义层

<https://github.com/Ewenwan/MVision/blob/master/CNN/HighPerformanceComputing/example/ncnn-%E6%96%B0%E5%BB%BA%E5%B1%82.md>

源码解读

https://blog.csdn.net/just_sort/article/details/111403398

<https://github.com/Tencent/ncnn/wiki/param-and-model-file-structure>

参数加载

`load_param("*.param")`

<https://github.com/Tencent/ncnn/wiki/operation-param-weight-table>

<https://github.com/Tencent/ncnn/wiki/param-and-model-file-structure>

.param: 描述神经网络的结构，包括层名称，层输入输出信息、层参数信息（如卷积层的 kernel 的大小等）

详细描述

`==> https://polariszhao.github.io/2020/09/21/ncnn%E5%89%8D%E5%90%91%E8%AE%A1%E7%AE%97%E6%B5%81%E7%A8%8B%E6%B5%85%E6%9E%90/`

```
1 7767517
2 75 83
3 Input          data          0 1 data 0=227 1=227 2=3
4 Convolution    conv1         1 1 data conv1 0=64 1=3 2=1 3=2 4=0 5=1 6=1728
5 ReLU           relu_conv1   1 1 conv1 conv1_relu_conv1 0=0.000000
6 Pooling        pool1        1 1 conv1_relu_conv1 pool1 0=0 1=3 2=2 3=0 4=0
7 Convolution    fire2/squeeze1x1 1 1 pool1 fire2/squeeze1x1 0=16 1=1 2=1 3=1 4=0 5=1 6=1024
8 ReLU           fire2/relu_squeeze1x1 1 1 fire2/squeeze1x1 fire2/squeeze1x1_fire2/relu_squeee
9 Split          splitncnn_0   1 2 fire2/squeeze1x1_fire2/relu_squeeze1x1 fire2/squeeze1x1
10 Convolution   fire2/expand1x1 1 1 fire2/squeeze1x1_fire2/relu_squeeze1x1_splitncnn_1 fire
11 ReLU           fire2/relu_expand1x1 1 1 fire2/expand1x1 fire2/expand1x1_fire2/relu_expand1x
12 Convolution   fire2/expand3x3 1 1 fire2/squeeze1x1_fire2/relu_squeeze1x1_splitncnn_0 fire
13 ReLU           fire2/relu_expand3x3 1 1 fire2/expand3x3 fire2/expand3x3_fire2/relu_expand3x
```

1) MagicNum

固定位 7767517

2) layer, blob 个数 (75, 83)

在 ncnn 网络初始化参数时，有两个 vector 容器

```
std::vector<Blob> blobs;
std::vector<Layer*> layers;
```

blobs 是每个网络层输出的数据容器，总数通常大于总层数

7767517		
75 83		
Input	data	0 1 data 0=227 1=227 2=3
Convolution	conv1	1 1 data conv1 0=64 1=3 2=1 3=2 4=0 5=1 6=1728
ReLU	relu_conv1	1 1 conv1 conv1_relu_conv1 0=0.000000
Pooling	pool1	1 1 conv1_relu_conv1 pool1 0=0 1=3 2=2 3=0 4=0
Convolution	fire2/squeeze1x1	1 1 pool1 fire2/squeeze1x1 0=16 1=1 2=1 3=1 4=0 5=1 6=1024
层类型	层名称	层输入数量、输出数量输入blob名称、输出blob名称
		没有圈黄的部分：层配置

- a. 层类型：Input、Convolution、ReLU
- b. 层名称：模型训练者为该层起得名字（毕竟相同类型的层可能多次使用，我们要区分它们）
- c. 层输入输出 包含：层输入 blob 数量，层输出 blob 数量，层输入、输出 blob 的名称
- d. 层配置参数

.param 文件的 id 索引问题

官方 wiki <https://github.com/Tencent/ncnn/wiki/param-and-model-file-structure>

operation param weight table <https://github.com/Tencent/ncnn/wiki/operation-param-weight-table>

- index : 0~19 对应整形或浮点型数据
- index: < -23000 减去 0~19 对应整形或浮点型数组
- 如果小于-23300，表示为数组，那么等号右边第一个参数就是数组长度，后面顺序就是数组内容，[array size],int,int,...,int 或[array size],float,float,...,float

```
0=1 1=2.5 -23303=2,2.0,3.0
```

数组实际的 id 为：

```
id = -id - 23300;
```

内存管理

参考资料

<https://zhuanlan.zhihu.com/p/335774042>

举例：比如内存按照 8 字节对齐的，就是我们分配的这个地址（转为整数） $addr \% 8 == 0$ 。随意分配的地址可能 $addr \% p != 0$ ，所以我们一般的做法是先分配一个比申请空间稍大的空间，然后将这个头地址，往前移动几个字节，使得新的 $addr$ 满足 $\% p == 0$

```
#define MALLOC_ALIGN 16

static inline void* fastMalloc(size_t size){

    unsigned char* udata = (unsigned char*)malloc(size +
sizeof(void*) + MALLOC_ALIGN);

    if(!udata) return 0;

    unsigend char** adata = alignPtr((unsigned
char**)udata+1,MALLOC_ALIGN);

    adata[-1] = udata;

    return adata;

}
```

- sizeof(void*)=> 8 bytes 为了存放分配的首地址, 保存下来是用于 Free
- MALLOC_ALIGN 就是为了对齐
- #define NULL (void*)0
- unsigned char* 是个指针类型, 指针类型占 8 字节的大小
- udata+1 是为了留出一个地方给分配的首地址用, 即 adata[-1]

ncnn:fastFree()

```
static inline void fastFree(void* ptr) {  
  
    if(ptr) {  
  
        unsigned char* udata = ((unsigned char**)ptr)[-1];  
  
        free(udata);  
  
    }  
  
}
```

与之相似的内存分配策略还有 MNN 和 OpenCV

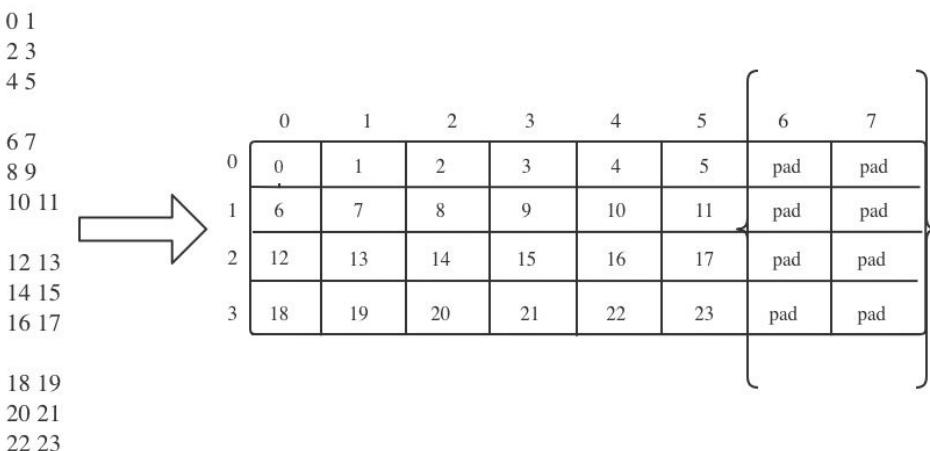
数据结构

Mat

<https://zhuanlan.zhihu.com/p/336359747>

DMA 请求，输出的数据会不会因为两个数组的起始地址不同，传输错误。

// w=2, h=3, c=4, elempack=1, float32, elemsize=4, 16-bytes aligned



// w=2, h=3, c=1, elempack=4, float32, elemsize=16, 16-bytes aligned

(0,6,12,18) (1,7,13,19)
(2,8,14,20) (3,9,15,21)
(4,10,16,22) (5,11,17,23)

总结, 你现在知道elempack的含义了嘛? 它表示有多少个数据打包在一起, elemsize = elempack * 元素类型所占字节数

一般elempack等于1, 4, 8, 主要作用就是使用指令集时, elempack个元素放在一个指令集里一起算, 起到加速的作用。

在x86平台下, 可以初略的认为:

elempack = 4: 用SSE2指令集
elempack = 8: 用AVX指令集

知乎 @OFShare

Vector (C++标准库中的)

向量 (Vector) 是一个封装了动态大小数组的顺序容器 (Sequence Container)。跟任意其它类型容器一样, 它能够存放各种类型的对象。可以简单的认为, 向量是一个能够存放任意类型的动态数组。

很多很多 API, 这里只记录目前遇到的

API 名称	功能
resize	调整容器的大小, 使其包含 n 个元素
begin	得到数组头的指针
end	得到数组的最后一个单元+1 的指针

Blob

.....

Operator

https://blog.csdn.net/sinat_31425585/category_9312419.html

算子 absval 解读：

重点语法：**虚函数**

基类指针指向派生类对象时，只能访问派生类的成员变量，但不能访问派生类的成员函数 !!!

为了消除这种尴尬，让基类指针能够访问派生类的成员函数，C++ 增加了**虚函数** (Virtual Function)。

使用虚函数非常简单，只需要在函数声明前面增加 **virtual** 关键字。

多态：有了虚函数，基类指针指向基类对象时就使用基类的成员（包括成员函数和成员变量），指向派生类对象时就使用派生类的成员。换句话说，基类指针可以按照基类的方式做事，也可以按照派生类的方式做事，它有多种形态，或者说有多种表现方式，我们将这种现象称为多态 (Polymorphism)。

!!! 虚函数的唯一作用就是构成多态

一些关键参数：

Class Layer 中的公有成员

```

bool one_blob_only           //one input and one output blob

bool support_inplace;       // support inplace inference

bool support_vulkan;        // support vulkan compute

bool support_packing;       // accept input blob with packed

storage

bool support_bf16_storage;  // accept bf16

bool support_fp16_storage;  // accept fp16

bool support_int8_storage;  // accept int8

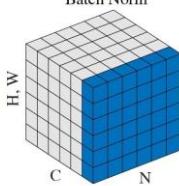
bool support_image_storage; // shader image storage

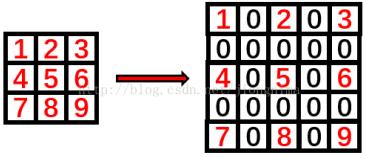
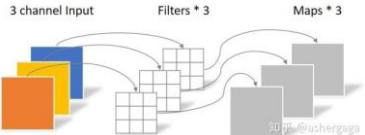
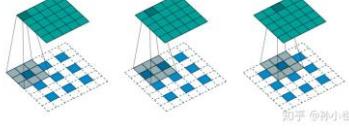
bool support_tensor_storage; // shader tensor storage

```

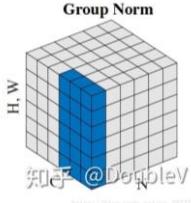
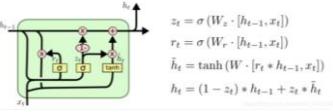
- support_inplace 原地推理
- forward 和 forward_inplace, 区别就是 inplace 操作原地替换，不会额外申请新内存空间
- shader 没看懂啥意思 ~~后面遇到再说

算子概述：

算子名称	简介	实现形式
absval	对输入所有元素求绝对值	ALU
argmax	对输入进行排序，取前 k 个最大值和序号	CPU
batchnorm	详见注释 	考虑算子融合

		
convolution1d	..	GEMM
convolution3d	..	GEMM
convolutiondepthwise	 https://zhuanlan.zhihu.com/p/92134485	GEMM
convolutiondepthwise1d	..	GEMM
convolutiondepthwise3d	..	GEMM
crop	裁剪层，一般用于全卷积神经网络中 (FCN)。 https://blog.csdn.net/wanyq07/article/details/77978490	CPU
deconvolution	反卷积是上采样的一种方式，反卷积也叫转置卷积。反卷积的本质还是卷积，只是在进行卷积之前，会进行一个自动的 padding 补充 0。  https://blog.csdn.net/zhsmkxy/article/details/107073350 可能需要更改一下算法	GEMM
deconvolution1d	..	GEMM
deconvolution3d
deconvolutiondepthwise
deconvolutiondepthwise1d
deconvolutiondepthwise3d
deepcopy	深拷贝	CPU
dequantize	反量化，三维时，c 维共享参数；二维时，h 维共享参数	ALU

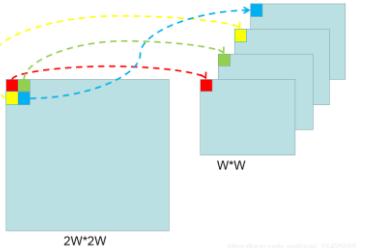
detectionoutput	对目标检测模型输出层做非极大值抑制（NMS），交并比（IOU）对检测结果进行过滤等。ssd 的检测输出层	CPU
dropout	随机失活	ALU
esinsum	全称 Einstein summation convention（爱因斯坦求和约定）。可以实现计算点积、外积、转置和矩阵向量或矩阵矩阵乘法等各种线性变换。 https://lonepatient.top/2018/05/25/einsum.html	/
eltwise	Eltwise 层的操作有三个：product（点乘），sum（相加减）和 max（取大值），要求四个维度都相同，逐元素操作	ALU
elu	激活函数，指数线性单元 $f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha * (e^x - 1) & \text{if } x \leq 0 \end{cases}$	CPU，或对激活量化采用查找表
embed	嵌入层，用在网络的开始层将你的输入转换成向量 https://blog.csdn.net/u010412858/article/details/77848878	ALU
exp	$f(x) = \begin{cases} e^{shift+scale*x} & \text{if bias == -1.0f} \\ bias^{shift+scale*x} & \text{else} \end{cases}$	CPU
expanddims	具体操作就是对输入 blob 沿着某个 axes 增加长度为 1 的维度，与 squeeze 刚好相反	CPU
flatten	就是将输入 blob 按照 channel 展开，并拉伸成一个一维数组	CPU
gelu	transformer 中使用，高斯误差线性激活单元 https://blog.csdn.net/liruihongbo/article/details/86510622	CPU，或对激活量化采用查找表

gemm	矩阵乘	GEMM
groupnorm	<p>GN, 组归一化</p>  <p>知乎 @Denislev https://zhuanlan.zhihu.com/p/35005794</p>	ALU 或者考慮融合
gru	<p>GRU-Gated Recurrent Unit 是 LSTM 最流行的一个变体，比 LSTM 模型要简单。</p> <p>https://blog.csdn.net/m_buddy/article/details/86549579</p> <p>https://zhuanlan.zhihu.com/p/46836364</p>  $\begin{aligned} z_t &= \sigma(W_z \cdot [h_{t-1}, x_t]) \\ r_t &= \sigma(W_r \cdot [h_{t-1}, x_t]) \\ \tilde{h}_t &= \tanh(W \cdot [r_t * h_{t-1}, x_t]) \\ h_t &= (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \end{aligned}$	分解
hardsigmoid	<p>激活函数</p> $f(x) = \begin{cases} 0 & x < -2.5 \\ 0.2 * x + 0.5 & -2.5 \leq x \leq 2.5 \\ 1 & x > 2.5 \end{cases}$	/
hardwish	<p>激活函数来自于 MobilenetV3</p> $HardSwish(x) = \frac{x * ReLU6(x)}{6}$ <p>ncnn 中没有对上限做处理</p>	ALU
innerproduct	全连接层，CNN 和 Transformer 中两种不同的调用形式，在 CNN 中就相当于二维矩阵乘	GEMM
input	数据输入层，ncnn 中似乎无作用	/
instancenorm	IN, instance norm, 在 H*W 维度上进行归一化	ALU 或者考慮融合

interp	<p>插值层 上下采样等，就是在 resize 图像时，需要对图像进行插值，主要分为最近邻插值、双线性插值和三次样方插值。</p> <p>https://blog.csdn.net/qq_39478403/article/details/105796249</p>	ALU 或专用单元
layernorm	<p>层归一化，在 H*W*C 维度上归一化</p>	ALU 或者考虑融合
log	<p>逐元素运算，取任意底对数 $f(x) = \log_{\text{base}}(\text{shift} + \text{scale} * x)$</p>	CPU
lrn	<p>Local Response Norm，局部相应归一化。AlexNet 首次提出，但模型泛化能力提高很少，以至于之后基本没用</p>	/
lstm	<p>$\Gamma_j^{(t)} = \sigma(W_j[a^{(t-1)}, x^{(t)}] + b_j)$ $\Gamma_i^{(t)} = \sigma(W_i[a^{(t-1)}, x^{(t)}] + b_i)$ $\tilde{c}^{(t)} = \tanh(W_c[a^{(t-1)}, x^{(t)}] + b_c)$ $c^{(t)} = \Gamma_f^{(t)} \circ c^{(t-1)} + \Gamma_i^{(t)} \circ \tilde{c}^{(t)}$ $\Gamma_o^{(t)} = \sigma(W_o[a^{(t-1)}, x^{(t)}] + b_o)$ $a^{(t)} = \Gamma_o^{(t)} \circ \tanh(c^{(t)})$</p>	分解
matmul	各种 shape 的矩阵乘算子	GEMM

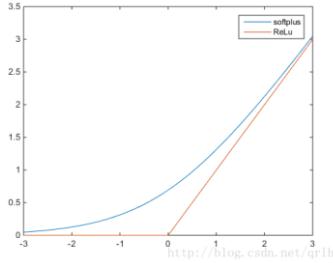
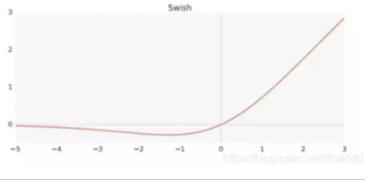
memorydata	从文件中读取一个 Mat 数据	/
mish	<p>激活函数 $Mish(x) = x * \tanh(\text{soft}(x))$</p> $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ $\text{soft}(x) = \ln(1 + e^x)$ $Mish(x) = x * \left(1 - \frac{2}{(1 + e^x)^2 + 1}\right) = x - \frac{2x}{(1 + e^x)^2 + 1}$	CPU, 或对激活量化采用查找表
multiheadattention	<p>transformer 中使用 https://finisky.github.io/2020/05/25/multiheadattention/</p> <pre> graph LR subgraph SDA [Scaled Dot-Product Attention] direction TB Q[Q] --> Scale[Scale] K[K] --> Scale Scale --> Mask[Mask (logits)] Mask --> SoftMax[SoftMax] SoftMax --> MHA[MHA] MHA --> V[V] end subgraph MHA [Multi-Head Attention] direction TB V[V] --> LinearV[Linear] K[K] --> LinearK[Linear] Q[Q] --> LinearQ[Linear] LinearV --> SDA LinearK --> SDA LinearQ --> SDA SDA --> Scale Scale --> Mask Mask --> SoftMax SoftMax --> MHA MHA --> Linear[Linear] Linear --> Conc[Concat] Conc --> V end </pre>	分解
mvn	<p>mean variance norm , 就是减去均值，除以标准差，将输入转换成 0 均值，方差为 1。</p> $f(x_i) = \frac{x_i - \text{mean}}{\mu}$ <p>两步操作，首先对每个 channel 进行操作，在将所有 channel 视为一个整体进行操作。</p>	CPU
noop	ncnn 中定义的空算子，不执行任何操作	/
normalize	<p>归一化操作</p> <p>直观来说，就是每个元素除以其模长，这里分为三种情况：</p> <p>第一种：直接考虑全局，就是 $W \times H \times C$ 个元素的模长；</p> <p>第二种：考虑每个 channel，就是计算当前 channel 上 $W \times H$ 个元素的模长；</p>	ALU

	第三种：考虑 $W \times H$ 个位置上，每个位置的 $C \times 1$ 个元素的模长；	
packing	ncnn 的 packing 策略，先不考虑	/
padding	共有三种类型： 常数、复制边界、复制边 $outptr[x] = v;$ $outptr[x] = ptr[0];$ $outptr[x] = ptr[left - x];$	CPU 或 ALU (最好实现算子融合)
permute	置换操作：就是按照某个 order 将输入 mat 进行重排	CPU
pixelshuffle	PixelShuffle 是一种上采样方法，可以对缩小后的特征图进行有效的放大。可以替代插值或反卷积的方法。 主要实现了这样的功能： $N \times (C \times r \times r) \times W \times H$ $>> N \times C \times (H \times r) \times (W \times r)$	CPU
pooling	全局池化（global pooling）不以窗口的形式取均值，而是以 feature map 为单位进行均值化。即一个 feature map 输出一个值。 在平均池化或最大池化中，您本质上是自己设置步幅和内核大小，将它们设置为超参数，在自适应池中，我们改为指定输出大小。并且会自动选择 stride 和 kernel-size 以适应需要。 注意其中包含了多种 padding	ACCEL
pooling1d
pooling3d
power	逐元素运算，取指数	CPU 或 ALU

	$f(x) = (shift + scale * x)^{power}$	
prelu	激活函数 (Parametric Rectified Linear Unit) $f(x) = \begin{cases} x & \text{if } x \leq 0 \\ slope * x & \text{else} \end{cases}$	ALU
priorbox	生成先验框	CPU
proposal	根据先验 anchors 和输入来生成边界框	CPU
psroipooling	位置敏感的候选区域池化 Position Sensitive roipooling https://codeantenna.com/a/Oxc7aSkTnn https://blog.csdn.net/a8039974/article/details/83113711	CPU 或 ALU
quantize	量化, float32 -> int8	ALU
reduction	将输入的特征图按照给定的维度进行求和或求平均等。	ALU
relu	根据 slope 同时包含 relu 和 leaky relu	ALU
reorg	yolov2 独有的层 , 一拆四层 , 一个大矩阵 , 下采样到四个小矩阵。 	CPU
requantize	连续量化, 详细见后文 ncnn 量化方案。	ACCEL
reshape	变换维度和维度的顺序	CPU
rnn	传统的 RNN 即 BasicRNNcell	分解

	<p>$a^{(t)} = \tanh(W_{ax}x^{(t)} + W_{aa}a^{(t-1)} + b_a)$</p> <p>$\hat{y}^{(t)} = \text{softmax}(W_{ya}a^{(t)} + b_y)$</p> <p>https://zhuanlan.zhihu.com/p/46836364</p> <p>双向 RNN :</p> <p>https://blog.csdn.net/Michale_L/article/details/126304723</p>	
roialign	<p>roipooling 存在一个很严重的问题，那就是两次量化之后存在较大偏差，这对于后面回归任务而言必然造成影响。roialign 直接采用浮点数计算，不进行向下取整的过程。</p> <p>对每个区域的每个元素进行双线性插值求均值（ncnn 代码里面是这样实现的，但是在一些其他的地方是双线性插值之后，再最大值池化。</p>	ALU+CPU
roipooling	<p>感兴趣区域池化</p> <p>Region of interest pooling，roipooling 源自于 faster rcnn 网络是用于目标检测任务的神经网络层。</p> <p>第一步根据 stride 得到目标在输出特征图上的位置。第一次量化。</p> <p>第二步根据输出维度划分区域。第二次量化。</p> <p>第三步最大值池化</p> <p>https://blog.csdn.net/sinat_314</p>	ALU+CPU

	25585/article/details/102927120	
scale	$f(x) = \begin{cases} scale * x & \text{if no bias} \\ scale * x + bias & \text{else} \end{cases}$	ALU
selu	激活函数 $f(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases}$	CPU, 或对激活量化采用查表
shufflechannel	源于 ShuffleNet, 首先将输入的 channel 分为 group 个组, 每个组对应 channel 数目为 , c/group, 然后打乱顺序 https://www.jianshu.com/p/44db6d72d6eb 	CPU
sigmoid	激活函数 $f(x) = \frac{1}{1 + e^{-x}}$	CPU, 或对激活量化采用查表
slice	concat 的反向操作, 通道分层, 适用于多任务网络。将输入 blob 按照某个 axis 切分为 N 个部分, 如输入为 6*H*W, 按照 axis=0, 将输入切分为三段, 每一段对应长度分别为 1,2,3, 那么输出为 : 1*H*W 2*H*W 3*H*W	CPU
softmax	$f(x_i) = \frac{e^{x_i}}{\sum_{j=0}^N e^{x_j}}$ 当 x 较大时, 就会导致 exp(x) 数值溢出, 所以一般处理时, 会对每个元素做一个减去最大值的处理	CPU, 或对激活量化采用查表
softplus	激活函数, 比 ReLU 光滑。 $\zeta(x) = \log(1 + e^x)$	CPU

		
split	将输入 blob 复制 n 份给输出 blob，对应操作为浅拷贝	CPU
spp	<p>空间金字塔池化 : SPP (Spatial Pyramid Pooling)</p> <p>作用就是利用池化技术，使得不同尺度输入图像都可以产生相同的输出维度。</p> <p>https://blog.csdn.net/sinat_31425585/article/details/102927120</p>	CPU
squeeze	将为 1 的维度压缩掉，例如： $a = [[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]]$ ，使用 squeeze 压缩后， $b=squeeze(a)=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$	CPU
statisticspooling	常使用在语信号处理中的一中池化方式，主要计算逐通道的均值和标准差，以及指数运算	ALU+CPU
swish	<p>激活函数</p> $f(x) = x * \text{sigmoid}(\beta x)$ 	CPU
tanh	<p>双曲正切激活</p> $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	CPU, 或对激活量化采用查表
threshold	$f(x) = \begin{cases} 1.0, & x > threshold \\ 0.0, & \text{else} \end{cases}$	ALU

tile	沿着某个维度，对输入 Mat 进行平铺操作，相当于将原始 Mat 沿着某个维度复制 tiles 份。比如原来是 1234，扩大两倍变成 11223344	CPU																																							
unaryop	一元操作：abs，sqrt，exp，sin，cos，conj（共轭）等。	ALU 或者 CPU																																							
yolodetectionoutput	inplace=True 为 yolov3detectionoutput 的部分。	CPU																																							
yolov3detectionoutput	<p>https://blog.csdn.net/xian0710830114/article/details/121425739?spm=1001.2101.3001.6650.1&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EOPENSEARCH%7ERate-1-121425739-blog-103353739.pc_relevant_default&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EOPENSEARCH%7ERate-1-121425739-blog-103353739.pc_relevant_default&utm_relevant_index=2</p> <table border="1"> <thead> <tr> <th>参数名</th> <th>解释</th> <th>默认值</th> </tr> </thead> <tbody> <tr> <td>2</td> <td>有2个输入</td> <td></td> </tr> <tr> <td>1</td> <td>1个输出</td> <td></td> </tr> <tr> <td>output1</td> <td>第一个输入的编号名称</td> <td></td> </tr> <tr> <td>output2</td> <td>第二个输入的编号名称</td> <td></td> </tr> <tr> <td>output</td> <td>一个输入的编号名称，程序中用这个值接返回值</td> <td></td> </tr> <tr> <td>0~20</td> <td>20个分类</td> <td>20</td> </tr> <tr> <td>1~3</td> <td>num_box</td> <td>5</td> </tr> <tr> <td>2~0.500000</td> <td>置信度阈值</td> <td>0.01</td> </tr> <tr> <td>3~0.300000</td> <td>nms阈值</td> <td>0.45</td> </tr> <tr> <td>-23304=12,xxxx</td> <td>anchor有12组,每组的值</td> <td></td> </tr> <tr> <td>-23305=6,xxxx</td> <td>anchor分配的顺序有6组,每组的index</td> <td></td> </tr> <tr> <td>-23306=2,xxx</td> <td>YOLO head有两组,每组的stride</td> <td></td> </tr> </tbody> </table>	参数名	解释	默认值	2	有2个输入		1	1个输出		output1	第一个输入的编号名称		output2	第二个输入的编号名称		output	一个输入的编号名称，程序中用这个值接返回值		0~20	20个分类	20	1~3	num_box	5	2~0.500000	置信度阈值	0.01	3~0.300000	nms阈值	0.45	-23304=12,xxxx	anchor有12组,每组的值		-23305=6,xxxx	anchor分配的顺序有6组,每组的index		-23306=2,xxx	YOLO head有两组,每组的stride		CPU
参数名	解释	默认值																																							
2	有2个输入																																								
1	1个输出																																								
output1	第一个输入的编号名称																																								
output2	第二个输入的编号名称																																								
output	一个输入的编号名称，程序中用这个值接返回值																																								
0~20	20个分类	20																																							
1~3	num_box	5																																							
2~0.500000	置信度阈值	0.01																																							
3~0.300000	nms阈值	0.45																																							
-23304=12,xxxx	anchor有12组,每组的值																																								
-23305=6,xxxx	anchor分配的顺序有6组,每组的index																																								
-23306=2,xxx	YOLO head有两组,每组的stride																																								

注释：

batchnorm :

<https://zhuanlan.zhihu.com/p/88347589>

机器学习领域有个很重要的假设：IID 独立同分布假设，就是假设训练数据和测试数据是

满足相同分布的，这是通过训练数据获得的模型能够在测试集获得好的效果的一个基本保障。BatchNorm 就是在深度神经网络训练过程中使得每一层神经网络的输入保持相同分布的。对于深度学习这种包含很多隐层的网络结构，在训练过程中，因为各层参数在变，所以每个隐层都会面临 covariate shift 的问题，也就是在训练过程中，隐层的输入分布老是变来变去，这就是所谓的“Internal Covariate Shift”，Internal 指的是深层网络的隐层，是发生在网络内部的事情，而不是 covariate shift 问题只发生在输入层。

BN 的优点

- BN 将 Hidden Layer 的输入分布从饱和区拉到了非饱和区，减小了梯度弥散，提升了训练速度，收敛过程大大加快，还能增加分类效果。
- Batchnorm 本身上也是一种正则的方式（主要缓解了梯度消失），可以代替其他正则方式如 dropout 等。
- 调参过程也简单多了，对于初始化要求没那么高，而且可以使用大的学习率等。

BN 的缺陷

- batch normalization 依赖于 batch 的大小，当 batch 值很小时，计算的均值和方差不稳定。

计算过程

$$\mu_j = \frac{1}{m} \sum_{i=1}^m Z_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m} \sum_{i=1}^m (Z_j^{(i)} - \mu_j)^2$$

$$\hat{Z}_j = \frac{Z_j - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

其中 ϵ 是为了防止方差为 0 产生无效计算。

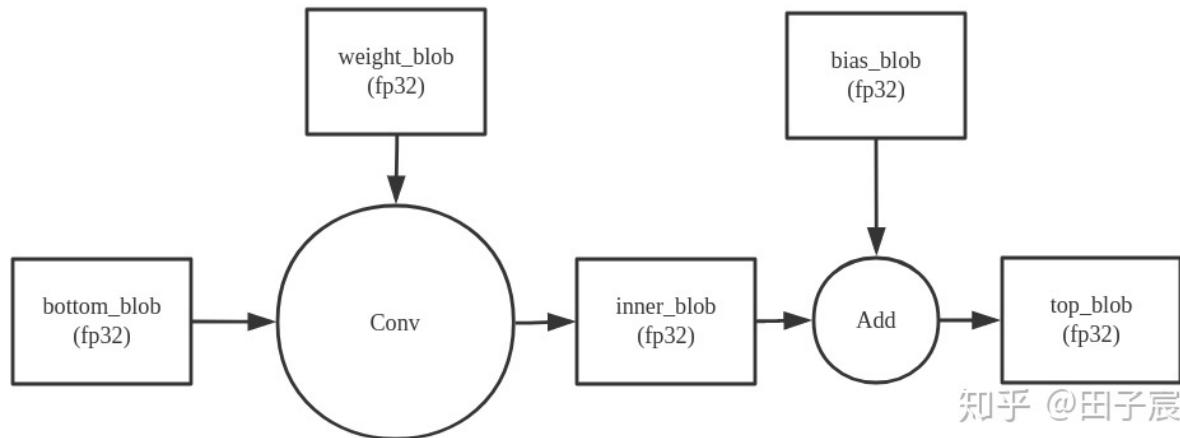
对每个特征进行独立的 normalization，使得第 1 层每个特征的输入的分布都是 0 均值，方差为 1，缓解了 ICS，但是让每一层网络的输入数据分布都变得稳定，但却导致了数据表达能力的缺失。也就是我们通过变换操作改变了原有数据的信息表达（representation ability of the network），使得底层网络学习到的参数信息丢失。另一方面，通过让每一层的输入分布均值为 0，方差为 1，会使得输入在经过 sigmoid 或 tanh 激活函数时，容易陷入非线性激活函数的线性区域。因此，BN 引入两个可学习的参数，对规范化后的数据进行线性变换，恢复数据本身的表达能力：

$$\tilde{Z}_j = \gamma_j \hat{Z}_j + \beta_j$$

NCNN 量化方案

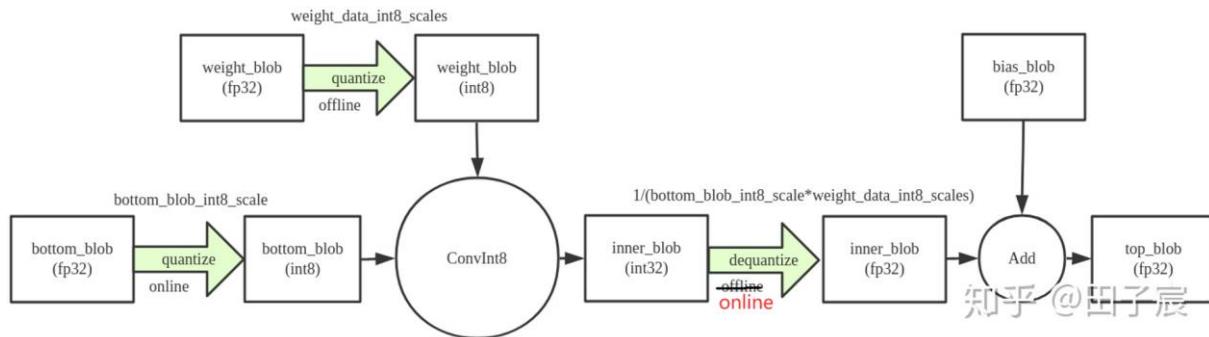
<https://zhuanlan.zhihu.com/p/71881443>

未量化下传统卷积方案：



fp32 Conv 计算流程，有时可不加 bias

在 NCNN Conv 进行 Int8 计算时：



量化公式：quantize OP

$$\begin{aligned}bottom_blob(int8) &= bottom_blob_int8_scale * bottom_blob(fp32) \\ weight_blob(int8) &= weight_data_int8_scale * weight_blob(fp32)\end{aligned}$$

so :

$$\begin{aligned}inner_blob(int32) &= (bottom_blob_int8_scale * weight_data_int8_scale) \\ &\quad * (bottom_blob(fp32) * weight_blob(fp32))\end{aligned}$$

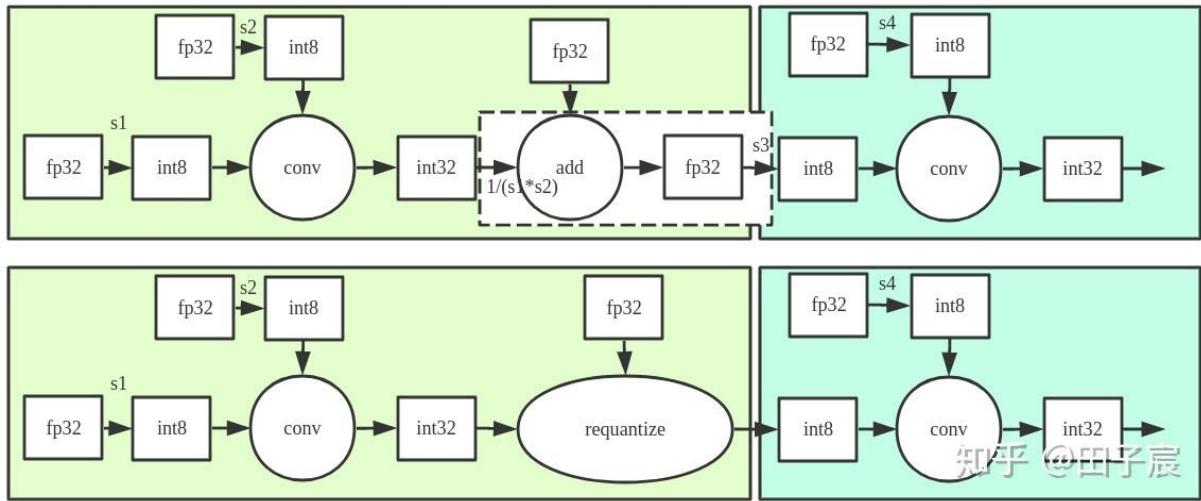
反量化公式：dequantize OP

$$dequantize_scale = 1 / (bottom_blob_int8_scale * weight_data_int8_scale)$$

$$inner_blob(fp32) = dequantize_scale * inner_blob(int32)$$

一个小操作：requantize OP

当一个 Conv1 后面紧跟着另一个 Conv2 时，NCNN 会进行 requantize 的操作。大致意思就是在得到 Conv1 的 INT32 输出后，会顺手帮 Conv2 做量化，得到 Conv2 的 INT8 输入。这中间不会输出 fp32 的 blob，节省一次内存读写。



requantize 相当于替换了原来的 dequantize+add_bias+下一层的 quantize
requantize 公式：

$$\text{Conv2_Input(int8)} = (\text{Conv1_Inner(int32)} * (1/s1 * s2) + \text{Conv1_Bias}) * s3$$

目前 NCNN 只有两种情况会用到 requantize：

- (1) Conv1 -> ReLU -> Conv2 , 且 Conv1 和 Conv2 都用 int8
- (2) Conv1 -> ReLU -> Split -> Conv2_1

-> Conv2_2
...
-> Conv2_x , 且这些 Conv 都用 int8

量化过程详解： ==> 量化脚本

<https://zhuanlan.zhihu.com/p/72375164>

ZCU104 使用记录：

芯片：MPSOC XCZU7EV-2FFVC1156

四核 Arm Cortex A53 + 双核 Arm Cortex R5

Table 1-1: Zynq UltraScale+ MPSoC ZU7EV Features and Resources

Feature	Resource Count
Quad core Arm Cortex-A53 MPCore	1
Dual core Arm Cortex-R5 MPCore	1
Mali-400 MP2 GPU	1
H.264/H.265 VCU	1
HD banks	Two banks, total of 48 pins
HP banks	Six banks, total of 312 pins
MIO banks	Three banks, total of 78 pins
PS-GTR transceivers (6 Gb/s)	Four PS-GTR transceivers
GTH transceivers (16.3 Gb/s)	20 GTH transceivers
System logic cells	504K
CLB flip-flops	461K
Maximum distributed RAM	6.2 Mb
Total block RAM	11 Mb
UltraRAM	27 Mb
DSP slices	1,728

交叉编译器

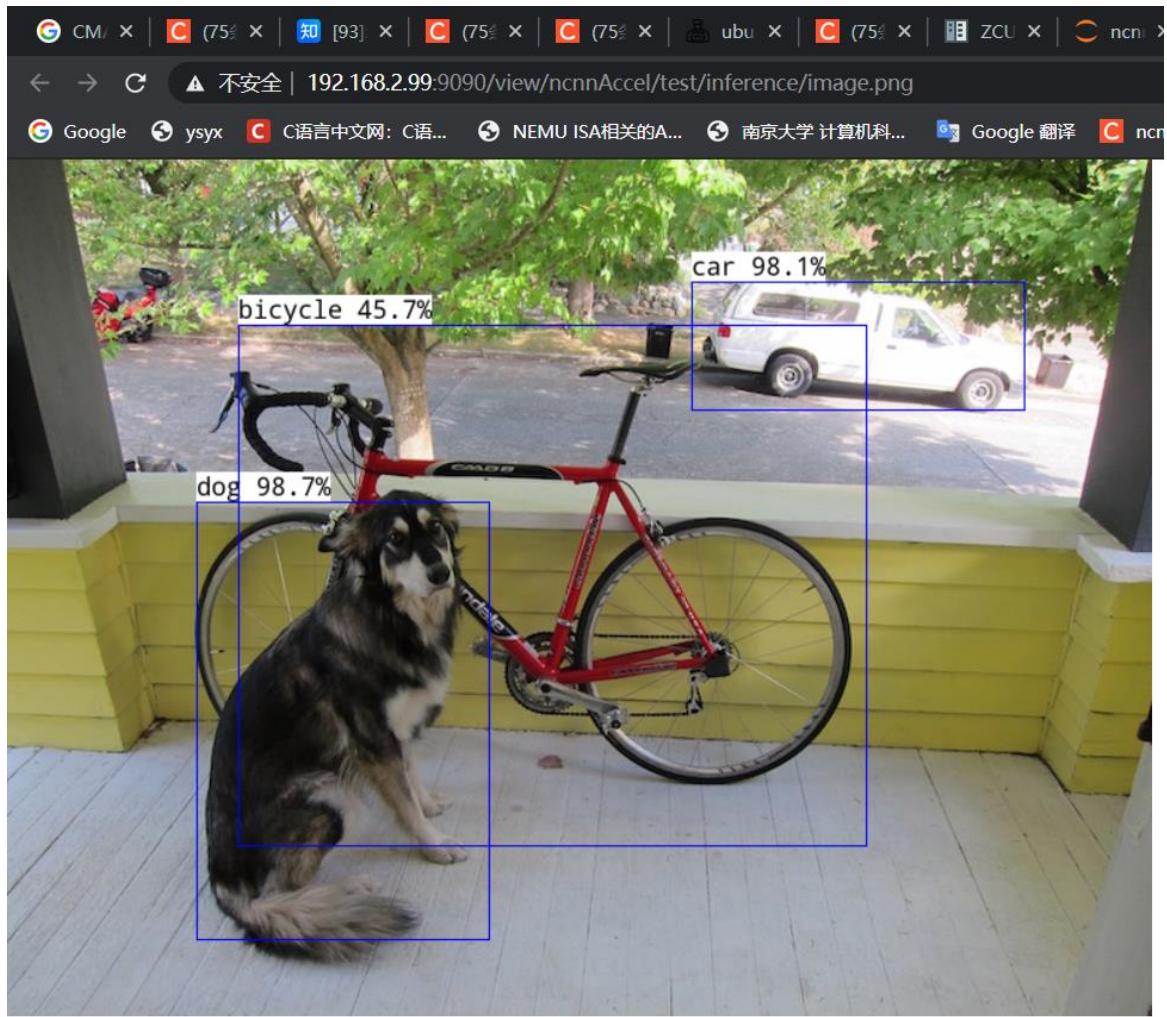
Cortex A53 | ARMv8 架构 | 64 位

install : https://blog.csdn.net/song_lee/article/details/105487177

查看 opencv 版本 : pkg-config --modversion opencv

opencv 安装 : https://blog.csdn.net/qq_36662437/article/details/97627520

开发板上的运行结果 :



base overlay:https://pynq.readthedocs.io/en/v2.5/pynq_overlays/zcu104/zcu104_base_overlay.html

Github:<https://github.com/xilinx/PYNQ/>

Table 2-4: Switch SW6 Configuration Option Settings

Boot Mode	Mode Pins [3:0]	Mode SW6 [4:1]
JTAG	0000/0x0	ON,ON,ON,ON
QSPI32	0010/0x2 ⁽¹⁾	ON,ON,OFF,ON
SDI	1110/0xE	OFF,OFF,OFF,ON

```
Welcome to PYNQ ZCU104, based on Xilinx Zynq® UltraScale+™ MPSoC Reference Design

Last login: Tue Sep 20 12:17:47 2022
xilinx@pynq:~$ uname -a
Linux pynq 5.4.0-xilinx-v2020.2 #1 SMP Wed Nov 17 23:55:17 UTC 2021 aarch64 aarch64 aarch64 GNU/Linux
xilinx@pynq:~$ █
```

MPSOC 相关 WIKI[https://xilinx-](https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/189595724/Zynq+UltraScale+MPSoC+Example+Designs)

[wiki.atlassian.net/wiki/spaces/A/pages/189595724/Zynq+UltraScale+MPSoC+Example+Designs](https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/189595724/Zynq+UltraScale+MPSoC+Example+Designs)

<https://github.com/ikwzm/ZynqMP-FPGA-Linux/tags>

<https://github.com/ikwzm/ZynqMP-FPGA-Linux>

在 ZCU102 上编译镜像

<https://www.wuquantai.com/post/%E7%BC%96%E8%AF%91pynq-%E5%9C%A8zcu102%E4%B8%8A%E7%9A%84%E9%95%9C%E5%83%8F/>

Rocket Chip

repository :

<https://github.com/chipsalliance/rocket-chip#what>

- chisel3:<https://github.com/ucb-bar/chisel3>
- firrtl:<https://github.com/ucb-bar/firrtl>
- hardfloat:<https://github.com/ucb-bar/berkeley-hardfloat>

Hardfloat 包含 Chisel 代码，该代码生成参数化的 IEEE 754-2008 兼容浮点单元，用于融合乘加运算、整数和浮点数之间的转换以及不同精度的浮点转换之间的转换。

- rocket-tools:<https://github.com/freechipsproject/rocket-tools>

我们标记了与此存储库中提交的 RTL 一起使用的 RISC-V 软件工具版本。

- torture:<https://github.com/ucb-bar/riscv-torture>

该模块用于生成和执行受约束的随机指令流，可用于对设计的核心和非核心部分进行压力测试。

- 创建自定义设备的顶层设计：<https://github.com/ucb-bar/chipyard>
- on FPGA:<https://github.com/sifive/freedom/blob/master/README.md>
- Docs: <https://chipyard.readthedocs.io/en/stable/Generators/Rocket-Chip.html>
- freedom:<https://github.com/sifive/freedom/blob/master/README.md>

Scala Packages:

- 1) **amba** 使用 diplomacy 来生成 AMBA 协议的总线实现，包括 AXI4、AHB-lite 和 APB。
- 2) **config** 提供了 Scala 接口，用于通过动态范围的参数化库配置生成器。
- 3) **coreplex** 通过将来自其他包的各种组件合在一起生成完整的 coreplex，包括： tiled Rocket cores, a system bus network, coherence agents, debug devices, interrupt

handlers, externally-facing peripherals, clock-crossers and converters from TileLink to external bus protocols (e.g. AXI or AHB).

- 4) **devices** 包含外围设备的实现，包括调试模块和各种 TileLink 从设备。
- 5) **diplomacy** 该包通过允许参数在模块间动态协商，以实现两阶段硬件细化，进而扩展了 chisel
- 6) **groundtest** 包含可综合的硬件测试器，它会发出随机的内存访问流以便对非核心的内存层次结构进行压力测试。
- 7) **jtag** 包含了用于生成 jtag 总线接口的定义
- 8) **regmapper** 用于生成具有标准接口的从设备以访问内存映射寄存器
- 9) **rocket** 包含 rocket 顺序流水线内核，以及 L1 指令 cache 和数据 cache。该库被 chip 生成器使用并在内存系统中实例化，连接到外部
- 10) **tile** 包含了可与内核组合以构成 construct tiles 的组件，比如 FPU 和加速器
- 11) **tilelink** 该包使用 diplomacy 实现 TileLink 总线，其中也包含各种适配器和协议转换器
- 12) **system** 顶层模块，使用 chisel 生成特定配置的 coreplex，以及相关的测试
- 13) **unitest** 包含了一个可综合的单个模块的硬件测试器架构
- 14) **util** 包含了一些 scala 和 chisel 的复用结构

Other Sources :

- 1) **bootrom** 包含第一阶段 bootloader (引导加载程序) 的源代码
- 2) **csrc** 用于 Verilator 仿真的 C 源码
- 3) **docs** 代码库某些部分的文档和教程等等
- 4) **emulator** Verilator 编译和运行的目录
- 5) **project** 使用 sbt 构建和编译 scala 的目录
- 6) **regression** Defines continuous integration and nightly regression suites.
- 7) **scripts** 用于解析模拟输出和操作源文件内容的程序
- 8) **vsim** 编译和运行 Synopsys VCS 仿真的目录
- 9) **vsrcc** 包含接口、约束和 VPI 的 Verilog 源代码

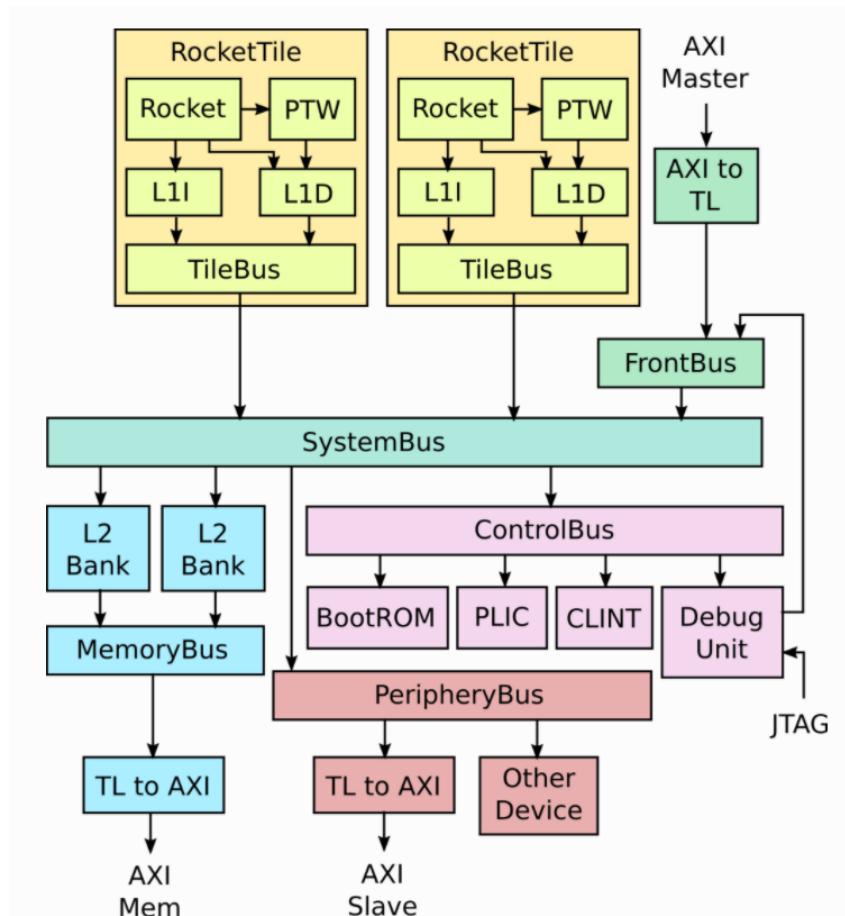
一些知识：

Rocket chip generator 的三个 Targets :

- 1) a high-performance cycle-accurate Verilator
- 2) Verilog optimized for FPGAs
- 3) Verilog for VLSI

支持 rocket chip 的一些软件工具集合：<https://github.com/chipsalliance/rocket-tools/blob/master/README.md>

- 1) Spike, the ISA simulator
- 2) riscv-tests, a battery of ISA-level tests
- 3) riscv-opcodes, the enumeration of all RISC-V opcodes executable by the simulator
- 4) riscv-pk, which contains bbl, a boot loader for Linux and similar OS kernels, and pk, a proxy kernel that services system calls for a target-machine application by forwarding them to the host machine.(一个代理内核，通过将目标机器应用程序的系统调用转发到主机为系统调用提供服务)

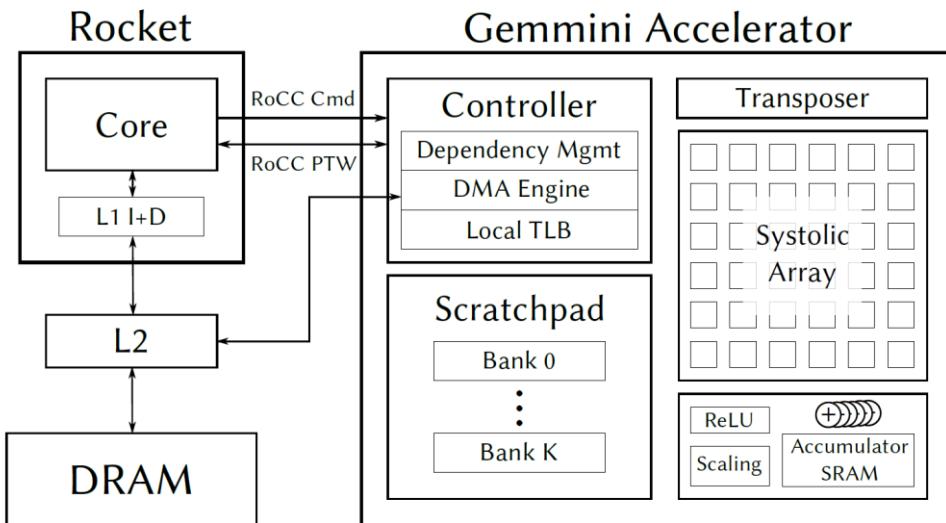


上图展示了一个双核的 Rocket 系统，每一个内核中都有一个页表遍历器 (PTW) , L1 指令

cache 和 L1 数据 cache，然后连接在 TileBus 上，Rocket 核心还可以换成 BOOM 核心，每一个 tile 都可以配置一个 RoCC 加速器，作为协处理器连接到内核。

Tiles 连接到 SystemBus，后者将其连接到 L2 cache。L2 cache 连接到 MemoryBus，后者通过 TileLink 到 AXI 转换器连接到 DRAM 控制器。

Gemmini



Customization

添加多核系统

<https://chipyard.readthedocs.io/en/stable/Customization/Heterogeneous-SoCs.html#adding-hwachas>

6.1.1. Creating a Rocket and BOOM System

Instantiating an SoC with Rocket and BOOM cores is all done with the configuration system and two specific config fragments. Both BOOM and Rocket have config fragments labelled

`WithN{Small|Medium|Large|etc.}BoomCores(X)` and `WithNBigCores(X)` that automatically create `X` copies of the core/tile [1]. When used together you can create a heterogeneous system.

The following example shows a dual core BOOM with a single core Rocket.

```
class DualLargeBoomAndSingleRocketConfig extends Config(
    new boom.common.WithNLargeBooms(2) ++ // add 2 boom cores
    new freechips.rocketchip.subsystem.WithNBigCores(1) ++ // add 1 rocket core
    new chipyard.config.AbstractConfig)
```

使用 MultiRoCC 将加速器分配给特定的 Tiles

6.1.3. Assigning Accelerators to Specific Tiles with MultiRoCC

Located in `generators/chipyard/src/main/scala/config/fragments/RoCCFragments.scala` is a config fragment that provides support for adding RoCC accelerators to specific tiles in your SoC. Named `MultiRoCCKey`, this key allows you to attach RoCC accelerators based on the `hartId` of the tile. For example, using this allows you to create a 8 tile system with a RoCC accelerator on only a subset of the tiles. An example is shown below with two BOOM cores, and one Rocket tile with a RoCC accelerator (Hwacha) attached.

```
class DualLargeBoomAndHwachaRocketConfig extends Config(
    new chipyard.config.WithMultiRoCC ++
    new chipyard.config.WithMultiRoCCHwacha(0) ++
    new hwacha.DefaultHwachaConfig ++
    new boom.common.WithNLargeBooms(2) ++
    new freechips.rocketchip.subsystem.WithNBigCores(1) ++
    new chipyard.config.AbstractConfig)
```

// support heterogeneous rocc
// put hwacha on hart-0 (rocket)
// set default hwacha config key
// add 2 boom cores
// add 1 rocket core

The `WithMultiRoCCHwacha` config fragment assigns a Hwacha accelerator to a particular `hartId` (in this case, the `hartId` of `0` corresponds to the Rocket core). Finally, the `WithMultiRoCC` config fragment is called. This config fragment sets the `BuildRoCC` key to use the `MultiRoCCKey` instead of the default. This must be used after all the RoCC parameters are set because it needs to override the `BuildRoCC` parameter. If this is used earlier in the configuration sequence, then MultiRoCC does not work.

- 多个加速器核采用 `WithMultiRoCCHwacha(0,1,3,6,.....)`
- `hartId` 的顺序是从下到上，所有上面的例子中 Rocket 是第一个

建议将自己的工程作为 git submodule 添加到工程中

<https://chipyard.readthedocs.io/en/stable/Customization/Custom-Chisel.html>

Adding a Custom Core

自己添加的 core 不支持 RoCC，RoCC 只支持 Rocket 和 BOOM

.....

RoCC vs MMIO

将加速器或者其他自定义 IO 设备添加到 SoC 系统中的途径：

- 1) MMIO Peripheral (a.k.a TileLink-Attached Accelerator) -> MMIO(Memory-mapped I/O)
- 2) Tightly-Coupled RoCC Accelerator

RoCC 加速器采用保留在 RISC-V ISA 编码空间的自定义非标准 ISA 指令

每个内核最多有四个由自定义指令集控制并与 CPU 共享 cache 的加速器

指令格式 : customX rd, rs1, rs2, funct

The X will be a number 0-3, and determines the opcode of the instruction, which controls which accelerator an instruction will be routed to. The `rd`, `rs1`, and `rs2` fields are the register numbers of the destination register and two source registers. The `funct` field is a 7-bit integer that the accelerator can use to distinguish different instructions from each other.

Note : 使用 RoCC 接口需要自定义软件工具链，而 MMIO 可以使用支持相应驱动的标准工具链

C 内嵌汇编是不是可以？？？

Adding a RoCC Accelerator

RoCC 加速器是扩展 LazyRoCC 类的 lazy modules，他们的实现必须继承 LazyRoCCModule 类

LazyRoCC 包含两个 TLOutputNode 实例，atlNode 和 tlNode。

The former connects into a tile-local arbiter along with the backside of the L1 instruction cache. The latter connects directly to the L1-L2 crossbar. The corresponding Tilelink ports in the module implementation's IO bundle are atl and tl, respectively.

另一个加速器可以利用的端口是 mem，他直接连接在 L1 cache 中；ptw 提供对页表遍历器（page-table walker）的访问；busy 信号表示加速器何时在处理指令；interrupt 可以用来中断 CPU。

添加 RoCC 加速器到一个 core 中可以通过重写配置中的 BuildRoCC 参数

如：当我们想给自己的加速器添加 custom0 和 custom1 指令时：

```
class WithCustomAccelerator extends Config((site, here, up) => {
  case BuildRoCC => Seq((p: Parameters) => LazyModule(
    new CustomAccelerator(OpcodeSet.custom0 | OpcodeSet.custom1)(p)))
})  
  
class CustomAcceleratorConfig extends Config(
  new WithCustomAccelerator ++
  new RocketConfig)
```

Note : 如果想在程序中添加 RoCC 指令，可以使用 test/rocc.h 中提供的宏，可以在文件 test/accum.c 和 charcount.c 中找到例子

Dsptools

dsptools 是一个 chisel 库，可以用来帮助编写自定义的信号处理加速器。

opencv 补充知识

Mat 作为 opencv 中使用率最高的类，它的数据包括两个部分：矩阵头和指向像素矩阵的指针。

矩阵头：描述像素矩阵，主要包括矩阵的尺寸、存储方式、存储地址等，矩阵头的大小固定。

矩阵指针：矩阵指针所指对象代表了图像本身，其尺寸会根据图像的不同而不同。像素矩阵一般比矩阵头大几个数量级，因此，拷贝图像会产生很大的计算量。

深拷贝和浅拷贝的区别：cv::Mat 的深拷贝和浅拷贝的区别就在于是否重新拷贝一份数据指针指向的数据？浅拷贝只是重新拷贝 Mat 的矩阵头，并不拷贝数据指针指向的数据，也就是说新生成的和之前的共用一块相同的数据空间，但是深拷贝则是连同数据一起重新拷贝一份。

cv::Mat

```
void Func(cv::Mat Input){    Input = cv::Mat::ones(4, 4, CV_32F); //修改了 Input，但是并不修改函数外的 Mat  //...}
```

这种方法只是将 Mat 的矩阵头拷贝了一份，此时在函数 Func()中可以更改 Input 的矩阵头中的参数，但是并不会影响到函数外的 Mat，因为这种方法对矩阵的头进行的是值传递。

const cv::Mat

```
void Func(const cv::Mat Input){    Input = cv::Mat::ones(4, 4, CV_32F); //错误，不能修改 Input  //...}
```

函数依然是按值传递，由于 const 的限制 input 的矩阵头中的数据不能修改，也就是

不能在函数里面修改矩阵的大小等参数，因为这些参数保存在矩阵头中。

cv::Mat&

```
void Func(cv::Mat& Input){    Input = cv::Mat::ones(4, 4, CV_32F); //修改了 Mat 的矩阵头，函数外的 Mat 也受影响  //...}
```

在使用引用的时候，修改矩阵 Input 将会影响到函数外的 Mat，也就是矩阵头中包含

的任何参数修改都会影响到外部的 Mat。

const cv::Mat&

```
void Func(const cv::Mat& Input){ Input = cv::Mat::ones(4, 4, CV_32F); //修改了 Mat 的矩阵头，函数外的 Mat 也受影响 //...}
```

在使用 const 引用的时候，将不能修改矩阵头参数。

!!!引用类型的函数形参请尽可能的使用 const

```
1. #include <cstdio>
2. using namespace std;
3.
4. double volume(const double &len, const double &width, const double &hei){
5.     return len*width*2 + len*hei*2 + width*hei*2;
6. }
7.
8. int main(){
9.     int a = 12, b = 3, c = 20;
10.    double v1 = volume(a, b, c);
11.    double v2 = volume(10, 20, 30);
12.    double v3 = volume(89.4, 32.7, 19);
13.    double v4 = volume(a+12.5, b+23.4, 16.78);
14.    double v5 = volume(a+b, a+c, b+c);
15.    printf("%lf, %lf, %lf, %lf, %lf\n", v1, v2, v3, v4, v5);
16.
17.    return 0;
18. }
```

volume() 函数用来求一个长方体的体积，它可以接收不同类型的实参，也可以接收常量或者表达式。

概括起来说，将引用类型的形参添加 const 限制的理由有三个：

- 使用 const 可以避免无意中修改数据的编程错误
- 使用 const 能让函数接收 const 和非 const 类型的实参，否则将只能接收非 const 类型的实参
- 使用 const 引用能够让函数正确生成并使用临时变量

Cmake 补充知识

基本语法格式 指令(参数 1 参数 2)

参数之间可以使用空格或者分号分隔开来

指令是大小写无关的，参数和变量是大小写相关的

变量使用\${}方式取值，如\${HELLO}，但是在 IF 控制语句中，直接使用变量名，如 if(HELLO)

重要指令

Hardware 篇

一、 Hardfloat 库

Github : <https://github.com/ucb-bar/berkeley-hardfloat>

<http://www.jhauser.us/arithmetic/HardFloat.html>

构建工程出现的问题

方案一、直接用 gemmini 的构建 sbt

方法二、修改 sbt 的托管管理

<https://yintianyu.github.io/2019/01/21/post-sbt-dependencies/>

<https://www.scala-sbt.org/1.x/docs/zh-cn/Library-Dependencies.html>

Berkeley HardFloat is a free, high-quality Verilog encoding of digital hardware modules for binary floating-point arithmetic. HardFloat **fully conforms to the IEEE Standard** for Floating-Point Arithmetic, supporting **all required rounding modes, exception flags, and special values, including subnormals**. These operations are implemented:

- addition/subtraction
- multiplication
- fused multiply-add
- division
- square root
- comparisons

- conversions between supported floating-point formats
- conversions to/from integers, signed and unsigned

A wide range of floating-point sizes is supported, with modules taking parameters that **independently determine the widths of the exponent and significand fields**. The set of possible formats includes the standard IEEE ones of 16-bit half-precision, 32-bit single-precision, 64-bit double-precision, and 128-bit quadruple-precision.

HardFloat converts IEEE-format floating-point inputs into a more convenient **recoded format** for its arithmetic operations. However, the results computed by HardFloat modules are always the exact values dictated by the IEEE Standard, just represented in the recoded format. Computed results can be converted back to a standard IEEE format at any time.

WARNING: These units are works in progress. They may not be yet completely free of bugs, nor are they fully optimized.

1、数据格式

- FN: 符合 IEEE 标准的浮点数
- IN: 整形
- recFN: 浮点寄存器文件存储的浮点数即为 recFN 格式，该格式对 IEEE 浮点数格式进行简单的变换，将部分非规格化数转换为规格化数（是指以 recFN 表示时为规格化数），因此可以简化一些浮点数处理的方法。
- rawFloat: 在进行浮点运算时，利用 rawFN 格式，该格式将浮点数的信息分解为 6 部分：符号位、指数位域、底数位域、非数标记、无穷标记、零标记。将浮点数信息拆解后，便于运算部件的处理，同时在 rounding 之前的中间结果也使用该格式进行表示。

2、常用对象和类

类型转换

(1) 转换到 RecFN:

INToRecFN、**recFNFFromFN**、**RecFNTToRecFN**、**RoundAnyRawFNTToRecFN**

(2) 转换到 rawFloat

rawFloatFromFN、**rawFloatFromIN**、**rawFloatFromRecFN**、**resizeRawFloat**

(3) 转换到 IN

RecFNTToIN

(4) 转换到 FN

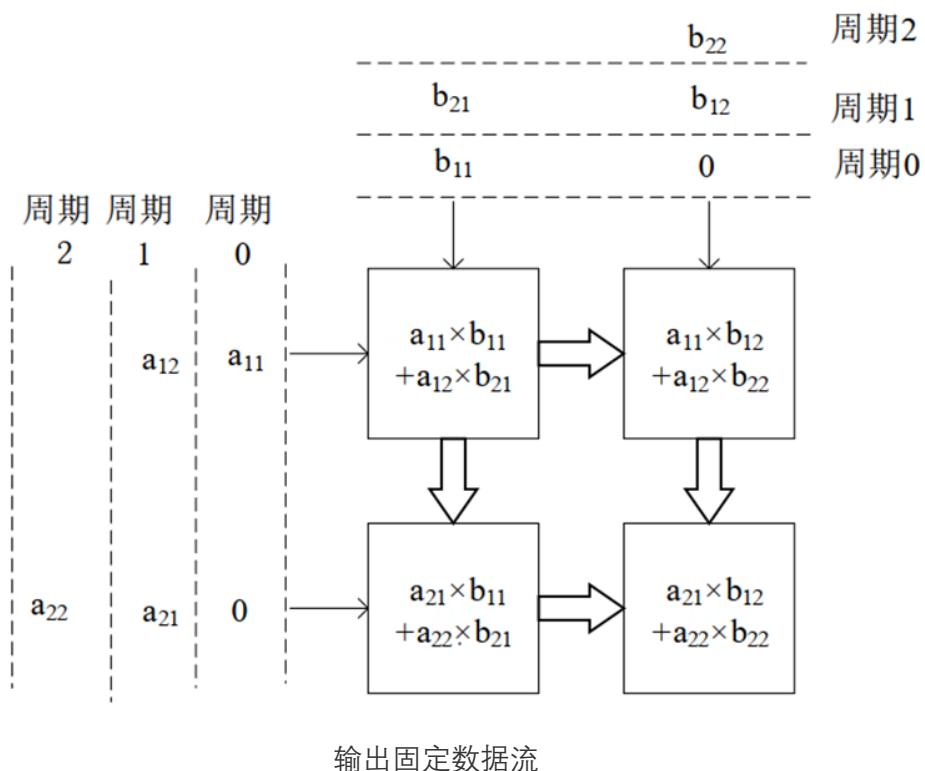
fNFromRecFN

运算

- **AddRecFN**
- **CompareRecFN**
- **MulAddRecFN**
- **MulRecFN**
- classifyRecFN 等，本项目用不到

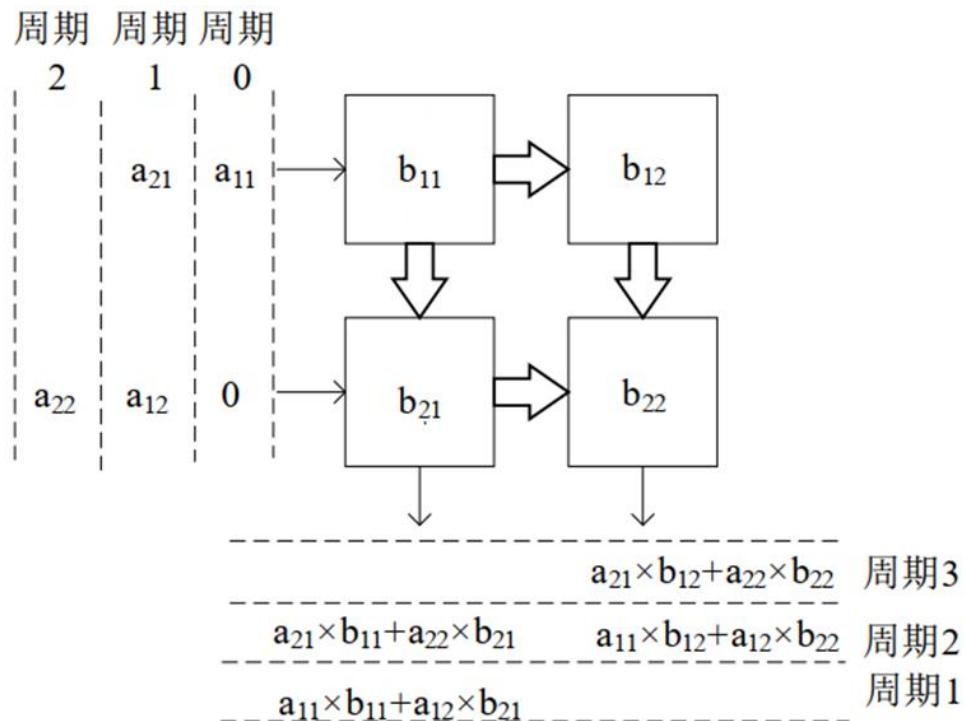
二、OS & WS

1、OS 输出固定



- 输出固定的数据流设计中，矩阵 A 和矩阵 B 在同一时钟周期输入脉动阵列，其计算结果保存在计算单元中。
- PE 单元之间通过寄存器传输数据，存在一个周期的时钟延时，为了在正确的时钟周期抵达同一个计算单元，脉动阵列每一行的输入相比上一行多插入一个时钟周期延时。

2、WS 权重固定



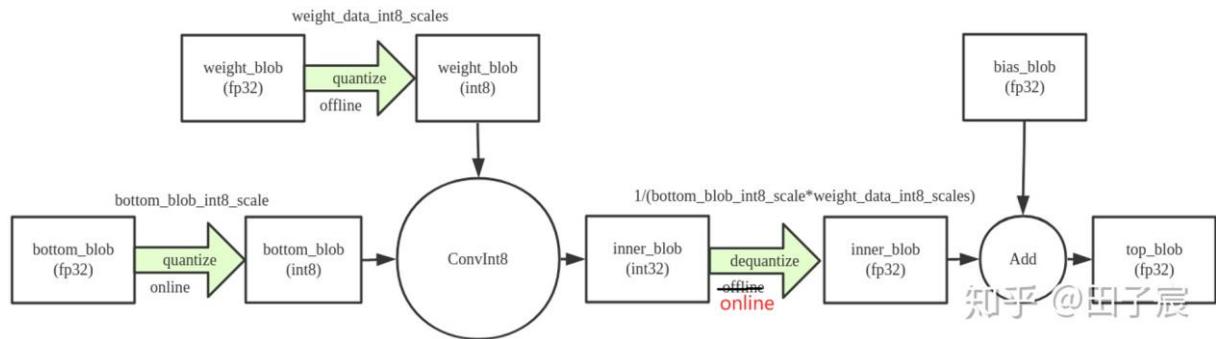
权重固定数据流

- 基于权重固定的数据流设计将矩阵 B 存入了脉动阵列中，数据复用性更高。
- 计算单元中加法器的数据位宽要求更低。
- 对于加速卷积神经网络，权重存入脉动阵列中，特征图左侧流入，计算结果下侧流出。
- 卷积计算更适合权重固定的数据流。

! 本设计中采用 WS 数据流

三、架构设计

1、ncnn 卷积流程（量化）



(1) float32_to_int8

=> FN +> RecFN +> IN

(2) make_padding

(3) im2col

(4) gemm

(5) scale

=> int32*float32 (IN +> RecFN) * (FN +> RecFN)

(6) bias

=> float32+float32 (FN +> RecFN) + (FN +> RecFN)

(7) activation

ReLU、LeakyReLU: 仍然利用 HardFloat 库，增加参数 latency

Sigmoid、Tanh 待设计

(8) (requantize)

=> float32_to_int8(float32*float32) ((FN +> RecFN) * (FN +> RecFN)) +> IN

2、数据接口和算力

为了满足不同软核硬核的需求，此处采用了双通道的 HP 接口设计（双 AXI 接口）。采用双接口设计可以很大程度上提高协处理器访问内存的带宽。之所以这么设计，实际上是参考了 xilinx 官方 HEVC IP 核的设计思想，其采用了 5 个 HP 接口设计，两个接口用于编码，两个接口用于解码，一个接口用于 MCU。

以下测试数据来自米联客 MPSOC 15EG 开发板的实测：

HP (AXI4) 带宽测试 (100MHz)

频率	四通道		双通道		单通道	
100MHz	HP0	862.04MBps	HP0	1100.20MBps	HP0	1114.73MBps
	HP1	840.32MBps				
	HP2	785.99MBps	HP1	1100.20MBps		
	HP3	643.37MBps				
Total	3133.77MBps		2200.41MBps		1114.73MBps	

HP (AXI4) 带宽测试 (150MHz)						
频率	四通道		双通道		单通道	
150MHz	HP0	976.83MBps	HP0	1489.41MBps	HP0	1560.62MBps
	HP1	896.75MBps				
	HP2	854.08MBps	HP1	1489.41MBps		
	HP3	753.03MBps				
Total	3480.69MBps		2978.82MBps		1560.62MBps	

HP (AXI4) 带宽测试 (200MHz)						
频率	四通道		双通道		单通道	
200MHz	HP0	1021.36MBps	HP0	1688.54MBps	HP0	1865.49MBps
	HP1	914.34MBps				
	HP2	870.85MBps	HP1	1688.54MBps		
	HP3	804.38MBps				
Total	3610.94MBps		3377.08MBps		1865.49MBps	

从以上的实测数据中可以分析得到，协处理器主频设置为 200MHz，HP 接口数量为 2 时较为合适。同时，为了更强系统的可扩展性，**系统应支持多核处理器模式**。

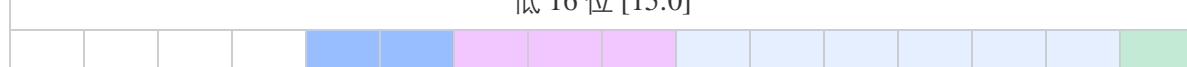
各模块峰值算力计算：(以 FPGA 200MHz 的主频为例)

- **ALU**：DMA 位宽为 2*128bit，支持数据类型为 int32 和 float32。并且支持矩阵点积和矩阵加法运算，因此提供的算力为 $256/32*2*200M = 3.2\text{GFlops}$ 。ALU 中数学函数计算的算力不计入。
- **Im2Col**：该过程包括量化、维度变换、Padding 三部分，其中量化部分提供了系统的算力。DMA 位宽为 2*128bit，支持数据类型为 float32。因此 Im2Col 单元提供的算力为 $256/32*200M = 1.6\text{GFlops}$ 。

3、详细设计

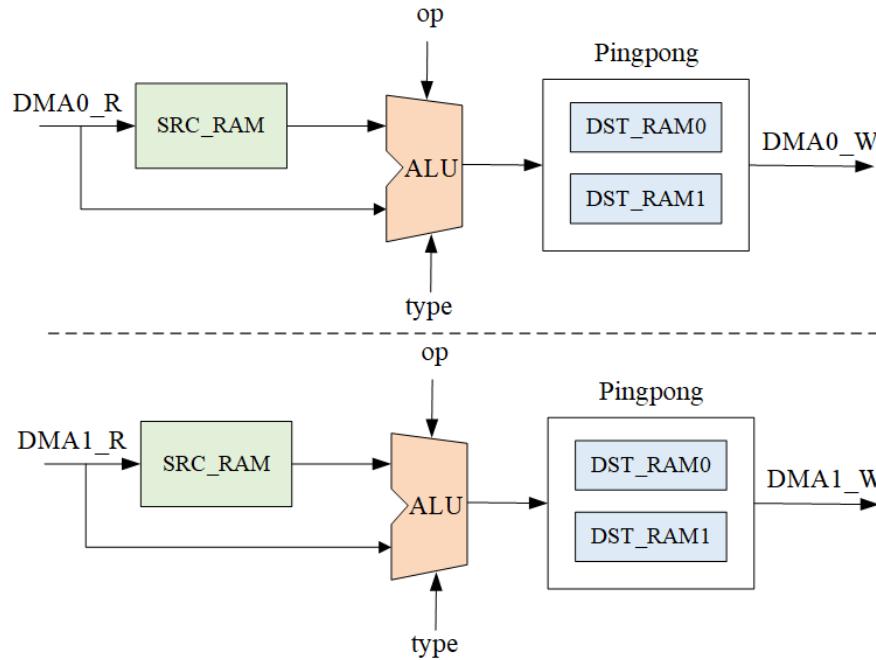
ALU Unit

ALU 控制寄存器定义如下：

ALU 单元控制寄存器 (alu_ctrl_reg)															
offset : 0x0030															
高 16 位 [31:16]															
															
低 16 位 [15:0]															
															

- [0:0] ALU 矩阵逐元素运算使能 上升沿有效
- [6:1] ALU 矩阵逐元素运算 OP 运算类型 0 : 矩阵加法、1 : 矩阵点积、2、矩阵绝对值、其他 : 保留
- [9:7] ALU 矩阵逐元素运算数据类型 1 : Int32、2 : float32、其他 : 保留
- [11:10] ALU 矩阵逐元素运算通道数 1 : 仅通道 1 有效、2 : 仅通道 2 有效、3 : 双通道全开
- [16:16] ALU 数学函数运算使能, 上升沿有效
- [22:17] ALU 数学函数类型 0 : sin、1 : cos、2 : atan、3 : sqrt、4 : iexp、5 : log2、其他、保留
 - (1) 对于矩阵的逐元素运算

ALU 矢量长度寄存器 0 (alu_veclen0_reg)	offset : 0x0034
ALU 矢量长度寄存器 1 (alu_veclen1_reg)	offset : 0x0038



(2) 对于数学函数运算

函数	功能描述
sin	$\sin(2\pi x), x \in R$
cos	$\cos(2\pi x), x \in R$
atan	$\arctan(x), x \in (-1, 1)$
sqrt	$\sqrt{x}, x > 0$
iexp2	$2^{- x }, x \in Z$
log2	$\log_2 x, x > 0$

- 所有数据输入均为 fp32
- 由于数据量较少，直接通过 AXI-LITE 传输即可
- 寄存器定义如下，同时承担着发送和写回的任务

ALU 数学函数数据寄存器 (alu_mathdata_reg)	offset : 0x003C
----------------------------------	-----------------

Im2Col Unit

im2col 单元本质上是完成 NCHW 到 NWHC 数据格式的变换。根据是否需要重量化，这里分为 float32 和 int8 两种数据类型。但是由于 int8 的数据类型在重量化之后，可以直接输出 NWHC 格式的特征图，因此不需要重复对特征图进行 im2col 单元处理。

模块控制寄存器如下：

Im2Col 单元控制寄存器 (im2col_ctrl_reg)															
offset : 0x0028															
高 16 位 [31:16]															
低 16 位 [15:0]															

- [0:0] im2col 单元使能 上升沿有效
- [2:1] ifm_mem 输入数据类型 0 : mat_fp32 1 : mat_int32 2 : ifm_fp32 3: ifm_int8
-

1、量化

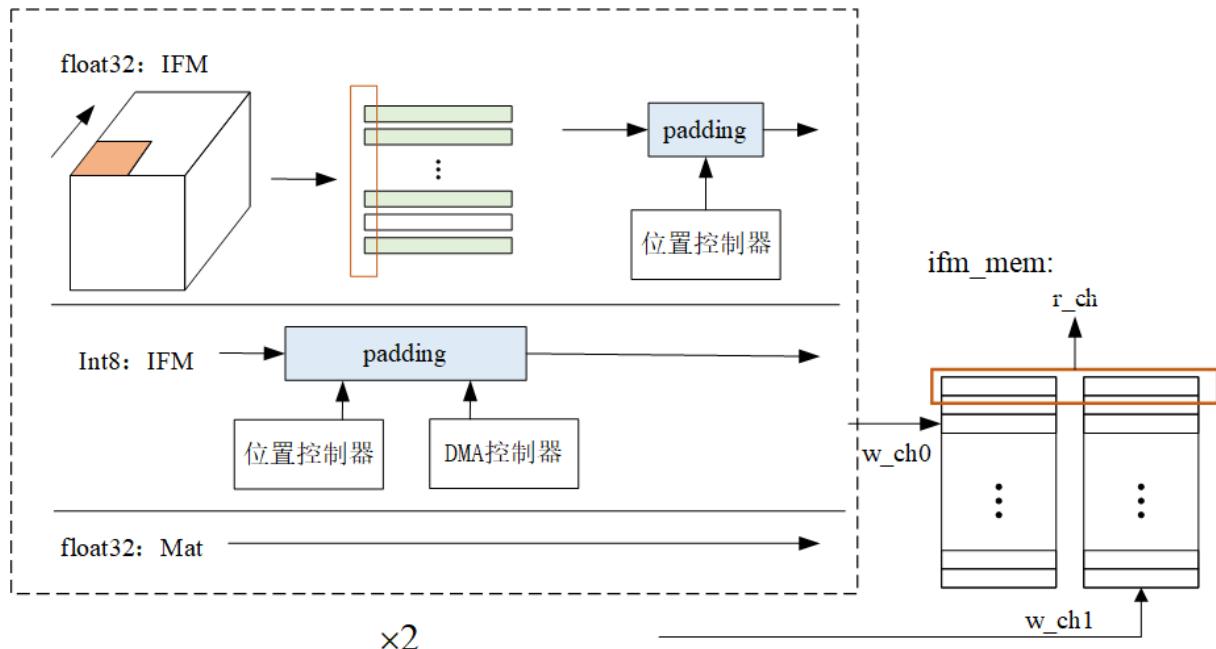
包含两种情况，第一种情况，整个输入特征图共用一个 scale，该数据可以通过 AXI-LITE 直接传输；第二种情况，输入特征图的最高维度共用一个 scale，此时缩放因子的个数等于最高维的尺寸。如一维情况下，scale 的个数等于其列数；二维情况下，scale 的个数等于其行数；三维情况下，scale 的个数等于其通道数。

量化缩放因子寄存器 (quant_scale_reg)															
offset : 0x002C															
高 16 位 [31:16]															
低 16 位 [15:0]															

- [31:0] scale_in, float32 数据格式

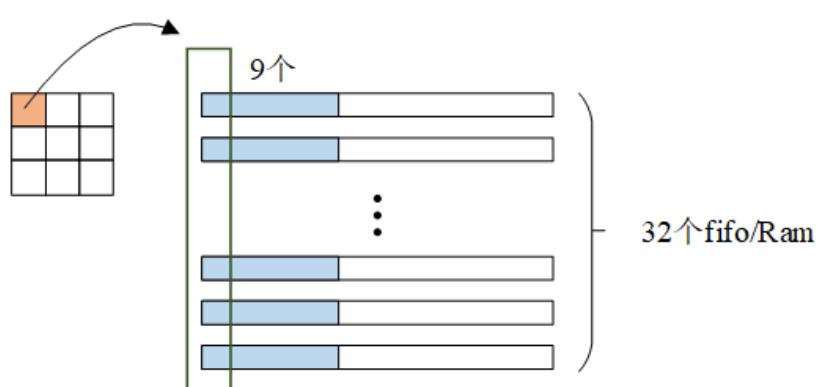
输入数据类型：

- 1) 输入 int8 的 conv ifm, N(WH)C 格式
- 2) 输入 int8 的 conv-depthwise ifm, N(WH)C 格式
- 3) 输入 float32 的 conv ifm, NC(WH)格式
- 4) 输入 float32 的 conv-depthwise ifm, NC(WH)格式
- 5) 输入 float32 的矩阵



ifm_mem 中优先排列 ic 方向，步进 32，把 w*h 视为一个维度，单个 ram 位宽为 32*8。

为了提高深度可分离卷积计算时的 PE 利用率，设计如下结构插入 ifm_mem 和 GEMM 之间



特征图分块：

目前的开发板，uram 3.9375MB、bram 3.275MB

对范围特征图在 h 方向分块，每块的大小小于 ifm_mem_size。注意给出 padding 标志位。

同时此处的分块策略不同于矩阵分块，两种策略需要同时用到，但是有区别：

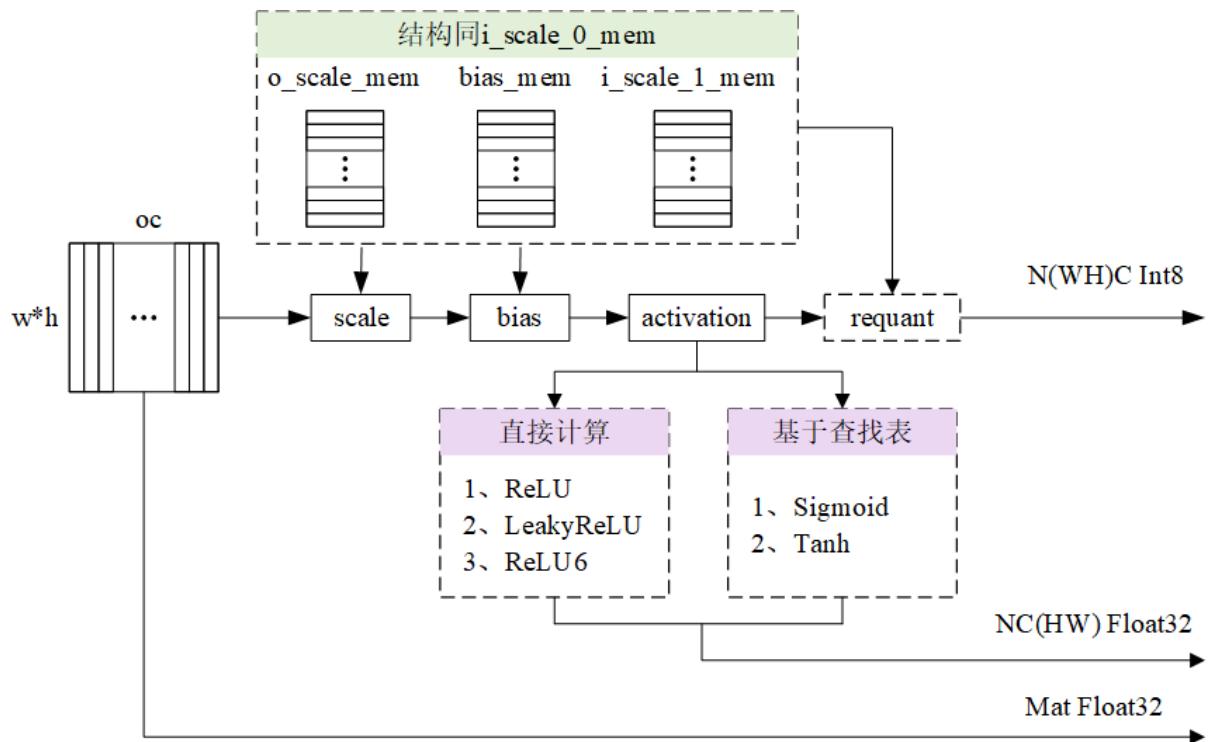
- 特征图分块，是由于 ifm_mem_size 的大小限制的，因此设计一种软件分块策略
- 矩阵分块，GEMM 的尺寸限制的，因此是一种硬件分块策略

GEMM Unit

PE

.....(肖飞豹)

Post-GEMM Unit

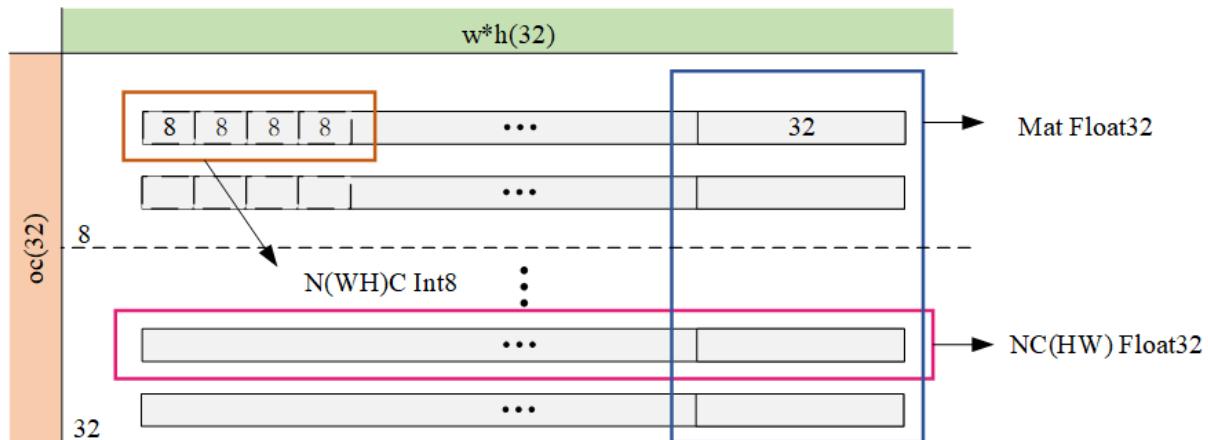


激活函数支持：目前仅支持 ReLU 和 LeakyReLU

数据加载优先级：由高到低 !!!

- i_scale_0、i_scale_1、o_scale、bias mem
- 如果涉及 Sigmoid、Tanh 查找表
- 权重
- 特征图

ofm_mem:



Pool Unit

目前只加速最大值池化和均值池化

..... (李铭宇)

Col2Im Unit

..... (吴俊杰)

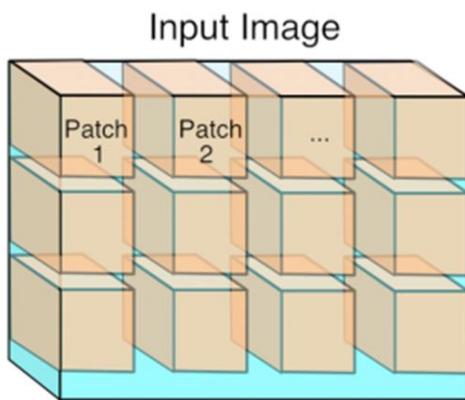
GEMM Unit

参考 TPU(256*256) 和 Gemmini(8*8, 16*16, 32*32), 脉动阵列尺寸暂设置为 64x32

周期计算：设二维矩阵每行 n 个元素，阵列尺寸为 x*x 时，dma 数据位宽为 w

权重填充(x) + 计算延时(n+x) = n+2x

特征图加载：对水平方向做数据复用， $n/3 * (x+2) / (w/32)$

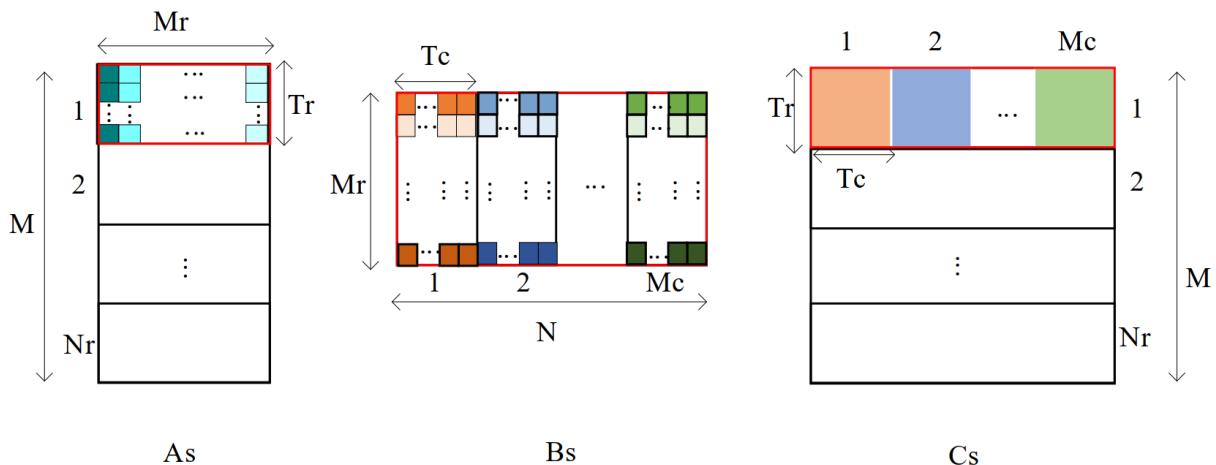


$$n/3 * (x+2) / (w/32) < n+2x \Rightarrow w > 32n(x+2) / 3(n+2x)$$

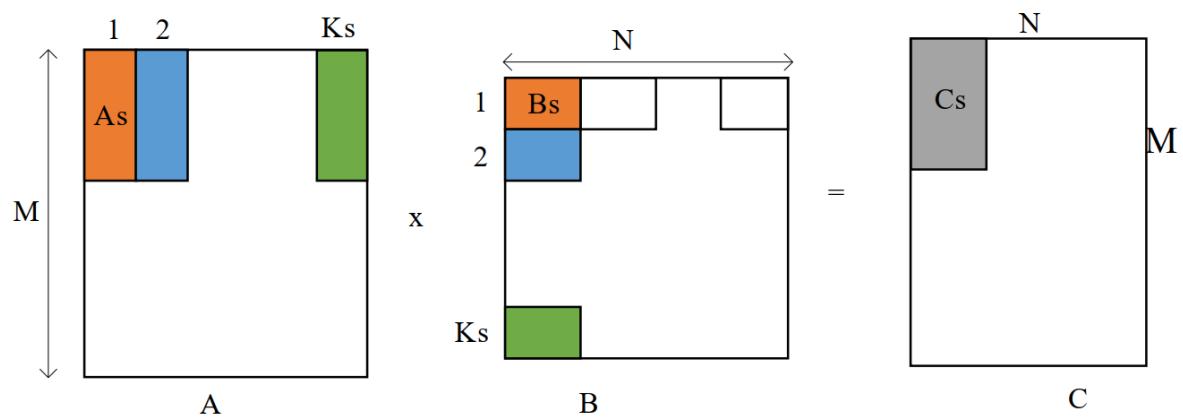
$$\text{取 } x = 32, w > n*32*34 / 3*(n+64)$$

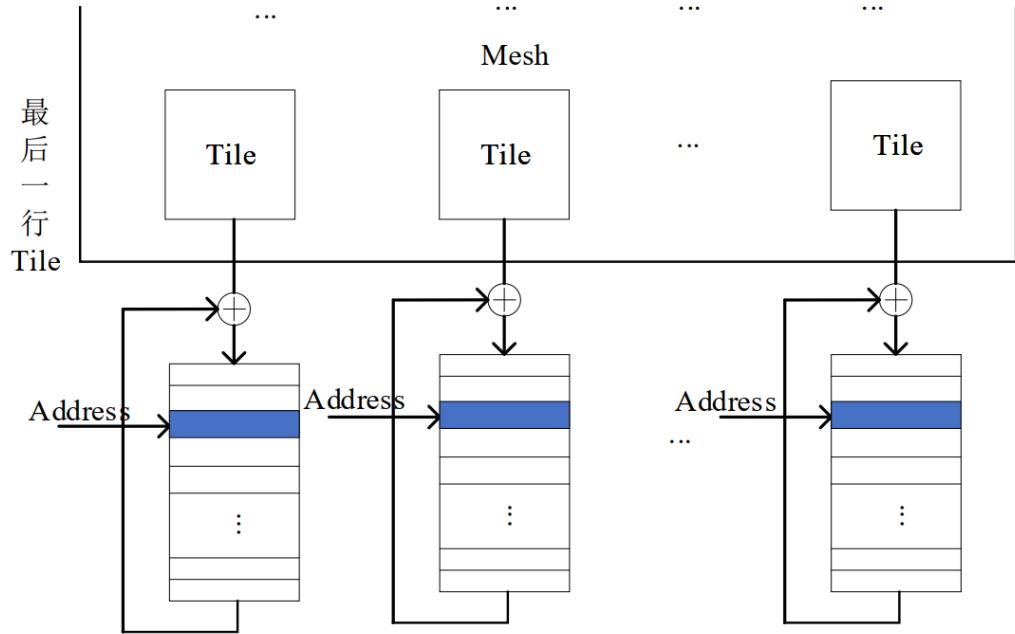
$$\Rightarrow w > n*32*34/3n = 363 \Rightarrow w = 512$$

矩阵分块：



累加器设计：

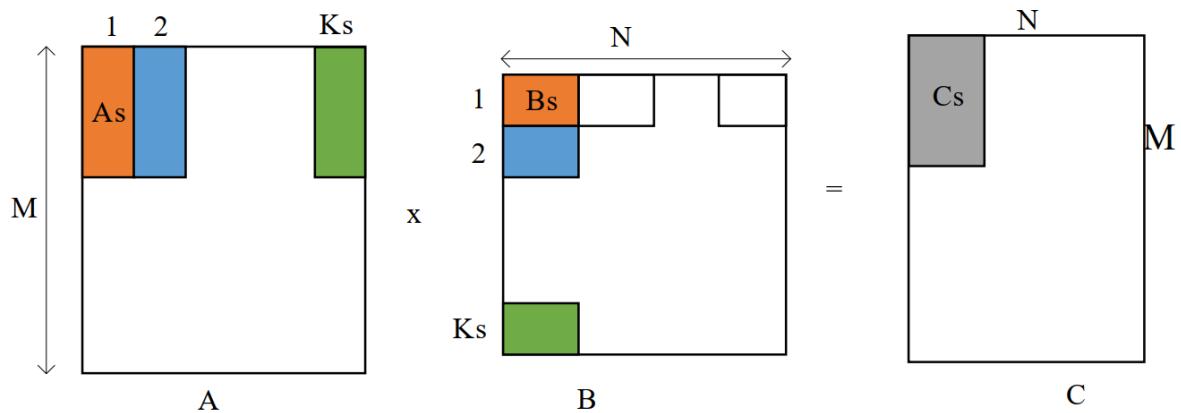




! 累加器模块要加 pingpong

之后进行逐元素的 scala , bias , activation , (requantize)

col2im Unit

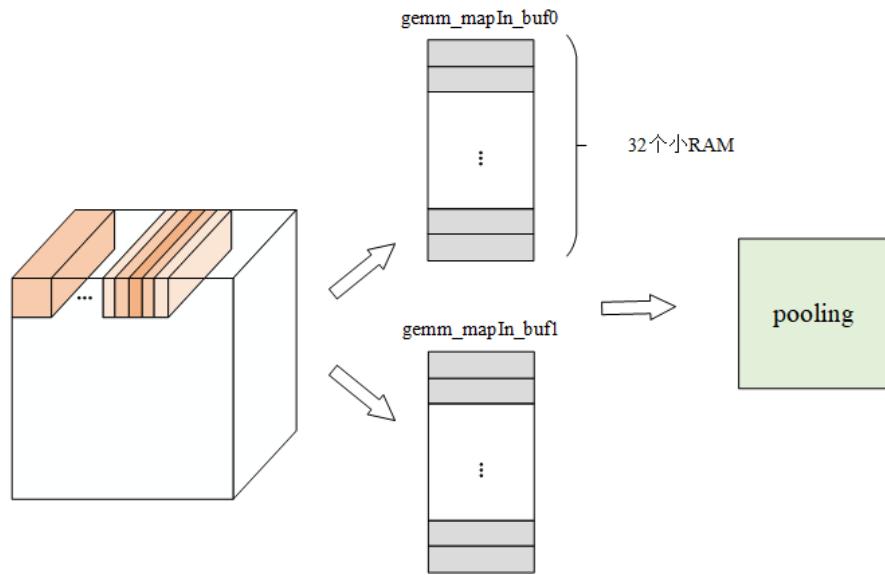


C 矩阵的列方向为输出特征图的一个通道，为了方便写回数据，应先对列方向进行输出。

首先需要从 CPU 获取每个通道的首地址，个数和卷积核的数量相同，如果通过 AXI-LITE 传输效率太低，可以 CPU +> DDR，然后从内存直接加载到片内 RAM

Pool Unit

- 目前只加速最大值池化和均值池化
- 复用矩阵乘特征图的输入 pingpong buffer
- 每次缓冲大小为 $32 * w * \text{kernel_h}$

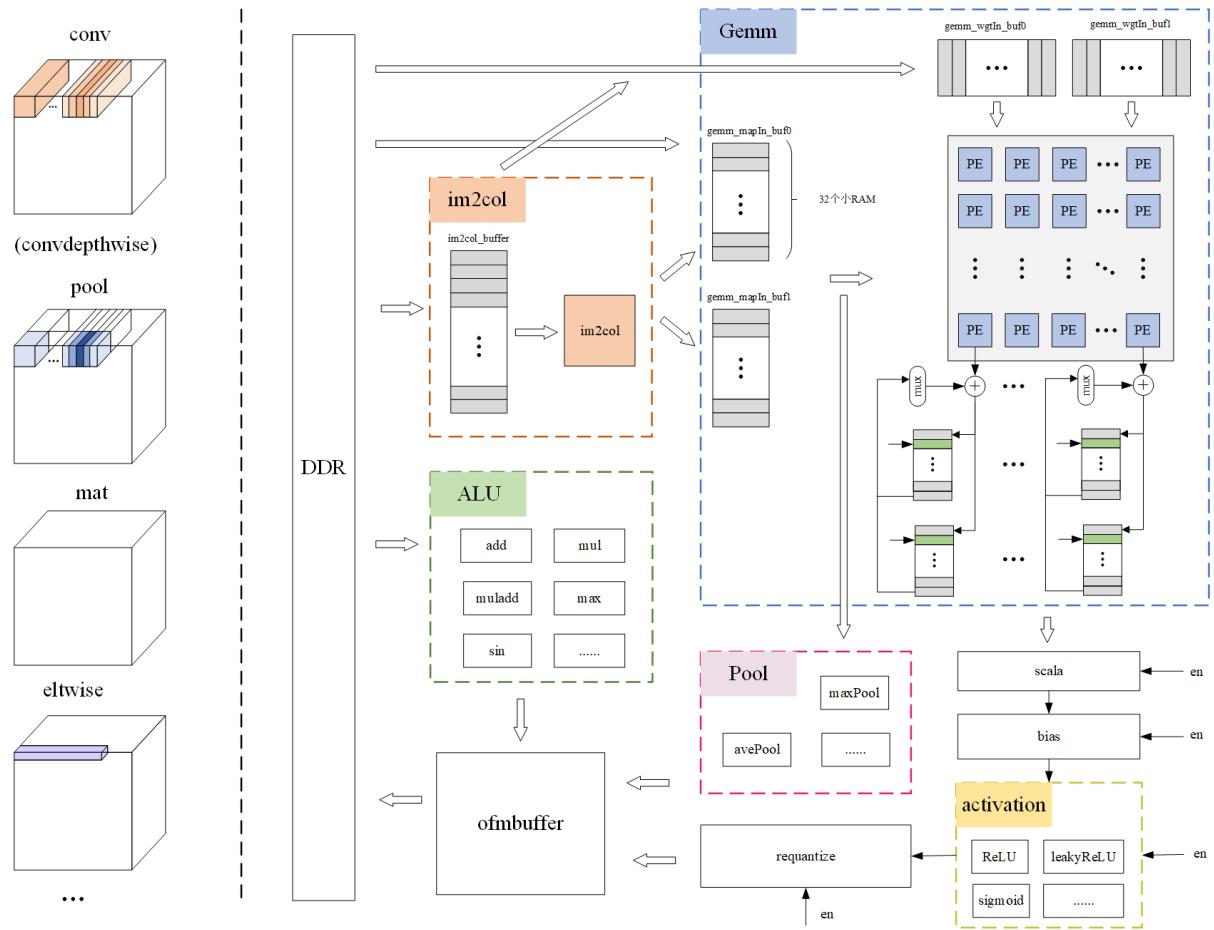


池化单元由 32 个子单元构成，实际上就是 地址控制器 + FPU 组成

四、算子映射

- convolutiondepthwise, 映射到 GEMM 中，会产生大量的零
- 矩阵乘法，需要矩阵转置单元
- deconvolution, 映射到 GEMM 中，会产生大量的零
-

五、整体数据流图



六、模拟器

- 采用 C++结合 Verilator 搭建模拟器，极大加速开发效率
- 可以实现软硬件协同仿真
- 同时还可以进行随机测试，验证硬件准确性
- 将模拟器作为后端接入 NCNN 推理引擎，可以在模拟器上运行神经网络、以及自己编写的 DSP 库、科学计算库

verilator 输入输出数据格式：https://codesearch.isocpp.org/actcd19/main/v/verilator/verilator_3.900-1/include/verilated.h

七、多核策略

Q1 : IOMMU

协处理器的大量数据传输需要采用 DMA 方式，但是由于 linux 系统的原因，地址采用虚拟地址的形式。因此，对于软件开发者而言，内存中的数据逻辑地址是连续的，物理地址是不连续的。原因在于 cache 与 DDR 之间采用了 MMU。

而对于外设而言，数据的传输不会经过 MMU 单元，因此我们需要给自己的外设也添加一个 IOMMU。例如对于 ARM 处理器，采用了 SMMU 实现了 IOMMU 的功能。

对于 ZCU104 而言，其包含了 4 核 A53，同样包含 SMMU。

<https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841981/Zynq+UltraScale+MPsoc+SMMU>

Q2 : Cache 一致性

软硬协同最常见的一个问题就是 cache 的问题，不管我们是裸核开发还是在 Linux 系统中，这个问题都不可避免。对于不同的开发方式，cache 一致性问题都可以从软件和硬件两个方向去解决。

软件实现方式：对于裸核开发而言，直接调用 API 冲刷 cache 即可。对于 Linux 系统而言，也可以软件冲刷 cache，但是由于在驱动层，实现起来可能有点复杂。

硬件实现方式：需要有硬件实现的 CCI 单元，通过配置 AXI 总线实现 cache 一致。对于 Linux 系统，虽然有硬件实现，但是也要有对应的软件支持。

<https://www.cnblogs.com/hankfu/p/11950376.html>

Q3 : Linux 驱动开发

目前主流的开发方式就是采用设备树的形式，但是需要自己编译内核，Boot 和根文件系统等，知识盲区有点多。第二种就是基于内核驱动提供的 API 编写，缺点有点臃肿，优点是可以直接由内核源码和驱动一起编译，使用时直接挂载外设。

目前先不考虑

八、DSP 库

一、DFT

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \vdots \\ X_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix}$$

FFT 本质就是 DFT 的加速算法，使用 DFT 可以同样满足需求

二、FIR

$$y(n) = b_0 x(n) + b_1 x(n-1) + \cdots + b_k x(n-k) = \sum_{k=0}^{M-1} b_k x(n-k)$$

注：待 GEMM 实现完成后，在考虑算法的映射问题

九、科学计算库

一、矩阵求逆

基于 Cholesky 分解实现：

$$A^{-1} = (LDL^H)^{-1} = (L^H)^{-1} D^{-1} L^{-1} = (L^{-1})^H D^{-1} L^{-1} = P^H C P$$

采用这种优化的 Cholesky 分解方法之后，求 A 的逆矩阵问题就被分解为三个子问题。

1、求出 A 分解后的矩阵 L、D 和 L^H。

2、分别求 L 和 D 的逆矩阵 P。

3、将得到的矩阵按公式中的顺序进行相乘，得到的结果便是 A 的逆矩阵^[26]。

基于上述算法过程设计矩阵求逆 IP 核流处理结构如图 3.6 所示。

相比于 QR 分解和 LU 分解，可行性最大。

第一步就是控制地址求出 L 和 D，第二步需要设计电路实现。第三步利用 GEMM。

二、矩阵转置

最好也采用硬件实现

十、仿真器结果

```
[ log ]: OMM_W = 20, OMM_H = 20, OMM_C = 255
[ log ]: kernel = 1, stride = 1
[ log ]: padding_left = 0, padding_right = 0, padding_top = 0, padding_down = 0
[ log ]: bias_en = 1, requant_en = 0, act_type = 0, act_param = 0.000000
[ log ]: quant_scale = 12.10376, dequant_ptr = 0x5623dd2f6200, bias_ptr = 0x562
3dd35a200
log: Yolov3DetectionOutput::forward
3 = 0.89808 at 456.78 81.88 235.81 x 80.99
3 = 0.88636 at 522.92 97.10 107.13 x 53.45
2 = 0.31109 at 61.16 77.38 531.25 x 422.51
[ log ]: imshow save image to image.png
[ log ]: FREQ = 200MHz, CYCLES = 5870602, TIME = 29.353012ms
[ log ]: POOL_TIME = 4.143480, 14.11603%
[ log ]: GEMM_TIME = 25.209520, 85.88393%
[ log ]: ALU_TIME = 0.000000, 0.00000%
[ log ]: simulation time: 34:33.62
make[1]: 离开目录 "/home/qin/ncnnAccel/simulation"
(nv38) qin@qin:~/ncnnAccel/simulation$
```

Yolov3_tiny

416x416 34FPS

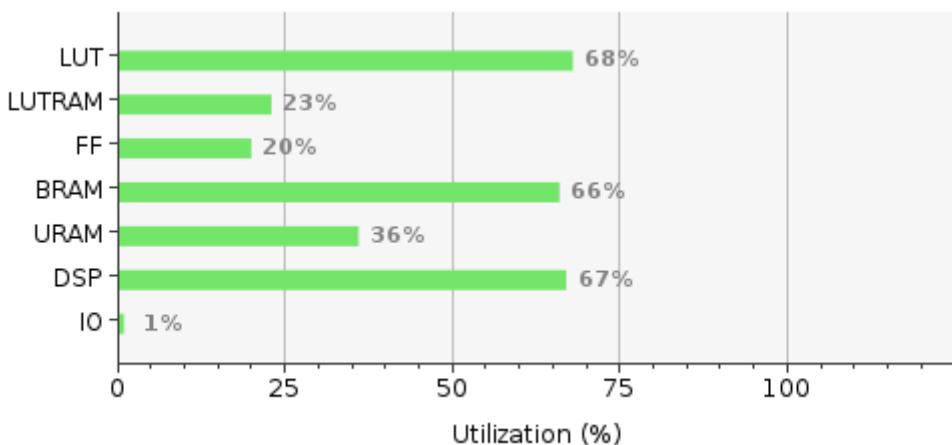
416x256 55FPS

256x256 90FPS

Yolov3

十一、综合结果

Resource	Utilization	Available	Utilization %
LUT	233128	341280	68.31
LUTRAM	41873	184320	22.72
FF	137950	682560	20.21
BRAM	492	744	66.13
URAM	40	112	35.71
DSP	2369	3528	67.15
IO	3	328	0.91



十二、可以优化的点

- 1、ifm mem 设计成数据预取的结构。
- 2、没有充分利用权重的复用，应该写一个程序算一下，权重缓冲区的大小、特征图大小、输出缓冲区的大小以及脉动阵列的尺寸和个数对推理速度的影响。
然后增加脉动阵列的个数，从而增加权重缓冲区的复用。
- 3、PE 的 MAC。
- 4、多核性能需要对 NCNN 进行调优（目前不考虑）。
- 5、激活、池化、偏置、量化、反量化 fp32 单元调优。

分工

肖飞豹：

- 1、PE 单元编写，每个单元支持 1 个单精度浮点 fp32,int32 和 2 个 int8 的乘加运算。同时可进行参数化配置，根据配置使能三种数据格式中的一种或者几种。

成舒婷：

- 1、负责硬件端 im2col+gemm 主模块的编写，软件端 ncnn 算子适配

秦海鹏：

- 1、负责硬件端 ALU 矩阵计算单元的编写，典型数学函数的实现
- 2、与其他人配合完成其他硬件模块的编写

吴俊杰：

- 1、负责完成科学计算需要的矩阵运算单元。

李铭宇：

- 1、负责硬件端池化单元的编写，要求支持最大值池化和均值池化，数据格式 float32
 - 2、负责硬件端激活函数单元的编写，要求支持 ReLU、leakyReLU，数据格式 float32；尽量支持一下 sigmoid。

刘奥：

- 1、与肖飞豹配合完成，可以尝试一下门级实现。