**总体介绍**

Diff\_core目前实现的是仅支持特定卷积操作的（即3\*3 stride1和5\*5 stride2），weight 默认8bit稠密，支持activation 8bit稀疏和4bit稠密两种模式的CNN卷积加速器，weight为8bit稠密。配置H，W，Ci，Co和bit\_mode，kernel\_mode即可完成一个一层CNN卷积。

目前仅实现了ReLU作为激活函数，不支持padding，pooling，DW卷积，但是可以较容易的增加这几种功能。相比之下增加其他卷积核大小支持较为困难。

注意window\_size不可简单重构，需要更改部分较多。

**稀疏处理方法**

主要应用1bit guard map，即每个正常位宽（8bit/4bit）的activation配备对应1bit的guard，用以指示该activation是否为0，单独存储。输入activation需要CPU生成guard存储在DDR指定位置，通过指令告诉处理核心（也可以简单片上生成，但是现在没有实现）。每层运算中会片上生成guard用于下一层计算。

运算中PE会根据guard map跳过0的计算节省周期（具体在PE中介绍），但是每window\_size个activation会浪费一周期准备数据。数据在计算中还是以dense形式暂存，一层计算结束写回时，写回生成的guard map，并且密集存储非0的activation。

**数据流**

不同行PE分配不同Co数据，不同列PE分配不同Ci数据。每一次分配中，PE计算一个windows\_size的activation和一层一个kernel的所有乘机。例如下图所示的一层网络输入，第一次assign和第二次assign的数据如图所示。





注意到activation每列是unicast，weight对于行列都是multicast，但是在一个Ci中是暂留在PE中的，由于weight认为是dense的，每一个Co，也就是每一行PE的行为是周期精确地相同的。对于一个PE，当一个Ci计算完成，会分配下一个属于这一列的Ci，直到计算完成所有Ci，与此同时加法树和AWB模块会累积部分和，直至计算完成一整个Co，由这一行对应的AWB写回一层的Co数据，之后又会开始相似的新的Co直至这一层计算结束。

**模块介绍**

Diff\_core主要由PE矩阵（PEM）、PE列控制器（PEC），加和写回模块（AWB），加和写回控制器（AWBC），顶层控制器（TC）和片上存储（BUF）构成（用于存储差分参考帧的存储没有画出）。PE阵列行列数可配置，但应该保证行数小于等于列数。

PE和PEC

PEC每个时间块发送一个window\_size的guard map给一列PE，PE每周期向BUF请求一个activation，计算一个activation所贡献的所有部分和（psum），根据guard map中非0值的位置在3\*（window\_size + 2）大小的寄存器组中更新psum值。



PE对于不同kernel采用同构，对于3\*3卷积，每周期计算9次乘法和6次加法。对于5\*5卷积，依据activation在feature map中的位置（奇数行奇数列，奇数行偶数列，偶数行奇数列，偶数行偶数列四种情况）分别计算9、6、6、4次乘法和6、6、4、4次加法，但是每个PE中的硬件固定9个乘法器和6个加法器，乘法后psum的存储比特数设置为32（有点大，可以作为参数调节）。



目前实现的window\_size为6，对于3\*3kernel，每个时间块生成8\*3（**A**）个psum，其中后2\*3（**A**）将留在PE中在下一个时间块内加和，前6\*3（**A**）个psum放入fifo。对于5\*5，每个时间块生成5\*3（**A**）个psum，其中后2\*3（**A**）和psum和下个时间快加和后变成8\*3（**A**）psum，即每两时间块写一次fifo。

这里需要注意，每个时间块由于稀疏度的不同，所需要的时钟周期也不同，因此Fifo在这里用来平衡sparsity分布不均的情况，这里不能采用更好的方法片上分配load，但是由于不同Ci的sparsity有一定随机性，可以调整fifo深度改善性能。

之后当同行不同列的fifo均不空时，同一行PE的3\*6psum对应位置由加法树加和送至AWB。

AWB

AWB负责暂存并加和psum，执行差分、非线性操作并且写回。

psum在AWB中的加和分为两部分，第一部分是连续3行（**B**恰好两种kernel计算pattern决定了都是3行）activation生成的3（**B**）\*6\*3（**A**）psum中对应行相加，得到一个Co中一部分Ci的加和结果；第二部分是所有部分Ci的psum加和起来生成一个完整的Co中的一层。

第一部分示意图如下，例如W31与卷积核相乘之后（5\*5kernel还需要加和）得到三行psum，W31-1、W31-2、W31-3，其中W31-1和已经在BUF存储的W21-2和W11-3加和生成一个Co中一个Ci的前6个结果，下一个时间块加和的是W32-2、W22-2、W12-2。硬件实现需要能够存储3（**A**）\*两行（**B**只需要存连续三行中的前两行）的BUF和6\*3个加法器；



第二部分需要能够存储完整的Co中（所有Ci）的一层的BUF，并且需要pingpong。

之后执行差分操作和ReLU（具体在4-bit和差分部分介绍），将每一个非零结果写回，并且每window\_size生成并写回一组guard map。

4bit

4bit操作目前仅仅实现稠密操作，实现方法是将两个Ci对应位置的activation高低位拼接，（第一层输入是直接8bit算还是要分成8bit和四比特？）在写回时需要额外的控制逻辑。其他设计中，需要乘法器支持和AWB中需要多6个32/2比特的加法器。

上面这种方法，实现稠密比较简单，但是要应用sparsity的话几乎不可能，后面设计稀疏4-bit时可能需要在6（windows\_size）个8bit中存12\*4bit连续的activation。每行每列多一个6输入LUT换取1.655倍的提升？

差分

目前实现简单版本的即单帧整网络计算，即片上保存所有层网络运算的结果作为参考帧，BUF需要与行数对应的bank数目。

**外部支持**

Weight存储模式

Weight总体规则按照由下至上，由右至左的顺序存放，5\*5较为特殊特殊还需按照深蓝、浅蓝、浅绿、橙色的顺序存放（即44, 24, 04, 42, 22, 02, 40, 20, 00, 43, 23, 03, 41, 21, 01, 34, 14, 32, 12, 30, 10, 33, 13, 31, 11）。Weight bank width为25\*8bit，因此3\*3卷积的weight每个bank存两个（9\*8bit）

