CNN-HWPE SPEC

——陈浩、陈强
2018-09-03

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[1]的 EAI (Extension Accelerator Interface)接口上。

特点

支持卷积层和 ReLU 层

通过矩阵乘法计算卷积（im2col on the fly）

支持 Kernel 大小从 3×3 到 11×11

支持数据类型 INT8，UINT8，EXP4（指数 scale 的 4bit），Ter2(Ternary2)

每周期完成 16 个 64bit 操作数的点乘运算

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。

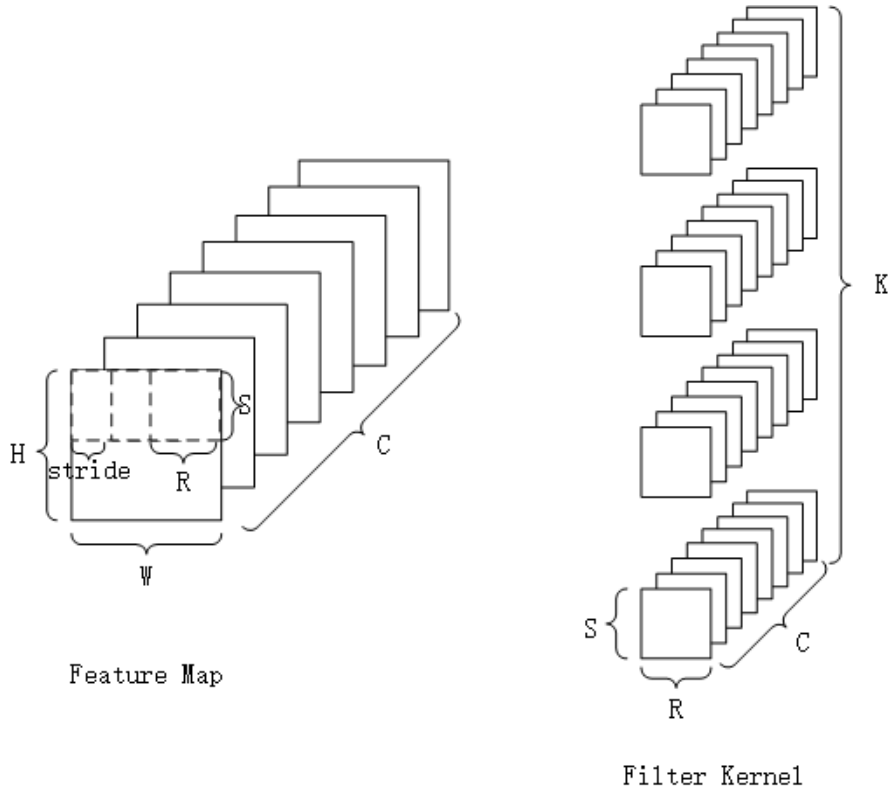

图 1　特征图和卷积核

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。

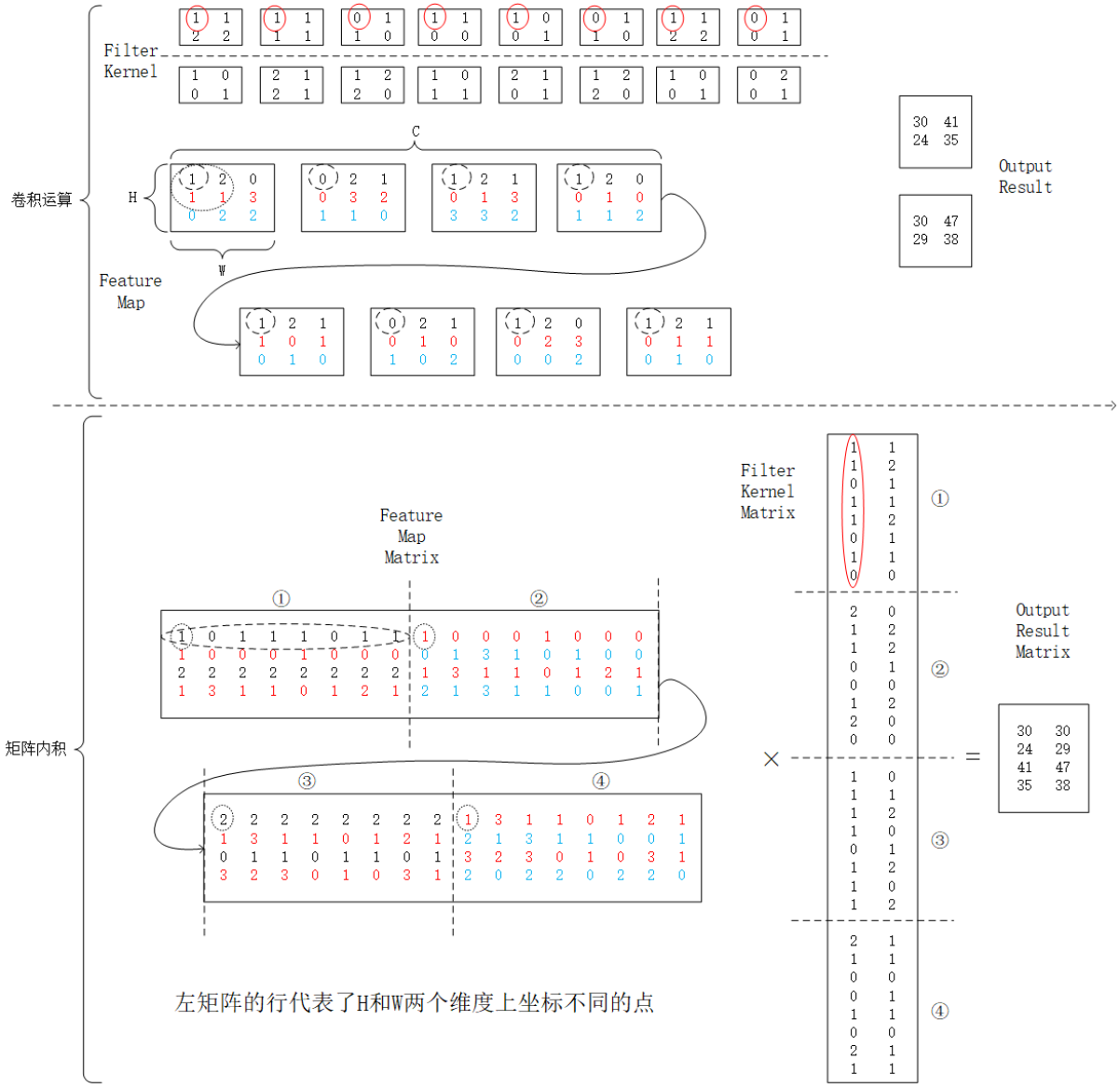

图 2　卷积转矩阵乘法的示例

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）。

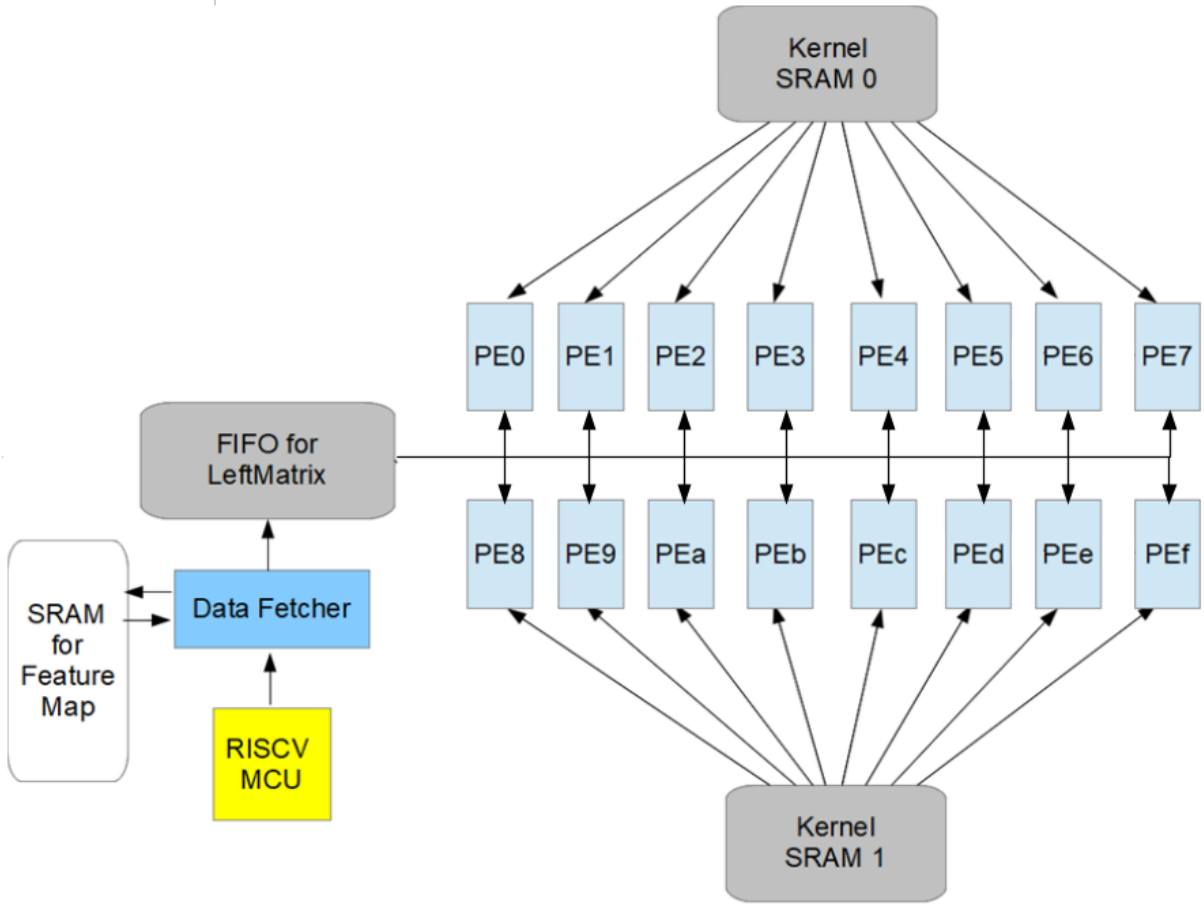

图 3　HWPE 的硬件架构图

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 左矩阵数据到 PE 中。
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送到 8 个 PE 中。

指令集

HWPE 的指令集遵守 RISCV 的定制指令扩展规则，RISCV 定制指令的格式如下：

表 1　RISC-V 指令映射表

31 25	24 20	19 15	14	13	12	11 7	6 0
funct7	rs2	rs1	xd	xs1	xs2	rd	opcode
7	5	5	1	1	1	5	7

Inst[4:2] Inst[6:5]	000	001	010	011	100	101	110	111 (>32b)
00	LOAD	LOAD-FP	***custom-0***	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	***custom-1***	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	***reserved***	***custom-2/rv128***	48b
11	BRANCH	JALR	***reserved***	JAL	SYSTEM	***reserved***	***custom-3/rv128***	≥80b

- 指令的第 0 位至至第 6 位区间为 Opcode 编码段，可以取值为 custom-0，custom-1，custom-2，或 custom-3。
- 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。

CNN-HWPE 的定制指令列表如下：

表 2　HWPE 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

Register	Value	Description
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(int8: C=8N; exp4: C=16N; ter2: C=32N) 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。

Register	Value	Description
H_count	Vrs1[15:0]	在 H 维度上需要进行几次跨越
W_count	Vrs1[31:16]	在 W 维度上需要进行几次跨越
H_stride	Vrs2[15:0]	每次在 H 维度上进行跨越需要增加的地址偏移量
W_stride	Vrs2[31:16]	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

HWPEReset 复位 HWPE

MCU 通过这条复位指令对 HWPE 的除 AccRegs 之外的寄存器做同步复位。AccRegs 的复位可以通过 *HWPEWriteAccReg* 置位来实现。

数据类型

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

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：

表 3　数据类型 EXP4 的真值表

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

Ternary，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

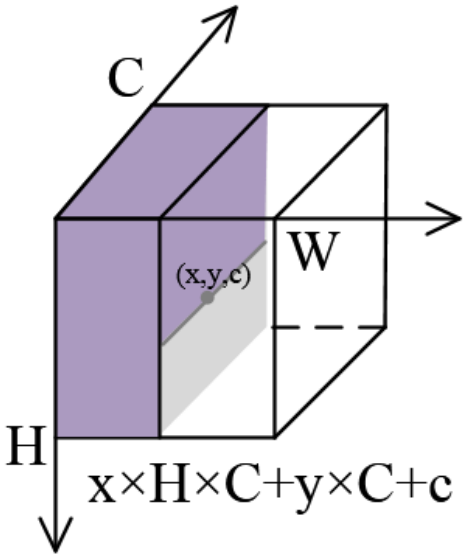

图 4 特征图上点的坐标示意图

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 H-W 平面再拆分成上下两部分，上下平移后重叠放置在一起，P0 P1 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 C→H→W 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 C→H→W 的方向。

Kernel SRAM

Kernel 的 SRAM 里的数据是提前重排列好的，对重排列的整个右矩阵来说，共有 K 列，对应 Kernel 的 K 维度，一列是一个 Kernel 的 RSC 三个维度的一维展开。同时，要求 R×S×C×N/64 是整数，否则在每列末尾补零使元素个数满足要求。对于输入层，R×S×3×8 不是 64 的倍数（意味着一行左矩阵无法恰好填满 64 bits，剩余位置补零），重排列的右矩阵需要 padding，使得一列的元素个数为 ceil(R×S×C/8)×8。这些在每列的末尾 padding 的零也是事先存储好的。

重排列好的 kernel 右矩阵在两块 Kernel SRAM 中如何存储？因为有 16 个 PE 每次需要 16 列 64bits 数据，因此以 16 列 64bit 为单位顺序存储 column0~column15，然后存储当前 16 列的下一个 64bits 数据直到当前的 16 列全部存储完毕。然后存储下一个 16 列，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 每周期更新一个 PE 的右矩阵参数，直至输出点的卷积完成。

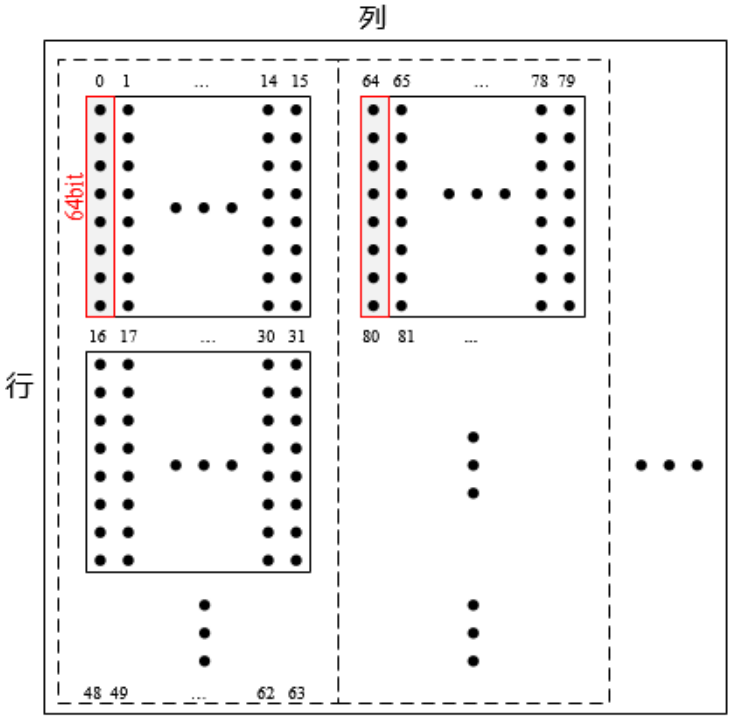

图 5 重排列的卷积核的存储顺序

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。（实际上，HWPE 也可以用来加速全连接层，唯一的问题是算力利用率只有 12.5%。但如果能凑够 8 个 batch，全连接层在 CNN-HWPE 里面计算时算力利用率是 100%）

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（int8: C=8N; exp4: C=16N; ter2: C=32N）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，大小可以是 3×3、5×5、7×7、11×11 等常用大小。因为一行左矩阵位数为 R×S×3×8 无法对齐 64bits，因此 FIFO 中最后一行 64 bits 数据没有被填满。此时 PE 的部分算力被浪费，导致 mac 利用率低于 100%。

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，FIFO 中的最后一行只有 1 个字节有效（对应的 kernel 的重排列补 7 个 0）。mac 利用率为 33/40=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，FIFO 中的最后一行只有 5 个字节有效（对应的 kernel 的重排列补 3 个 0）。mac 利用率为 21/24=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，FIFO 中的最后一行只有 7 个字节有效（对应的 kernel 的重排列补 1 个 0）。mac 利用率为 15/16=93.75%

3×3×3：按照上述的对每一次连续访存补零的方法，mac 利用率将为 9/16=56.25%。接近一半的算力被浪费了。为了增加利用率，HWPE 对这种情形做了特殊处理，包括前述的特殊的存储方式——将两个点交错存储。

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），需要得到的两行左矩阵如下：

```
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 中是彼此临近交错存储的。考虑到地址不对齐的情形，最多需要 3 个周期取来这 16 bytes 数据。然后对这些点做一次转秩操作，得到想要的给 FIFO 的两行左矩阵。

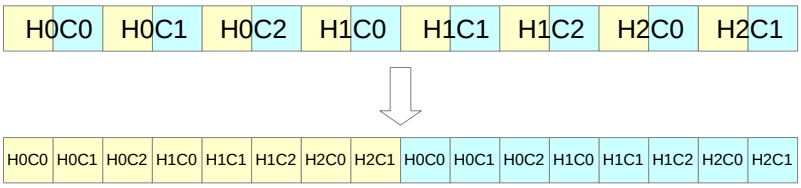

图 6 交错数据点的转置

对于第 2 轮和第 3 轮，因为跨 W 维度，情况比第 1 轮更差，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 的利用率提高为 27/32=84.38%。在进行 PE 的点乘运算前，只需要配置 4 个地址寄存器，每个地址会取来 2 行给 FIFO 的左矩阵。

在硬件实现上，FIFO 被分成两个，一个负责奇数 entry，一个负责偶数 entry。这样可以满足两行数据同时 push 到 FIFO 的需求；但在 Pop 的时候，每周期仍然只 pop 一个。

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 $((\frac{H-S}{stride}+1)\times(\frac{W-R}{stride}+1))\times(C\times R\times S)$，右矩阵的维度为(C×R×S)×K。在硬件上，经过逐步往往 FIFO 里灌 64 bits 数据逐渐实现 im2col，而右矩阵则预先重排列并存储好的。

一行左矩阵对应一个卷积窗口的元素的重排列；一列右矩阵对应一个 kernel 的元素的重排列。这一行与一列的点乘得到该卷积窗口的卷积输出，对应到输出 HW 平面上的一个点。在硬件上，PE 里的左右矩阵是 64 bits 的，所以理论上的一行与一列的点乘是被分解成若干轮 64bit 的点乘完成的（每一轮 PE 都顺序处理 8 行 64 bits 左矩阵）。

如图 7 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 64 bits，只能处理 8 个数，理论上的一行被分解为 4 轮点乘，完成一行的 R×S×C=32 个数的点乘才得到一个输出点的实际卷积结果。

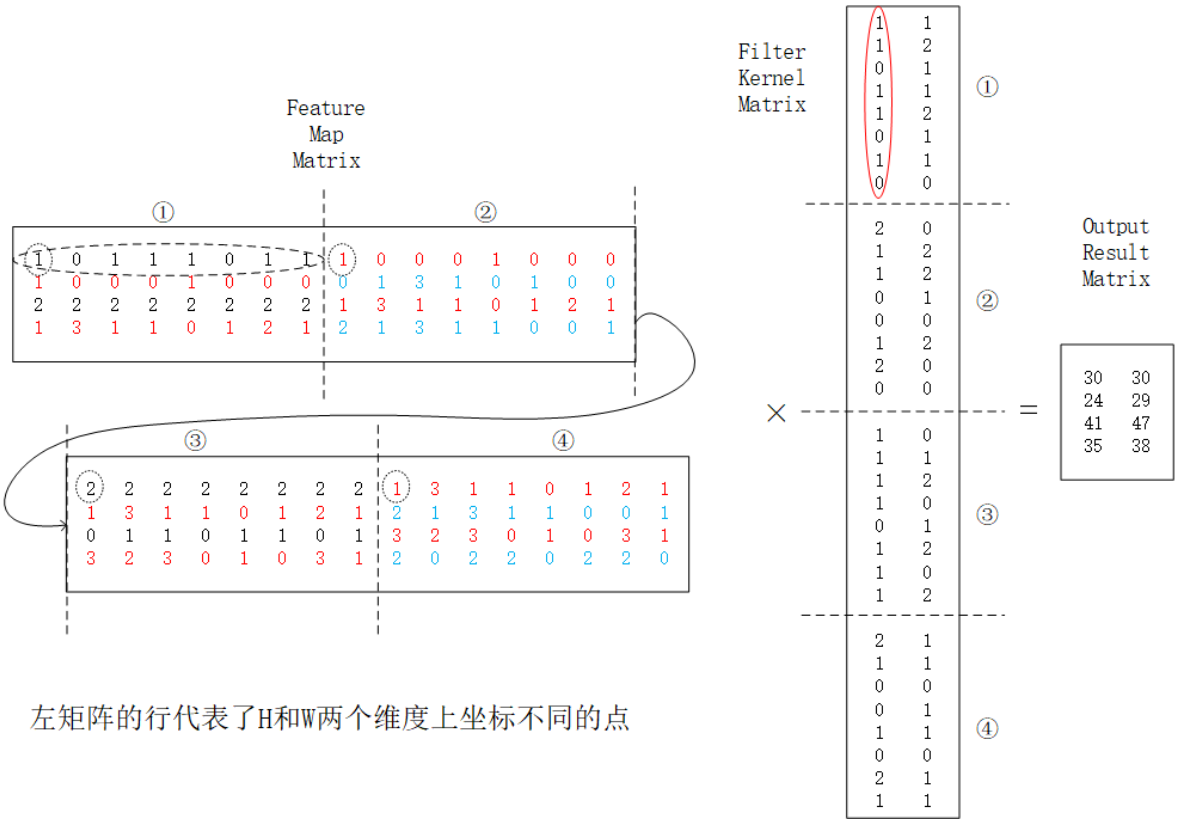

图 7 矩阵乘法的示例图

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。当一个卷积窗口计算完毕后，移动卷积窗口逐渐覆盖整个 H-W 平面。如图 8 所示，H-W 平面被分割为 8 个部分（每部分的第一个卷积窗口的第一个元素地址就是 8 个基地址）。每个部分有 3×4 个卷积窗口，对应 H_count=3，W_count=4。8 个卷积窗口先 H 后 W 方向滑动覆盖完各部分时，整个 H-W 平面的卷积窗口也就计算完毕了。

16 个 PE 每一轮都处理 8 个行，这 8 行数据来自于图中的 8 个不同部分。经过 C×R×S×N/64 轮 PE 运算的点乘得到 8×16=128 个点的卷积结果。之后滑动 8 个卷积窗口，取得新的卷积窗口 im2col 的数据作为下 8 行左矩阵。经过 H_count×W_count 次滑动卷积窗口，覆盖整个 HW 平面。然后更换卷积核——16 列右矩阵（16 个 PE，每次更换 16 列），卷积窗口回到初始基地址所在窗口并且重新移动覆盖整个 H-W 平面。经过 K/16 次右矩阵更换，所有 K 个卷积核都参与计算完毕，整个卷积任务完成。

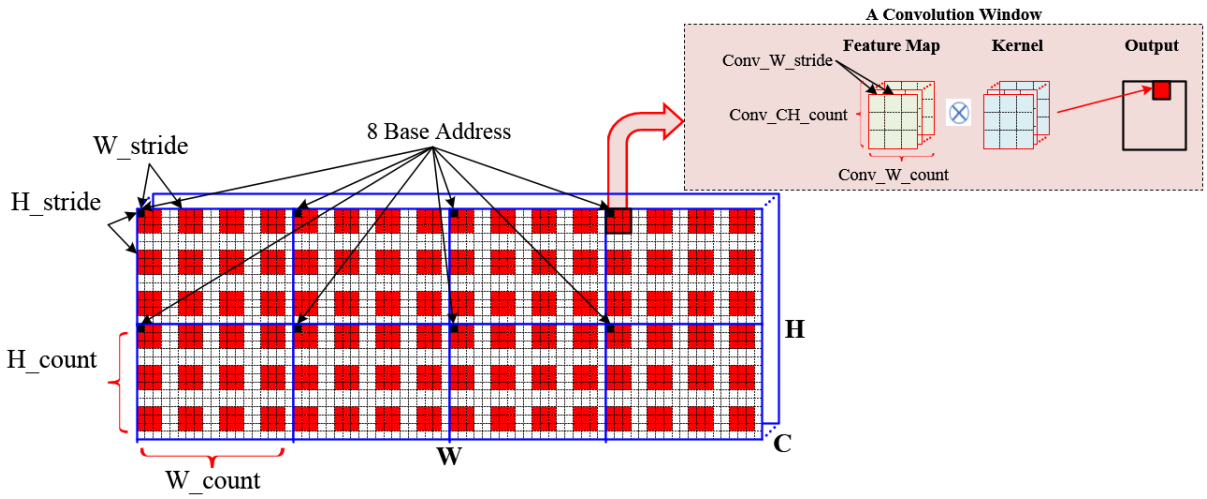

图 8 卷积窗口滑动过程示例图

Head address of convolution windows:

- FmapConvAddr[i] = FmapAddrBase[i] + x·W_stride + y·H_stride,

0≤x<H_count; 0≤y<W_count

SRAM memory access address:

- FmapSramAddr[i] = FmapConvAddr[i] + m·Conv_W_stride + n·Conv_CH_stride,

0≤m<Conv_CH_count; 0≤n<Kernel_size

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 里存储右矩阵的列的 ping-pong buffer 也获取来自 Kernel sram 的新数据。Kernel sram 的两套接口每周期给 ping-pong buffer 输送两个新列。8 周期是一个时间单位，完成一轮点乘。

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 **FmapSramAddr** 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，完成新一轮的点乘。

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 **FmapConvAddr**，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，开始跑新一层卷积。

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

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。总共 H_count×W_count×K_count 次 128 点卷积。

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。经过若干次的点乘将得到卷积结果。然后根据 H_stride、W_stride 更新卷积窗口首地址取来新的 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 64 bits 行数据。

计算更新后的卷积窗口地址 FmapConvAddr[8]。

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 FmapConvAddr[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;
        }
    }
}
```

上述代码给出了每次访存的起始地址的计算方式，只适用于 Kernel_333=1'b0 的卷积层。从计算到的 FmapSramAddr 地址取来 8bytes 数据的方式则会受到地址对齐的影响。因为 SRAM 采用 64 bits 位宽，当 FmapSramAddr 对齐到存储的边界时（中间层），读取该 entry 就可以取来 64 bits 的期望数据。当不对齐时（非 3x3x3 的输入层），地址 FmapSramAddr~FmapSramAddr+8 跨越两个 entry，需要双周期访存。此时会通过两套访存地址同时访存，保证平均每周期供应 FIFO 一行左矩阵。

当 kernel size=3×3×3 时，输入图像数据是交叉存储的且不对齐 64 bits。地址 FmapSramAddr 的更新略复杂一些，按照下面 4 轮列出的元素顺序来访存。前两轮准备两行 64 bits 左矩阵需要 16 bytes 数据，第三轮需要 10bytes 数据，第四轮需要 8bytes 数据。其中第二轮和第三轮还会跨越 W 维度。因此每轮的非对齐访存的最差情况为：3 周期、4 周期、4 周期、2 周期。硬件上固定每轮的访问周期为 4，通过两套 SRAM 访存接口保证 4 个周期取得 4 行左矩阵。

```
1st elements (P0 P1): W0H0C0 W0H0C1 W0H0C2 W0H1C0 W0H1C1 W0H1C2 W0H2C0 W0H2C1
2nd elements (P0 P1): W0H2C2 W1H0C0 W1H0C1 W1H0C2 W1H1C0 W1H1C1 W1H1C2 W1H2C0
3rd elements (P0 P1): W1H2C1 W1H2C2 W2H0C0 W2H0C1 W2H0C2 W2H1C0    0      0
4th elements (P0 P1): W2H1C1 W2H1C2 W2H2C0 W2H2C1 W2H2C2    0       0      0
```

下表列出了访存地址的更新的规律，红色标志了当前轮的最后访存地址也是下一轮的起始访存地址的情况。

表 4　kernel_333=1 时访存地址 FmapSramAddr 的更新规律

	两周期		三周期	
第一轮	addr	对齐 取 8 个数	addr	不对齐 取 16 个数
	addr+8	对齐 取 8 个数	addr+8	
			addr+8+8	

<table>
<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="4">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>addr+2offW
对齐 取 8 个数</td><td>addr+2offW
addr+2offW+8
不对齐 取 8 个数</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，需要保证分割边缘的行(列)所在卷积窗口的完整性。因此需要重叠 Kernel_size-STRIDE 行(列)，保证卷积所需数据正确读取。

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 H-W 平面先上下分割再左右分割，两次分割都要做重叠。

配置寄存器的值

配置寄存器的取值受卷积本身的类型、选取的 8 个初始点、H-W 平面的分割方式的影响。下面是一个例子用来说明如何设置这些寄存器的值。

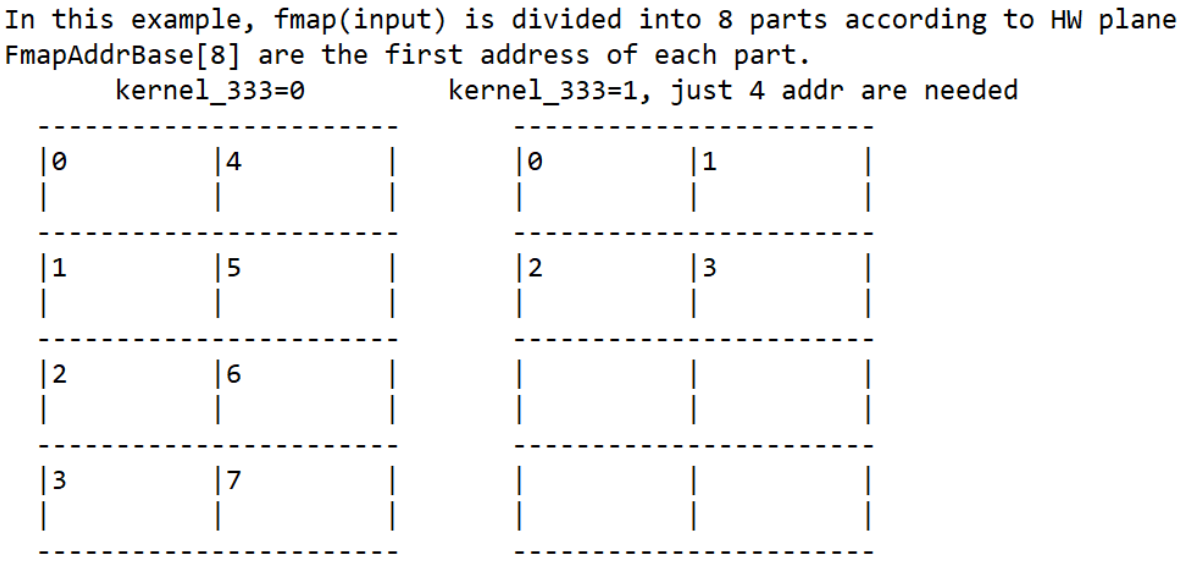

图 9 特征图分割的示例

表 5 配置寄存器赋值的示例

Registers	Values
FmapAddrBase[8]	First address of each part
H_count	((H-Kernel_size)/STRIDE+1)/4
W_count	((W-Kernel_size)/STRIDE+1)/2
H_stride	Kernel_333? 2*STRIDE*C*N/8 : STRIDE*C*N/8
W_stride	Kernel_333? H_333*C*STRIDE*N/8 : H*C*STRIDE*N/8
Conv_CH_count	Kernel_333? 3 : Layer_type ? ceil(C*Kernel_size*N/64) : C*Kernel_size*N/64
Conv_W_offset	Kernel_333? C*H_333*N/8 : C*H*N/8

注：H_333=H + Kernel_size – STRIDE，因为部分行是重叠的被重复存储在 memory 中。详情参照 FmapSram 数据存储的重叠 这一节。

8 个首地址 FmapAddrBase[8]如下：

```
uint32_t FmapAddrBase_333[8]=
{
        0,
        uint32_t(W_count*W_stride),
        uint32_t(H_count*H_stride),
        uint32_t(W_count*W_stride+H_count*H_stride),
        0,0,0,0
};
uint32_t FmapAddrBase_n333[8] =
{
        0,
        uint32_t(H_count*H_stride),
        uint32_t(2*H_count*H_stride),
        uint32_t(3*H_count*H_stride),
        uint32_t(W_count*W_stride),
        uint32_t(W_count*W_stride+H_count*H_stride),
```

```
        uint32_t(W_count*W_stride+2*H_count*H_stride),
        uint32_t(W_count*W_stride+3*H_count*H_stride)
};
if (Kernel_333)
        FmapAddrBase = FmapAddrBase_333;
else
        FmapAddrBase = FmapAddrBase_n333;
```

Reference

[1] https://github.com/SI-RISCV/e200_opensource

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