GEMM Engine W4MXINT8 ()

Achronix
Data Acceleration

– June 15, 2025

Copyrights, Trademarks and Disclaimers

Notice of Disclaimer

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

Website: www.achronix.com
E-mail : info@achronix.com

Table of Contents

Chapter 1 : Introduction

The W4MXINT8 variant of the matrix multiplication engine takes MXINT4 weights and MXINT8 activation inputs, and produces floating point results. The MXINT4 input consists of int4 weight, with an exponent (5-bit or 8-bit) per block of int4 values. (Note: An exponent is equivalent to a scaling factor of the form 2^e, just as used for floating point formats.) Likewise, the MXINT8 input consists of int8 activation values with an exponent per block of int8 values. The matrix multiplication engine uses the same input and output protocol as the original fp version, with the input sizes adjusted for the MXINT formats. This document clarifies the order of inputs and outputs, relative to the matrix.

Chapter 2 : Weight Input

For efficiency, the model weights should be converted to MXINT4 format once, then saved. They should be stored in that format in DRAM, to get the bandwidth benefit of the reduced value size. Compatible models with int4 weights may already be available, or a model with floating point weights can be converted as outlined in the document on Block-Floating Point format.

The diagram illustrates how the int4 weights are organized and input to the matrix multiplication engine.

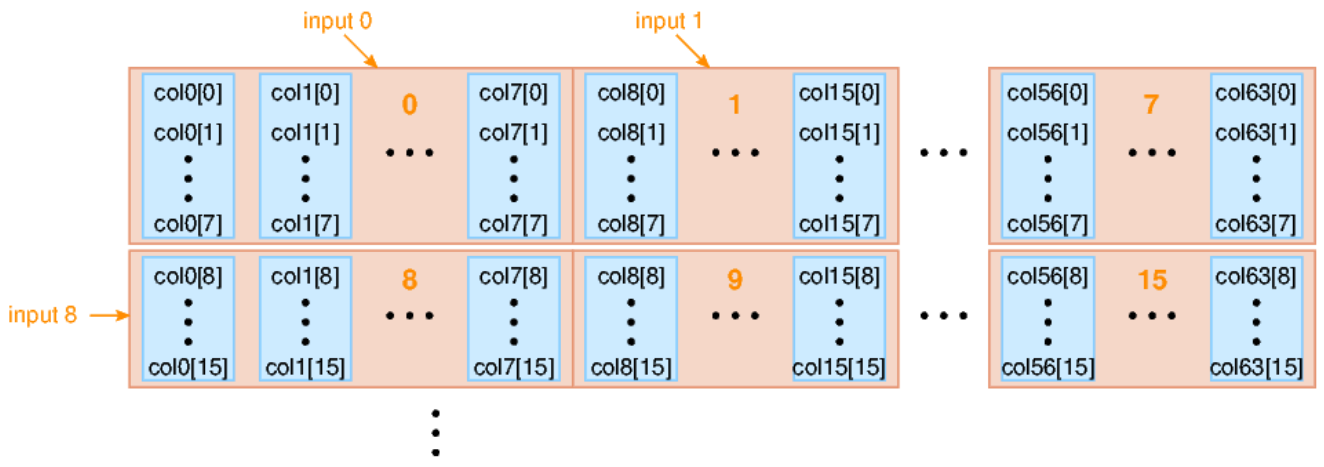

Figure 1 • Weight storage in block-row major order

Weights are stored in external DRAM in block-row major order: First, each column of the weight matrix is divided into blocks of 8 values (the blue blocks in the diagram), then those are arranged in row-major order. Each block is 8 * 4 = 32 bits. Since the i_b input is 256 bits wide (as is the NAP output), each input consists of 8 blocks, of 8 different columns: one orange block in the diagram corresponds to one clock cycle input. The inputs occur in the order indicated.

Each blue block has an associated exponent (a choice of 5 bits or 8 bits). These exponents will have to be read from DRAM separately, and must be passed to the matrix multiplication engine via the i_b_scale input, together with the weights. To reduce the bandwidth overhead of reading the exponents, it makes sense to share the same exponent between several blocks, so that an exponent can be read from DRAM once and passed to the matrix multiplication engine multiple times.

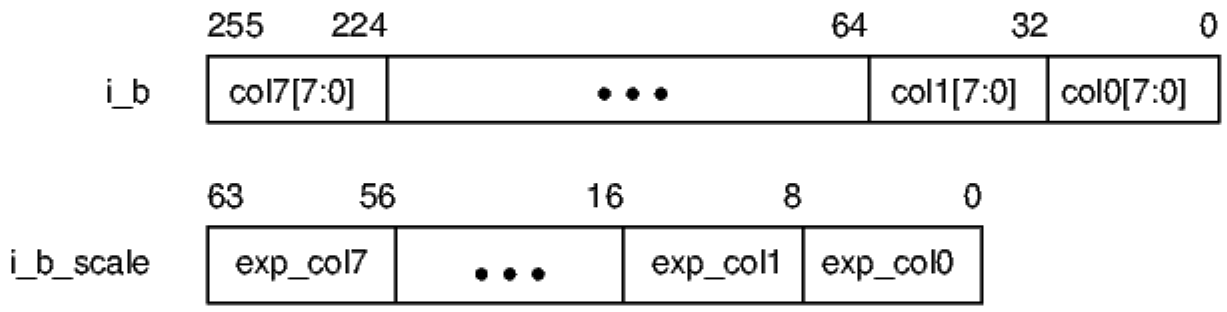

Figure 2 • First weight input to matrix multiplication engine

The above diagram shows how the weights and exponents are input to the matrix multiplication engine (if 5-bit exponents are selected, then i_b_scale is only 40 bits wide).

Chapter 3 : Activation Input

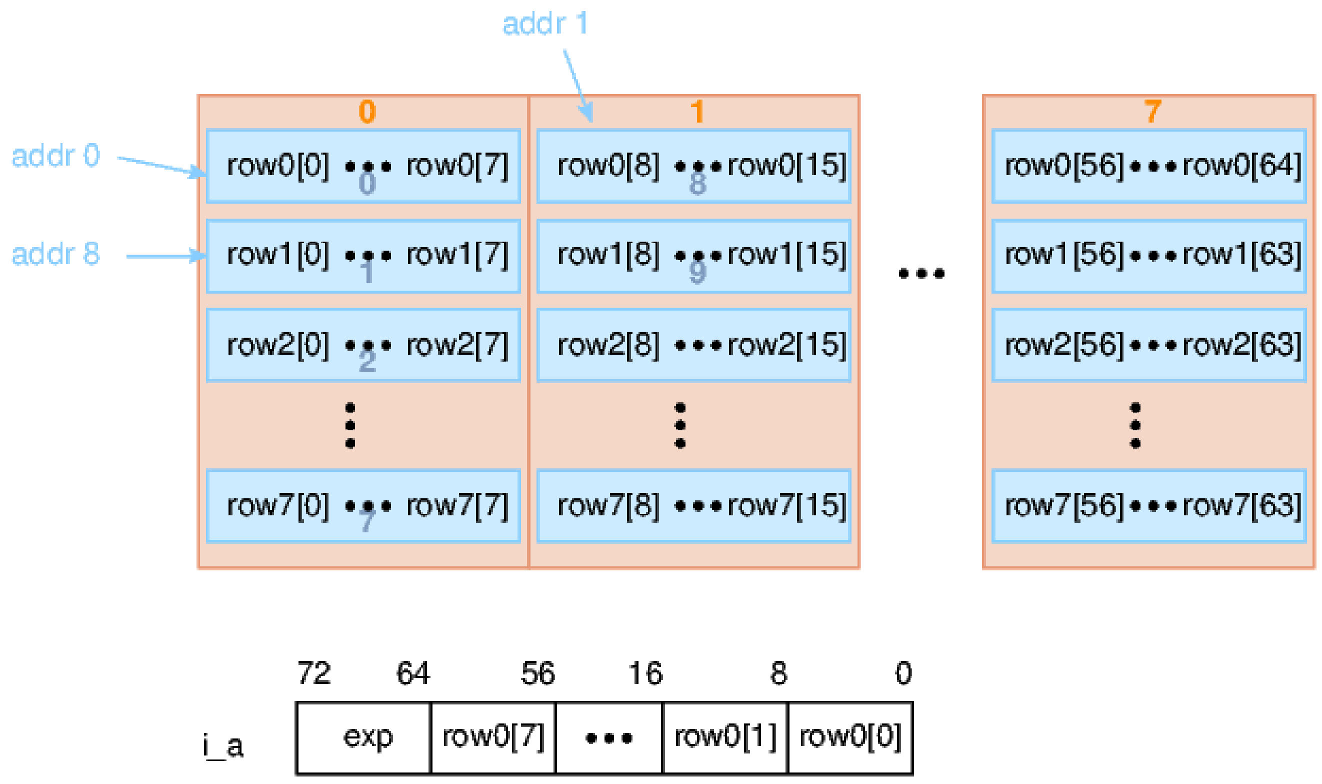

Figure 3 • Activation input order

The activation input is in MXINT8 format, consisting of blocks of 8 int8 values, with an exponent. When storing in BRAM, it is easiest to store in row major order, but the input should iterate over the rows before moving on to the next column. The input is one (blue) block per cycle. Hence, the first input is row0[7:0], the next is row1[7:0], etc., till the last row, followed by row0[15:8], row1[15:8], etc..

ⓘ The first version of the matrix multiplication engine only allows up to 8 rows. The next release will allow a larger number of rows.

Unlike the weight input, there is no separate input for the exponent. Instead, the exponent is appended to the block as value for input i_a, as shown above.

Chapter 4 : Activation Output

The result output follows the same order as the activation input, with blocks of 8 values per row, and then iterating over the rows before moving to the next 8 columns. However, the 8 values per row are output over two cycles of 4 values. Hence, the output starts with row0[3:0], row0[7:4], row1[3:0], ..., as illustrated in the following diagram.

A future release may have the option to output 8 values per clock cycle.

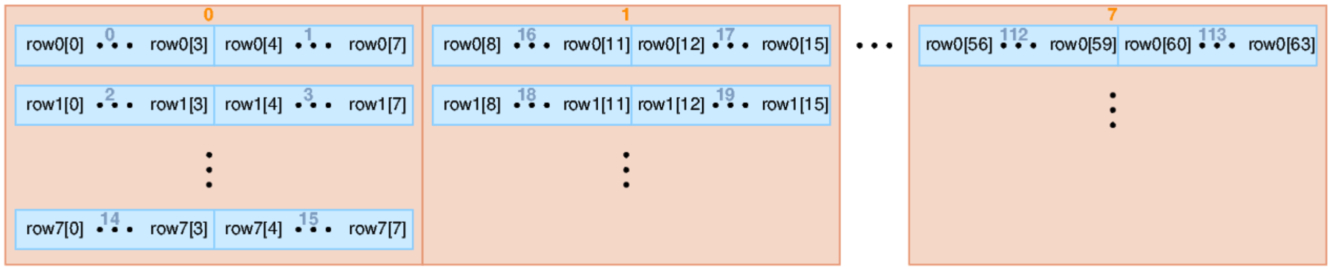
0
1
7
row0[0] 0 row0[3] row0[4] 1 row0[7]
row1[0] 2 row1[3] row1[4] 3 row1[7]
row7[0] 14 row7[3] row7[4] 15 row7[7]
row0[8] 16 row0[11] row0[12] 17 row0[15]
row1[8] 18 row1[11] row1[12] 19 row1[15]
row0[56] 112 row0[59] row0[60] 113 row0[63]

Figure 4 • Order of result output

Note that the result values are floating point values. Typically some non-linear operations are performed, and the resulting floating point values form the activation input for the next matrix multiplication. However, this means that those floating point values must be transformed, on the fly, into the MXINT8 format of input i_a. A method for performing that transformation is outlined in the document on Block-Floating Point. Achronix will provide a Verilog module that performs this transformation, but users can implement their own versions as well.