# CNN-HWPE SPEC

——Hao Chen; Qiang Chen
2018-09-03

## Objective

A hardware coprocessor for CNN acceleration based on RISC-V instructions set, which is linked with Hummingbird E200 MCU[1] through extension accelerator interface (EAI).

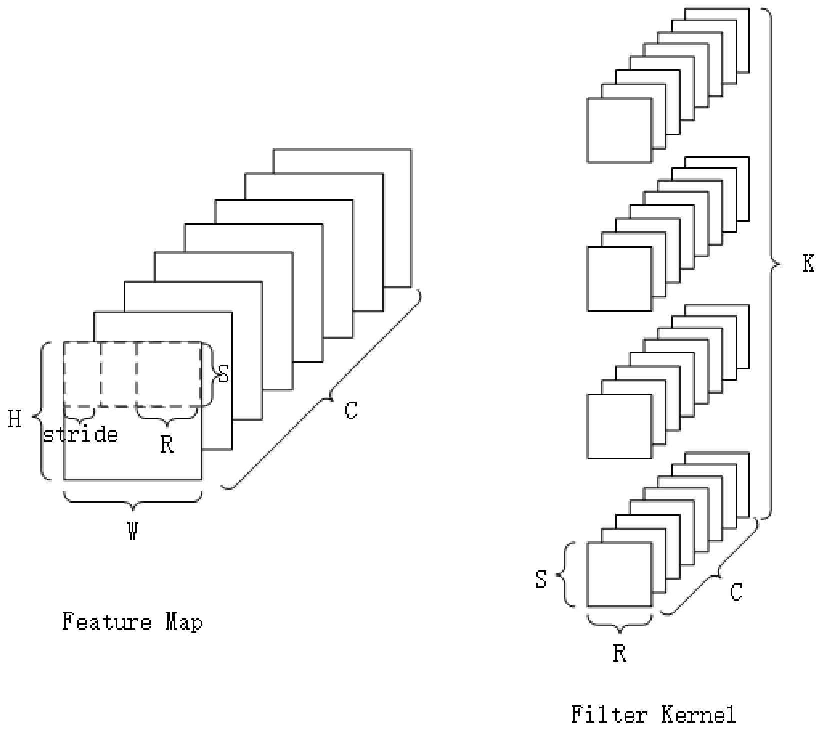
## Features

- Support Convolution layer and ReLU layer
- Transform convolution into matrix multiplication (im2col on the fly)
- Kernel size from 3×3 to 11×11
- Support data type INT8， UINT8, EXP4 (4 bits of exponential scale) and Ternary
- 16 dot-product operations of 64-bit operands (8 INT8, 16 EXP4 or 32 Ternary) per cycle

## Theoretical Principle (im2col)

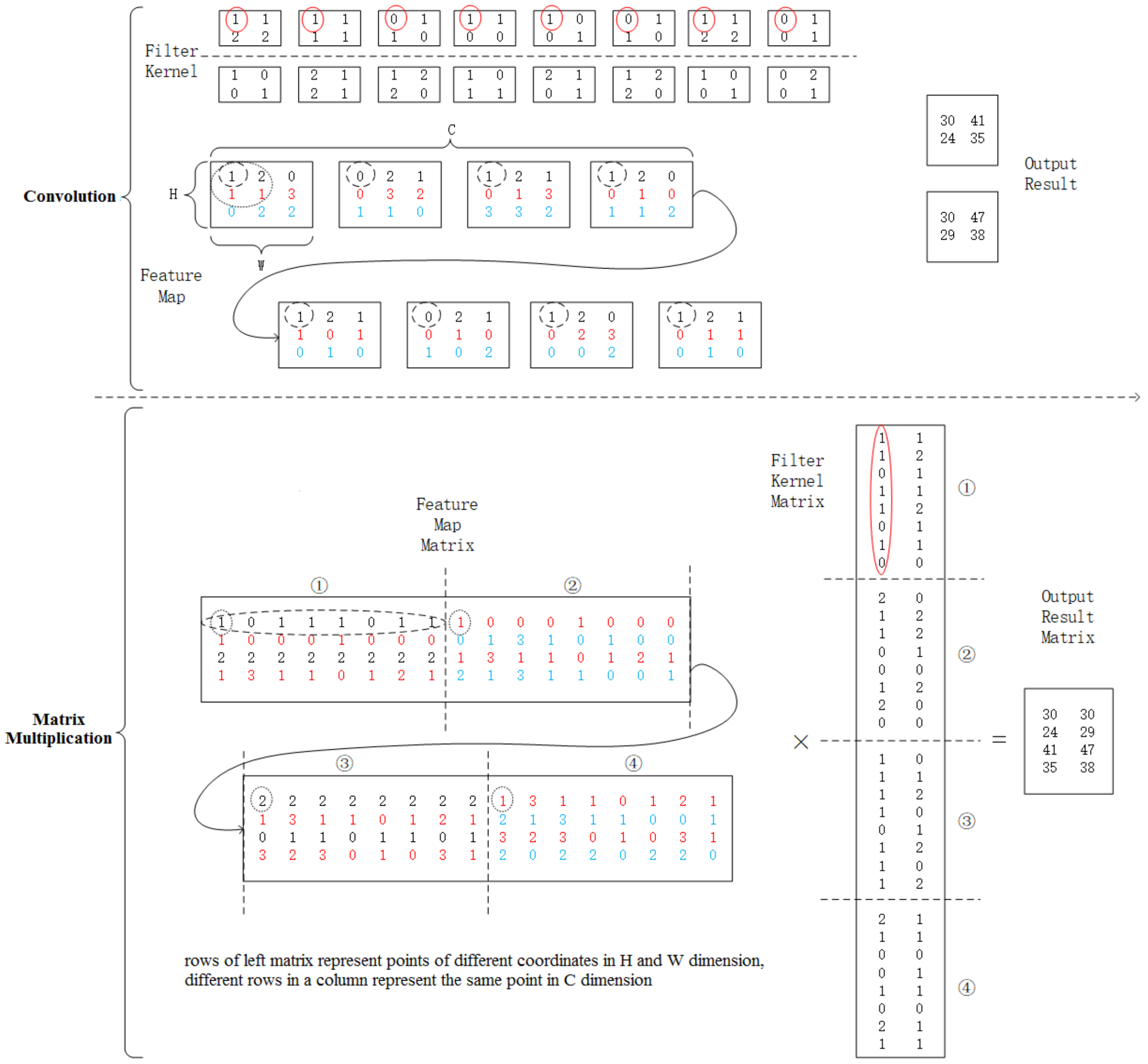
The following figure illustrates the Feature Map (Fmap) and Filter Kernel, which are two operands in a convolution. The feature map has three dimensions, H, W and C. The kernel has 4 dimensions, S, R, C and K.

**Figure 1 The Dimensions of Feature Map and Filter Kernel**



Convolution of CNN can be computed by im2col and matrix multiplication. The following figure illustrates the results of convolution of a 3×3×8 feature map and a 2×2×8×2 kernel. A 4×32 left matrix can be got by transforming feature map in the way of im2col according to the order of C→H→W. Rearranging the filter kernel accordingly will get a 32×2 right matrix. As you can see, the results of matrix multiplication are just the same as convolution, which is also a rearranged matrix for convolution results according to the order of C→H→W.

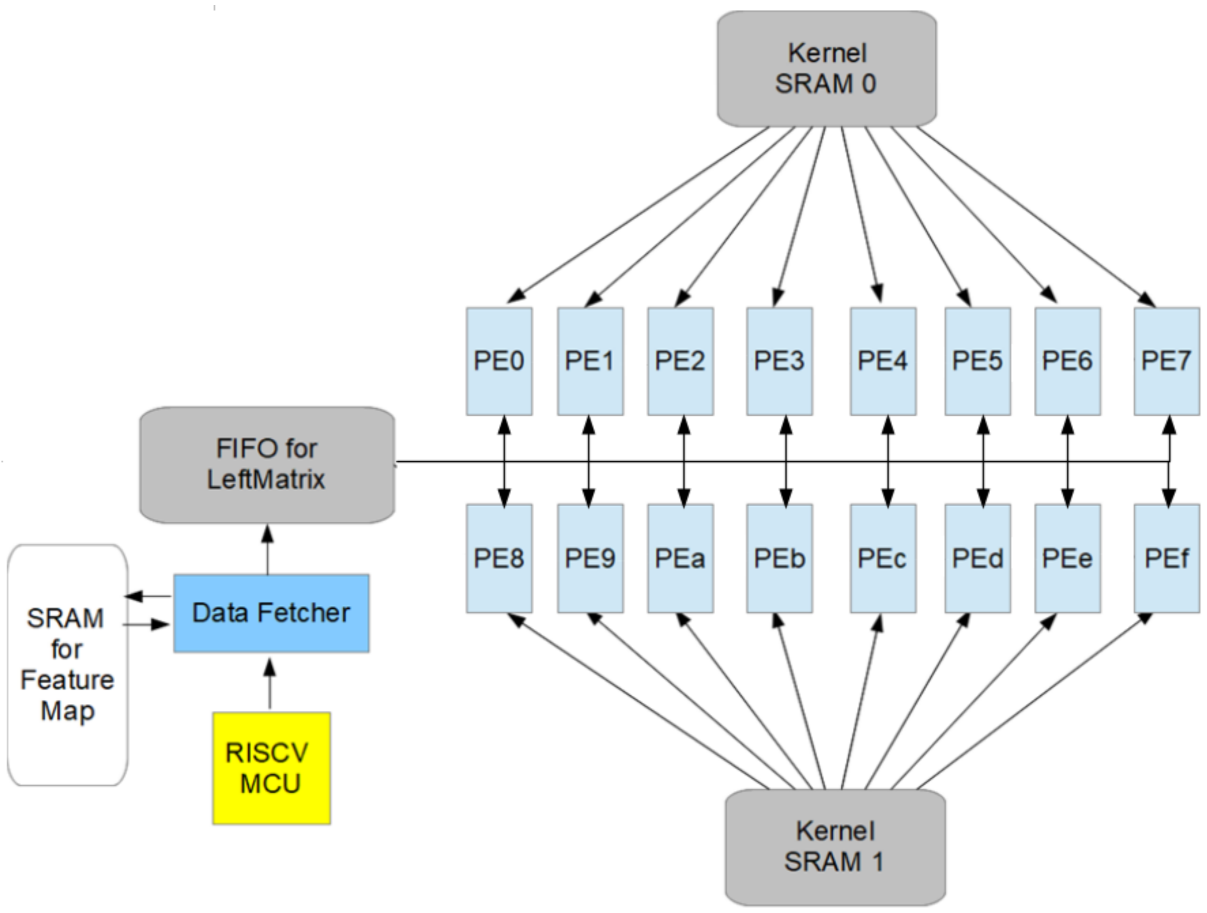
**Figure 2 An Example of Im2col**


Convolution
Filter Kernel
Feature Map
C
H
W
Output Result
Matrix Multiplication
Feature Map Matrix
Filter Kernel Matrix
Output Result Matrix
①
②
③
④
×
=
rows of left matrix represent points of different coordinates in H and W dimension, different rows in a column represent the same point in C dimension

## Hardware Implementation

HWPE can be integrated into the SOC environment of hummingbird through the EAI interface, which is very similar to the coprocessor interface RoCC[2] of Rocket core.

Figure 3	Architecture Diagram of HWPE



Traditionally, the GPU does im2col by expanding the image and storing it into a small chunk of *col* memory. However, in this design, the im2col action is completed gradually by continuously pushing data to FIFO, and there isn't specific *col* memory space to store *col*. Kernel is stored in SRAM and has been rearranged in advance.

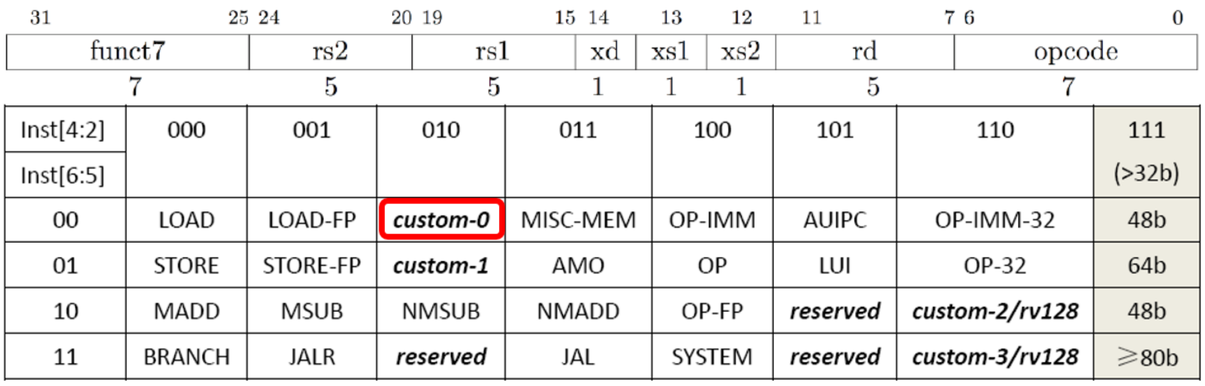
The components are as follows:

1. A FIFO (width=64 bits, depth=4) is used to buffer the left matrix, and data fetcher constantly pushes new data into it. The FIFO pops the left matrix data out to PE in order every cycle.
2. Data fetcher accepts the command from MCU and fetches data from Feature Map SRAM and push them into FIFO. Feature Map SRAM has at least two banks since data fetcher may have to fetch two 64-bit data at the same time.
3. 16 PEs, each of which stores a column (64 bits) of the right matrix, and a column of the result matrix (which requires eight 32 bits result registers called *AccRegs*).
4. Two big Kernel SRAMs, each one is responsible to send the right matrix data to 8 PEs.

## Instruction Set

The instructions of HWPE are designed based on custom-0 opcode of RISC-V extended instruction set. The format of 32-bit instruction is as follows.

Table 1 RISC-V Basic Opcode Mapping Table



| 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 |

Table 2 The Customized Instructions Table of HWPE

| 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 |

Table 2 is the instructions table of HWPE. There are only 7 instructions. The following are the details.

***HWPEWriteFmapAddrReg*** is a configuration instruction. The HWPE has eight base address registers named *FmapAddrBase* for left matrix which point to the eight initial points of the H-W plane. This instruction sets two address *FmapAddrBase*[rd] and *FmapAddrBase* [rd+1] (so rd can be 0, 2, 4 or 6) once executed. Only 4 addresses are needed to be configured for input layer if the filter kernel is 3×3×3.

***HWPEWriteCfgReg*** sets the configuration registers with the values of general registers rs1 and rs2 of MCU.

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] | Numbers of 64 bits of continuous data in C-H dimension within a convolution window |

| | | |
|---|---|---|
| Conv_W_offset | Vrs1[31:16] | The offset of address spanning W dimension within a convolution window |
| Kernel_size | Vrs2[3:0] | The size of filter kernel (R=S) |
| Data_type | Vrs2[5:4] | 2’b01/2’b10/2’b11/2’b00 represent ternary2 / exp4 / int8 / uint8, the data type of right matrix is always int8 when Date_type = 2’b11 / 2’b00 |
| Layer_type | Vrs2[6] | 1’b0: intra layer (int8: C=8N; exp4: C=16N; ter2: C=32N); 1’b1: input layer (C=3) |
| Kernel_333 | Vrs2[7] | Kernel size is 3×3×3 if it is high level |
| AccReg_shift | Vrs2[12:8] | Shifting amount for ReLU within a range of 0~24 to transform the *Accregs* to be an 8 bits data |
| K_count | Vrs2[22:13] | The K dimensions of filter kernel divided by 16 |

***HWPEMatrixMac*** configures two registers and starts the convolution computation. { W_count, H_count} = Vrs1; {W_stride, H_stride}=Vrs2。

| Register | Value | Description |
|---|---|---|
| H_count | Vrs1[15:0] | Iterations needed in the H dimension of input feature map |
| W_count | Vrs1[31:16] | Iterations needed in the W dimension of input feature map |
| H_stride | Vrs2[15:0] | The offset of address spanning H dimension when convolution window slides |
| W_stride | Vrs2[31:16] | The offset of address spanning W dimension when convolution window slides |

***HWPEWriteAccReg*** writes the value of general register of MCU to a AccReg of a PE, in which the rd, rs2 and rs1 are the indexes of *AccReg*, PE and general register of MCU. This instruction can be used to reset or set *AccRegs* with an initial value, which is also called a bias in a convolution layers.

***HWPEReadAccReg*** reads the *AccRegs* back to the MCU general registers, in which the rs1[2:0], rs2, rd are the indexes of AccReg, PE and general register of MCU. The most significant bit rs1[4] is the enable flag for HWPE, which indicates convolution continues after moving *AccRegs* when it asserts.

***HWPEReLuMemWriteAccReg*** executes ReLU operation and transfers the 32 bits AccRegs to 8 bits, then stores the results into the specified memory address. The objectives of this instruction are the No. rs2 *AccRegs* of 16 PEs which actually belong to the same output point.

The value of rs1 register is the address of memory, and rs2[2:0] is the index of *AccReg* (every PE has eight 32-bit *AccReg* with index from 0 to 7). The most significant bit rs2[4] is the enable flag for HWPE, which indicates convolution continues after writing memory when it asserts.

HWPE needs 4 cycles to do ReLU and transformation for 16 32-bits *AccRegs*. The shifting amount of transforming 32 bits to 8 bits is configured by register CfgReg1.AccReg_shift.

*Note: There are two ways to move the results from AccRegs in PEs. One way is to write 32-bit data back to general registers of MCU with **HWPEReadAccReg** and MCU will handle the next step. The other way is to execute ReLU and transform to 8 bits for the 16 channels of an output point, then writing results back into specified memory with **HWPEReLUMemWriteAccReg**.*

***HWPEReset*** executes synchronous reset for all flip-flops except *AccRegs* of HWPE, which can be reset by being set to zero with *HWPEWriteAccReg*.

## Data Type

HWPE supports INT8, UINT8, EXP4 (4 bits of exponential scale) and Ternary.

The range of 4 bits of exponential scale is within the set [-64、-32、-16、-8、-4、-2、-1、0、1、2、4、8、16、32、64]. The MSB of EXP4 is sign bit and the others 3 bits represent exponent (it equals to actual exponent plus 1). The value is zero when exponent is 3’b000.

For multiplication, two exponent add together to get the result, while the special case is that the result is 0 if any exponent is 3’b000. For addition, one-hot encoding is needed before normal addition.

$$\begin{cases} (-1)^{a_3} \bullet 2^{a_2 a_1 a_0 - 1} & a_2 a_1 a_0 \neq 000 \\ 0 & a_2 a_1 a_0 = 000 \end{cases}$$

Table 3 Truth Table of EXP4

| 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 |

As for Ternary, 2’b00=0; 2’b01=1; 2’b11=-1。

## Data Storage Format

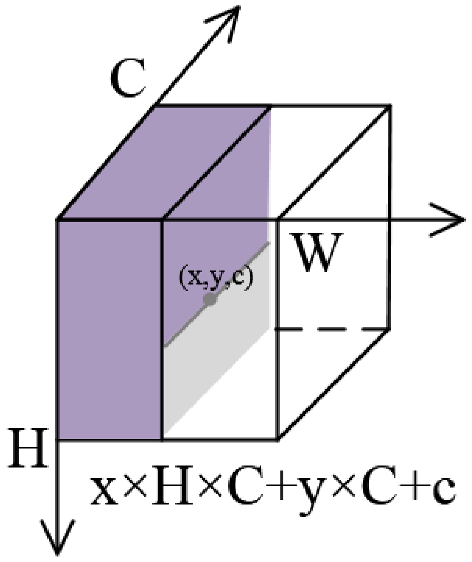
### Feature Map SRAM

SRAM for feature map are divided into two parts storing left and right half of the H-W plain. The HWPE is designed to access two entries data at the same time, so the number of interfaces to access memory must be two, but the number of SRAMs can be 2 or 4 or more.

The data of feature map is stored into memory according to the order C→H→W. The SRAM address of a point of feature map is:

$x \times H \times C + y \times C + c \quad 0 \leqslant x < W, \quad 0 \leqslant y < H, 0 \leqslant c < C$

Figure 4 The Coordinate of a Point In a Feature Map



**Special case**: For input layer, the input data must be stored into memory in a special format when the size of filter kernel is 3×3×3.

According to the storage order C→H→W, two points from different local parts marked P0 and P1 are not contiguous. The two points in storage are as following:

```
P0W0H0C0 P0W0H0C1 P0W0H0C2 P0W0H1C0 P0W0H1C1 P0W0H1C2 P0W0H2C0 P0W0H2C1
P1W0H0C0 P1W0H0C1 P1W0H0C2 P1W0H1C0 P1W0H1C1 P1W0H1C2 P1W0H2C0 P1W0H2C1
```

Imaging that the H-W plain of an input image in SRAMs are divided equally into upper part and lower part, then put one part on top of another and the P0 and P1 are overlapped, which indicates that P0 and P1 are in the same relative location in themselves part. P0 and P1 can be contiguous in memory if the dimension P is in the innermost order of storage. Now the data of input image are stored in the order of P→C→H→W:

```
W0H0C0P0 W0H0C0P1 W0H0C1P0 W0H0C1P1 W0H0C2P0 W0H0C2P1 W0H1C0P0 W0H1C0P1
W0H1C1P0 W0H1C1P1 W0H1C2P0 W0H1C2P1 W0H2C0P0 W0H2C0P1 W0H2C1P0 W0H2C1P1
```

This is the special storage format for filter kernel of 3×3×3. First store the two points of two local parts, then store data according to normal order C→H→W.

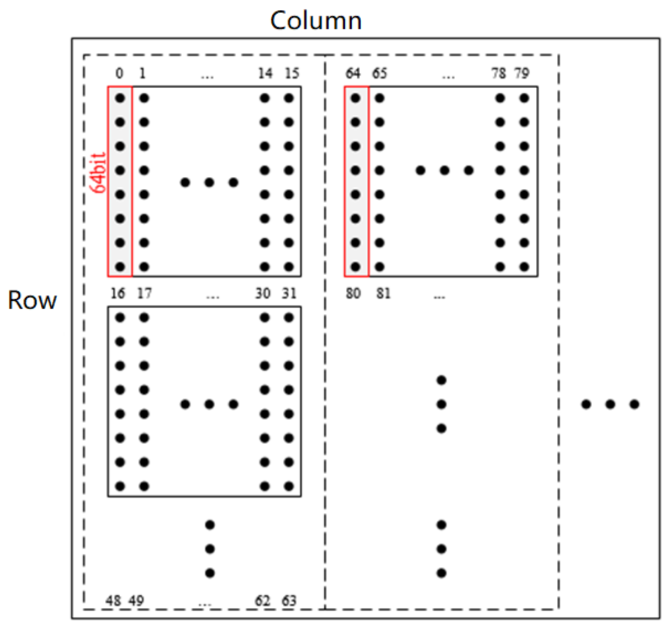
### Kernel SRAM

The data of filter kernel in SRAMs are rearranged in advance. First, change the four-dimensional filter kernel (R, S, C, K) into two-dimensional matrix which has R×S×C rows and K columns. For input layers whose input data type is INT8, if R×S×C, which is the number of elements in a convolution window that will fill FIFOs, are not the multiple of 8, rows of rearranged right matrix have to be padded to the multiple of 8 with zeros. A column of the matrix has ceil(R×S×C/8)×8 elements after padding.

How to store the rearranged matrix into SRAMs? Because there are 16 PEs and each will calculate a column of the matrix at the same time. So store the 16 64-bit data in order from column 0~15 into SRAM address 0~15, then store the next 16 64-bit data from column 0~15 into SRAM address 16~31 until all elements of first 16 columns are stored. Next, store the 16 64-bit data from the beginning to the end of the next 16 columns until all K columns data are stored.

The kernel SRAMs have two interfaces which provide the first 8 columns and the last 8 columns for every 16 columns of the matrix. Each interface update right matrix data for 8 PEs until the convolution is done.

**Figure 5 The Storage Order of Filter Kernel**



*Note: By the way, the width of feature map and filter kernel must be the same.*

## Supported Kernel Size of CNN

The HWPE supports the convolution of input layer and internal layer, while fully-connected layer and pooling layer are still calculated by MCU. (Actually, HWPE can accelerate fully-connected layer. The only problem is that utilization of macs is 12.5% for one batch fully-connected layer. The utilization can be 100% if 8 batches fully-connected layers are provided to HWPE.) The width of register *Kernel_size* is 4 bits which can support a maximum 11×11 Frequently-used size of filter kernel.

For **internal layer**, the restrictions for feature map channels (int8: C=8N; exp4: C=16N; ter2: C=32N) provide an aligned 64-bit data. Data Fetcher can fetch an aligned 64-bit data from SRAMs as left matrix per cycle. The utilization of mac array is 100%.

For **input layer,** the kernel size can be 3×3, 5×5, 7×7, 11×11 and so on with a fixed channel number (C=3). Because the number of elements of a convolution windows is R×S× 3, and the whole vector for multiplication is R×S×3×8 bits, which is not aligned to 64 bit operand, some mac array in PEs will be wasted at the last time computation of a row, and the utilization cannot be 100%.

11×11×3: The data of a convolution window in SRAMs are neighbouring in channel C and H, so Data Fetcher can fetch 11×3=33 bytes continuously and push it to FIFO every 8 bytes. The last row of left matrix in FIFO has only 1 byte data and the remaining 7 bytes (56 bits) of operand are filled with zeros. At the same time, the corresponding column of rearranged matrix of filter kernel must be padded with 7 zeros in advance. The utilization of mac array is 33/40=82.5%.

7×7×3: Just like the analysis for 11×11×3, Data Fetcher can fetch 7×3=21 bytes continuously and push it to FIFO every 8 bytes. The last left matrix in FIFO has 5 bytes data and the remaining 3 bytes (24 bits) of operand are filled with zeros. The corresponding column of rearranged matrix of filter kernel must be padded with 3 zeros in advance. The utilization of mac array is 21/24=87.5%.

5×5×3: Data Fetcher can fetch 5×3=15 bytes continuously and push it to FIFO every 8 bytes. The last left matrix in FIFO has 7 bytes data and the remaining 1 bytes (8 bits) of operand are filled with zeros. The corresponding column of rearranged matrix of filter kernel must be padded with 1 zero in advance. The utilization of mac array is 15/16=93.75%.

As for **3×3×3**, the utilization of mac array is only 9/16=56.25% if we handle this case just like the previous method. To improve the utilization, besides the particular storage format, a special data manipulation is also proposed. There are the 4 rounds computation to complete a convolution for total 27 elements as following:

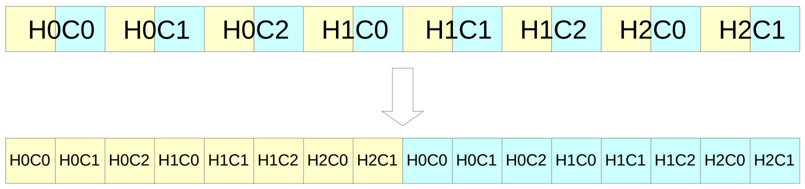
```
1st elements: W0H0C0 W0H0C1 W0H0C2 W0H1C0 W0H1C1 W0H1C2 W0H2C0 W0H2C1
2nd elements: W0H2C2 W1H0C0 W1H0C1 W1H0C2 W1H1C0 W1H1C1 W1H1C2 W1H2C0
3rd elements: W1H2C1 W1H2C2 W2H0C0 W2H0C1 W2H0C2 W2H1C0    0      0
4th elements: W2H1C1 W2H1C2 W2H2C0 W2H2C1 W2H2C2   0       0      0
```

Take the first round as an example, from the point of view of matrix multiplication, eight elements W0H0C0 W0H0C1 W0H0C2 W0H1C0 W0H1C1 W0H1C2 W0H2C0 W0H2C1 for two different parts named P0 and P1 should be accessed. That is to say the following data are filled into two entries of FIFO as two left matrices for mac array:

```
P0W0H0C0 P0W0H0C1 P0W0H0C2 P0W0H1C0 P0W0H1C1 P0W0H1C2 P0W0H2C0 P0W0H2C1
P1W0H0C0 P1W0H0C1 P1W0H0C2 P1W0H1C0 P1W0H1C1 P1W0H1C2 P1W0H2C0 P1W0H2C1
```

According to the storage format, data of P0 and P1 are stored neighbouring and interleaved. A transposition for these points is necessary to separate the interleaving data of P0 and P1, as the Figure 6 illustrates.

**Figure 6 The Transposition For Interleaving Data From Two Parts**


H0C0 H0C1 H0C2 H1C0 H1C1 H1C2 H2C0 H2C1
H0C0 H0C1 H0C2 H1C0 H1C1 H1C2 H2C0 H2C1 H0C0 H0C1 H0C2 H1C0 H1C1 H1C2 H2C0 H2C1

Two rows left matrix for FIFO are prepared after transposition. These 16 bytes of data are accessed in at most 3 cycles taking data's unalignment into consideration. For 2nd ~ 4th rounds, the case is worse since the W dimension are spanned, and 4 cycles at most are needed for accessing data. In summary, the HWPE gets 4 rows of left matrix for FIFO in 4 cycles at most by accessing memory through the two interfaces of SRAMs at the same time, which is acceptable and has the same efficiency of one row per cycle compared to the other types layer.

Under this circumstance, the utilization of mac array is 27/32=84.38%, which is much higher than 9/16=56.25% (the method of padding zeros for continuous memory access).

As for hardware implementation, the FIFO are separated into two instances to accept two rows left matrix at the same time, and one FIFO is for even entry, the other for odd entry. But there is only one row of data per cycle out from FIFO to PE when FIFO pops.

## How PE works

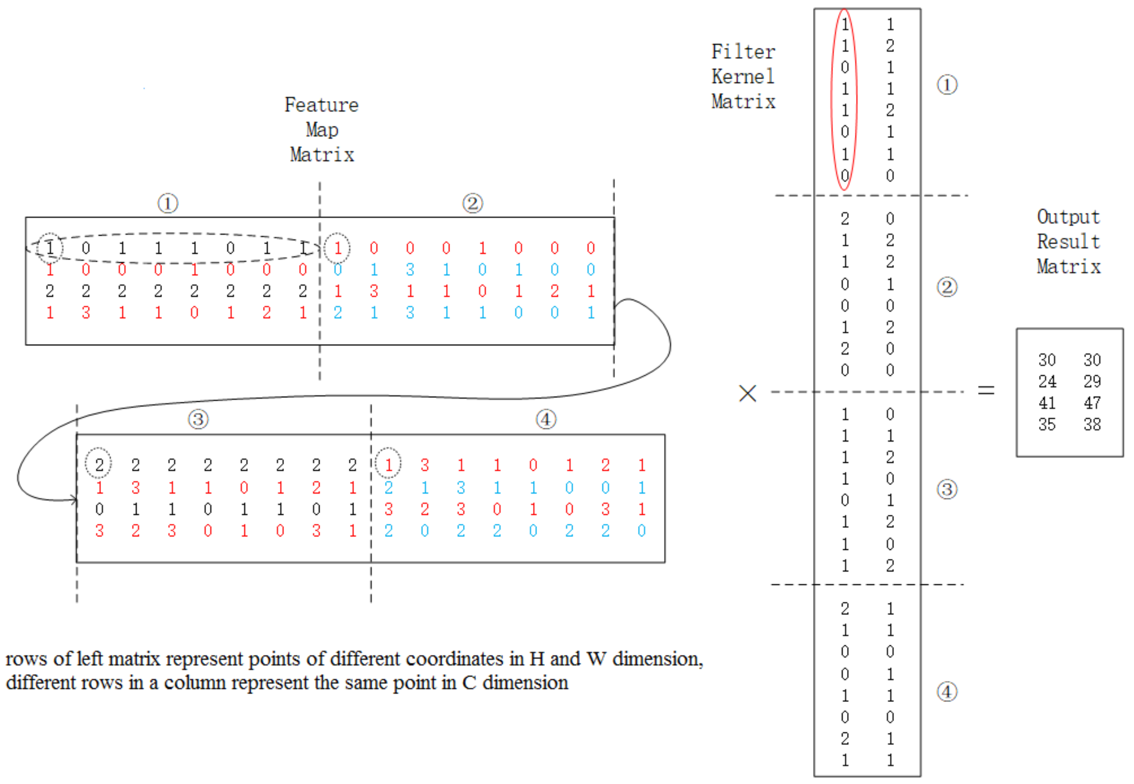
The work of PE is matrix multiplication for a left matrix by $((\frac{H-S}{stride}+1)\times(\frac{W-R}{stride}+1))\times(C\times R\times S)$ and a right matrix by (C×R×S)×K in theory. The im2col for left matrix is realized gradually by fetching data and pushing it into FIFO according to a certain order. The im2col for right matrix are rearranged in advance and already stored in memory.

From the point of view of convolution, a row of left matrix (C×R×S elements) is the arrangement of a convolution window, and a column of right matrix (C×R×S elements) is the arrangement of a filter kernel (K filter kernels in total). The dot-product between a row and a column get a convolution output point in the output H-W plain.

From the point of view of hardware, the left matrix in a FIFO is 64-bit data, and one PE executes a dot-product operation of 64-bit operands per cycle, so an output point is calculated after C×R×S×N/64 rounds, in which N is the bit-width of data type.

As shown in Figure 7, if the data type is INT8 (N=8), and the dimension of left matrix after im2col is 4×32, there have to be 4 rounds to complete a row 32 numbers with one round dot-product handling 8 numbers data.

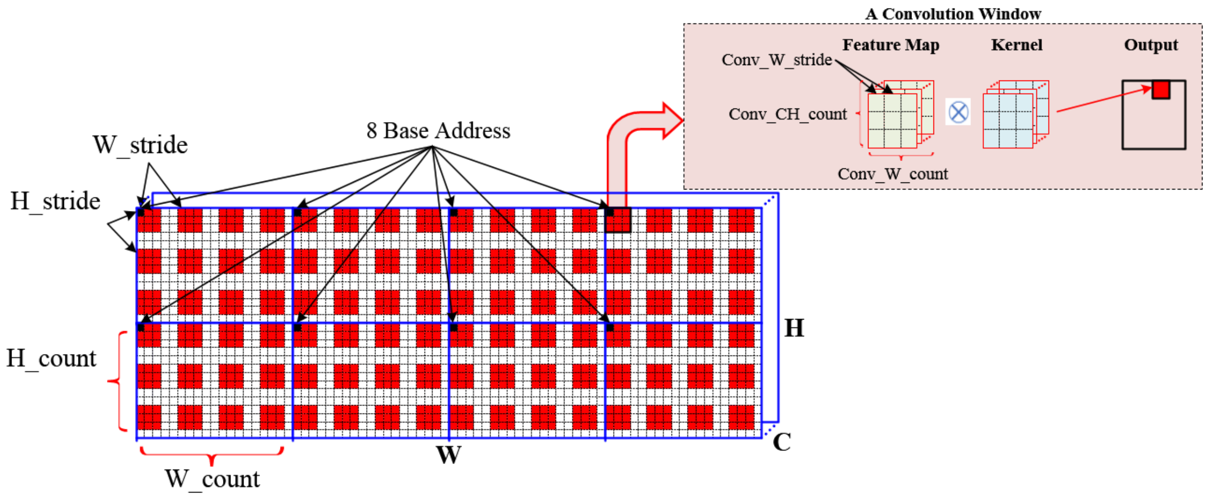
**Figure 7  An Example of Matrix Multiplication**


Feature Map Matrix
Filter Kernel Matrix
Output Result Matrix
①
②
③
④
×
=
30 30
24 29
41 47
35 38
rows of left matrix represent points of different coordinates in H and W dimension, different rows in a column represent the same point in C dimension

Generally speaking, the results are got after C×R×S×N/64 rounds computation if the bit-width is N. The convolution window slides to cover the H-W plane after the computation is done. As shown in Figure 8, the H-W plane are divided into 8 parts, and each part has 3×4 convolution windows (H_count=3, W_count=4), where the address of the first element in the first window is the base address *FmapAddrBase*. When 8 convolution windows slide and cover each part in directions H→W, the convolution windows of the whole H-W plane are calculated.

The 8 rows of data calculated in 16 PEs come from 8 different parts in figure 8. After C×R×S×N/64 rounds dot-product, 8×16=128 output points are obtained. Then slide the 8 convolution windows and use the im2col data from new windows as next 8 rows of left matrix. After H_count×W_count times slides of windows, the whole H-W plane are covered. Then change the next 16 columns of right matrix, set the 8 convolution windows back to initial base location, then slide the windows to cover H-W plane again and again. When all K different filter kernels are calculated after K/16 times changing columns of right matrix, the entire convolution task is completed.

Figure 8 An Example of Sliding Convolution Windows



Head address of convolution windows:

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

$0 \le x < H\_count;\ 0 \le y < W\_count$

SRAM memory access address:

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

$0 \le m < Conv\_CH\_count;\ 0 \le n < Kernel\_size$

The detailed operating procedure is as following:

1. Configure the 8 base address registers FmapAddrBase and configuration registers H_count, W_count, K_count, H_stride, W_stride, Kernel_size, Kernel_W_offset;
2. Convolution starts. The Data Fetcher accesses memory and pushes a row of 64-bit left matrix to FIFO and FIFO pops a row to PE. The HWPE ensures that there must be a row of 64-bit left matrix coming into FIFO from Data Fetcher, and a row outgoing from FIFO to PE every cycle.
3. A row of 64-bit left matrix is broadcasted to 16 PEs. A round multiply-accumulate is done in 8 cycles as a time unit. In a round, the 16 PEs deal with 8 rows left matrix, and ping-pong buffer in PEs storing 16 64-bits right matrix obtains two new 64-bit data per cycle from the two interfaces of kernel SRAMs.
4. In the next 8 cycles, Data Fetcher updates memory access address **FmapSramAddr** and fetches new 8 rows left matrix for second round computation according to the configuration registers Kernel_size, Conv_CH_count, Conv_W_offset. The ping-pong buffer changes the entry to multiplies the left matrix and completes the new round computation.
5. Repeat 4 until 128 output points are calculated after C×R×S×N/64 rounds computation. Then an interrupt signal is generated and the task on HWPE suspends. MCU sends commands to move AccRegs to general registers of itself or write back into specific address of memory when it receives the interrupt signal. The task goes on after all AccRegs are moved out. (The last command of *HWPEReadAccReg* or

*HWPEReLuMemWriteAccReg* to move AccRegs has a high level enable flag to restart convolution task for HWPE)

6. The convolution task continues. The Data Fetcher updates the address of convolution window **FmapConvAddr** to get the next 8 point in H-W plane according to the registers H_count, W_count, H_stride, W_stride, while the address of kernel SRAM loops back to the beginning of the current 16 columns of the rearranged right matrix. Repeat 3 and 4 to get the 128 output points after several rounds computation, then repeat 5 to handle the results of output.
7. Repeat 6 until interrupts are generated H_count×W_count times. Now the entire H-W plane are covered. Then reset the address of convolution window **FmapConvAddr** to be the value of **FmapAddrBase**, and update the address of kernel SRAM to be the beginning of next new 16 columns of rearranged right matrix. Complete the computation task for the new 16 channels of K dimension.
8. Repeat 7 and the whole task is finished until the columns of right matrix are changed K/16 times. In other words, the task for PE are finished after H_count×W_count×(K/16) times interrupts.
9. Back to 1, reconfigure the configuration registers and run a new convolution task.

*Note: the address of convolution window indicates the address of the least member whose coordinate is the minimum in three dimension of H, W, C within a convolution window. In other words, it is the head address of a convolution window.*

In summary, there are 3 nested loops. The inner loop is to finish the dot-product between 8 rows left matrix and 16 columns right matrix and get 128 output points. The middle loop is to update the address of convolution window and complete H_count×W_count times convolutions (128 output points each convolution). The outer loop is to update columns of right matrix, which is the K dimension of filter kernel, and get all outputs. There are H_count ×W_count×K_count times convolution in total.

## Address Update of Data Fetcher

Data Fetcher fetches 8 rows of left matrix's data and push them to FIFO gradually from FmapAddrBase[0]~ FmapAddrBase[7] after decoding the command *HWPEMatrixMac*. Then the memory access address **FmapSramAddr** is updated to fetch new 64-bit data belonging to the same convolution window according to the configuration registers Kernel_size, Conv_CH_count, Conv_W_offset. The output point of current window is calculated after C×R×S×N/64 rounds computation. Next, slide the convolution window, update the head address of convolution window **FmapConvAddr** to get new 8 rows left matrix. And then repeat updating **FmapSramAddr** then **FmapConvAddr** until the task is done.

The code for updating address of convolution window **FmapConvAddr** is as following:

```
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;
                }
        }
}
```

The code for updating memory access address **FmapSramAddr** is as following:

```
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;
                }
        }
}
```

The above code gives the way to calculate the starting address when accessing memory, which is only applicable to the convolution layers when Kernel_333=1'b0. The way to fetch 8 bytes of data from calculated addresses FmapSramAddr is affected by address alignment. Because SRAM width is 64 bits, it's ok that just fetch the expected 64-bit data from this entry when FmapSramAddr is aligned to the storage boundary (middle layer). When the address is not aligned (non-3×3×3 kernel of input layer), double-cycle memory access is required since the address FmapSramAddr~FmapSramAddr+8 spans two entries. In this case, fetch data from two interfaces of SRAMs to ensure that a row left matrix is supplied to FIFO per cycle on average.

When kernel size is 3×3×3 (Kernel_333=1'b1), input image is stored interleaved and unaligned. The update of the address FmapSramAddr is slightly more complex and in the order of the elements listed in the following four rounds. In order to prepare two rows 64-bit left matrix, the first two rounds need 16 bytes of data, the third round need 10 bytes data, the fourth round need 8 bytes data, and don't forget that the second and third rounds will also span the W dimension. Therefore, the worst situations for each round non-alignment memory access is: 3 cycles, 4 cycles, 4 cycles, 2 cycles. The memory access period is fixed at 4 cycles every round on hardware implementation, so that it is guaranteed that 4 rows of left matrix are supplied to FIFO every 4 cycles through two interfaces of SRAMs.

```
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
```

The following table lists the behavior of update of the address FmapSramAddr. The address marked with red color indicates that it's the last memory access address of current round and the first memory access address of next round at the same time.

**Table 4 The Behavior of Address FmapSramAddr When Kernel_333 Asserts**

| | | | | |
|---|---|---|---|---|
| 1st round | Two cycles | | Three cycles | |
| | addr | aligned, fetch 8 bytes | addr | unaligned fetch 16 bytes |
| | addr + 8 | aligned, fetch 8 bytes | addr + 8 | |
| | | | addr + 8 + 8 | |
| 2nd round | Three cycles | | Four cycles | |
| | addr + 8 + 8 | fetch 2 bytes | addr + 8 + 8 | fetch 2 bytes |
| | addr + offW | fetch 6 or 8 bytes | addr + offW | fetch 2 or 4 bytes |
| | addr + offW + 8 | fetch 8 or 6 bytes | addr + offW + 8 | fetch 8 bytes |
| | | | addr + offW + 8 + 8 | fetch 4 or 2 bytes |
| 3rd round | Three cycles / Four cycles | | Two cycles / Three cycles | |
| | addr + offW + 8 | fetch 2 bytes | addr + offW + 8 + 8 | fetch 4 bytes |
| | addr + offW + 8 + 8 | fetch 2 bytes | addr + 2offW<br>aligned, fetch 8 bytes | addr + 2offW<br>addr + 2offW + 8<br>unaligned, fetch 8 bytes |
| | addr + 2offW<br>aligned, fetch 8 bytes | addr + 2offW<br>addr + 2offW + 8<br>unaligned, fetch 8 bytes | | |
| 4th round | Two cycles | | Two cycles | |
| | addr + 2offW + 8 | aligned, fetch 8 bytes | addr + 2offW + 8 | unaligned, fetch 10 bytes |
| | addr + 2offW + 8 + 8 | aligned, fetch 2 bytes | addr + 2offW + 8 + 8 | |

## The Overlap of Data Storage for Feature Map

When dividing the HW plane, we need to ensure the integrity of the convolution window which belongs to the row(column) at the edge of the divided line. Therefore, Kernel_size-STRIDE rows (columns) are overlapped and stored twice (two copies in two SRAM) to ensure that the convolution data can be fetched correctly.

For example, STRIDE = 1, Kernel_size = 3, H = W = 10, so the output is $8 \times 8$. To divide the H-W plane into left and right, and each half plane will compute $4 \times 8$ points. Then, column 0~5 should be stored in Feature map SRAM1, and column 4~9 in Feature map SRAM2, with two overlapped columns 4 and 5.

For the input layer of $3 \times 3 \times 3$, the H-W plane needs to be divided upper and lower half first, then left and right, and Kernel_size-STRIDE rows and columns must be overlapped because it has been divided twice.

## The Values of Configuration Registers

The values of some configuration registers are determined by convolution type, the initial 8 points and the shape of divided input feature map. The following is an example to illustrate how to set the values of all these registers. Figure 8 shows how the feature map is divided for kernel_333 equaling to 1’b1 or 1’b0. Table 5 shows the formulas of computation under the circumstance of figure 8.

**Figure 9 An Example of Divided Input Feature Map**

```
In this example, fmap(input) is divided into 8 parts according to HW plane
FmapAddrBase[8] are the first address of each part.
       kernel_333=0             kernel_333=1, just 4 addr are needed

  -----------------------      -----------------------
  |0         |4          |     |0         |1          |
  |          |           |     |          |           |
  -----------------------      -----------------------
  |1         |5          |     |2         |3          |
  |          |           |     |          |           |
  -----------------------      -----------------------
  |2         |6          |     |          |           |
  |          |           |     |          |           |
  -----------------------      -----------------------
  |3         |7          |     |          |           |
  |          |           |     |          |           |
  -----------------------      -----------------------
```

**Table5 An Example of the Values of Configuration Registers**

| 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 |

*Note: H_333=H + Kernel_size – STRIDE, because some rows are overlapped and stored twice. Refer to the chapter “The Overlap of Data Storage for Feature Map” for more details.*

The First address for each part FmapAddrBase[8] are as follows:

```
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