Block-Floating Point (MXINT) ()

Achronix
Data Acceleration

– June 02, 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 MLP72 supports both integer and floating point arithmetic. When implementing machine learning models, floating point arithmetic is a natural choice. However, a floating point implementation has two possible drawbacks: floating point numbers are somewhat large (the MLP supports 16- and 24-bit floating point formats), and floating point operators are more expensive than integer operators. The size of the numbers limits the size of models that can be stored, as well as the effective bandwidth (values per second) when transferring data between FPGA and off-chip memory. Because of the relative cost of the operators, the MLP72 supports only two parallel floating point multipliers. If instead we could use small integers, such as int8, not only would the size of numbers be halved (thus doubling effective storage capacity and bandwidth), but the compute power would increase by 8x, because the MLP72 has 16 parallel int8*int8 multipliers. (If fractions are needed, integers can be interpreted as fixed-point numbers; this does not change the implementation.) These are attractive advantages, but the downside is that an int8 number has a very small range compared with a floating point number. For that reason, the MLP72 supports a mixed form of operation, called block-floating point.

Chapter 2 : Block-Floating Point Basics

Floating Point Background

Conceptually, a floating point number has the form m * 2^e, where m is the mantissa and e is the exponent (we ignore, for now, the exact binary representation of such numbers). The exponent is a signed integer, and the mantissa is a signed fixed-point number. Suppose we have vectors A and B of such numbers,

A = { ma[0]*$2^{ea[0]}$, ma[1]*$2^{ea[1]}$, ..., ma[n-1]*$2^{ea[n-1]}$ } and

B = { mb[0]*$2^{eb[0]}$, mb[1]*$2^{eb[1]}$, ..., mb[n-1]*$2^{eb[n-1]}$ }.

The dot-product A*B is defined as the sum of products

A*B = (ma[0]*$2^{ea[0]}$) * (mb[0]*$2^{eb[0]}$) + (ma[1]*$2^{ea[1]}$) * (mb[1]*$2^{eb[1]}$) + ... + (ma[n-1]*$2^{ea[n-1]}$) * (mb[n-1]*$2^{eb[n-1]}$)

Block-Floating Point Definition

We define a block as a vector of floating point numbers that all have the same exponent, such as

A = { ma[0]*2^{ea}, ma[1]*2^{ea}, ..., ma[n-1]*2^{ea} }.

Such a vector can be stored efficiently as a vector of integers

mA = { ma[0], ma[1], ..., ma[n-1] } together with the single block exponent ea.

Depending on the number of elements in the block (the block size, n), this provides significant size savings compared with regular floating point vectors. Furthermore, per our earlier definition, the dot-product of two such vectors (each with their own block exponent) is

A*B = (ma[0]*2^{ea}) * (mb[0]*2^{eb}) + (ma[1]*2^{ea}) * (mb[1]*2^{eb}) + ... + (ma[n-1]*2^{ea}) * (ma[n-1]*2^{eb})

= (ma[0]*mb[0] + ma[1]*mb[1] + ... + ma[n-1]*mb[n-1]) * 2^{ea} * 2^{eb}

= mA * mB * 2^{ea+eb}

Here, the product mA * mB involves only integer operations, which means we can use the MLP72's multiply-accumulate (MAC) operation. As mentioned above, the integer MAC has 8x the parallel compute power of the floating point MAC (16 int8 multipliers compared with 2 floating point multipliers).

The block exponent can be seen as a scaling factor for the vector of integers. In the context of machine learning, the **Open Compute Project (OCP)**[1] refers to block-floating point as **"Microscaling" (MX) formats**[2]. The block-floating point formats implemented in the MLP72 match the OCP MXINT*n* formats, e.g., MXINT8 when int8 is used.

1 https://www.opencompute.org/

2 https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf

Chapter 3 : MLP72 Support

The MLP72 has hardened support for block-floating point (OCP MXINT*n* formats). In block-floating point mode, the MLP72 takes blocks as input, where each block consists of a vector of integers plus an exponent. The integer multiplier-adder tree is used to compute the mA * mB sum of products, and an integrated block-floating point unit combines that product with the exponents to produce an fp24 number. This includes normalization and standard floating point rounding (round-to-nearest with ties to even).

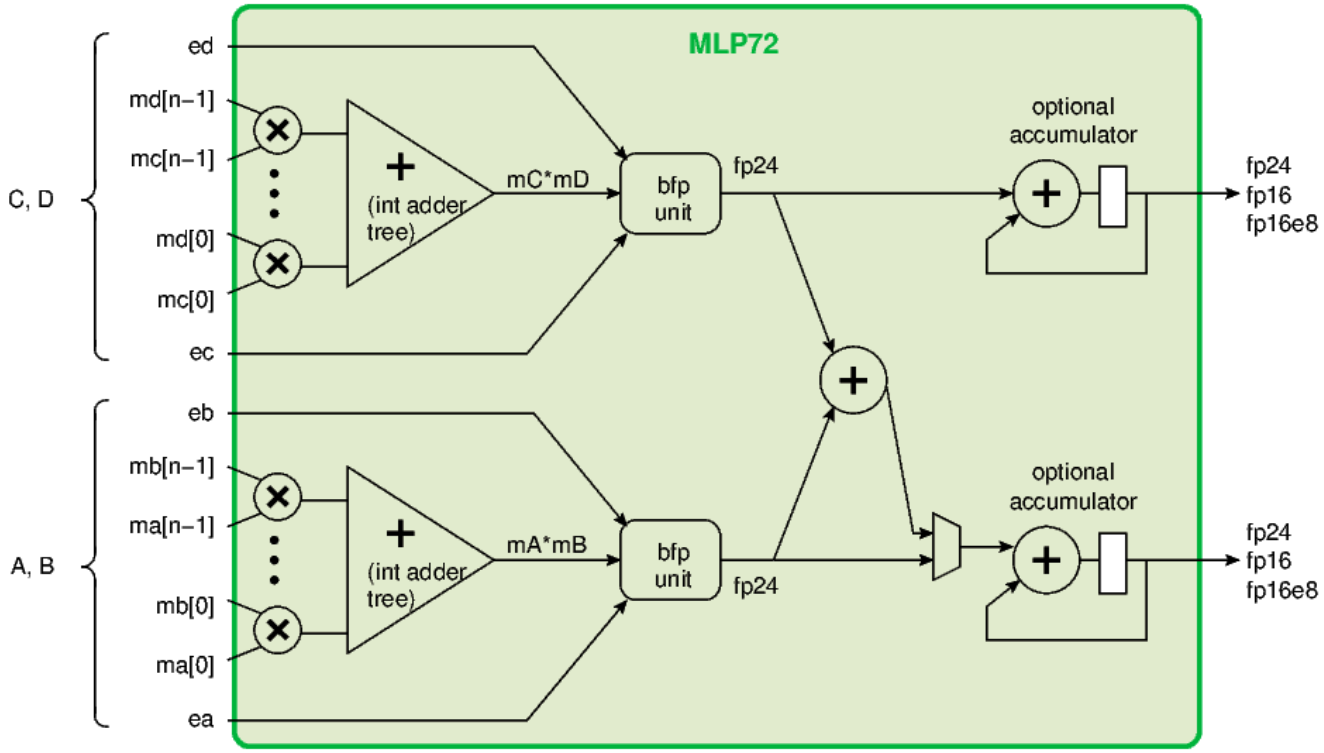

Figure 1 • Schematic of MLP72 in block-floating point mode

The diagram shows a schematic of the MLP72 block-floating point operation (for clarity, details that are not mode-specific, such as input cascade and LRAM, are not shown). As can be seen, the MLP72 has two parallel multiplier-adder trees, and hence can compute two dot-products, A*B and C*D, in parallel. Optionally, the central (floating point) adder can be used to add the two products together; use of the (floating point) accumulators is optional as well.

The number of multipliers depends on the selected integer size, as shown in this table.

int size	block size	multipliers per MLP72	int format	exponent bits
3, 4	16	32	smag or 2's comp	5 or 8
6, 7, 8	8	16	smag or 2's comp	5 or 8
16	2	4	2's comp	5 or 8

Table 1 • Supported integer sizes for block-floating point, with number of parallel multipliers

The block size is half the number of multipliers, because each of the two multiplier-adder trees takes a block as input (a smaller block size results in higher precision). The integers can be represented in the standard 2's complement format, or (except for int16) signed magnitude format. The user can select 5 or 8 bits for the exponent, regardless of integer size. Note that when computing the dot-product A*B, both blocks must use the same integer size (e.g., both int4).

As an example, consider using int8 for the integer values, with an 8-bit exponent. The block size is 8, hence a block of 8 integers and an exponent consists of 72 bits total. The MLP72 is designed to accommodate such blocks: the MLP72 has 72 direct fabric inputs (and can "borrow" another 72 inputs from the adjacent BRAM72K), there are four 72-bit wide input cascades, and the internal register file (LRAM) can be configured as 72 or 144 bits wide. Likewise, the BRAM72K can be configured as 72 or 144 bits wide, and the direct-connect between the BRAM72K and MLP72 is 144 bits wide as well.

Packing Fabric Inputs

The MLP72 has 72 fabric inputs that can be driven with arbitrary logic via the FPGA routing network (the routing "fabric"). In addition, the MLP72 has several special input methods that do not require routing: inputs can come from the adjacent tightly-coupled BRAM72K; via the input cascade from an adjacent MLP; or from the internal LRAM. Finally, an MLP72 has a wide mode where it "borrows" (or shares) an additional 72 fabric inputs from the adjacent BRAM72K (in practice, this means that BRAM72K cannot be used).

With int8, to use all 16 multipliers, 4 blocks (A, B, C, D) are needed as input, though in some configurations a block may be used twice (e.g., A and C might be equal). Since that exceeds the available fabric inputs, one or more of the special input methods must be used. Nevertheless, there are situations where at least one of the arguments in a dot-product A*B must come from the fabric. To maximize the number of multiplications in such a case, the MLP72 supports packing the smaller integer sizes (int3, 6, and 7) in a 72-bit block. In this mode, only one dot-product A*B is computed. Either A or B can be routed via the fabric, or both if wide mode is used. The table shows the number of parallel multipliers that can be used in this mode.

int size	block size	multipliers per MLP72
3	20	20
4	16	16
6	10	10
7	9	9
8	8	8

Table 2 • Number of parallel multipliers usable with fabric inputs

Number Format Details

The integer values are interpreted as fixed-point numbers with one integer bit. For instance, an 8-bit value in signed magnitude format has the form

sx.yyyyyy

where s is a sign bit (0 = +, 1 = -), x is an integer bit, and the remaining 6 bits are fraction bits. The bfp unit in the MLP72 takes the implied position of the binary point into account when generating the floating point value. This format matches that of the mantissa of regular floating point numbers, except that for normalized floating point numbers x is always 1 and therefore not stored (the so-called "hidden 1"). For block-floating point, there is no hidden 1, and all bits must be explicit. While signed magnitude format is convenient because it matches floating point formats, the number can also be input in standard 2's complement format. (For int_size=16, only 2's complement is supported.)

Likewise, exponents use the same format as in floating point numbers: exponent e is stored as unsigned value e+bias, where bias = 15 for 5-bit exponent fields, and bias = 127 for 8-bit exponent fields. As for floating point numbers, the exponent field has two special values: 0 means the entire block is 0; and an all-1 field (31 for 5-bit, 255 for 8-bit) means the entire block is ∞ (infinity). Unlike regular floating point numbers, in case of block floating point, the sign of 0 and infinity is ignored (always +).

Chapter 4 : Conversion from Floating Point

In block-floating point mode, the MLP72 takes blocks of integers with a shared exponent as input, and produces floating point values as output. Often this process must be repeated, meaning the floating point outputs must be converted to another block of integers with shared exponent. This conversion is not part of the MLP72, and must be implemented with soft logic instead. Here we outline a possible way to perform this conversion.

Converting a Single Value

As example, we first show the conversion of a single floating point number to block-floating point format.

1. Unless the exponent is 0, add a '1' in front of the mantissa field
2. Truncate or use rounding to fit the number in int_size-1 bits
3. Add the sign bit in front
4. Copy the exponent field

As example, consider this conversion of fp16 to int8 (signed magnitude) and 5-bit exponent: The fp16 binary representation of -5.79296875 is

sign	exponent	mantissa
1	1_0001	01_1100_1011

Conversion to int8: Step 1 produces 101_1100_1011. In step 2 we reduce this to 7 bits by removing the 4 lsb: 101_1100; if we choose to use rounding, then in this example that would force an increment, yielding 101_1101. We then add the sign bit in front, resulting in 1101_1101, and copy the exponent:

int (smag)	exponent
1101_1101	1_0001

In step 2, using rounding is potentially more accurate than truncation, but requires additional logic.

The same example with fp16e8, converting to int8 with 8-bit exponent: The fp16e8 binary representation of -5.78125 is

sign	exponent	mantissa
1	1000_0001	011_1001

Step 1 produces 1011_1001. In step 2 we remove the lsb: 101_1100. Then we add the sign bit and copy the exponent:

int (smag)	exponent
1101_1100	1000_0001

Note that even though fp16e8 has an 8-bit mantissa field, we still need to remove one mantissa bit to make space for the "hidden 1".

Converting a Block of Values

In reality we must convert a block of multiple floating point values to a block of integers with one exponent. Typically, not all floating point values will have the same exponent, which requires an additional step. We equalize the exponents by using the identity

$$m * 2^e = (m/2) * 2^{e+1} = (m>>1) * 2^{e+1}$$

Hence we can repeatedly right-shift the mantissa while incrementing the exponent (right-shifting is the reason that there can be no hidden 1).

1. Unless the exponent is 0, add a '1' in front of each of the mantissa fields
2. Find the number with the largest exponent, e_lg; this is the exponent we will use
3. For each number with a smaller exponent, shift the mantissa field (as modified by step 1) right while incrementing the exponent, until the exponent is e_lg or the mantissa is 0
4. Truncate or round each of the shifted mantissas
5. Add each of the sign bits
6. Use e_lg as the shared exponent

In step 3, numbers that have smaller exponents will lose some precision due to the shifting, and numbers that have sufficiently small exponents, relative to e_lg, will become 0. This is not unreasonable: in many cases the largest (absolute) values will dominate the dot-product, whereas the smallest numbers have almost no effect. Note that the above method is just one way of converting floating point numbers to blocks; it is possible that other methods work better for some data sets.