CNN-HWPE_spec_v1.1.6

——陈浩、陈强
2018-05-29

目标

用于 CNN 硬件加速的协处理器，挂在 RISCV 指令集的蜂鸟 E200 MCU 的 EAI 接口上。

特点

支持 Kernel 的 RS 维度大小最大 11×11

输入层卷积 kernel：C=3

中间层卷积 kernel：Int8 C=8N，Exp4 C=16N，Ter2 C=32N

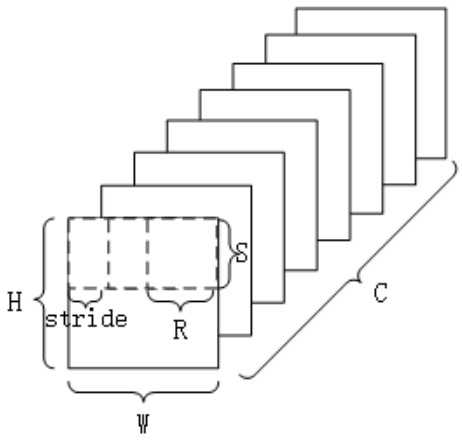
每周期完成 16 个乘积操作，乘积的操作数是 64bit

支持 INT8，EXP4（指数 scale 的 4bit），Ter2(Ternary2)

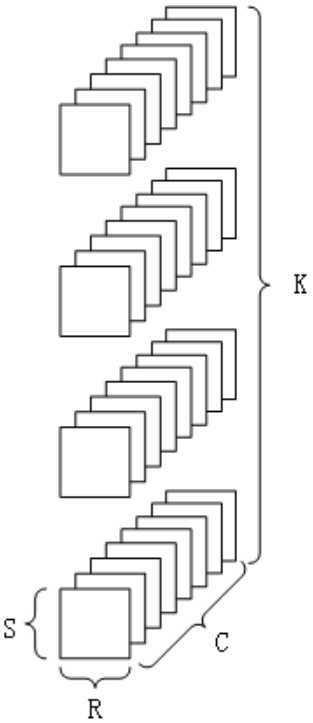
注：N 指代整数 1,2,3… 文中后续 N 指代数据位宽

理论基础——卷积转矩阵乘积

下图是卷积操作的对象特征图(Feature Map)和卷积核(Filter Kernel)的示意图，Feature Map(以后简称 Fmap)有 3 个维度，H、W、C。Kernel 有 4 个维度，S、R、C、K。

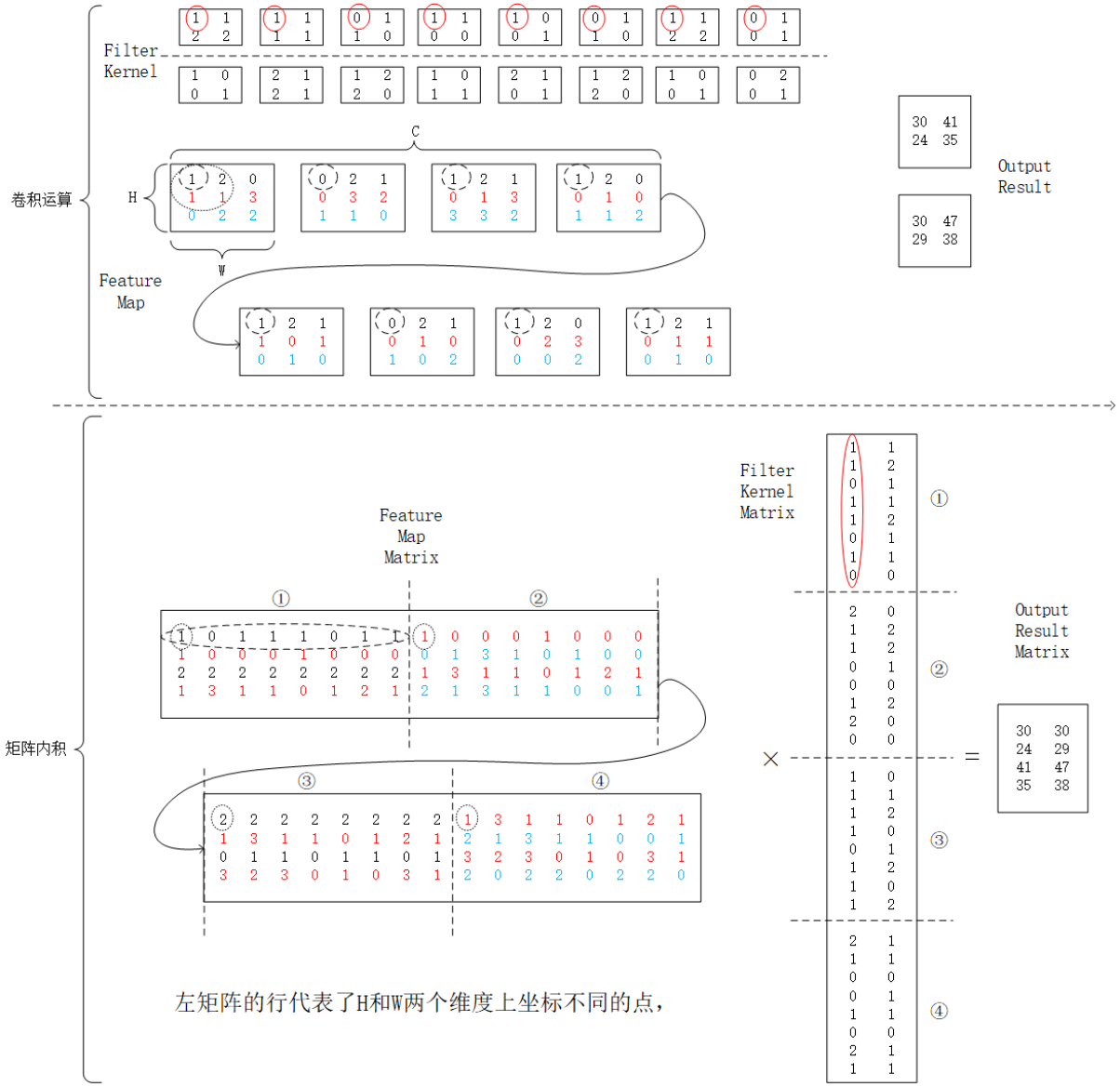


Feature Map



Filter Kernel

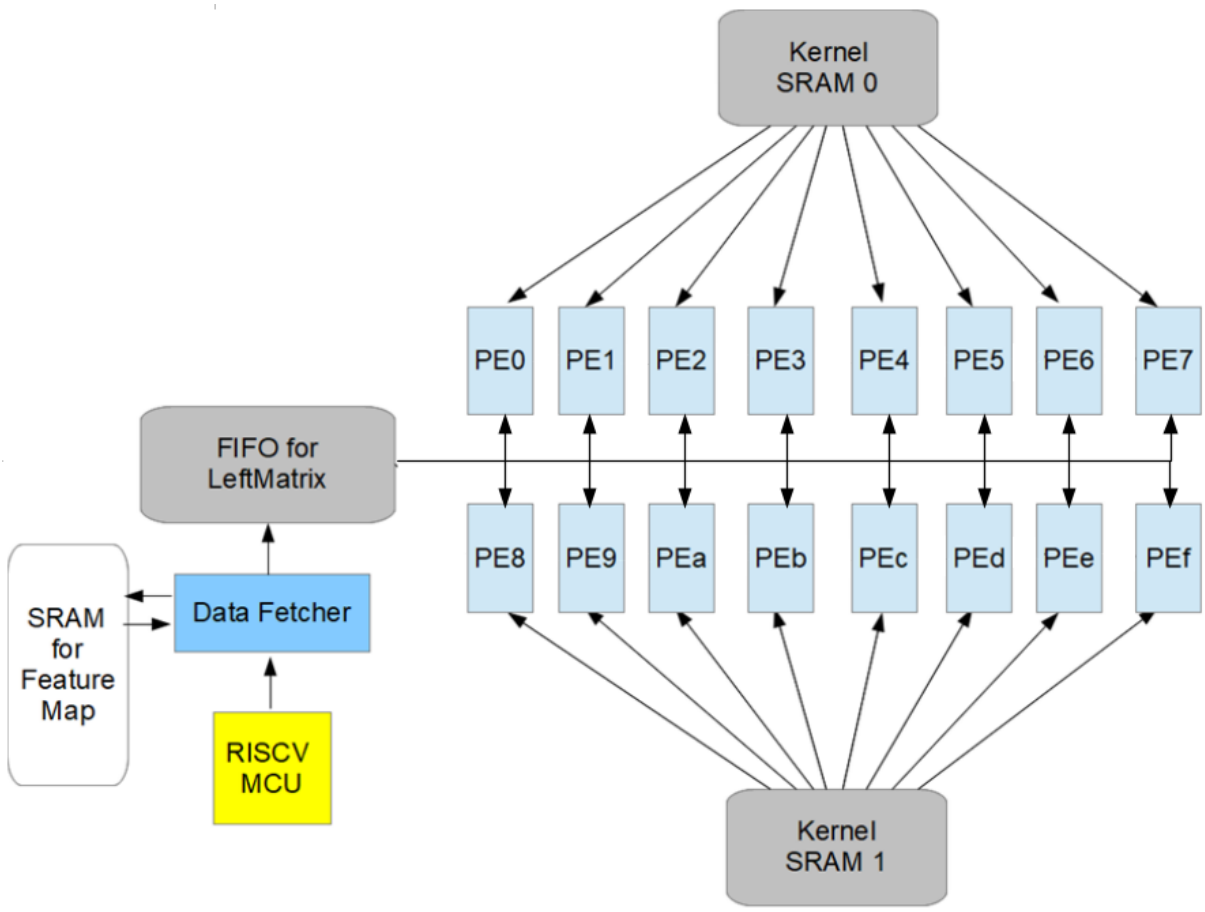
CNN 的卷积运算可以通过 im2col+矩阵乘积来实现。如下图，是 3×3×8 的特征图与 2×2×8×2 的卷积核做卷积运算的结果。将特征图按照 C→H→W 的顺序做 im2col 得到一个 4×32 的左矩阵，将卷积核按照对应的顺序做重排列得到一个 32×2 的右矩阵。可以看到左右矩阵做乘积得到的结果与卷积结果是相同的，区别只在于矩阵乘积结果是输出结果的重排列的形式。



硬件实现方案

HWPE 通过蜂鸟的 EAI 接口接入到蜂鸟的 SOC 环境。EAI 接口类似 Rocket Core 的协处理器接口 RoCC（Rocket Custom Coprocessor）。

https://github.com/SI-RISCV/e200_opensource/blob/master/doc/Hummingbird_E200_Coprocessor_Extension_Quick_Start_Guide.pdf



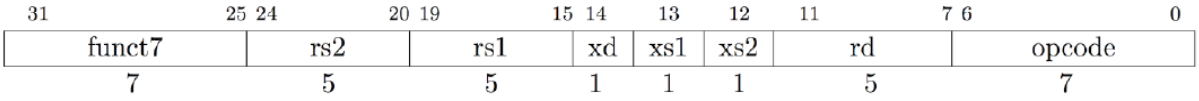
传统上 GPU 做 im2col 的方式，是把 image 的局部展开后存在一小块专门放 col 的内存里，但在我们的方案里并不是这样，im2col 的动作是通过逐步往 FIFO 里面灌数据的方式慢慢实现的，并没有专门开辟 RAM 空间来放 col。而 Kernel 是事先重排列好后存储在 SRAM 中的。

各个部件包括：

1. 一个 FIFO 用来缓存左矩阵，一个 entry 是一行，Data Fetcher 不停地向它内部 push 新数据。它按照顺序依次向外 pop 出左矩阵数据。左矩阵有 8 行，FIFO 内部 4 个 entry（分成 2 个 FIFO，一个存奇数，一个存偶数）。FIFO 带有转秩写入功能。(写入 FIFO 的时候，需要转秩的功能时，以 2 个表项为单位来 push，否则仍然是 1 个表项，pop 的时候，总是以 1 个表项为单位。2 个表项转秩 push 是为了满足 3×3×3 的卷积核的需求)。
2. Data Fetcher 接受 MCU 的命令，从 Fmap(Feature Map) SRAM 取来数据，push 到 FIFO 中。Fmap SRAM 可以是 2 个或者 4 个，或者更多，但其共有 2 套访问 SRAM 的接口，可供 Fetcher 同时取来 2 组数据。
3. 16 个 PE，每个内部保存右矩阵的一个列（64bits），以及结果矩阵的一个列（需要 8 个 32bit 的结果寄存器）。
4. 2 块大的 Kernel SRAM，每块把右矩阵的数据广播到 8 个 PE 中。

指令集

HWPE 的指令集遵守 RISCV 的定制指令扩展规则，32 位指令格式如下：



- 指令的第 0 位至至第 6 位区间为 Opcode 编码段，可以编码 custom-0，custom-1，custom-2，和 custom-3 四种指令组。
- xs1，xs2，和 xd 比特分别用于控制是否需要读源寄存器和写目标寄存器 rd。如果 xs1 位的值为 1，则表示该指令需要读取由 rs1 比特位指示的通用寄存器作为源操作数 1，如果 xs1 位的值为 0，则表示该指令不需要源操作数 1；同理，如果 xs2 位的值为 1，则表示该指令需要读取由 rs2 比特位指示的通用寄存器作为源操作数 2，如果 xs2 位的值为 0，则表示该指令不需要源操作数 2；如果 xd 位的值为 1，则表示该指令需要写回结果至由 rd 比特位指示的目标寄存器，如果 xd 位的值为 0，则表示该指令不需写回结果。
- 指令的第 25 位至第 31 位为 funct7 区间，可作为额外的编码空间，用于编码更多的指令，因此一种 custom 指令组可以使用 funct7 区间编码出 128 条两读一写（读取两个源寄存器，写回一个目标寄存器）协处理器指令。

指令列表：

Instruction	funct	rd	xd	rs1	xs1	rs2	xs2
HWPEWriteFmapAddrReg	1	addr_idx	0	addr1	1	addr2	1
HWPEWriteCfgReg	2	——	0	cfg	1	cfg	1
HWPEMatrixMac	4	——	0	W/H_count	1	H/W_stride	1
HWPEWriteAccReg	8	AccReg_ID	0	M(idx)	1	PE_ID	0
HWPEReadAccReg	16	M(idx)	1	Acc_Reg_ID	0	PE_ID	0
HWPEReLUMemWriteAccReg	32	——	0	M(addr)	1	AccReg_ID	0
HWPEReset	64	——	0	——	0	——	0

HWPEWriteFmapAddrReg 传两行 Fmap 首地址

MCU 传来的 Vrs1 和 Vrs2 是两个地址，作为两行数据的起始访存地址。rd 是 8 个地址寄存器的索引。

HWPE 有 8 个左矩阵地址寄存器 FmapAddrBase，存储 8 行左矩阵将要访存的地址，指令每次写两个地址 0-1/2-3/4-5/6-7

注意，当处理输入层的 3×3×3 的卷积时，配置 4 个地址即可。

HWPEWriteCfgReg 寄存器配置

MCU 配置 HWPE 的寄存器。

编号为 0 的 CfgReg：Conv_CH_count 和 Conv_W_offset

编号为 1 的 CfgReg：Kernel 相关信息

CfgReg0 = { Conv_W_offset, Conv_CH_count }

CfgReg1 = { K_count, AccReg_shift, Kernel_333, Layer_type, Data_type, Kernel_size }

10 5 1 1 2 4

Conv_CH_count=Vrs1[15:0] 卷积窗口中，CH 维度上的连续数据含有的 64bit 左矩阵个数

Conv_W_offset=Vrs1[31:16] 卷积窗口中跨越 W 维度时地址偏移量

Kernel_size=Vrs2[3:0] rs2 指定 kernel 的大小；

Data_type=Vrs2[5:4] 取 01/10/11/00 分别代表 ternary2/exp4/int8/uint8；11/00 时右矩阵始终为 int8

Layer_type= Vrs2[6] 指定该层是中间层还是输入层。0 中间层(C=8N) 1 输入层(C=3);

Kernel_333=Vrs2[7] 为高时指示 kernel 大小为 3×3×3。

AccReg_shift=Vrs2[12:8] 对 AccReg 转 8bit 时的移位量，0~24 之间。

K_count=Vrs2[22:13] kernel 的 K 维度/16

HWPEMatrixMac 矩阵乘运算

配置 HWPE 的寄存器，并开始矩阵乘运算直至任务完成。

{ W_count, H_count} = Vrs1。{W_stride, H_stride}=Vrs2。

H_count，16 位，在 H 维度上需要进行几次跨越；

W_count，16 位，在 W 维度上需要进行几次跨越；

H_stride，16 位，每次在 H 维度上进行跨越需要增加的地址偏移量；

W_stride，16 位，每次在 W 维度上进行跨越需要增加的地址偏移量；

注：在第一次开始运算前 PE 里的右矩阵需要提前准备好。所以该指令下发后，配置寄存器的同时，KernelSRAM 开始往 PE 里传输右矩阵。16 个右矩阵准备好时，Fetcher 正好从 FmapAddrBase[0]地址取来第一行左矩阵存入 FIFO 了。

HWPEWriteAccReg 从 MCU 寄存器写 PE 结果寄存器

将 MCU 中 rs1 的寄存器值写入到对应 rs2 的 PE 里的第 rd 个 AccReg。用来清零或者置位 AccReg。

HWPEReadAccReg 读 PE 结果寄存器到 MCU 寄存器

将 rs2 的 PE 里的第 rs1[2:0]个 AccReg 读取，写入到 MCU 中 rd 寄存器。

rs1 的 MSB 为 **HWPE 使能标志位**，为 1 时，指示 HWPE 将在此次 AccReg 搬运完成后，继续卷积任务。

HWPEReLuMemWriteAccReg 将 PE 的结果寄存器写回到 MCU 的存储器

将属于同一个输出点的 16 个 channel 的数据全部做 ReLU 和转 8bit，再写回存储。

MCU 的 rs1 寄存器给将要写入的存储地址；rs2 指定 PE 的 AccReg_ID(0-7，PE 中的 8 个 32bit 累加寄存器)。

HWPE 对 16 个 PE 的第 rs2[2:0]个 AccReg 的 32bit 数据做 ReLU 再转换成 8bit 数据连续写出。共 4 个周期完成。转 8bit 时的移位量根据配置寄存器 CfgReg1.AccReg_shift 字段完成。

rs2 的 MSB 为 **HWPE 使能标志位**，为 1 时，指示 HWPE 将在此次 AccReg 搬运完成后，继续卷积任务。

注：PE 运算的结果的处理有两种方式，一是 MCU 写回存储，一条指令会将属于一个输出点的 16 个通道结果写回，写回之前做 ReLU 和转 8bit。二是通过 MCU 对 PE 结果寄存器的读取将结果返回到 MCU 的通用寄存器中，再由 MCU 进行后续操作。两种操作对应的两条指令——HWPEReLUMemWriteAccReg / HWPEReadAccReg

数据类型

支持 INT8，EXP INT4（指数 scale 的 4bit），Ternary2

左矩阵可配置为 INT8（Data_type=2'b11）或者 UINT8（Data_type=2'b00）。

EXP INT4 表示范围为-64、-32、-16、-8、-4、-2、-1、0、1、2、4、8、16、32、64，MSB 为符号位，低三位为 0 表示 0，其余编码减一表示阶码。乘法时阶码相加即可（对 0 特殊处理），加法时先做 one-hot 译码，后做正常的加法。

$$\begin{cases} (-1)^{a_3} \bullet 2^{a_2a_1a_0-1} & a_2a_1a_0 \neq 000 \\ 0 & a_2a_1a_0 = 000 \end{cases}$$

真值表如下：

Hex	Binary	Value	Hex	Binary	Value
0	0000	0	8	1000	0
1	0001	1	9	1001	-1
2	0010	2	A	1010	-2
3	0011	4	B	1011	-4
4	0100	8	C	1100	-8
5	0101	16	D	1101	-16
6	0110	32	E	1110	-32
7	0111	64	F	1111	-64

Ternary2，2'b00=0；2'b01=1； 2'b11=-1。

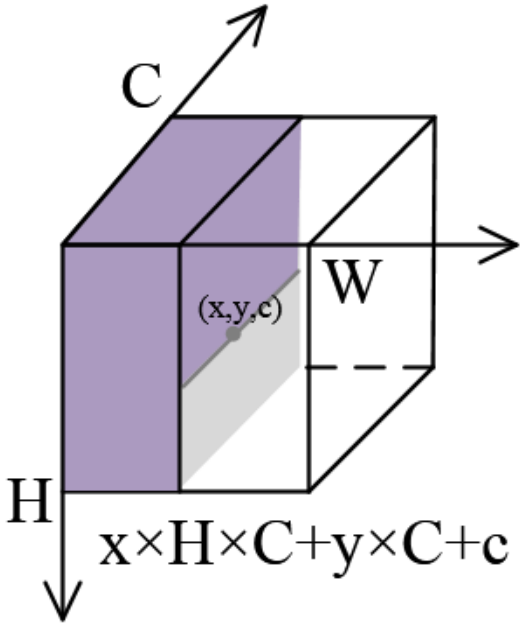
数据存储格式

Fmap SRAM

Fmap 的 SRAM 有 2 套访问 SRAM 的接口，分别存储 HW 平面的左半和右半部分。HWPE 需要有两个访问 SRAM 的接口，能同时取来两个 entry 的数据。SRAM 的个数可以是 2 个或者 4 个或者更多。

Fmap SRAM 的数据存储按照先 C 再 H 后 W 的方式存储。即 SRAM 地址按照如下计算给出:

x×H×C+y×C+c　0<=x<=W，0<=y<=H, 0<=c<=C



特例：对于输入层，当其卷积核是 3×3×3 的时候，需要将输入数据按照特殊的数据格式存储。

将 HW 平面两个不同的局部位置记为 P0 和 P1，则这两个位置的点在前面提到的 CHW 的存储顺序下不是相邻的，如下：

P0W0H0C0 P0W0H0C1 P0W0H0C2 P0W0H1C0 P0W0H1C1 P0W0H1C2 P0W0H2C0 P0W0H2C1

P1W0H0C0 P1W0H0C1 P1W0H0C2 P1W0H1C0 P1W0H1C1 P1W0H1C2 P1W0H2C0 P1W0H2C1

可以想象把一块 Fmap SRAM 里存储的 HW 平面再拆分成上下两部分，上下平移后重叠放置在一起，P0P1 就是位置重叠的两个部分，如果把 P（Part）这个维度塞在最里面，在 memory 中彼此临近。再按照 CHW 的顺序即是如下的存储顺序：

W0H0C0P0 W0H0C0P1 W0H0C1P0 W0H0C1P1 W0H0C2P0 W0H0C2P1 W0H1C0P0 W0H1C0P1

W0H1C1P0 W0H1C1P1 W0H1C2P0 W0H1C2P1 W0H2C0P0 W0H2C0P1 W0H2C1P0 W0H2C1P1

这就是在 3×3×3 的 kernel 下的特殊存储方式，两个不同局部要存储在一起，然后才是 CHW 的方向。

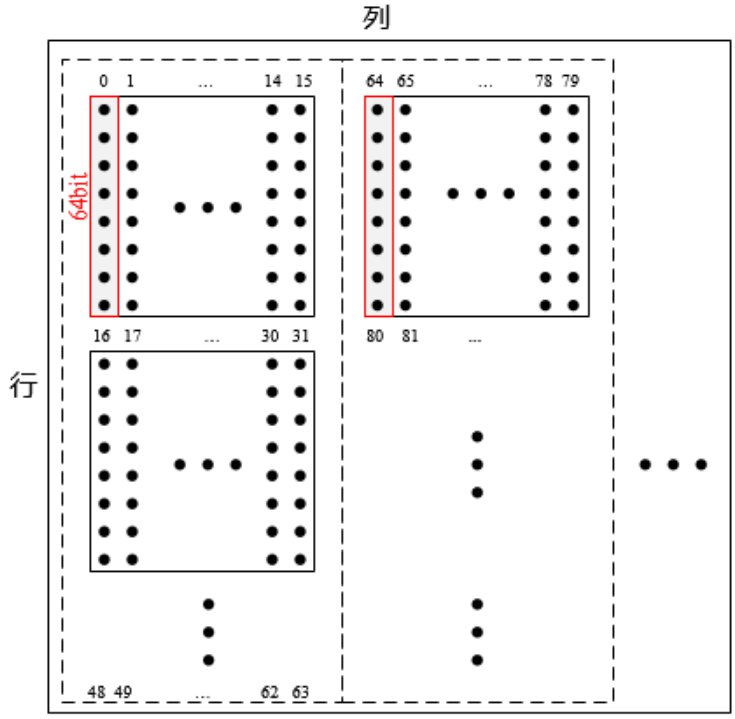
Kernel SRAM

Kernel 的 SRAM 里的数据是提前重排列好的，一个“整列”是一个 Kernel 的 RSC 三个维度，总共有 K 个整列，对应 Kernel 的 K 维度，输出 K 个 channel。当 FIFO 中的一行左矩阵的实际数据没有填满 64bit 时(当 Kernel 为 5×5×3 或 7×7×3 等时，此时存在向上取整)，对应的右矩阵相应位置填 0，也是事先就存储好的。

注：“整行”说法都指重排列后的整个左矩阵的一行，是卷积窗口元素的重排，对应于 Kernel 的 RSC 维度排列，而不是在 FIFO 中的位宽为 64bit 的一行左矩阵；“整列”的说法指重排列后的 Kernel 的 RSC 维度，是整个右矩阵的一列，而不是在 PE 里中的位宽为 64bit 的一列右矩阵；

对于重排列好的 kernel 矩阵在两块 Kernel SRAM 中如何存储。以 16 列 64bit 为块，先行后列。16 列 64bit 在存储的最里面，然后是下一块 16 列 64bit，将 R×S×C×(0-15)这 16 “整列”存储完毕后再跨列存储下一个 16 列整列 R×S×C×(16-31)维度，直到所有 R×S×C×K 都存储完毕。两块 Kernel SRAM 各存储 16 列里的前 8 列和后 8 列。

在做矩阵乘法时，两个 Kernel SRAM 每周期更新对应 PE 的右矩阵参数，直至输出点的卷积完成。



Fmap（N 位宽）和 Kernel（M 位宽）的数据位宽需要相同，暂不支持两者数据位宽不同的情况。

CNN 卷积大小支持

HWPE 支持输入层和中间层的卷积，非卷积和全连接层的计算由 MCU 支持。Kernel_size 寄存器为 4 位，故支持的常用的 kernel 的大小最大 11×11。

对于**中间层**， Fmap 按 C 存储，对 C 的要求使得数据是正好对齐的，每个周期都可以取来指定地址存储器中对齐的 64bit 数据为一行左矩阵。mac 利用率为 100%。

对于**输入层**，支持的 kernel 的通道 C 为 3，大小不限，可以是 3×3、5×5、7×7、11×11 等常用大小。只要配置好相应的配置寄存器、准备好补零的重排列好的 kernel 数据即可。不同的 kernel 大小，mac 利用率不同。

11×11×3：数据在跨 W 维度时存储是分开的，但 C 和 H 两个通道是相邻存储的，此时可连续取 11×3=33 个字节，向上取整到 40 个字节存入左矩阵（对应的 kernel 的重排列补 7 个 0）。访存时可能为非对齐存储，需要两个周期才能取来。对两个 Fmap SRAM 访存通道同时访存，保证两个周期取得两行左矩阵。mac 利用率为 82.5%。

7×7×3：数据在跨 W 维度时存储是分开的，但 C 和 H 两个通道是相邻存储的，此时可连续取 7×3=21 个字节，向上取整到 24 个字节存入左矩阵（对应的 kernel 的重排列补 3 个 0）。访存时可能为非对齐存储，需要两个周期才能取来。对两个 Fmap SRAM 访存通道同时访存，保证两个周期取得两行左矩阵。mac 利用率为 87.5%。

5×5×3：数据在跨 W 维度时存储是分开的，但 C 和 H 两个通道是相邻存储的，此时可连续取 5×3=15 个字节，向上取整到 16 个字节存入左矩阵（对应的 kernel 的重排列补 1 个 0）。访存时可能为非对齐存储，需要两个周期才能取来。对两个 Fmap SRAM 访存通道同时访存，保证两个周期取得两行左矩阵。mac 利用率为 93.75%

3×3×3：该大小需要特殊处理，完成一次卷积需要 27 个元素，分 4 轮完成：

第 1 轮处理的元素：W0H0C0 W0H0C1 W0H0C2 W0H1C0 W0H1C1 W0H1C2 W0H2C0 W0H2C1

第 2 轮处理的元素：W0H2C2 W1H0C0 W1H0C1 W1H0C2 W1H1C0 W1H1C1 W1H1C2 W1H2C0

第 3 轮处理的元素：W1H2C1 W1H2C2 W2H0C0 W2H0C1 W2H0C2 W2H1C0 0 0

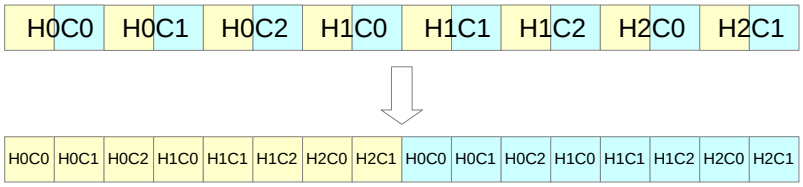
第 4 轮处理的元素：W2H1C1 W2H1C2 W2H2C0 W2H2C1 W2H2C2 0 0 0

以第 1 轮为例，我们需要从 2 个 SRAM 访存通道中读出对应于 2 个不同点的 W0H0C0 W0H0C1 W0H0C2 W0H1C0 W0H1C1 W0H1C2 W0H2C0 W0H2C1，以填满左矩阵。这 2 个不同点记为 P0 和 P1（P 是 Part 的首字母），为了填满左矩阵的头两行，需要的两行左矩阵如下：

P0W0H0C0 P0W0H0C1 P0W0H0C2 P0W0H1C0 P0W0H1C1 P0W0H1C2 P0W0H2C0 P0W0H2C1

P1W0H0C0 P1W0H0C1 P1W0H0C2 P1W0H1C0 P1W0H1C1 P1W0H1C2 P1W0H2C0 P1W0H2C1

按照前面介绍的特殊的存储格式，假如这两个点是 HW 平面对折后在一起的两个点，则其在 memory 中是彼此临近的。考虑到不对齐的情况，我们最多需要 3 个周期就能取来它们了。对于这些点，只需进行一次转秩操作，就得到两行左矩阵塞到 FIFO 里了：



在硬件实现上：FIFO 做转秩写入，两个表项一起 push；Pop 的时候，只 pop 一个——（在实现中，作为两个 FIFO 实现，一个负责奇数 entry，一个负责偶数 entry）

对于第 2 轮到第 4 轮，情况比第 1 轮要更差，因为跨 W 维度了。这时最多需要 4 个周期，两个 SRAM 读取地址，则 4 个周期能取来 4 行左矩阵所需要的数据——仍然是可以接受的。

在该 kernel 下，mac 的利用率为 27/32=84.38%。在进行 PE 的矩阵乘累加运算前，只需要配置 4 个地址寄存器，每个地址会取来 2 行左矩阵。

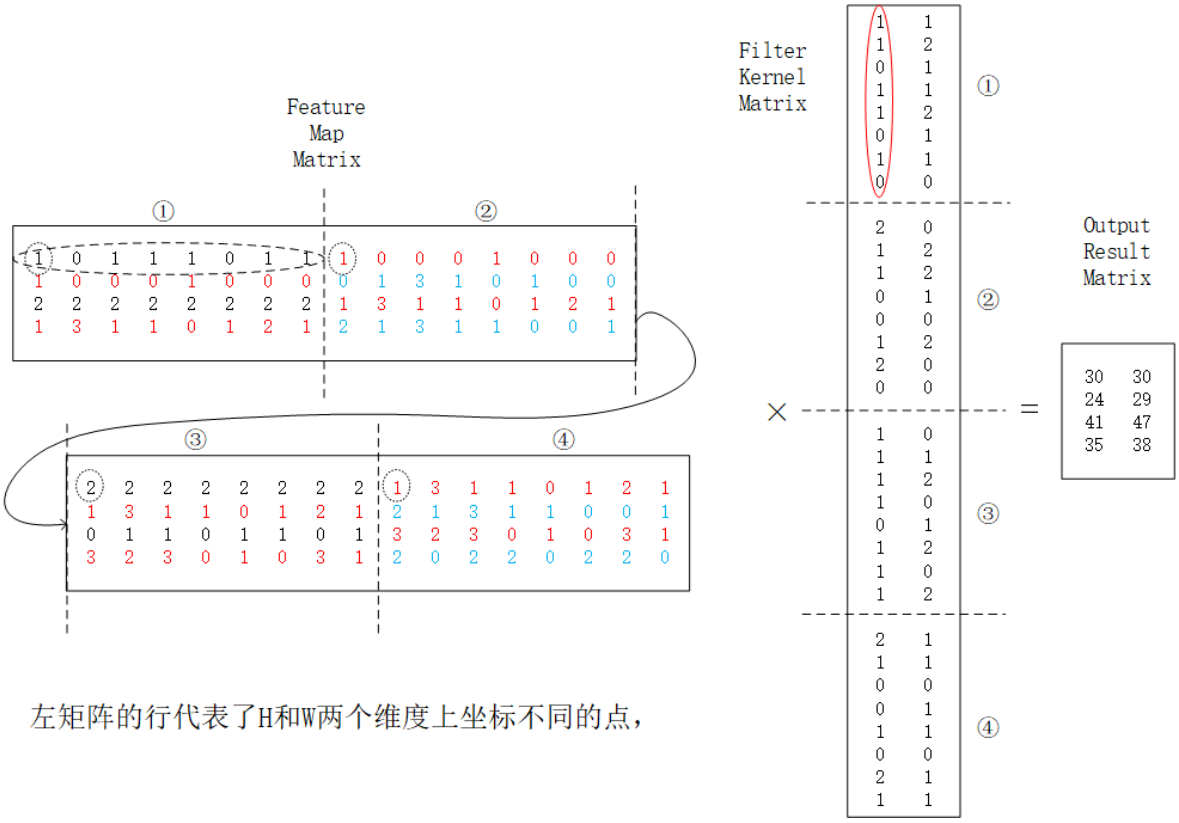
PE 的工作模式

PE 完成的是矩阵乘法，理论上 im2col 后整个左矩阵的维度为 $(C\times R\times S)\times((\frac{H-S}{stride}+1)\times(\frac{W-R}{stride}+1))$，右矩阵的维度为(C×R×S)×K。在硬件上，经过逐步往往 FIFO 里灌 64bit 数据慢慢实现 im2col，FIFO pop 出来的 64bit 左矩阵数据已经是 im2col 重排列好了。而右矩阵来自预先排列好的 Kernel。

重排列好的左矩阵，各个不同的行代表了 H 和 W 两个维度上坐标不同的点，（一列的不同行则对应于相同的 C 维度），一“整行”对应一个 kernel 的重排列。相对应的一“整行”与一“整列”的乘积得到的是 HW 平面上该点的卷积输出。

在硬件上，FIFO 里的左矩阵是 64bit 的，所以理论上的一“整行”与一“整列”的矩阵乘累加是被分解成若干轮 64bit 的矩阵乘累加完成的（每一轮都顺序处理 8 行 64bit 左矩阵）。如下图所示例子中，若数据类型为 INT8（即 N=8），im2col 后为 4×32 的左矩阵，PE 一次运算一行左矩阵 64bit 只能处理 8 个数，一“整行”被分解为 4 轮矩阵乘累加，每一轮得到一个输出点的部分乘累加。完成“整行”上的 R×S×C=32 个数的乘积才得到一个输出点的实际卷积结果。

更一般的情况是：fmap 数据位宽为 N，则计算输出 fmap 的一个点，需要 C×R×S×N/64 轮 PE 运算得到结果。16 个 PE 每一轮都处理 8 个行，在 C×R×S×N/64 轮 PE 运算的乘累加过程中，得到 8×16=128 个点的卷积结果。



不同行代表了 HW 平面上的不同的点，多轮次的 PE 运算完成 RSC 三个维度的乘累加即得到该 HW 平面上该点的一个输出点，因此不断的更换左矩阵可以逐渐覆盖想要的 HW 平面的所有计算点，但是此时只计算了输出的部分 channel（右矩阵的不同列代表了 K 维度，此时的输出点对应于右矩阵的前 16 个 K，是输出结果的前 16 个 channel）。此后，更换右矩阵的 R×S×C×16，重复取 HW 平面需要算卷积的点，经过 K/16 次右矩阵更换，将计算得到所有 K 维度的输出点。（K 是指输出 Fmap 中 channel 的数量）

具体的执行过程如下：

1. HWPE 先配置好 8 个地址基址寄存器 FmapAddrBase，以及配置寄存器：H_count、W_count、H_stride、W_stride、Kernel_size、Kernel_W_offset；

2. 任务开始。Data Fetcher 通过访问 SRAM 的接口读取左矩阵的一行，push 到 FIFO 中，保证每周期有左矩阵的一行进入 FIFO。FIFO pop 左矩阵的一行到 PE，保证每周期有左矩阵的一行到 PE。

3. 左矩阵的一行被广播到 16 个 PE 中，16 个 PE 在 8 个周期内处理 8 行左矩阵，同时 PE 里保存右矩阵的列的 ping-pong buffer 也获取来自 Kernel sram 的新数据，

Kernel sram 每周期给 ping-pong buffer 输送一个新列。8 周期是一个时间单位，完成一轮矩阵乘累加。

4. 下一个 8 周期，Fetcher 根据配置寄存器 Kernel_size、Conv_CH_count、Conv_W_offset 控制访存地址更新，取来第二轮的 8 行左矩阵，ping-pong buffer 则换一个 entry 同左矩阵的行做乘积，完成新一轮的矩阵乘累加。

5. 重复 4，在 C×R×S×N/64 轮 PE 运算后，得到了 128 个输出点的结果，发出中断信号，任务挂起。MCU 发指令将结果寄存器的值搬回 MCU 的通用寄存器或者写回存储。数据输出完毕后，任务唤醒继续。（搬运 AccReg 的最后一条指令 HWPEReadAccReg /HWPEReLuMemWriteAccReg 中 HWPE 使能标志位置为高）

6. 任务继续，Fetcher 根据配置寄存器 H_count、W_count、H_stride、W_stride 更新卷积窗口地址，取 HW 平面上的下 8 个点作为左矩阵，Kernel SRAM 则回环到当前的 16 “整列” 即 R×S×C×16 列的起始地址开始输送右矩阵，重复 3 和 4，经过多轮次的运算，得到 128 个输出点时，重复 5 完成输出点的处理。

7. 重复 6，直到发出 H_count×W_count 次中断之后，任务中的 HW 平面计算完毕。之后卷积窗口地址重置为 FmapAddrBase[8]，Kernel SRAM 地址更新为 K 维度上新的 R×S×C×16 列的起始地址。完成对应输出点的新的 16 个 channel 的计算。

8. 重复 7，当右矩阵的列更换了 K/16 次后，任务完成。也就是说在总共发出 H_count×W_count×(K/16)次中断之后，PE 所配置的任务才算完成。

9. 回到 1，更新若干配置寄存器的值，重新开始跑新的了。

总的来说，可以认为有三层循环，最内层是完成左矩阵的 8 行 RSC 维度与右矩阵的 16 列 RSC 维度的矩阵乘积得到 128 个输出点。中层循环是更新卷积窗口地址，完成 H_count×W_count 次 128 点卷积。外层循环是更新卷积核的 K 维度，最终得到所有输出点。

DataFetcher 地址更新

HWPEMatrixMac 这条指令下发之后，开始从 FmapAddrBase 寄存器的地址逐次 load 8 行的左矩阵数据 Push 到 FIFO 中去，之后根据 Kernel_size、Conv_CH_count、Conv_W_offset 更新访存地址取来同属于一次卷积窗口的其他 64bit 行数据。经过若干次的矩阵乘累加将得到卷积结果。然后根据 H_stride、W_stride 更新访存地址取来新的 HW 平面上的 8 个点，在完成这 8 个点的卷积过程中依然是根据 Kernel_size、Conv_CH_count、Conv_W_offset 更新访存地址取来同属于一次卷积的其他行数据。

配置寄存器给出：

FmapAddrBase[8]，8 个基地址；

H_stride，每次取新点卷积时在 H 维度上进行跨越需要增加的地址偏移量；

W_stride，每次取新点卷积时在 W 维度上进行跨越需要增加的地址偏移量；

H_count，在 H 维度上需要进行几次跨越；

W_count，在 W 维度上需要进行几次跨越；

Kernel_size，就是 kernel 的大小，S；

Conv_CH_count，一次窗口卷积中连续的 CH 维度上有多少个 64bit；

$$Conv\_CH\_count=C\times S\times N/64$$

Conv_W_offset，一次窗口卷积中跨 W 维度时地址的偏移量；

$$Conv\_W\_offset=C\times H\times N/8$$

计算更新后的卷积窗口地址 FmapConvAddr[8]。

注：卷积窗口地址——卷积窗口的最小元素，在 H、W、C 这三个维度上的坐标都是最小的地址

计算方法如下：

```
for(int countW=0, offsetW=0; countW<W_count; countW++, offsetW+=W_stride) {
    for(int countH=0, offsetH=0; countH<H_count; countH++, offsetH+=H_stride) {
        for(int i=0; i<8; i++) {
            FmapConvAddr[i] = FmapAddrBase[i] + offsetW + offsetH;
        }
    }
}
```

在一次卷积窗口中，由 FmapAddr[8]计算访存地址 FmapSramAddr 如下：

```
for(int conv_countW=0, conv_offsetW=0; conv_countW<S; conv_countW++,
conv_offsetW+=Conv_W_offset) {
    for(int conv_countCH=0, conv_offsetCH=0; conv_countCH< Conv_CH_count;
    conv_countCH++, conv_offsetCH+=8) {
        for(int i=0; i<8; i++) {
            FmapSramAddr[i] = FmapConvAddr[i] + conv_offsetW + conv_offsetCH;
        }
    }
}
```

因为 SRAM 中的数据存储有三种类型：中间层（数据对齐）、非 3×3×3 输入层（数据不对齐）、3×3×3 输入层（数据折叠不对齐）

数据的对齐与否会影响到准备一行左矩阵的访存周期，因为访存的方案也不同。但数据始终是紧凑存储的，所以对卷积窗口地址的计算更新是相同的。但是访存地址的计算更新不完全相同。具体而言：

中间层（数据始终对齐）

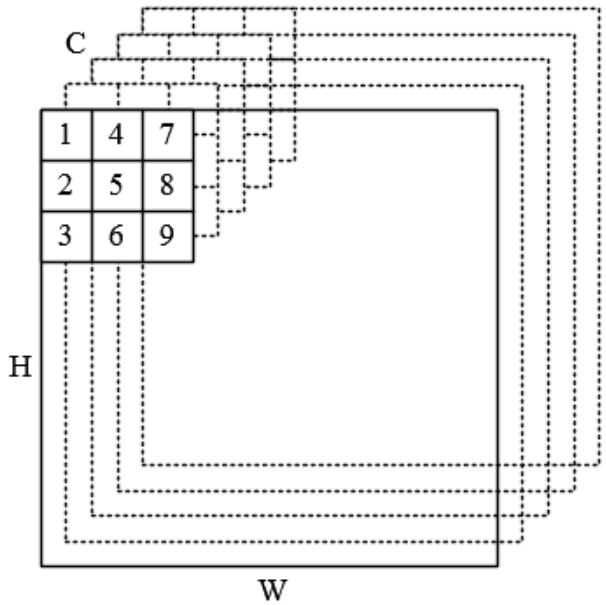
1. 访存方式：每个时钟周期取来的 64bit 数据正好就是需要的一行左矩阵。因此，对 Fmap SRAM 只使用 1 个地址访存。

2. 卷积窗口地址 FmapConvAddr 的更新：按照上述 FmapConvAddr 的伪代码逐次更新。

3. 访存地址 FmapSramAddr 的更新：按照上述 FmapSramAddr 的伪代码逐次更新。

一个举例。如下图所示，假设 **kernel 为 3×3×64、数据为 INT8 时，stride=2**。

第一次卷积的访存地址的计算：初始配置的 FmapAddrBase[0]指向输入 Fmap 的 H0W0C0，即图中标 1 的 C0 位置，此时 FmapSramAddr=FmapConvAddr[0]=FmapAddrBase[0]。当矩阵乘开始时，访存 1 点的 C0-C7 为第一行左矩阵，然后 FmapAddrBase[1-15]的 15 个地址取其余 15 行左矩阵完成一轮计算。第二轮计算开始时，FmapSramAddr 更新为 FmapAddr[0]+8，继续取 1 点 C8-C15 完成第 2 轮运算，第 C×N/64=8 次更新后地址将指向 2 点 C0 的位置，经过 Conv_CH_count=3×64×8/64=24 次更新后，地址指向 H 方向的下一点(H3W0C0)，此时若要继续卷积则地址应该跨越 W 维度指向 H0W1C0，即 4 的 C0 位置。跨 W 时，地址相对卷积窗口首地址的总偏移量是 Conv_W_offset=8×8H，（实际地址相对上次 FmapSramAddr 偏移量为 8×8H-24×8）。可以看出之后便是重复 Conv_CH_count=24 次地址+8 的更新和 1 次 Conv_W_offset=64H-24×8=64(H-S)×N/8，总共 3×Conv_CH_count 次地址+8，2 次 Conv_W_offset。



一个卷积窗口完成之后，需要更新窗口的首地址 FmapConvAddr。这个计算和由 FmapConvAddr 计算 FmapSramAddr 的过程是相似的。FmapAddrBase[0]指向 1 点 C0，卷积 kernel 的滑动步长为 2，则下一个卷积窗口地址 FmapAddr[0]是 3 点 C0。则 H_stride=2×64×8/64×8=128，FmapConvAddr[0]=FmapAddrBase[0]+128。当更新 H_count 次后，卷积窗口跨 W 维度，地址偏移为 W_stride，总共 H_count×W_count 次更新后完成任务要求的 HW 平面覆盖。

之后，FmapSramAddr=FmapConvAddr[0]=FmapAddrBase[0]，更换右矩阵为新的 K 维度的 16 列，循环 HW 平面计算输出点，直到对应 K 维度的右矩阵的所有的输出点都计算完毕。

输入层（非 3×3×3，C=3，数据存储不对齐）

1. 访存方式：从给定的基地址取来 64bit 的数据需要两个周期，因而使用 2 个 Fmap SRAM 地址同时访存两块 SRAM，才能满足 FIFO 的带宽需求。

2. 卷积窗口地址 FmapConvAddr 的更新：按照上述 FmapConvAddr 的伪代码逐次更新。

3. 访存地址 FmapSramAddr 的更新：按照上述 FmapSramAddr 的伪代码逐次更新。双周期访存中，只有第一个周期访存后更新地址（计入 Conv_CH_count 和 Conv_W_count），第二个周期访存后地址保持到下一轮的第一个周期再访存一次。

对于非 3×3×3 的卷积核，H×C 向上取整到 8N，则每次跨 W 维度时取来的 64bit 数据都只有部分数据是有效的（因为 kernel sram 的右矩阵参数是事先准备好的，取整的位置已经补了 0，因此在 Fmap sram 的访存中无须对数据做特殊处理，无效的部分在矩阵乘时结果为 0，不会影响结果）。和中间层相比，数据的非对齐并不影响访存地址的更新，在跨 W 时直接加上 Conv_W_offset 偏移地址即可。

输入层（3×3×3 数据存储不对齐）

1. 访存方式：从给定的基地址取来两行 64bit 的数据最差需要四个周期，因而使用 2 个 Fmap SRAM 地址同时访存两块 SRAM，才能满足 FIFO 的带宽需求。

2. 卷积窗口地址 FmapConvAddr 的更新：按照上述 FmapConvAddr 的伪代码逐次更新。

3. 访存地址 FmapSramAddr 的更新：按照上述 FmapSramAddr 的伪代码逐次更新。但有轻微的区别——每次访存前更新地址与否要视情况而定。

输入层的卷积核为 3×3×3 时，需要注意，此时一次从两个 SRAM 地址取来 4 行左矩阵，周期不定，最差为 4 周期。（中间层取数 1 个周期 1 行，非 3×3×3 的输入层每轮取数 2 个周期 2 行，都是确定的）存储方式上是将两个点存在一起的，完成一次卷积窗口的矩阵乘时，每次跨 W 时都是取了 9×2=18 个数，对应的一定是 3 次地址，

因此 Conv_CH_count=3，Conv_W_count=3，Conv_W_offset=3(H+1) (上下平面重叠时会有一行重叠存储)。周期虽然不定，但地址的偏移依然是规律的按照伪代码来更新的。

轮次如下：(注意，两个点相邻存储，实际取数的个数乘 2)

第 1 轮处理的元素：W0H0C0 W0H0C1 W0H0C2 W0H1C0 W0H1C1 W0H1C2 W0H2C0 W0H2C1

第 2 轮处理的元素：W0H2C2 W1H0C0 W1H0C1 W1H0C2 W1H1C0 W1H1C1 W1H1C2 W1H2C0

第 3 轮处理的元素：W1H2C1 W1H2C2 W2H0C0 W2H0C1 W2H0C2 W2H1C0 0 0

第 4 轮处理的元素：W2H1C1 W2H1C2 W2H2C0 W2H2C1 W2H2C2 0 0 0

下表列出了每轮不同周期情况下的地址更新情况，红色标出的地址下，新一轮时的访存地址是保持上一轮最后地址的。addr 为卷积窗口起始地址，offW 是 Conv_W_offset=3*H_333。

注：H_333=H+Kernel_size-STRIDE，即包含了重叠的几行

<table>
<tr><td rowspan="4">第一轮</td><td colspan="2">两周期</td><td colspan="2">三周期</td></tr>
<tr><td>addr</td><td>对齐 取 8 个数</td><td>addr</td><td rowspan="3">不对齐 取 16 个数</td></tr>
<tr><td>addr+8</td><td>对齐 取 8 个数</td><td>addr+8</td></tr>
<tr><td colspan="2"></td><td>addr+8+8</td></tr>
<tr><td rowspan="5">第二轮</td><td colspan="2">三周期</td><td colspan="2">四周期</td></tr>
<tr><td>addr+8+8</td><td>取 2 个数</td><td>addr+8+8</td><td>取 2 个数</td></tr>
<tr><td>addr+ offW</td><td>取 6 或 8 个数</td><td>addr+offW</td><td>取 2 或 4 个数</td></tr>
<tr><td>addr+offW+8</td><td>取 8 或 6 个数</td><td>addr+offW+8</td><td>取 8 数</td></tr>
<tr><td></td><td></td><td>addr+offW+8+8</td><td>取 4 或 2 个数</td></tr>
<tr><td rowspan="5">第三轮</td><td colspan="2">三周期/四周期</td><td colspan="2">两周期/三周期</td></tr>
<tr><td>addr+offW+8</td><td>取 2 个数</td><td>addr+offW+8+8</td><td>取 4 个数</td></tr>
<tr><td>addr+offW+8+8</td><td>取 2 个数</td><td rowspan="2">addr+2offW
对齐 取 8 个数</td><td rowspan="2">addr+2offW
addr+2offW+8
不对齐 取 8 个数</td></tr>
<tr><td rowspan="2">addr+2offW
对齐 取 8 个数</td><td rowspan="2">addr+2offW
addr+2offW+8
不对齐 取 8 个数</td></tr>
<tr><td></td><td></td></tr>
<tr><td rowspan="3">第四轮</td><td colspan="2">两周期</td><td colspan="2">两周期</td></tr>
<tr><td>addr+2offW+8</td><td>对齐 取 8 个数</td><td>addr+2offW+8</td><td rowspan="2">不对齐 取 10 个数</td></tr>
<tr><td>addr+2offW+8+8</td><td>对齐 取 2 个数</td><td>addr+2offW+8+8</td></tr>
</table>

FmapSram 数据存储的重叠（HW 平面的分割）

平分 HW 平面时，需要重叠 Kernel_size-STRIDE 行/列保证卷积窗口位于分割边缘时能够在该 SRAM 里取到全部所需数据。

例如，STRIDE=1，Kernel_size=3，H=W=10；输出则是 8×8；将 HW 平面左右平分，左平面需要计算出 4×8，则需要在 FmapSRAM1 存储 0~5 列，在 FmapSRAM2 存储 4~9 列，重叠了 4~5 列。

对于 3*3*3 的输入层，需要将 HW 平面先上下分割再左右分割，两次分割都要做重叠。