White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of Leeds.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/245052/

Version: Accepted Version

Proceedings Paper:
Islam, M. and Mikaitis, M. (Accepted: 2026) Simulation of Custom-Precision OCP MX Block Floating-Point Formats and Arithmetic. In: IEEE Conference on High Performance Extreme Computing (HPEC). 2026 IEEE High Performance Extreme Computing (HPEC), 14-18 Sep 2026, Virtual. IEEE. (In Press)

This is an author produced version of a conference paper accepted for publication in IEEE Conference on High Performance Extreme Computing (HPEC) made available via the University of Leeds Research Outputs Policy under the terms of the Creative Commons Attribution License (CC-BY), which permits unrestricted use, distribution and reproduction in any medium, provided the original work is properly cited.

Reuse
This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here: https://creativecommons.org/licenses/

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS
University of Sheffield
UNIVERSITY of York

Simulation of Custom-Precision OCP MX Block Floating-Point Formats and Arithmetic

Maliha Islam and Mantas Mikaitis
School of Computer Science, University of Leeds, Leeds, UK

***Abstract*—We present MXsim v0.1: a MATLAB library for simulating the OCP MX block floating-point arithmetic and formats. The library is built on top of a custom-precision floating-point simulator, CPFloat, which allows custom variants of non-block IEEE 754 floating-point formats to be simulated. MXsim is a MATLAB simulator, and is targeted at mixed-precision algorithm development community to perform experimentation in OCP MX block floating-point arithmetic by using a simple and well-documented interface in MATLAB, which is especially useful when access to the latest NVIDIA and AMD GPUs is not available. Furthermore, due to full customisation of exponents, precisions of input and output formats, and the precision of the accumulator within the dot product operation, MXsim allows experimenting with custom variants of the OCP MX block floating point for research. Finally, since the bit-wise numerical behavior of OCP MX formats is not documented by GPU vendors, and the precision and rounding are not prescribed by the OCP specification itself, MXsim allows experiments with various possible settings in order to determine the effects on mixed-precision numerical algorithms.**

I. Introduction

Computing hardware is increasingly incorporating low- and mixed-precision floating-point formats and arithmetic, motivated by the need to increase efficiency, whilst reducing memory usage and hardware complexity. In 2023, The Open Compute Project (OCP) released two floating-point format specifications, in which they standardised 4-8 bit representations, with a framework similar to block floating-point (BFP), named *microscaling (MX) formats* [1], [2]. These formats are now present in NVIDIA Blackwell [3] and AMD CDNA4-based GPUs [4]. Because access to these GPUs is scarce we have created a simulator MXsim[1] for the OCP MX formats which will be useful to those wanting to experiment with OCP MX easily and to those who want to experiment with MX-like block floating-point arithmetic in general. Our own motivation is, in subsequent work, to use it for research on algorithms for emulating high-precision computations with the MX formats.

BFP is an alternative to standard floating point which does not appear commonly in literature but has recently been appearing in hardware accelerators targeted at machine learning. It is now present in commercial hardware and has been standardised by OCP, as well as IEEE in the form of the upcoming P3109 standard [5]. BFP hardware could allow for the use of more-efficient low-precision arithmetic, whilst preserving accuracy, in scientific computing.

Although our library is inspired by the OCP MX specification [2] which contains five block floating-point formats, the exponents, precisions of input and output formats, and the precision of the accumulator within the dot product operator can be customised, allowing for the simulation of many more than five custom MX-like BFP formats.

[1]https://github.com/north-numerical-computing/MXsim

II. Previous work

MXsim has CPFloat [6] as a dependency, a MATLAB library for simulating custom floating-point formats. CPFloat provides a general-purpose IEEE 754 floating-point format simulation that supports customisable precisions, exponent widths, and rounding modes. Our software in effect extends CPFloat, providing MX block floating-point structures, with shared scale factors, as well as operations on these block formats.

A software called gfloat [7] provides generic floating-point encode and decode logic in Python, supporting OCP MX formats alongside IEEE 754 and P3109 formats. However, like CPFloat, it does not implement MX operations such as the dot product or matrix multiplication of two vectors of floating-point values represented in user-specified formats, scaled by block scales (see. Section III-C).

Microsoft's MX PyTorch emulation library [8] provides emulation of MX-compatible formats in Python. It requires a CUDA-capable GPU and a PyTorch installation, and supports a fixed set of OCP MX formats (see. Section III-C) without the ability to customise element precision, block size, or accumulation format. Hence, this emulation library is targeted at machine learning users and supports only currently-available OCP formats. In contrast, MXsim targets the scientific computing community, providing full customisation beyond the standard OCP formats and can run on any hardware that supports MATLAB.

Several hardware implementations of MX formats have been proposed. Samson et al. [9] present an open-source FPGA implementation of the full MX standard, supporting all concrete formats for conversion and the standard-defined operations. Gorodecky and Sousa [10] propose an FPGA-based hardware converter from IEEE 754 fp32 to MX formats, supporting six MX format variants—it includes one MX format that uses fixed-point arithmetic for elements of vectors. They implement the conversion algorithm specified in the OCP MX specification [2, Sec. 6.3]. Regarding exceptional values, they note that the shared scale X (see Section III-C) can represent NaN but not infinity. If X is a NaN, the vector represented by the MX vector with X a NaN contains elements

that are all NaNs. This is not specified by OCP, but is derived from Equation 3.

Abdurakhmanov and Fahmy [11] present a detailed exploration of systolic array architectures for MX format matrix multiplication on FPGAs, evaluating two accumulation strategies across multiple MX formats and pipeline depths. One of these accumulation strategies is to generate exact results, and the following formula is given to determine the bitwidth required for exact accumulation:

$$\text{accwidth} = 2 \times (2^{E_{\text{width}}} + M_{\text{width}}) + \lceil \log_2(k) \rceil, \quad (1)$$

where E_{width} is the number of exponent bits and M_{width} is the number of stored mantissa bits (i.e. the fractional part of the significand m defined in Section III-A, excluding the implicit leading bit), and k is the block size.

For the OCP 8-bit formats (see. Section III-B), the minimum bit widths of the accumulator are 73 and 43 for element formats E5M2 and E4M3, respectively. This formula is also used by Samson et al. [9].

The second accumulation strategy proposed by [11] is BFloat16 accumulation, which provides better performance than exact accumulation. BFloat16 is chosen over fp16 due to its exponent being 8 bits wide, allowing for a larger dynamic range than E5M2.

Islamoglu et al. [12] and Wipfli et al. [13] propose RISC-V ISA extensions for native MX dot product execution, achieving significant speedup over software emulation. This work is targeted at hardware implementations and does not contribute to the research in applied mathematics, where simulation of new formats in an easy-to-use manner is a very useful tool for algorithmic research and numerical experiments. MXsim provides a software tool for experimentation with MX arithmetic without requiring access to specialised hardware.

III. Background

A. IEEE 754 floating-point formats

A floating-point number in a binary format with precision p and exponent range $[e_{\min}, e_{\max}]$ is represented as

$$x = (-1)^s \cdot m \cdot 2^e \quad (2)$$

where $s \in \{0, 1\}$ is the sign bit, m is the significand satisfying $1 \leq m < 2$ for normal numbers ($0 \leq m < 1$ for subnormal numbers), and $e \in [e_{\min}, e_{\max}]$ is the exponent. The exponent range satisfies $e_{\min} = 1 - e_{\max}$ [14]. The significand m has p bits, giving $p - 1$ stored mantissa bits. The IEEE 754 standard [14] defines several binary formats, including fp32 (binary32, $p = 24$, $e_{\min} = -126$, $e_{\max} = 127$) and fp64 (binary64, $p = 53$, $e_{\min} = -1022$, $e_{\max} = 1023$).

B. Definition of OCP 8-bit formats

The OCP 8-bit floating-point specification [1] defines two formats: E4M3 and E5M2. Both follow the general floating-point representation of Section III-A, with a sign bit, exponent bits, and significand bits, including an implicit bit and subnormals, with one deviation for E4M3 described further.

E4M3 represents the exponent e in 4 bits and the fractional part of the significand m in 3 bits, with $p = 4$, $e_{\min} = -6$ and $e_{\max} = 8$. Therefore E4M3 violates the IEEE 754 format pattern of $e_{\min} = 1 - e_{\max}$.

E5M2 represents the exponent e in 5 bits and the fractional part of the significand m in 2 bits, giving a total precision $p = 3$, with $e_{\min} = -14$ and $e_{\max} = 15$, providing a larger dynamic range at the cost of reduced precision.

C. Definition of OCP MX formats

An MX-compliant format consists of a block of k private elements, where each element P_i is a floating-point number, and a shared scale factor X. All elements P_i must have the same floating-point format and X must be a power of 2. Each value V_i within the block is given by

$$V_i = X \cdot P_i. \quad (3)$$

When converting a block to MX, the shared scale X is computed by taking the largest power of two less than or equal to the largest absolute value within the block, divided by the largest power of two representable in the element format [2].

The specification defines five floating-point variants (MXFP8 E5M2/E4M3, MXFP6 E3M2/E2M3, MXFP4 E2M1) [2] and one integer variant (MXINT8), all with a fixed block size of $k = 32$. MXsim focuses on floating point element formats, but may be extended in the future to include MXINT8 if there is demand for it.

D. Differences of OCP formats compared with IEEE 754 formats

In IEEE 754, each number carries an exponent part of its bit pattern. In MX formats, each element similarly retains its own exponent, but additionally shares a common scale factor X across a block of k elements.

In IEEE 754 style formats, the maximum exponent bit pattern is reserved for representing infinities and multiple NaN values. In E4M3, this bit pattern is instead used for finite numbers, with only one significand pattern (all-ones) reserved for NaN. This change causes a violation of the IEEE property $e_{\min} = 1 - e_{\max}$, and gives one more finite exponent value than an equivalent IEEE style format would have. The maximum finite value is $2^8 \times (1 + 6/8) = 448$ [1]. If E4M3 had followed the IEEE style for that last binade, the maximum value would be $2^7 \times (1 + 7/8) = 240$; here we have $7/8$ instead of $6/8$ for the fraction since the all-ones significand would not be reserved for the NaN.

Similar to IEEE 754 [14, Sec. 9.4], the OCP specification leaves the internal precision and the order of operations of the dot product implementation-defined [2].

IV. Methods

A. Functional requirements

1) ***Format selection****:* Formats can be selected through a structure defined by the user. We will refer to this structure as `MX`. The `MX` struct configures the block floating-point format. The element type is set via `element_opts`, with block size

k. Accumulation and output formats are configured by structs accumulator_opts and output_opts respectively, and the shared scale is parameterised by scale_bits. These options allow for full precision customisation of the MX structure and its operations. If a field of MX is left undefined, it will default to its corresponding value denoting OCP MXE4M3 [2].

TABLE I
FIELDS OF THE MX CONFIGURATION STRUCT.

Field	Type	Default
MX.element_opts	struct	E4M3
MX.k	integer	32
MX.accumulator_opts	struct	fp32
MX.output_opts	struct	fp32
MX.scale_bits	integer	8

As shown in Table I, some of the fields of MX are structures, each of which can specify any floating-point format supported by CPFloat [6]. CPFloat provides a number of predefined formats (including E4M3 and E5M2), as well as support for custom formats—this is exploited in MXsim to provide customisation in the simulation of MX formats.

*2) **Accumulation**:* The accumulation precision for MX operations is unspecified by OCP. The default accumulation format is fp32, which is provided by CPFloat. CPFloat uses fp64 as its base arithmetic and rounds to the specified precision, which limits the maximum precision it can support to 26 bits due to double rounding effects [15]. Consequently, exact accumulation for the OCP 8-bit formats, which requires 43 and 73 bits for E4M3 and E5M2 respectively (see Equation 1), cannot be reliably achieved through CPFloat. Even native fp64 accumulation, which would in principle provide exact results for E4M3, cannot be enabled via CPFloat. Future work will explore providing an option for fixed-point accumulation, by using an alternative to CPFloat.

Alternatively, for non-exact accumulation, Abdurakhmanov and Fahmy [11] favour the use of BFloat16, due to its 8-bit exponent, offering a wider dynamic range than fp16 (IEEE binary16 format).

Recent work characterising the numerical behaviour of NVIDIA B200 Tensor Cores [16] found that the accumulator for 8-bit formats uses 25-bit fixed-point arithmetic with 2 integer bits. It is highly likely that MX hardware will share a similar accumulation strategy. In future work, we plan to set the default accumulator to match this precision, using a dedicated fixed-point model, as CPFloat does not currently support fixed-point arithmetic.

The accumulation format in MXsim can be configured by the user, via MX.accumulator_opts.

*3) **Conversion**:* Once MX has been defined, a vector V of any length can be converted to the chosen format via the function MXconvert(V,MX). The function will return a structure, with the field scale containing a set of shared scales and the field vector containing a set of vectors of length k. If the input vector is not divisible by k, it will first be padded with zeros, so that its length is a multiple of k, before being split into blocks.

The scale and vector components of each block are calculated by the internal block convert function BlockConvert, which is an implementation of Section 6.3 of the OCP MX specification [2]. The scale X is computed for each block as described in Section III-C. The input vector is then divided by X and converted to the format specified by element_opts, using CPFloat, to give the private elements P_i. If the element format is E4M3, the maximum value of 448 is forcibly set within the conversion algorithm, as CPFloat has a maximum value of 480 when e_{max} is set to 8.

The conversion to the element format uses round-to-nearest with ties-to-even by default (see. Section IV-A5), but this is configurable via element_opts.round. On overflow, values exceeding the maximum finite value of the element format are clamped to the maximum by default, as specified by OCP [2]. This behaviour can be configured via MX.element_opts.saturation. Further detail on this can be found in Section IV-B.

The function MXconvert_inv(M,n) converts a vector from MX format to standard floating point. This function is designed to be the inverse of MXconvert. The argument M must be an MX structure, containing a set of vectors and a set of scales. The argument n is optional, and is the length of the original vector. This argument reverses the effects of the zero padding which occurs during conversion, if the original vector length is not a multiple of k, and sets the length of the backwards converted vector to n. The function multiplies each block vector by its corresponding scale and concatenates the results into a single resulting vector. If n is not supplied, the resulting vector will be a multiple of k, which may not match the length of the original vector before the conversion to MX.

*4) **Operation support**:* There are two operations defined by OCP [2].

The dot product of two single blocks of length k is defined [2, Sec. 6.1] as

$$\mathrm{Dot}(A,B) = X^{(A)}X^{(B)}\sum_{i=1}^{k} P_i^{(A)}P_i^{(B)}, \tag{4}$$

where $X^{(A)}$ and $X^{(B)}$ are the shared scales of blocks A and B respectively (see. Section III-C), and $P_i^{(A)}$, $P_i^{(B)}$ are the i-th floating-point elements of each block.

Operation 4 is supported by the function BlockDot(A, B, MX), and computes the dot product of a single block. The input values A and B should be in MX format, and with vector components of length k. The accumulation precision is determined by MX.accumulator_opts. The sum is currently computed recursively, which may not reflect the accumulation behaviour of NVIDIA hardware. In future work we plan to utilise the accurate models of NVIDIA Tensor Cores presented in [16] to provide an option for hardware-accurate accumulation.

The second operation specified in OCP MX [2, Sec. 6.2] is the dot product of a vector of any length:

$$C = \text{DotGeneral}(A, B) = \sum_{j=1}^{n} \text{Dot}(A_j, B_j) \tag{5}$$

where A_j and B_j are the j-th blocks of vectors A and B respectively, each of length k.

Operation 5 is supported by the function `GeneralDot(A,B,MX)`. Inputs `A` and `B` are MATLAB vectors of any length. The sum is computed recursively, using the precision defined by `MX.output_opts`.

MXsim also contains a matrix multiplication operation, `BlockMatMul(A,B,MX)` which is not present in the OCP specification. The arguments `A` and `B` are matrices of sizes $m \times n$ and $n \times p$ respectively. The result is computed via a sum of inner products computed by `GeneralDot`, iterating over rows of A and columns of B, accumulated recursively.

*5) **Rounding mode**:* Round-to-nearest with ties-to-even is used by default at each of the three sources of rounding: when rounding inputs to the element format, when rounding during accumulation, and when rounding the accumulator to the output format. Each of these can be customised via the corresponding options in the `MX` structure, using the CPFloat rounding options documented in `help cpfloat`.

B. Treatment of exceptional values

*1) **Exponent limit**:* The shared scale factor X in OCP-defined MX formats is E8M0 and contains only exponent bits. E8M0 has one representation for NaN: the all-ones bit pattern. It does not have a representation for infinity, as all other bit patterns are used for finite values. E8M0 is set as the default format for X, but this is configurable via `MX.scale_bits`. This field must be set to an integer, as it determines the bitwidth of the scale. The scale will contain an exponent part, with no significand bits [2].

All scale formats will mirror E8M0, with a single NaN representation and no infinity representation. This ensures that the exception handling within blocks stays consistent, regardless of shared scale bitwidth.

If the computed exponent exceeds the maximum representable scale exponent, X is clamped to its maximum value $2^{E_{\max}}$, where $E_{\max} = 2^{s-1} - 1$ and s is the number of scale bits.

*2) **Infinities and saturation**:* As mentioned, the exponent part of a block can not be set to infinity. If a vector containing infinities is converted to MX, the shared exponent within its block will be set to its maximum value. Treatment of infinities within vector parts of a block can be configured by the user, using the `saturation` field of `MX.element_opts`. By default, `saturation` $= 1$, clamping all values exceeding the element maximum, including infinities, to the element maximum. This default was chosen as it is specified in the OCP MX conversion algorithm [2]. If the user sets `saturation` to 0, infinities may be present in the block vector, and will remain in their corresponding positions held before the conversion. If the element format is E4M3, infinities in the vector will be represented as NaN, in accordance with OCP [1].

If a vector contains only NaNs and infinities, the scale is computed from the infinite values, resulting in the maximum representable scale. NaN positions are preserved in the element vector. This ensures that Inf and NaN places from the original vector are preserved when using `MXconvert_inv`, whilst also ensuring a NaN outcome if `BlockDot` is used.

*3) **NaNs**:* The OCP specification mentions that if the shared scale is NaN, then a backwards conversion would result in an all-NaN vector [2], and MXsim preserves this behaviour. However, the handling of a vector containing NaNs when converting to MX is left as implementation defined by OCP.

When a vector containing a NaN is converted to MX via `MXconvert`, the shared scale X will be calculated based on the non-NaN values within the vector. The NaNs in the resulting block will preserve their corresponding positions in the vector part of the MX format. This is consistent with Samson et al. [9], which mentions setting NaNs at the element level if possible.

If the input vector contains only NaNs, it is not possible to compute a finite shared scale, as there are no finite values to compute it from. Hence, the scale is set to NaN, as well as all the elements.

If a dot product is performed on any MX-vector containing a NaN, the resulting value will be a NaN.

An alternative approach would be to set the shared scale to NaN, if any NaNs are present, resulting in a vector part of all NaNs. We chose not to use this method, as it does not preserve the other values within the vector when backwards converting, and sets them all to NaN. The chosen method preserves NaN positions as well as finite values (up to conversion error), while still producing NaNs after a dot product.

*4) **Zero vectors**:* If all elements of a block are zero, the shared scale is set to 1 and the private elements are set to zero. This avoids the undefined behaviour that would result from computing $\lfloor \log_2(0) \rfloor$ in the scale computation. This behaviour is left implementation defined by OCP [2].

*5) **Subnormals and negative zero**:* The OCP MX specification requires support for subnormal numbers in FP8 formats [1], with minimum subnormal magnitudes of 2^{-9} and 2^{-16} for E4M3 and E5M2 respectively. These are handled by CPFloat [6].

Negative zero is preserved through conversion, which is consistent with the OCP specification [1].

C. Structure of the MATLAB simulator

The repository is organized into `MXsim/` and `tests/` directories, with CPFloat [6] as an external dependency that must be installed separately and added to the MATLAB search path.

*1) **`simulator/`**:* The core simulator functions are located in `simulator/`. The function `MXconvert` converts a vector of any length to MX format, returning a structure containing block vectors and their corresponding scales. The internal function `BlockConvert` implements the OCP conversion

algorithm for a single block [2]. Operations are provided by `BlockDot`, `GeneralDot`, and `BlockMatMul`, which compute the block dot product, general dot product, and matrix multiplication, respectively. Inverse conversion back to standard floating-point is provided by `backwardsConvert`, which operates on a single block, and `MXconvert_inv`, which operates on a vector of any length. Default formats are set by `default_MX`. Fields of `MX` unspecified by the user are set to their default values via `merge_MX`.

*2) **tests/:*** The `tests/` directory contains integration tests in `tests/integration/` and error bound tests in `tests/expected_errors/`. All tests can be run via `runalltests.m`.

D. Testing of the correctness

Since the documentation of MX-compliant hardware is not precise, particularly regarding the ordering of additions and internal precision of the accumulator, there is no gold standard reference implementation against which MXsim can be directly validated. Instead, correctness is verified through two approaches.

*1) **Integration tests:*** Integration tests are located in `tests/integration/`, and cover the core functions `BlockConvert`, `BlockDot`, `MXconvert`, and `backwardsConvert`. These test edge cases and hand computed results against MXsim results.

a) BlockConvert: `BlockConvert` is the core function behind MX conversions, and explicitly follows the OCP Section 6.3 conversion algorithm [2]. Hence, it has been thoroughly tested. Tests include verification against a variety of hand computed results. The input vectors for the hand computed tests include the following.

- Multiples of 2 in E4M3, verifying correct scale and quantised values.
- Prime numbers in E4M3, verifying correct handling of non-power-of-two values.
- Mixed-sign values in E4M3, verifying correct sign handling.
- Decimal values in E5M2, verifying correct quantisation.
- Uniform in E5M2, where all elements are equal, verifying all elements produce the same output.
- Large dynamic range values in E5M2, verifying correct scale selection.

In addition, edge cases and exception handling (see. Section IV-B) have been tested. Each of the following criteria have been given a dedicated test:

- The scale is always an exact power of two.
- The output vector length matches the input length.
- Values exceeding the E4M3 maximum of 448 are clamped under saturation mode.
- A vector containing infinities produces a finite scale.
- NaN positions are preserved in the output vector, with the scale computed from non-NaN values.
- An all-NaN vector produces a NaN scale.
- A zero vector produces a scale of 1 and a zero output vector.

b) BlockDot: Hand-computed test cases verify the dot product of two MX blocks using the formula in Section 6.1 of the OCP MX specification [2]. Two test cases are included: one using E4M3 format and one using E5M2 format, each with known scales and element vectors. fp16 accumulation was used for handwritten test cases. Edge cases tested include NaN handling from both the element vector and the scale factor.

c) backwardsConvert: Tests verify correct reconstruction of values from a single MX block for a range of scale factors, including positive and negative powers of two, and vectors containing positive, negative, mixed-sign, and zero values. Edge cases include zero scale, single element blocks, and NaN handling from both the scale and the element vector.

*2) **Expected error tests:*** Error bound tests are located in `tests/expected_errors/`, and verify that the results of `BlockDot` and `GeneralDot` fall within the worst-case forward error bound for recursive summation [17, Eq. 4.4].

The vector length, vector values and unit roundoff are subbed into the theoretical bound at each iteration of this test, to find the maximum accumulation error. This is then plotted against errors from the randomly generated test cases, to show that the results fall within the expected range.

a) BlockDotErrorBoundTest: The error bound for `BlockDot` is derived from the standard model of floating-point arithmetic [17], in which each operation satisfies $\mathrm{fl}(a \operatorname{op} b) = (a \operatorname{op} b)(1+\delta)$ with $|\delta| \le u$, where $u = 2^{-24}$ is the unit roundoff of the fp32 accumulation format, and $\operatorname{op} \in +, -, \times, /$. `BlockDot` computes k products, and performs a recursive summation. Substituting these terms into the error bound for recursive summation shown by Higham [17, eq. (4.4)], the error E_k incurred by $\texttt{BlockDot}(A, B)$ satisfies:

$$|E_k| \le |X^{(A)}X^{(B)}|(k-1)u \sum_{i=1}^{k} |P_i^{(A)}P_i^{(B)}| + O(u^2) \quad (6)$$

The multiplications $P_i^{(A)}P_i^{(B)}$ can be performed exactly in fp32 for E4M3 and E5M2 elements, and the final scale multiplication $X^{(A)}X^{(B)}$ is exact since both are powers of two (assuming no overflow or underflow). Hence, no rounding error is introduced by these multiplications; the bound reflects only the accumulation error, scaled by the shared scale factors.

b) GeneralDotBounds: `GeneralDot` takes two input vectors of length n and splits them each into k sized blocks. It then calls `BlockDot` on every corresponding pair of blocks (see Equation 5). In these tests, input error introduced by the initial conversion to MX format is removed, so that the bound reflects only the accumulation error. Hence, the bound is based on the accumulation error. The accumulation precision used is also equivalent to the output precision.

Substituting into [17, eq. 4.4] and including the per-block errors from Equation 6, the error E incurred by

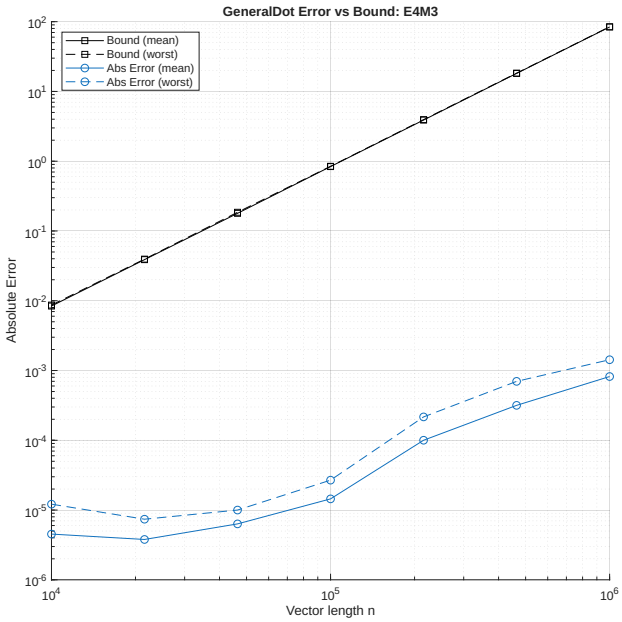


Fig. 1. Absolute error and theoretical bound for `GeneralDot` across vector lengths from 10^4 to 10^6, with entries independently generated from a uniform distribution on $(-1, 1)$, for E4M3 element formats. Mean and worst-case values over 5 trials are shown.

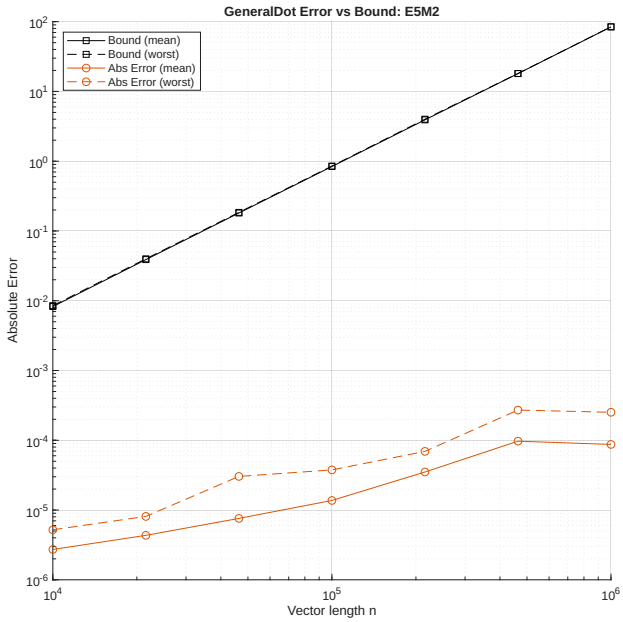


Fig. 2. Absolute error and theoretical bound for `GeneralDot` across vector lengths from 10^4 to 10^6, with entries independently generated from a uniform distribution on $(-1, 1)$, for E5M2 element formats. Mean and worst-case values over 5 trials are shown.

`GeneralDot`(A, B, MX) satisfies:

$$|E| \leq (\lceil n/k \rceil - 1)u \left(\sum_{j=1}^{\lceil n/k \rceil} |A_j B_j| + (k-1)u \sum_{j=1}^{\lceil n/k \rceil} |X^{(A_j)} X^{(B_j)}| \sum_{i=1}^{k} |P_i^{(A_j)} P_i^{(B_j)}| \right) + O(u^2) \tag{7}$$

where A_j and B_j are the j-th blocks of A and B respectively after MX conversion.

Figures 1 and 2 show the results of plotting this bound against `GeneralDot` errors. All errors fall below the bound.

V. Results

The following example shows the configuration for simulating MXFP8 (E5M2) formats, with an output in fp16 and accumulation in fp32.

```
MX.element_opts.format = 'E5M2';
MX.k = 32;
MX.accumulator_opts.format = 's';
MX.output_opts.format = 'h';
```

The following example demonstrates `GeneralDot` usage, computing the dot product of two fp64 (binary64 IEEE 754 format) vectors of size 10^4, using the MX format configuration described above.

```
>> rng(500);
% Choose two random vectors
A = (rand(10^4,1) - 0.5) * 2;
B = (rand(10^4,1) - 0.5) * 2;
>> GeneralDot(A,B,MX)
ans = 4.4375
```

The following example demonstrates `BlockMatMul` usage with the same configuration.

```
>> rng(500);
>> C = (rand(4,4) - 0.5) * 2;
>> D = (rand(4,4) - 0.5) * 2;
>> BlockMatMul(C,D,MX)
ans =
    0.1094   -0.5820   -0.7812   -0.9897
   -0.0352   -0.3828   -0.5586    0.4756
    0.6416    0.4724    0.1628    0.2886
   -0.0371    0.0449   -0.0293   -0.3716
```

VI. Conclusion

We have presented MXsim v0.1, a MATLAB simulator for OCP MX block floating-point arithmetic, supporting the standard OCP operations with full customisation of element format, accumulator format, output format, block size, rounding and scale bits. In the future work we plan to extend MXsim to support the IEEE P3109 standard [5], incorporate accurate models of NVIDIA Tensor Core accumulation [16], and add fixed-point accumulation support, for error-free accumulation. The software MXSim v0.1 is available at https://github.com/north-numerical-computing/MXsim.

Acknowledgment

Claude Opus 4.1 was used to produce an initial draft of Sections II, IV and IV-D1 , which were subsequently reviewed, verified, and significantly revised by the authors.

References

[1] P. Micikevicius, S. Oberman, P. Dubey, M. Cornea, A. Rodriguez, I. Bratt, R. Grisenthwaite, N. Jouppi, C. Chou, A. Huffman, M. Schulte, R. Wittig, D. Jani, and S. Deng, "OCP 8-bit floating point specification (OFP8)," p. 16, Jun. 2023, version 1.0. [Online]. Available: https://www.opencompute.org/documents/ocp-8-bit-floating-point-specification-ofp8-revision-1-0-2023-12-01-pdf-1

[2] B. D. Rouhani, N. Garegrat, T. Savell, A. More, K.-N. Han, R. Zhao, M. Hall, J. Klar, E. Chung, Y. Yu, M. Schulte, R. Wittig, I. Bratt, N. Stephens, J. Milanovic, J. Brothers, P. Dubey, M. Cornea, A. Heinecke, A. Rodriguez, M. Langhammer, S. Deng, M. Naumov, P. Micikevicius, M. Siu, and C. Verrilli, "OCP microscaling formats (MX) specification," p. 16, Sep. 2023, version 1.0. [Online]. Available: https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf

[3] NVIDIA, "NVIDIA Blackwell architecture technical brief," 2025. [Online]. Available: https://resources.nvidia.com/en-us-blackwell-architecture

[4] Advanced Micro Devices, "CDNA4 Instruction Set Architecture," Aug. 2025. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/amd-instinct-cdna4-instruction-set-architecture.pdf

[5] IEEE SA Working Group P3109, "Arithmetic formats for machine learning: Public materials," https://github.com/P3109/Public, 2024, unapproved interim drafts of IEEE SA WG P3109. Accessed: 2026-04-20.

[6] M. Fasi and M. Mikaitis, "Cpfloat: A c library for simulating low-precision arithmetic," *ACM Transactions on Mathematical Software*, vol. 49, no. 2, pp. 1–32, 2023.

[7] G. Research, "gfloat: Generic floating-point types in Python," 2023. [Online]. Available: https://github.com/graphcore-research/gfloat

[8] Microsoft, "MX PyTorch emulation library," 2023. [Online]. Available: https://github.com/microsoft/microxcaling

[9] E. Samson, N. Mellempudi, W. Luk, and G. A. Constantinides, "Exploring FPGA designs for MX and beyond," in *2024 IEEE International Conference on Field-Programmable Technology (FPT)*, 2024, pp. 304–311.

[10] D. Gorodecky and L. Sousa, "Hardware for converting floating-point to the microscaling (MX) format," 2024. [Online]. Available: https://arxiv.org/abs/2411.03149

[11] A. Abdurakhmanov and S. A. Fahmy, "Exploring microscaling MX minifloat systolic arrays on FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 18, no. 4, 2025.

[12] G. Islamoglu, L. Bertaccini, A. S. Prasad, F. Conti, A. Garofalo, and L. Benini, "MXDOTP: A RISC-V ISA extension for enabling microscaling (MX) floating-point dot products," in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2025.

[13] M. Wipfli, G. Islamoglu, N. Kunhi Purayil, A. Garofalo, and L. Benini, "VMXDOTP: A RISC-V vector ISA extension for efficient microscaling (MX) format acceleration," in *Design, Automation and Test in Europe (DATE)*, 2026.

[14] *IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2019 (revision of IEEE Std 754-2008)*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers, Jul. 2019.

[15] S. M. Rump, "IEEE-754 precision-p base-β arithmetic implemented in binary," *ACM Transactions on Mathematical Software*, 2023.

[16] F. A. Khattak and M. Mikaitis, "Accurate models of NVIDIA tensor cores," *arXiv preprint arXiv:2512.07004*, 2025. [Online]. Available: https://arxiv.org/abs/2512.07004

[17] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM, 2002.