In-memory data formats

Overview

DLA engine supports various types of CNN layers like convolution layer, pooling, ReLU, LRN, etc. To improve the performance DLA engine also applies some options like Winograd, weight compression and multi-batch mode. To support these layers DLA engine uses specified input and output data formats.

There are two main types of input formats. They are weight data and activation data. The weight formats include below branches:

- weight for direct convolution
- weight for image input
- weight for Winograd convolution

Two options for weight formats:

- channel post-extension for image input mode
- sparse compression

The activation formats supported by DLA engine includes:

- feature data format
- pixel format (ROI input)

The output formats supported by DLA engine includes:

- feature data format

Besides, DLA engine will fetch below auxiliary formats from external memory

- bias data
- PReLU data
- batch-normalization data
- element-wise data

Channel extension refers to a set of mapping rule for both weight data and activation data to fit with accelerator. It includes:

- Channel extension for Winograd convolution
- Channel pre-extension for image input mode
- Channel post-extension for image input mode

The channel post-extension for image input mode is an option for performance. Other two are mandatory for their working mode. HW handles all channel extension on feature/pixel data, while SW shall do channel extension to weight data accordingly.

All data formats should be mapping in memory with rules.

Precision Type

NVDLA engine pipeline support three types of data precision. They are int8, int16 and fp16. For int8, one element of data refers to an 8-bit signed integer. For int16, one element refers to a 16-bit signed integer. For fp16, one element refers to a 16-bit floating point data, which is also named as half-precision floating-point format.

All input feature data should belong to one of three precision types. And all image input data will be converted to one precision type before calculation. For example, DLA engine can take a T_R10G10B10A2 image as input (for first layer) and convert the component to int8, int16 or fp16.

Precision Conversion

NVDLA engine supports dynamical precision conversion. There are some rules:

- NVDLA convolution pipeline supports precision conversion for image input mode only.
- Direct convolution (DC) mode and Winograd convolution mode do not support precision conversion
- For image input mode (please see section 6.1.1.4), pipeline allows conversion from integer to all 3 types. Floating point images can only be converted to fp16.
- Batch-normalization and element-wise layer (implemented in SDP) support free conversion of int16 <-> fp16 and int8 <-> int16 for DC mode only.
- LRN layer (implemented in CDP) does not support any precision conversion
- Pooling layer (implemented in PDP) does not support any precision conversion.

Here is the summary:

Table 29 - Precision conversion for convolutional layer

Configured input format	Configured output precision	Real precision in pipeline	Corresponding weight precision
image input (uint8/ int16/uint16)	int8	int8	int8
	int16	int16	int16
	fp16	fp16	fp16
image input (fp16)	int8	Invalid case	Invalid case
	int16	Invalid case	Invalid case
	fp16	fp16	fp16
int8 feature data	int8	int8	int8
	int16	Invalid case	Invalid case
	fp16	Invalid case	Invalid case
int16 feature data	int8	Invalid case	Invalid case
	int16	int16	int16
	fp16	Invalid case	Invalid case
fp16 feature data	int8	Invalid case	Invalid case
	int16	Invalid case	Invalid case
	fp16	fp16	fp16

Table 30 - Precision conversion for SDP layer (offline mode)

Configured input format	Configured output precision	Real precision in pipeline
int8 feature data	int8	int32
	int16	int32
	fp16	Invalid case
int16 feature data	int8	int32
	int16	int32

Configured input format	Configured output precision	Real precision in pipeline
	fp16	int32
fp16 feature data	int8	Invalid case
	int16	fp32
	fp16	fp32

Table 6-3 precision conversion for LRN layer

Table 31 - Precision conversion for LRN layer

Configured input format	Configured output precision	Real precision in pipeline
int8 feature data	int8	int8
	int16	Invalid case
	fp16	Invalid case
int16 feature data	int8	Invalid case
	int16	int16
	fp16	Invalid case
fp16 feature data	int8	Invalid case
	int16	Invalid case
	fp16	fp16

Table 32 - Precision conversion for pooling layer

Configured input format	Configured output precision	Real precision in pipeline
int8 feature data	int8	int8
	int16	Invalid case
	int16	Invalid case
int16 feature data	int8	Invalid case
	int16	int16
	fp16	Invalid case
fp16 feature data	int8	Invalid case
	int16	Invalid case
	fp16	fp16

For pixel formats, the conversion to int8/int16/fp16 follows the equation below.

$$d_{int8} = truncate2int8\left((d_{\text{pixel}} - offset) * SF\right)$$

$$d_{int16} = truncate2int16\left((d_{\text{pixel}} - offset) * SF\right)$$

$$d_{fp16} = int2fp\left((d_{\text{pixel}} - offset) * SF\right)$$

Equation 1 pixel precision conversion

Here *SF* refers to scaling factor, *offset* refers to offset value. They are both given by programmable register fields.

For conversion between int16 and int8, the equations are:

$$d_{int8} = truncate2int8\left((d_{int16} - offset) * SF\right)$$

$$d_{int16} = truncate2int16\left((d_{int8} - offest) * SF\right)$$

Equation 3 precision conversion between int8 and int16

The CDMA and SDP convert precision individually. When working in on-flying mode, SDP takes precision of convolution pipeline output as input precision then do another precision conversion, but the input precision and output precision should have the same bit-depth.

FP16 Supporting

This section describes NVDLA how to support fp16 in data-path.

- Infinity

NVDLA treats infinity value as different normalized value module by module:

Sub-module	INF converted values
Convolution pipeline	+/-65536 (DC/IMG) +/-65504 (Winograd)
SDP	+/-3.40282e+38
CDP	+/-4292870144
PDP	+/-4292870144 (For AVE) INF (For Max/Min)

There won't be any INF output from any NVDLA sub-module, if saturation happens, NVDLA will output the maximum representable (+/-65504 for FP16, 32767/-32768 for INT16, 127/-128 for INT8).

- NaN

NVDLA won't generate NaN since no infinity value involves in any operation. But it supports NaN propagation. If input data have NaN, any result related to NaN operand will be NaN (mantissa propagation behavior is undefined).

NVDLA provides a register field to flush NaN to Zeros. If the register is set, all input NaNs are treated as zero value in float point data-path and output data cube doesn't have any NaN. Otherwise input NaNs propagate to output.

NVDLA also provide input/output NaN counting registers that summarize total NaN number in input/output data cube. The counting registers are updated when layer is done. When done interrupts arrives, FW can poll NaN counting registers to figure out whether input/output data cubes have any NaN value.

- Denormalized value

NVDLA supports denormalized value for both input and output. The dealing of denormalized value is completely following the requirement of IEEE754 standard.

Actually, NVDLA internal float point data-path often provide fp17/fp32 value for better precision. These fp17 and fp32 format doesn't support denormalized value during calculation. Even though these formats have better precision than fp16 with denormalized value. Before writing back to memory, fp17/fp32 will convert to fp16 with denormalized value.

- Rounding

NVDLA supports Rounding to Nearest (or RN) in calculation except overflow case. If the result is exceeding maximal normal value, it will be clipped to max normalized value.

Feature Data Format

DLA engine maintains a private data format for all supported HW-layers. The data format is called feature data format. This format is only generated by DLA engine itself.

All elements of feature data for one layer are organized as a 3D data cube. Three dimensions are width (W), height (H) and channel size (C). The memory mapping rules are:

- Adding data into end of channel if the original data is not 32byte aligned in C direction.
- The attached data can be any value except NaN when it's fp16.
- Split the data cube into 1x1x32byte small atom cubes.
- Reordering atom cubes in by progressively scanning the data cube. Scanning order: W (line) -> H (height) -> C (channel).
- Map all atom cubes into memory by scanning sequence.
- All atom cubes in the same line are mapped compactly.
- Atom cube mapping at line boundary and/or surface boundary can be either adjacently or incompactly. But they are always 32-byte aligned.
- In conclusion, mapping in memory follows pitch linear format. The order is C' (32byte) -> W -> H -> C (surfaces). Here C' changes fastest and C changes slowest.

Fig. 10 is a case of feature data that all small cubes are mapped compactly. This is called packed feature data. If the line or surface of small cubes is not mapped compactly, it is called unpacked. See Fig. 11.

Note
Line stride and surface stride of feature data shall always align to 32bytes. Start address has same alignment as well. This is mandatory requirement.

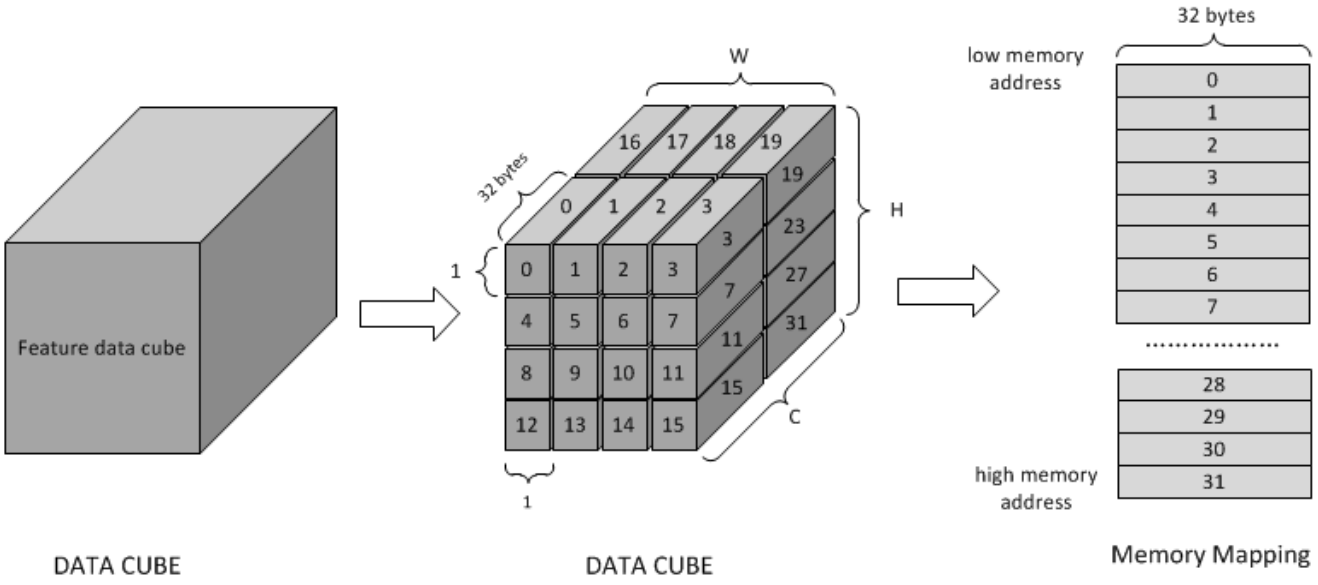


Fig. 10 - Packed feature data

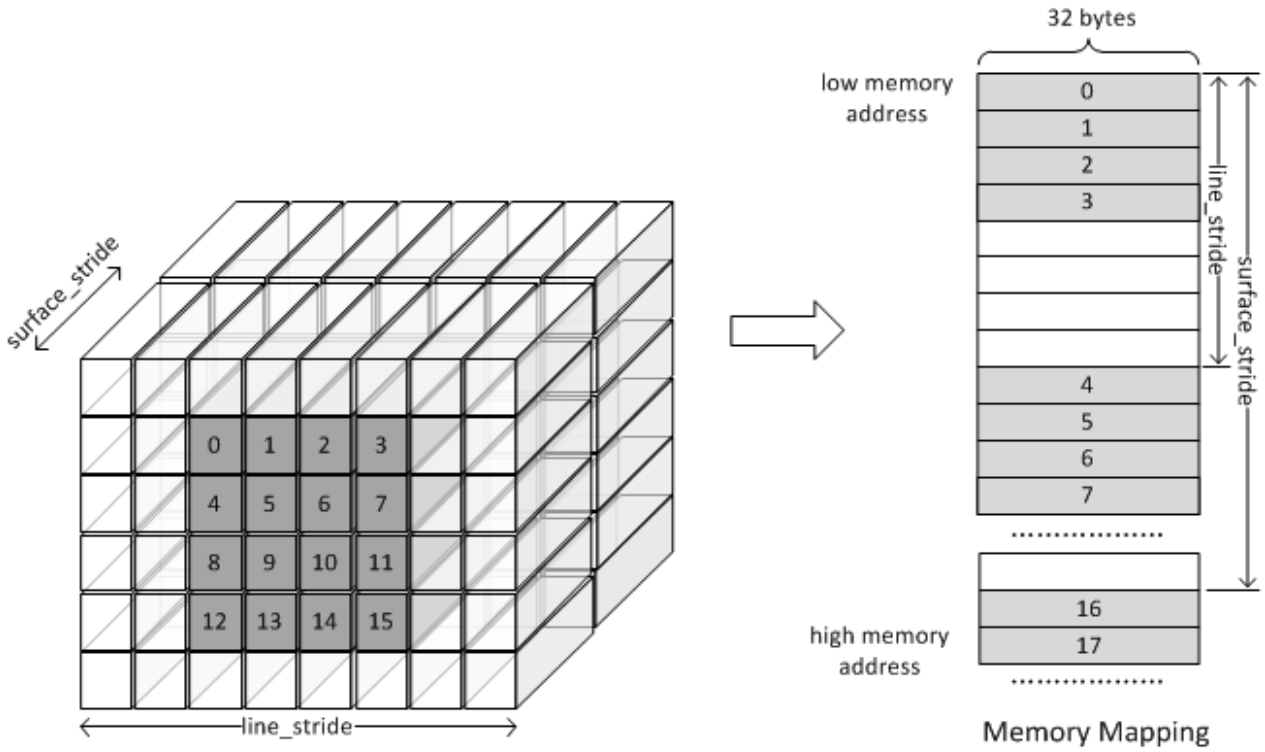


Fig. 11 - Unpacked feature data

If a 1x1xC feature data cube maps as surface-packed, NVDLA can treat it like (C/32) x1x 32 cube to save bandwidth.

Mapping of feature data cube is done by NVDLA core logic. Falcon does not involve in mapping procedures.

Pixel Format

DLA engine supports pixel data for ROI. The pixel data comes from a part or a whole image. The pixel formats are listed in table Table 33.

When NVDLA takes image as input data, there are some limits of configuration.

- Channel size. The valid channel size highly depends on each format. Please see table Table 33.
- Input precision. The input precision highly depends on pixel each format. Please see table Table 33. DMA logic will turn unsigned integer value to signed integer value automatically.
- **Both start address and line stride of pitch linear shall aligned to 32 bytes. This is mandatory requirement.**
- It may have redundant data between 32-byte aligned address and first element. NVDLA use x offset to indicate how many redundant data are. The unit of offset is pixel.

Table 33 - Pixel formats and valid setting

Format Name	# of planar	Valid channel size setting	Valid input precision setting	Valid X offset range
T_R8	1	1	int8	0~31
T_R10	1	1	int16	0~15
T_R12	1	1	int16	0~15
T_R16	1	1	int16	0~15
T_R16_I	1	1	int16	0~15
T_R16_F	1	1	int16	0~15
T_A16B16G16 R16	1	4	int16	0~3
T_X16B16G16 R16	1	4	int16	0~3

Format Name	# of planar	Valid channel size setting	Valid input precision setting	Valid X offset range
T_A16B16G16 R16_F	1	4	fp16	0~3
T_A16Y16U16 V16	1	4	int16	0~3
T_V16U16Y16 A16	1	4	int16	0~3
T_A16Y16U16 V16_F	1	4	fp16	0~3
T_A8B8G8R8	1	4	int8	0~7
T_A8R8G8B8	1	4	int8	0~7
T_B8G8R8A8	1	4	int8	0~7
T_R8G8B8A8	1	4	int8	0~7
T_X8B8G8R8	1	4	int8	0~7
T_X8R8G8B8	1	4	int8	0~7
T_B8G8R8X8	1	4	int8	0~7
T_R8G8B8X8	1	4	int8	0~7
T_A2B10G10R 10	1	4	int16	0~7
T_A2R10G10B 10	1	4	int16	0~7
T_B10G10R10 A2	1	4	int16	0~7
T_R10G10B10 A2	1	4	int16	0~7
T_A2Y10U10V 10	1	4	int16	0~7
T_V10U10Y10 A2	1	4	int16	0~7
T_A8Y8U8V8	1	4	int8	0~7
T_V8U8Y8A8	1	4	int8	0~7
T_Y8___U8V8 _N444	2	3	int8	0~31
T_Y8___V8U8 _N444	2	3	int8	0~31
T_Y10___U10 V10_N444	2	3	int16	0~15
T_Y10___V10 U10_N444	2	3	int16	0~15
T_Y12___U12 V12_N444	2	3	int16	0~15
T_Y12___V12 U12_N444	2	3	int16	0~15
T_Y16___U16 V16_N444	2	3	int16	0~15
T_Y16___V16 U16_N444	2	3	int16	0~15

Weight Format

Unlike pixel data or feature data, weight data are generated long before convolution operation. And DLA engine never changes them during operation. Software should map weight data with property rules to fit with the calculation sequence in DLA.

The original weight data has 4 dimensions: width, height, channel and number of kernels. They can construct as a group of 3D data cubes. One data cube is called a kernel. See Fig. 12.

DLA engine support 4 types of weight data. They are weight for direct convolution, weight for Winograd convolution, weight for image input and weight for deconvolution. There are two options for weight to improve DLA performance: sparse compression and channel post-extension.

DLA engine support 4 basic formats of weight data for different operation mode:

- weight for direct convolution
- weight for image input
- weight for deconvolution
- weight for Winograd convolution

There are some mandatory requirements for some formats:

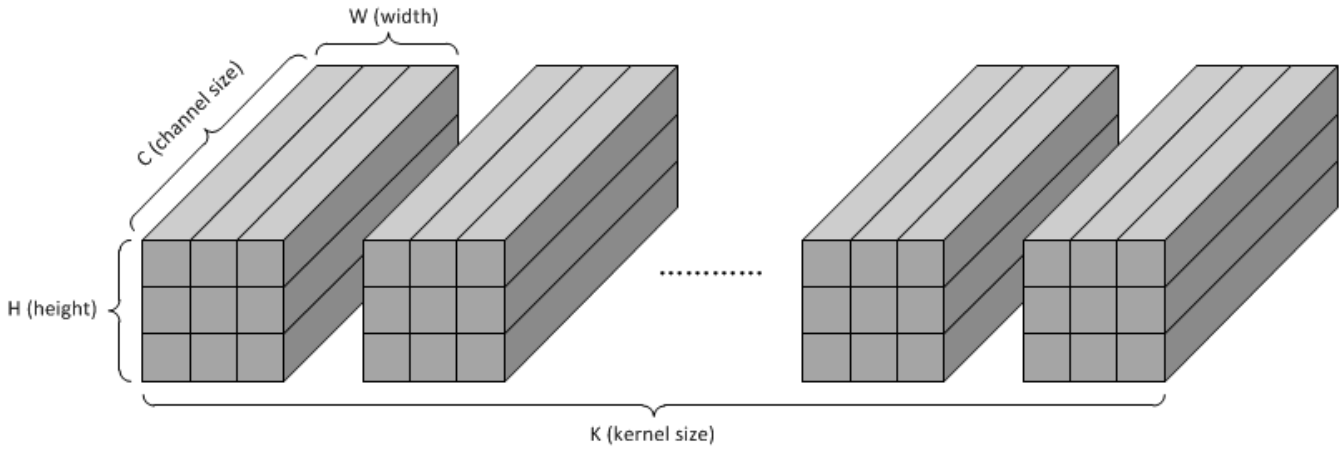
- channel pre-extension for image input
- channel extension for Winograd
- Set split for deconvolution

And two options for weight formats:

- channel post-extension
- sparse compressing

Table 34 - Weight formats and options

Weight types	Sparse compression option	Post-extension option
Weight for DC	Support	NOT support
Weight for Winograd	Support	NOT support
Weight for image input	Support	Support
Weight for deconvolution	Support	NOT support



(../_images/format_original_weight_data.svg)

Fig. 12 - Original weight data

Basic Weight for Direct Convolution

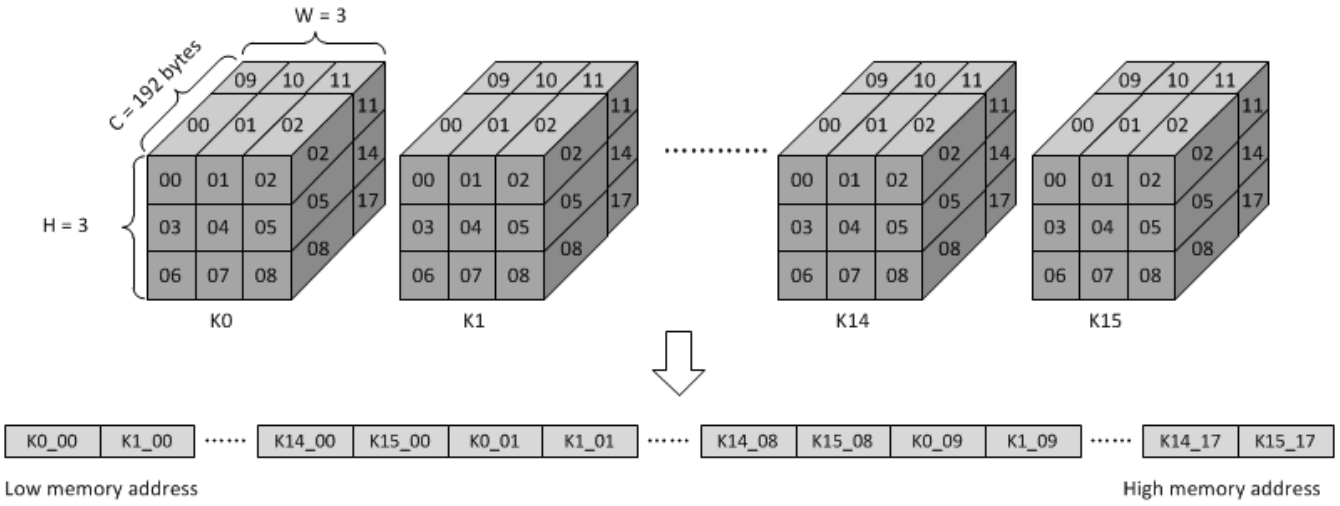
Basic weight for direct convolution is the most basic and common weight format. Other weight formats are all extended from this format.

The mapping rules of uncompressed weight for direct convolution are:

- Distribute the kernels into groups. For int16 and fp16 weight, one group has 16 kernels. For int8, one group has 32 kernels. Last group can have fewer kernels.
- Divide each kernel to 1x1x64-element small cubes. For int16/fp16 the small cube is 128 bytes each; and for int8 the small cube is 64 bytes each. Do not append 0 if channel size is not divisible by 128/64.
- After division, all weights are stored in 1x1xC' small cubes, where C' is no more than 128 bytes.

- Scan the 1x1xC' small cubes in a group with C'->K->W->H->C sequence. Here C' changes fastest and C changes slowest. And map them compactly as scanning sequence.
- Map the weight groups compactly. Do not append any 0s between group boundaries.
- Append 0s at end of all mapped weight for 128-byte alignment.

Diagram below shows how a group of 3x3x192Byte kernel maps for direct convolution.



Weight mapping in memory

(../_images/format_dc_weight_mapping.svg)

Fig. 13 - Weight mapping for direct convolution inside one group

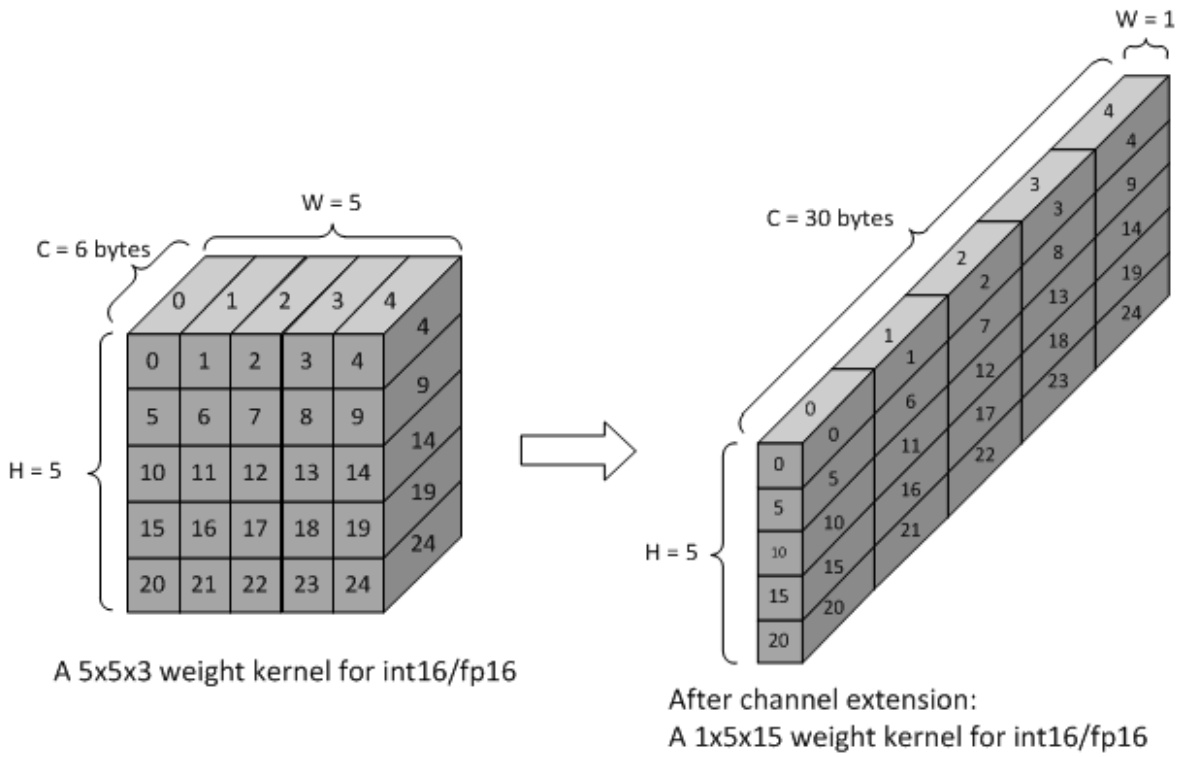
Basic Weight for image input

Weight mapping for image input is like weight for direct convolution. The main difference is that image weight needs an additional channel extension step ahead of mapping steps for direct convolution weight.

The channel pre-extension for image weight is a mandatory requirement, while channel post-extension is an option to improve performance.

Note
Channel pre-extension for image weight is different from channel extension for Winograd convolution.

The key idea of per-extension is to turn all weights in same line to a single channel. Fig. 14 is a case for an int16 image input whose channel size is 3.



(../_images/format_dc_channel_extension_for_image_for_weight.svg)

Fig. 14 - Channel extension for image weight

Channel pre-extension is the first step for image weight. Then all extended kernels follow the same steps of weight for direct convolution. That is, SW still need to do group and channel distribution after channel extension.

Basic Weight for Winograd Convolution

The memory mapping of Winograd weight is very different from direct convolution. There are two phases to process the weights. Phase 1 is to do channel extension and conversion for each kernel. Phase 2 is to group the kernels and map small cubes in memory.

Steps of phase 1:

- Divide kernels to 1x1x32Byte small cubes. If the channel size is not divisible by 32, append 0s.
- Do channel extension in if convolution stride is not 1. The new width and height of a kernel should be 3 after extension.
- Convert the kernel from 3x3xC cube to a 4x4xC cube. The equation is GWGT. Here W is each 4x4x1 of weight cube, G is a 4 x 3 matrix and GT is transpose matrix.
- During conversion, a scaling factor may involve. Please see the Winograd convolution documentation for reference.
- The width and height of a kernel should be 4 after conversion.

$$G = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0.5 & -0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix}$$

Matrix for weight transfer for Winograd

Steps of phase 2:

- Distribute the converted kernels into groups. For int16 and fp16 weight, one group has 16 kernels. For int8, one group has 32 kernels.
- Divide converted kernels to 4x4x4 elements small cubes. For int16/fp16 small cube is 128 bytes each. For int8 small cube is 64 bytes each. The channel size should always divisible by 4.
- Scan the 4x4x4 elements small cubes in a group with K->C sequence. Take int16 for example, the scan order is small cube 0 of K0, small cube 0 of K1, small cube 0 of K2, ..., small cube 0 of K15, small cube 1 of K0, small cube 1 of K1, ..., small cube 1 of K15, ..., small cube N of K15.
- Maps the 4x4x4 elements small cubes closely with scanning order
- Maps the weight groups one by one closely

The phase 2 is similar to weight for direct convolution except the small cube size is 4x4x4 elements.

Figure below shows how to do channel extension to one kernel and map the data.

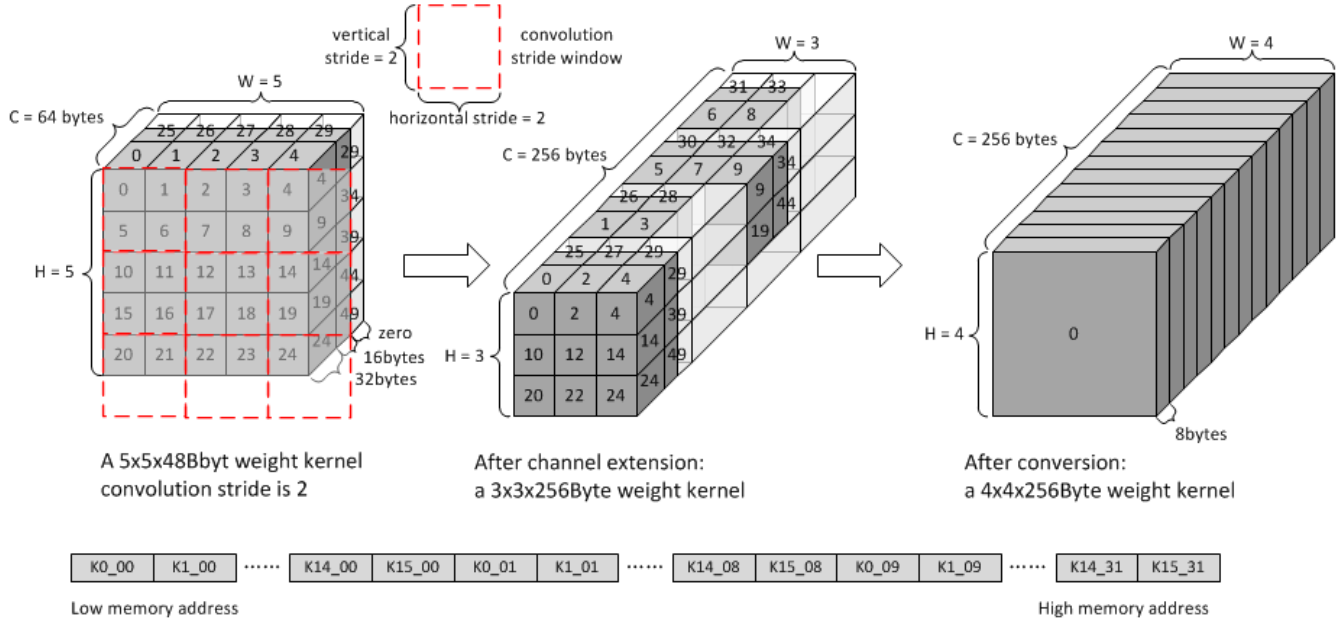


Fig. 15 - Channel extension and conversion for Winograd

Weight Channel Post-extension for image input

Weight channel post-extension is an option to enhance MAC efficiency when channel size is less than 32. It is available for image input mode only.

Key idea of channel post-extension is to combine two neighbor lines to saving the efficiency. It allows two-line (C<=32) or four-line (C<=16) combination. 1, 2 and 4 parameters are available.

If this option is enabled, NVDLA manage to post-extend input feature (or image) data in CSC sub units. And SW needs to adjust weight mapping order.

The channel post-extension is done after pre-extension. Figure below shows one case which parameter is 2.

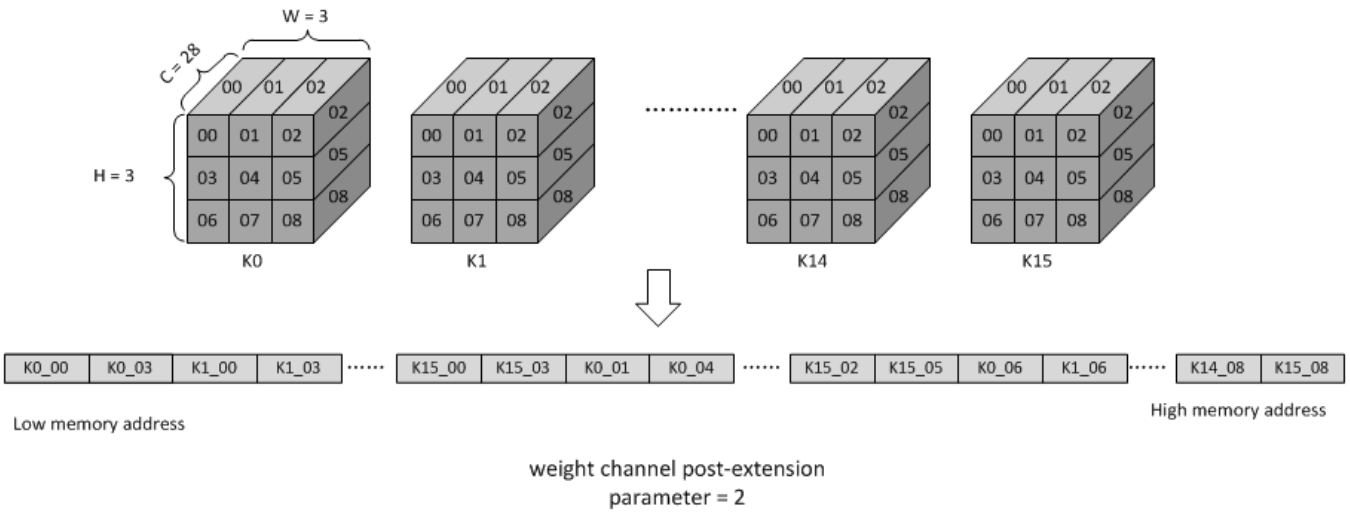


Fig. 16 - Weight channel post-extension, parameter = 2

Flow of pre-extension, post-extension, mapping and compression option for image weight:

- Do pre-extension
- Do post-extension
- Remap weight data
- Do weight compression.

Some tips for post-extension:

- Channel post-extension cannot be used in Winograd convolution
- Channel post-extension only support 2-line and 4-line.
- If weight height is not divisible by 2 (2-lines) or 4 (4-lines), do NOT append 0s. This is unlike channel extension for Winograd.

Sparse Compression option

To reduce the bandwidth and power consumption on memory interface, NVDLA engine support weight sparse compression option. All four weight formats can support sparse compression. This option requires additional steps after basic mapping and post-extension option.

Sparse algorithm uses one-bit tag to indicate a weight element is zero or not. Bit tags of one kernel group compose a weight mask bit group, or WMB. WMBs reside in a dedicate memory surface. Since 0 values are marked by bit tags (assign 0 to corresponding bit), they can be removed from original weight memory surface. A third memory surface recodes remaining byte number of each kernel group (WGS).

The steps of weight compression are:

- Always use 1 bit to indicate 1 element of weight. For int16 and fp16, 1 bit represents 2 bytes of weight data; for int8, 1 bit represents 1 byte of weight data.
- Compress weight group by group. Assembly of bits for one weight group is called WMB. The bits in WMB are stored as little-endian.
- Align WMB surface to 128-byte by adding 0 bits in the end
- Remove all 0 weights in original surface and pack them compactly.
- Align compressed weight surface to 128-byte by adding 0s in the end.
- Calculate the byte number of each compressed group. The remaining byte number of each group is called weight group size or WGS. One WGS is of 32-bit wise.
- Store WGS, WMB and compressed weight into three separated memory surfaces.

The diagram below shows the memory mapping of compressed weight format.

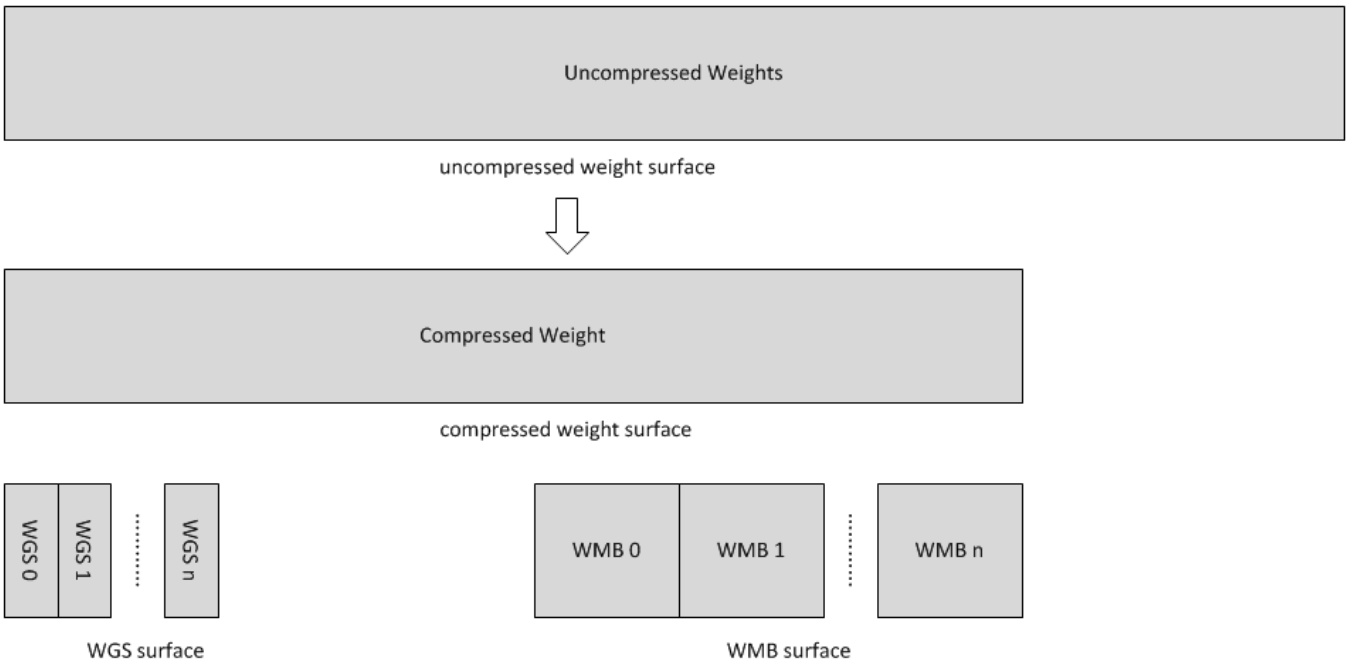


Fig. 17 - Memory mapping of compressed weight

Bias Data Format

Bias data is another optional input data for convolution layers. When this option is enabled, DLA engine will add the bias data to result of convolution before writing back to memory.

There are three types of bias data,

- Per layer bias data
- Per channel bias data
- Per element bias data

They both store in memory for DLA engine to fetch.

If the output feature data cube is WxHxC, check below table for the corresponding bias cube size:

Per Layer	1x1x1 (Register)
Per Channel	1x1xC
Per Element	WxHxC

For INT pipeline, bias data can be either INT8 or INT16, and FP16 type of bias data is in16-bit fp16 format. They are generated along with CNN network.

The memory mapping of bias data is described as below:

Per Channel:

- Two bytes per element with INT16/FP16 or 1 byte per element with INT8

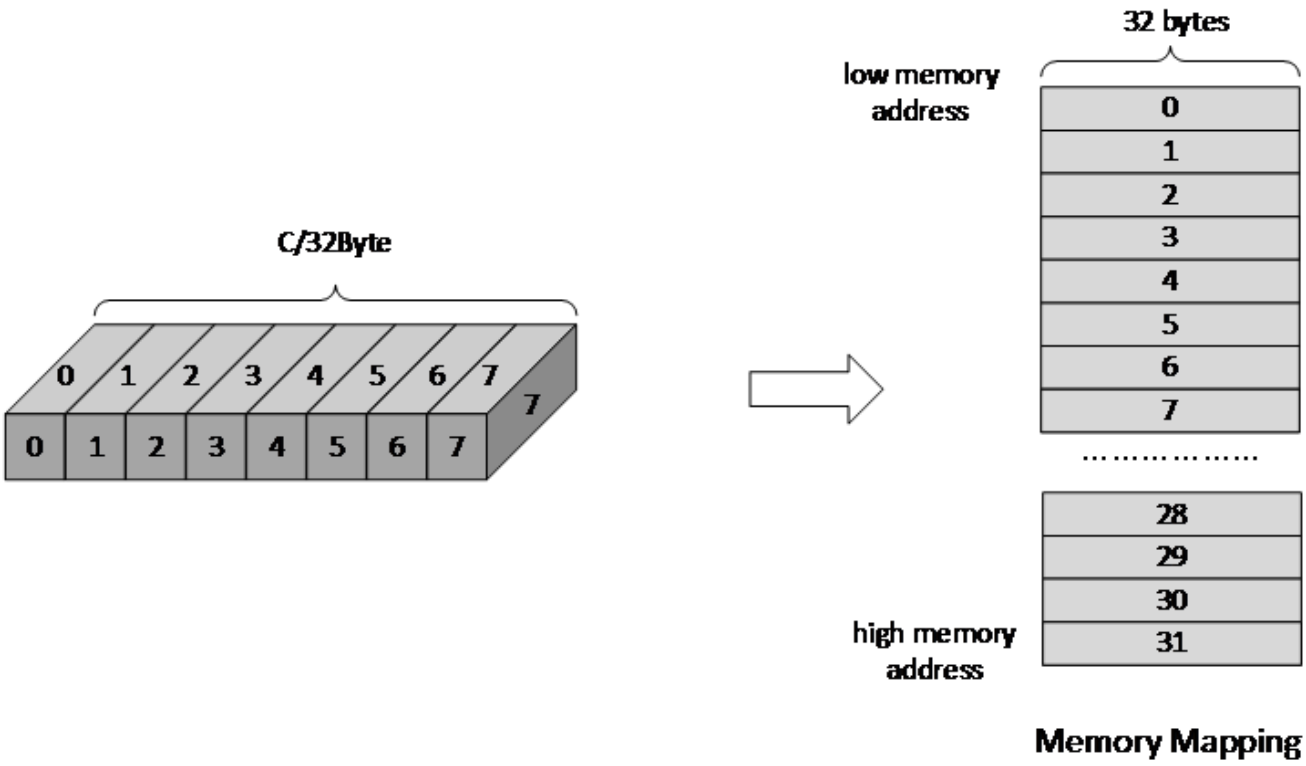


Fig. 18 - Memory Mapping of Per Channel Bias Data (Case 1)

- 2 bytes per element with INT8:

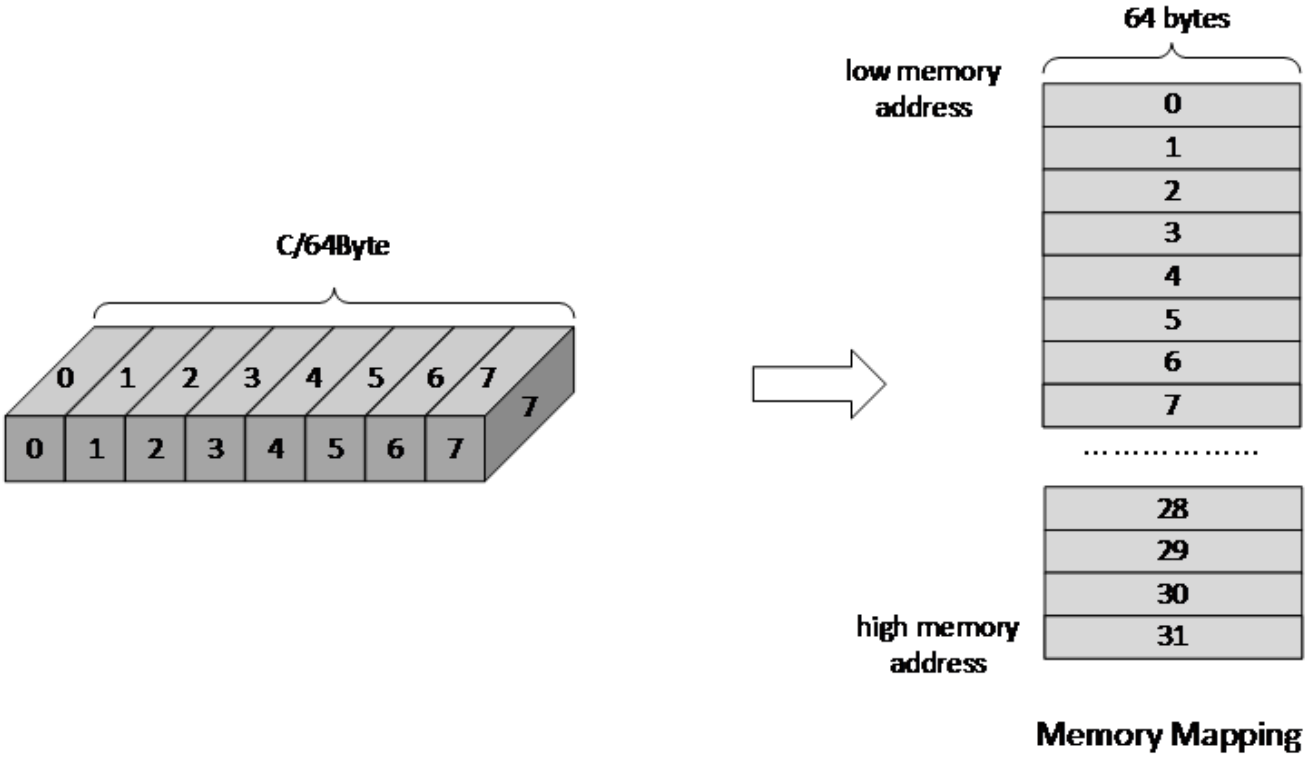


Fig. 19 - Memory Mapping of Per Channel Bias Data (Case 2)

- 2 bytes per element with INT8:

Per Element:

- Two bytes per element with INT16/FP16 or 1 byte per element with INT8

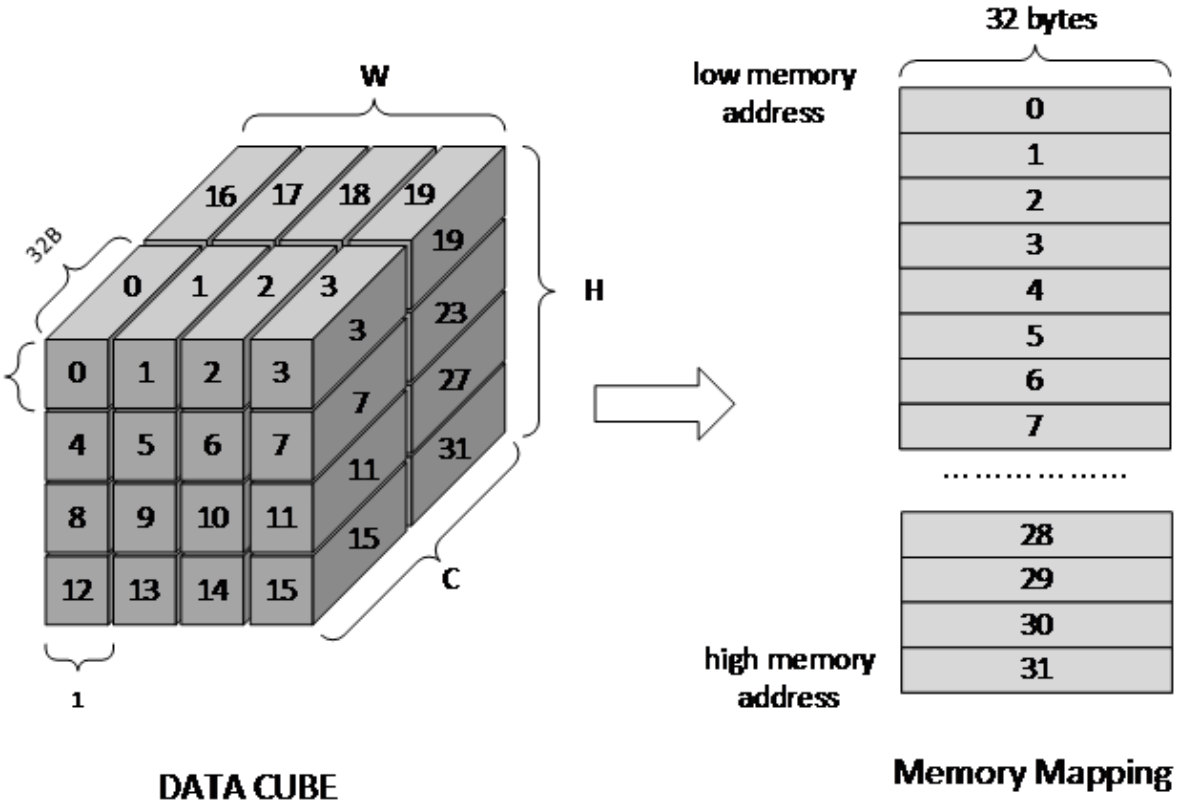


Fig. 20 - Memory Mapping of Per Element Bias Data (Case 1)

- 2 bytes per element with INT8:

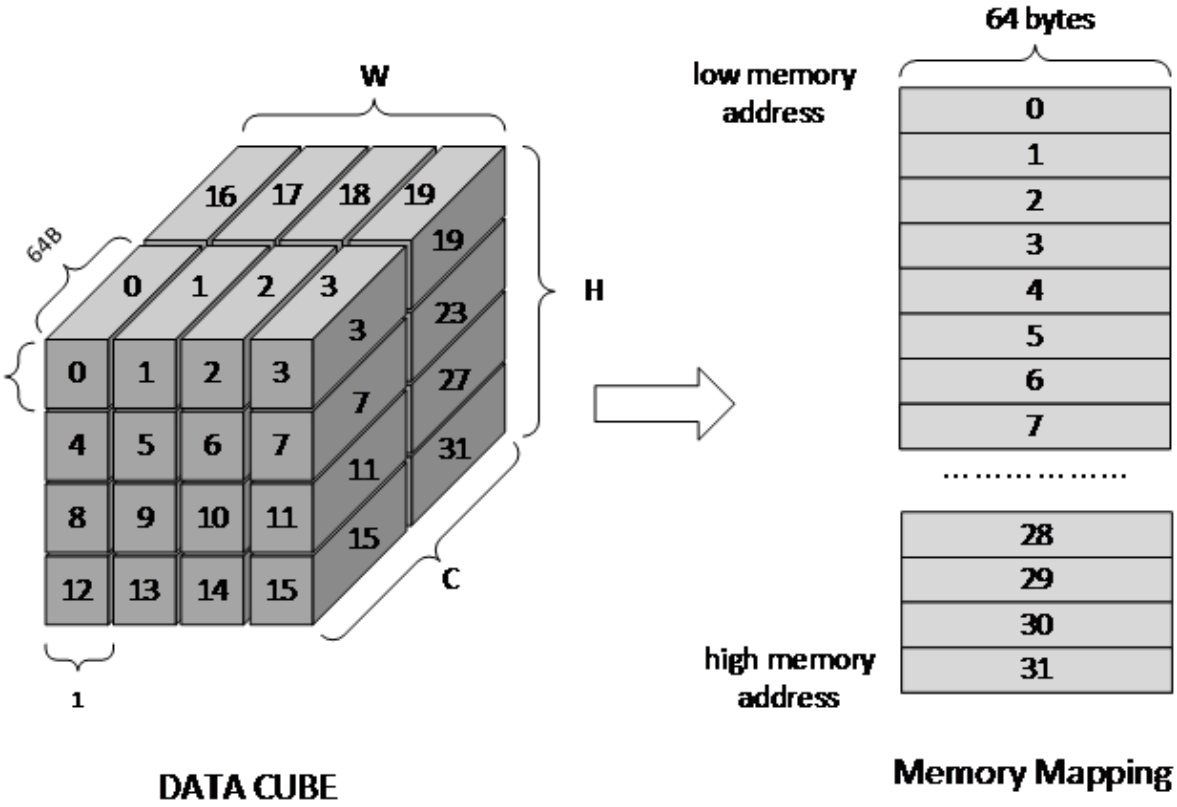


Fig. 21 - Memory Mapping of Per Element Bias Data (Case 2)

PReLU Data Format

Each PReLU data just have one component and it will be fed into multiplier of SDP.

PReLU always operated per-channel thus there is only one type of PReLU data:

- Per channel PReLU data

Per channel PReLU data is stored in memory in a continuous 1x1xC space. Be noted that C is in unit of channel.

- For INT8/16, each channel can occupy 1 or 2 bytes depending on B/N/E RDMA_DATA_SIZE
- In FP16 types, each channel need 2 bytes data

The memory mapping of PRelu data is described as below:

- Two bytes per element with INT16/FP16 or 1 byte per element with INT8

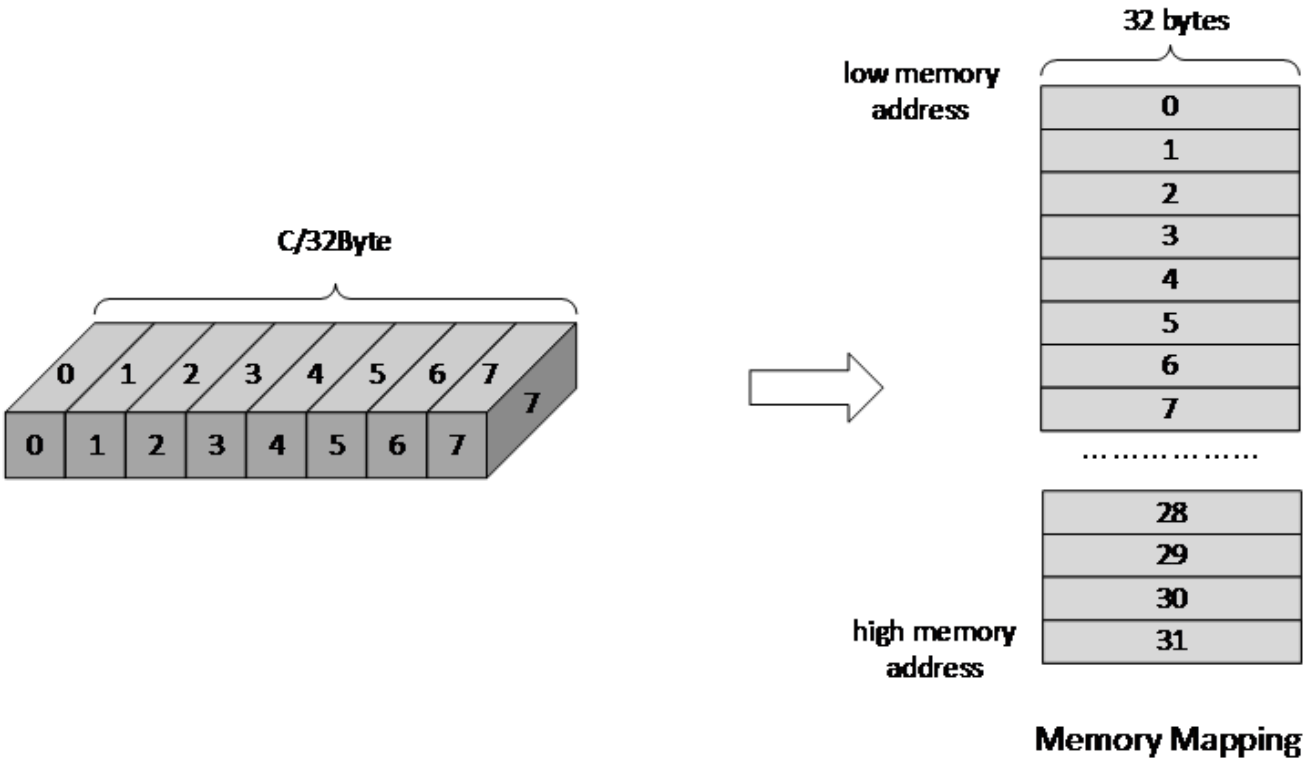


Fig. 22 - Memory Mapping of Per Channel PReLU Data (Case 1)

- 2 bytes per element with INT8:

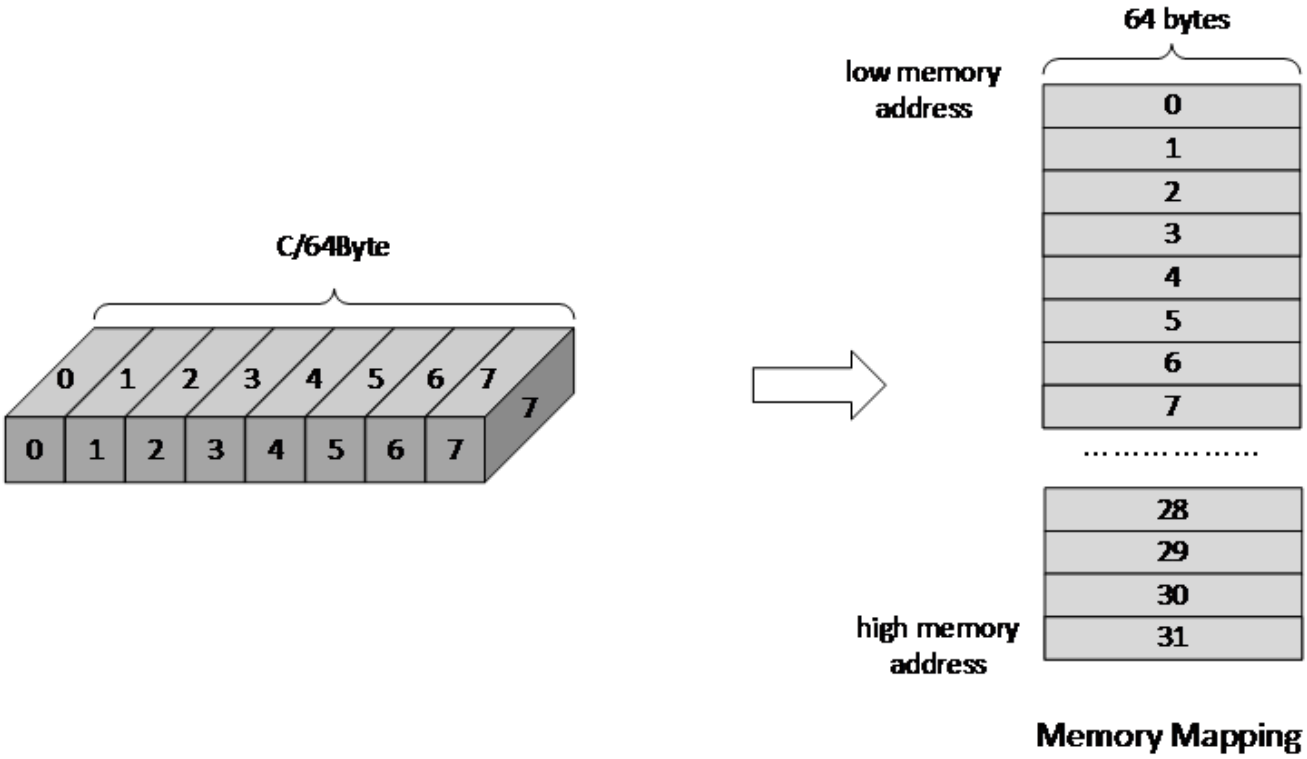


Fig. 23 - Memory Mapping of Per Channel PReLU Data (Case 2)

Batch Normalization Data Format

Batch Normalization data is another optional input data for batch normalization layers.

Each normalization data consists of two parts, one is to add onto the feature data and the other is to multiple with the result after addition.

There are two types of batch normalization data

- Per channel batch normalization data
- Per layer batch normalization data

Per channel batch normalization data is stored in memory in a continuous 1x1xC space. Be noted that C is in unit of channel.

- In INT8/16 types, each of the two parts of normalization data can be either 1 byte or 2 bytes, so each channel need 2*1 or 2*2 bytes data
- In FP16 types, each of the two parts of normalization data is 2 byte, so each channel need 4 bytes data

The pair data of each element are always packed together in memory. The memory mapping of data is described as below:

- Two bytes per element with INT16/FP16 or 1 byte per element with INT8

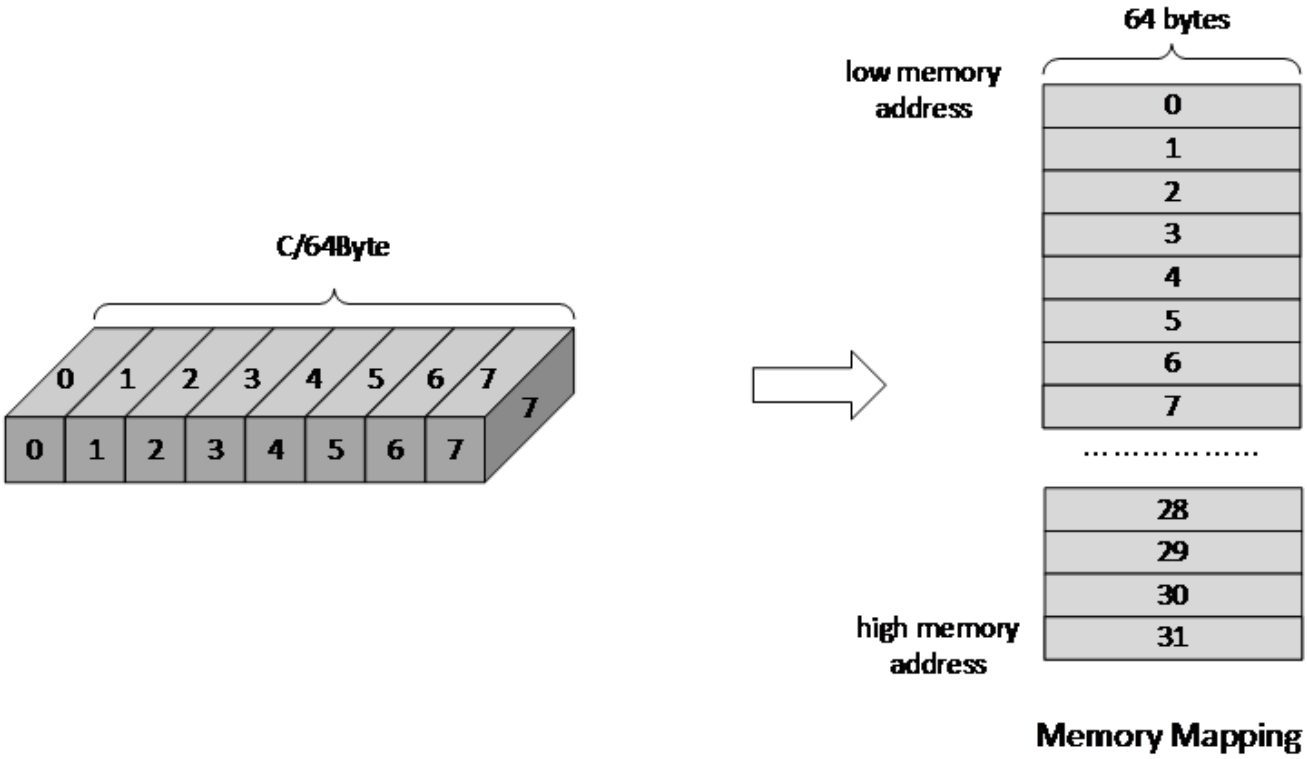


Fig. 24 - Memory Mapping of Batch Normalization Data (Case 1)

- 2 bytes per element with INT8:

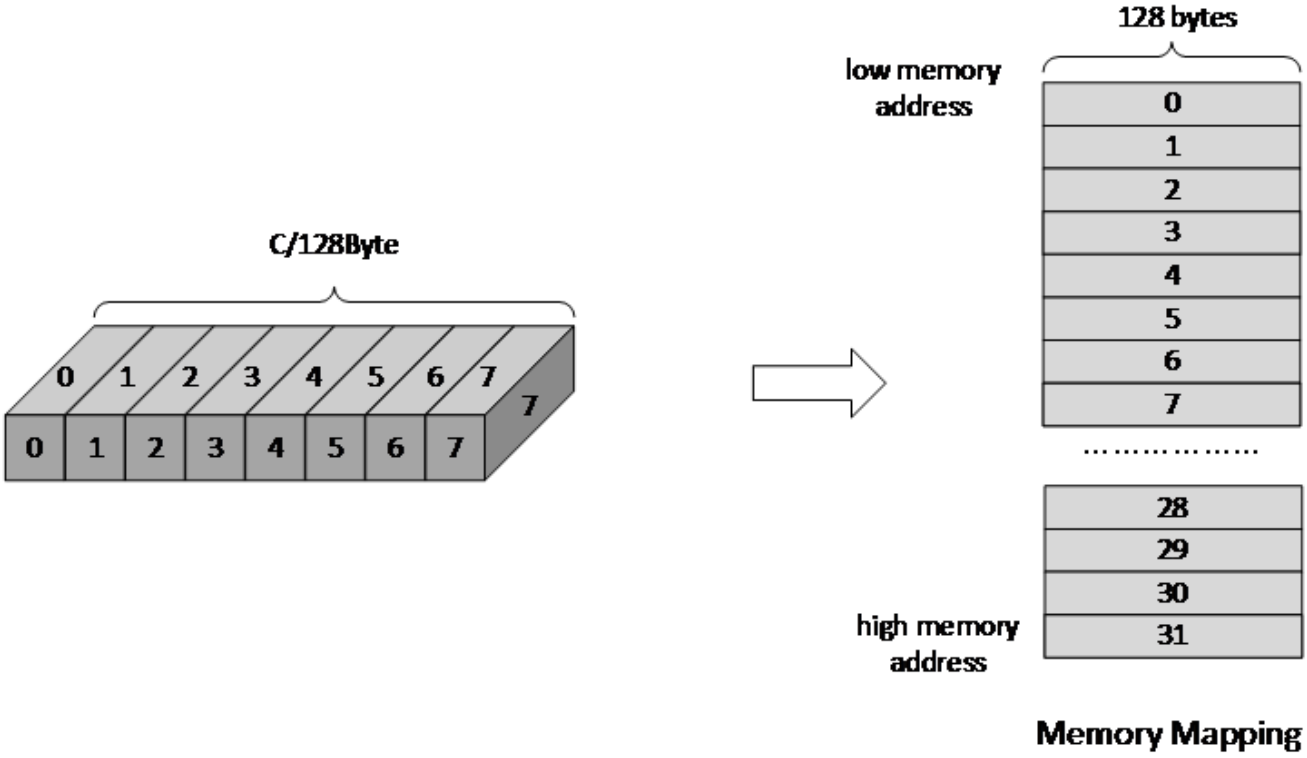


Fig. 25 - Memory Mapping of Batch Normalization Data (Case 2)

Per layer batch normalization data is stored in register.

Be noted that INT8 and INT16 here means the processing precision, so when the layer is running from INT16 to INT8 or INT8 to INT16 precision conversion, batch normalization data need set to processing precision which is always INT8.

Element-Wise Data Format

Element-Wise data is another optional input data for Element-Wise layers.

Each Element-Wise data consists of just one part and either for ALU or multiplier.

There are one type of element-wise data

- Per element Element-Wise data

Per element Element-Wise data is stored in memory with size of W x H x C.

- In INT8 /16types, each of the two parts of element-wise data can be either 1 byte or 2 bytes, so each element need 1/2 bytes data
- In FP16 types, each of the two parts of element-wise data is 2 bytes, so each element need 2 bytes data

From algorithm perspective, element-wise employs ALU or MUL only but never both, however, DLA hardware support employ both operations for per-element operation, in this case, each element size should be x2 of description above;

The memory mapping of data is described as below:

- Two bytes per element with INT16/FP16 or 1 byte per element with INT8

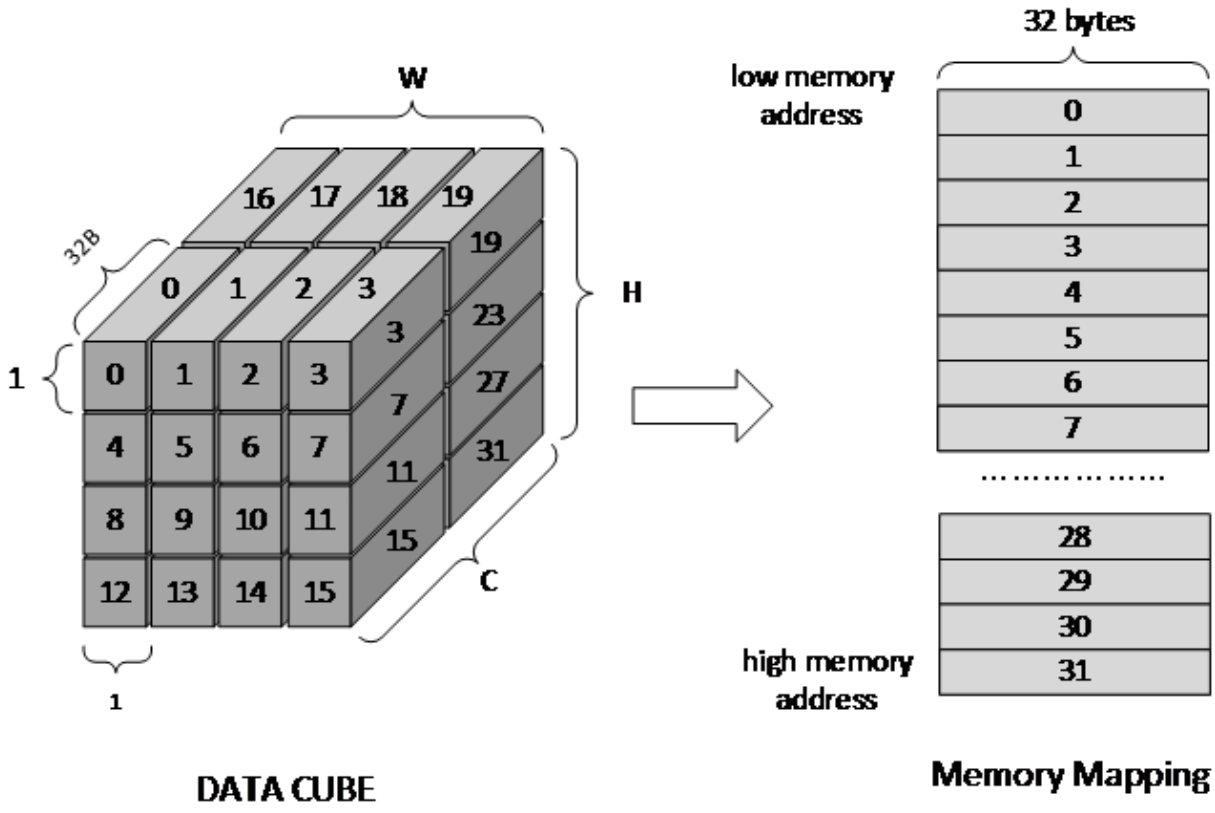


Fig. 26 - Memory Mapping of Element Wise Data (Case 1)

- 2 bytes per element with INT8:

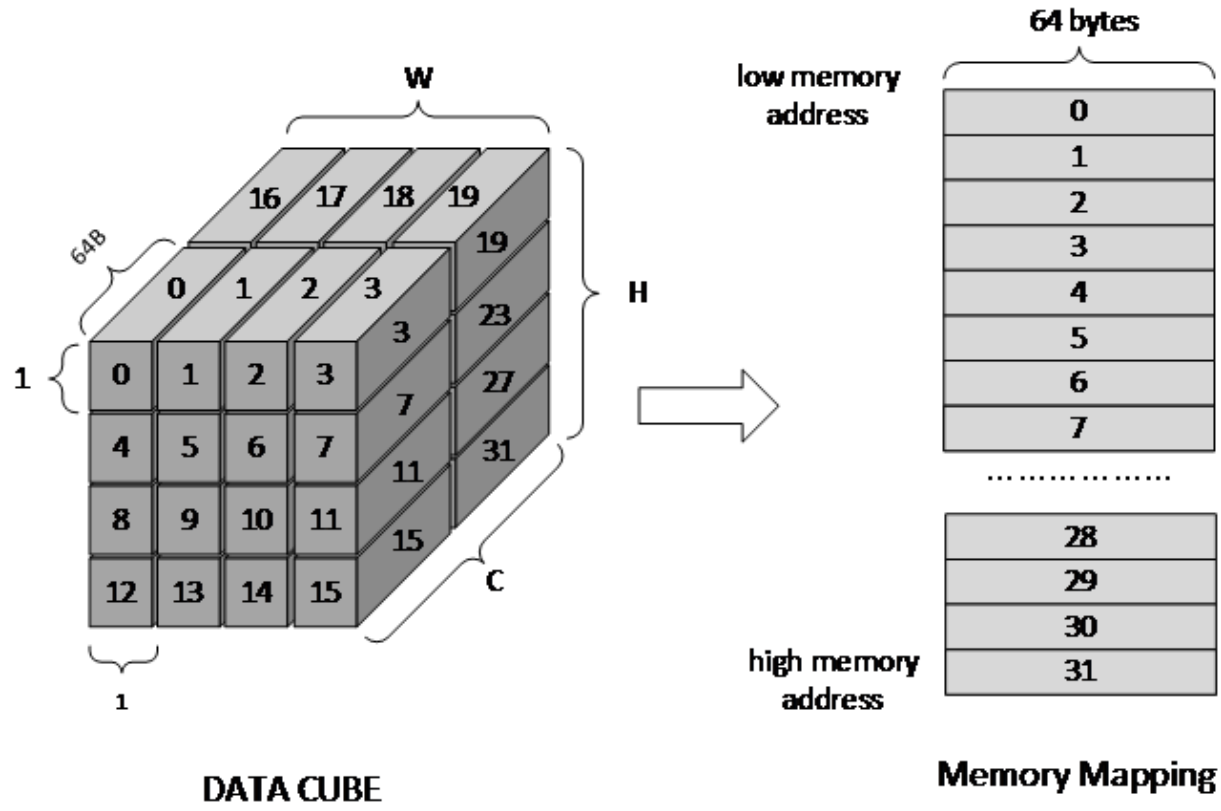


Fig. 27 - Memory Mapping of Element Wise Data (Case 2)

Be noted that INT8 and INT16 here means the processing precision, so when the layer is running from INT16 to INT8 or INT8 to INT16 precision conversion, Element-Wise data need set to processing precision which is always INT8.

Normally, one atom contains 1x1x32Bytes data, but it's no longer true for:

- Bias data format;
- PReLU data format;
- Batch normalization data format;
- Element-wise data format

The bytes-per-atom for those formats should be computed by:

BytesPerAtom=ElementPerAtom * ComponentsPerElement * BytesPerComponent

Where ElementPerAtom is decided by PROC_PRECISION of SDP data pipeline:

PROC_PRECISION	ElementPerAtom
INT8	32
INT16/FP16	16

ComponentsPerElement is decided by use case (or DATA_USE register):

Use case	ComponentsPerElement
Bias	1
PReLU	1
BatchNormalization	2
Element-wise (Only ALU or MUL enabled)	1
Element-wise (Both ALU/MUL are enabled)	2

BytesPerComponent is decided by precision (or DATA_SIZE register)

DATA_SIZE	BytesPerComponent
ONE_BYTE	1
TWO_BYTE	2

Alignment of Start Address and Stride

Here is the conclusion of requirements of alignment:

Table 35 - Requirements of alignment

Data format	Alignmen t of start address	Alignmen t of line stride	Alignmen t of surface stride	Alignmen t of planar/ cube stride	Alignmen t of data size
Feature data cube	32 bytes	32 bytes	32 bytes	32 bytes	NA
uncompre ssed/ compress ed weight	256 bytes	NA	NA	NA	128 bytes
WMB	256 bytes	NA	NA	NA	128 bytes
WGS	256 bytes	NA	NA	NA	128 bytes
Pitch linear pixel	32 bytes	32 bytes		NA	NA
Bias	32 bytes	32 bytes	32 bytes	NA	NA
PReLU	32 bytes	N/A	N/A	NA	NA
Batch Normaliz ation	32 bytes	NA	NA	NA	NA
Element- wise	32 bytes	32 bytes	NA	NA	32bytes