Accurate Matrix Multiplication with Low- and Mixed-Precision Matrix Multiply-Add Units

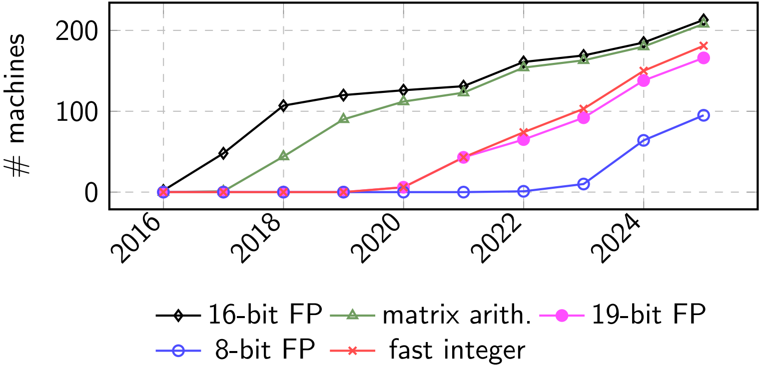
Determining Non-Standard Floating-Point Features and Developing Efficient Algorithms

Mantas Mikaitis

School of Computer Science, University of Leeds, Leeds, UK

Intel VSSAD seminar (virtual)
21 August, 2025

Non-standard floating point on the TOP500 (June 2025)



Devices counted: P100, V100, A100, H100, MI210, MI250X, MI300X, Intel Data Center GPU, from `https://www.top500.org`.

NVIDIA Blackwell throughputs (FLOPS)

fp8 (9×10^{15}) fp16 (4.5×10^{15}) fp64 (0.04×10^{15}).

The many floating-point formats

Format	precision	min pos.	max pos.	ulp(1)/2
binary64 (double)	53	2^{-1022}	$\sim 1.798 \times 10^{308}$	2^{-53}
binary32 (single)	24	2^{-126}	$\sim 3.403 \times 10^{38}$	2^{-24}
tf32 (19-bit)	11	2^{-126}	$\sim 3.401 \times 10^{38}$	2^{-11}
bfloat16	8	2^{-126}	$\sim 3.389 \times 10^{38}$	2^{-8}
binary16 (half)	11	2^{-14}	65504	2^{-11}
fp8-E4M3	4	2^{-6}	448	2^{-4}
fp8-E5M2	3	2^{-14}	57344	2^{-3}
fp6-E2M3	4	2^{0}	7.5	2^{-4}
fp6-E3M2	3	2^{-2}	28	2^{-3}
fp4-E2M1	2	2^{0}	6	2^{-2}

New standards in development

This may change quite significantly soon with the IEEE P3109 standard for fl. point for machine learning almost complete.

Mixed-precision matrix multipliers

Many low-precision formats are available as input formats to matrix multiply-accumulate operation.

$$\begin{array}{ccccccc} D & = & C & + & A & \times & B, \\ \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\substack{\text{binary16 or}\\ \text{binary32}}} & = & \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\substack{\text{binary16 or}\\ \text{binary32}}} & + & \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{8/16/19-bit FP}} & \times & \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{8/16/19-bit FP}} \end{array}$$

Hardware matrix multipliers in mixed precision

- Example above is 4×4, but dimensions differ across architectures.
- Reduction ops not standardised by IEEE 754: internal dot product precision, rounding, subnormal support, sum order, carry bits, monotonicity (see Clause 9.4 in 754-2019).

Part 1: Testing Features of Undocumented Matrix Multipliers

1982: Paranoia software (Kahan in 1982, and then others ported to Python, C, Fortran)

`https://www.arithmazium.org` lists the following questions Paranoia tackles:

- Is the arithmetic **binary**, octal, decimal, hexadecimal, or even logarithmic?
- **How many significant digits in the radix 2**, 8, 10, or 16 are carried?
- **Are excess digits in a result truncated, rounded off, or something else**?
- What is the **largest finite number**? The **smallest nonzero number**? Do the extreme values have any unusual behaviors in arithmetic?
- How accurate are $\sqrt{x}$ and y^x?
- **Does the arithmetic behave according to the then-emerging IEEE floating point standard?**

1982: Paranoia software, example test

```
def find_precision_big_B_to_nth(b):
    """Compute the number of B-digits in the arithmetic and
       the power of B sufficient to have the ones place fall
       off the right.
    Args:
        b: the global radix B, accepted as an argument
    Returns:
        precision: number of B digits in arithmetic
        power of B such that the low-order digit is the B's place
    """
    big_b = ONE
    precision = ZERO
    while True:
        precision = precision + ONE
        big_b = big_b * b
        y = big_b + ONE
        if y - big_b != ONE: break
    return precision, big_b
```

GPU Paranoia [Hillesland and Lastra, 2004]

- R300: 16-bit significand
- NV30: 23-bit significand (perhaps 24)
- Found guard bits in all operations
- Found no correct rounding, and no "chopping"

ULP errors:

Operation	R300/arbfp	NV30/fp30
Addition	[-1.000, 0.000]	[-1.000, 0.000]
Subtraction	[-1.000, 1.000]	[-0.750, 0.750]
Multiplication	[-0.989, 0.125]	[-0.782, 0.625]
Division	[-2.869, 0.094]	[-1.199, 1.375]

Used a special set of significands of [Shryer, 1981]:

- 1.100..., 1.010..., 1.001, ...
- 1.000..., 1.100..., 1.110, ...
- 0

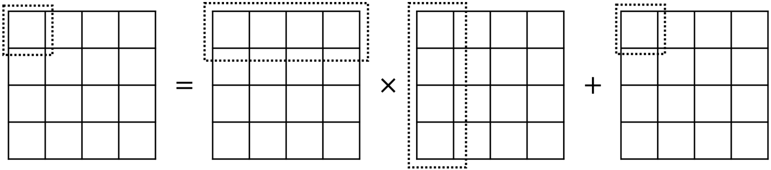
FPGA Paranoia [Tan, Boland, Constantinides, 2012]

Test Name	**Altera v11.0**	**Flopoco v2.2.1**	**Xilinx v6.0**
Basic Arithmetic			
Basic Arithmetic	☑	☑	☑
Division by Zero	☑	☒	☑
Add/Sub Rounding	☑	☑	☑
Multiplication Rounding	☑	☑	☑
Division Rounding	☒	☑	☑
Guard Digits	☑	☑	☑
Sticky Bit	☒	☑	☑
Sqrt Rounding	☑	☑	☑
Exponentiation			
x^y where $x, y \in \mathbb{Z}$	☑	☑	☑
$\lim_{x \to 1} x^{\frac{x+1}{x-1}} = e^2$	☒	☑	NI
Underflow and Overflow			
Thresholds	☑	☑	☑
PseudoZero	☑	☒	☑
X!=Z but X-Z=0	☑	☒	☑
Gradual Underflow	☒	☒	☒

☑	Passed this test
☒	Did not pass this test
NI	Not implemented in hardware

Flopoco $0 \div 0 = \infty$ (IEEE asks for NaN); Altera division not RN in $(1.5 - 2^{-23}) \div (1 - 2^{-23})$.

Now turn to matrix multiplier testing



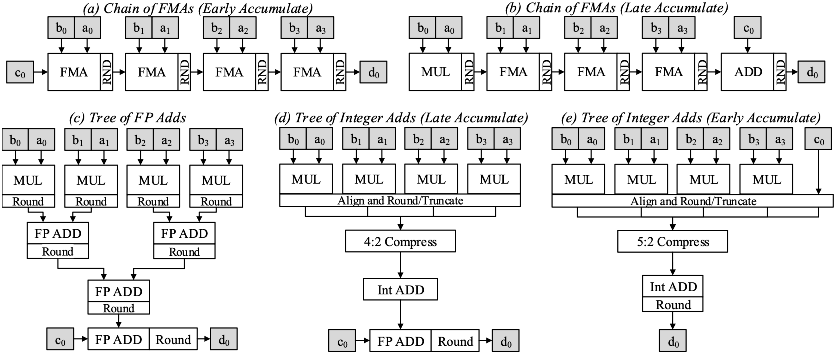
To simplify, we look at any of the 16 inner products:

$$d = a_1 b_1 \times a_2 b_2 \times a_3 b_3 \times a_4 b_4 + c$$

then either assume all 16 behave identically, or repeat tests.

Here a and b are vectors in one of low-prec. formats (4/6/16/19 bits), and c, d are in high-prec. output format (32/64 bits).

NVIDIA V100 tests [Hickmann and Bradford, 2019]



NVIDIA V100 tests [Hickmann and Bradford, 2019]

$$d = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4 + c$$

1. Take $a_1 = b_1 = a_2 = 2^{15}$, $b_2 = -2^{15}$, $a_3 = 2^{-14}$, $b_3 = 1$.
2. After products, this results in summation $2^{30} - 2^{30} + 2^{-14}$.
3. Then run all permutations of inputs.
4. Internal rounding points should produce 0.0 and 2^{-14} across all permutations:
    1. $\mathrm{fl}(\mathrm{fl}(2^{30} - 2^{30}) + 2^{-14}) = 2^{-14}$
    2. $\mathrm{fl}(2^{30} + \mathrm{fl}(-2^{30} + 2^{-14})) = 0$
5. Authors found 0 returned in all permut. (designs a, b, c ruled out)

Why 2^{30} and 2^{-14}?

Need big and small numbers to cause rounding. Also, 2^{16} is not representable in binary16, so $2^{15} \times 2^{15}$ is largest power-of-two product. Additionally, 2^{-14} is the smallest normalised binary16.

NVIDIA V100 tests [Hickmann and Bradford, 2019]

What is the internal accumulator's precision?

$$d = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4 + c$$

1. Take $a_1 = b_1 = a_2 = 2^{15}$, $b_2 = -2^{15}$, $a_3 = 2^X$, $b_3 = 2^Y$.
2. Vary X and Y such that $X + Y = -28...30$ in that order.
3. When $2^{30} - 2^{30} + 2^{X+Y} \neq 0$, internal precision of accum. $30 - (X + Y) + 1$.
4. Authors found internal precision 24.

[Fasi et al. 2021] subsequently applied similar testing to NVIDIA A100, finding 25-bit precision.

NVIDIA V100 tests [Hickmann and Bradford, 2019]

Are intermediate additions in the accumulator's precision rounded?

$$d = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4 + c$$

1. Take $a_1 = b_1 = 1$, $a_2 = 2^{-10}$, $b_2 = 2^{-13}$, $a_3 = 2^{-10}$, $b_3 = 2^{-14}$.
2. Results in summation $1 + 2^{-23} + 2^{-24}$.
3. Negative version: $-1 - 2^{-23} - 2^{-24}$.
4. 2^{-24} "falls off" the 24-bit precision.
5. Tests returned $\pm(1 + 2^{-23})$, meaning that 2^{-24} was not used for rounding up.

Testing NVIDIA Turing and Ampere [Fasi et al. 2021]

$$d = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4 + c$$

Testing the number of carry bits is at least two in the internal accumulator.

Take $c = 1.00000000000000000000011$,

and set the rest of inputs to produce the addends

1.00000000000000000000000
1.00000000000000000000000
1.00000000000000000000000
0.00000000000000000000001

and then permute the placement of the smallest input—we need it to be added last.

Testing NVIDIA Turing and Ampere [Fasi et al. 2021]

001.00000000000000000000011+ (24 bits, starting point for binary32)
001.00000000000000000000000 =
010.00000000000000000000011+ (25 bits)
001.00000000000000000000000 =
011.00000000000000000000011+
001.00000000000000000000000 =
100.00000000000000000000011+ (26 bits)
000.00000000000000000000001 =
100.00000000000000000000100
001.00000000000000000000001 (final normalise)

If the carry bits were not present, the bottom two 1's would have disappeared in the intermediate calculations since we would shift right to avoid overflows.

Testing NVIDIA Turing and Ampere [Fasi et al. 2021]

As part of this work we discovered a monotonicity test, a property that breaks when computing with denormalised values.

On the V100, set inputs in

$$d = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4 + c$$

such that the computation is

$$d = 2^{-24} + 2^{-24} + 2^{-24} + 2^{-24} + c$$

then with

- $c = 1$ we get $d = 1$,
- $c = 1 - 2^{-24}$ we get $d = 1 + 2^{-23}$.

The ordering of inputs is unchanged. See [Mikaitis, 2024] for detail.

AMD matrix engine testing [Li et al. 2024]

Inputs	GPU	Subnormal inputs handled?	Subnormal outputs handled?	Extra bit present? How many?	Rounding mode exhibited	FMA unit width	Order within one FMA unit is controllable?	Rounding mode for: 1. outputting FP16/BF16 (only for FP16/BF16 inputs) 2. product (only for FP32/FP64 inputs)
FP16	V100	✓	✓	0	truncate	4	✗	RTN-TE
	A100	✓	✓	1	truncate	8	✗	RTN-TE
	H100	✓	✓	≥ 2	truncate	≥ 16	✗	RTN-TE
	MI100	✓	✓	3	RTN-TE*	4	✗	RTN-TE
	MI250X	✗	✗	3	RTN-TE	1	N.A.	RTN-TE
BF16	A100	✓	✓	1	truncate	8	✗	N.A.**
	H100	✓	✓	≥ 2	truncate	≥ 16	✗	RTN-TE
	MI100	✓	✓	3	RTN-TE	2	✗	RTN-TE
	MI250X	✗	✗	3	RTN-TE	1	N.A.	RTN-TE
TF32(NVIDIA) FP32(AMD)	A100	✓	✓	1	RTN-TE	4	✗	N.A.
	H100	✓	✓	≥ 2	truncate	4	✗	N.A.
	MI100	✓	✓	3	RTN-TE	1	N.A.	RTN-TE
	MI250X	✓	✓	3	RTN-TE	1	N.A.	RTN-TE
FP64	A100	✓	✓	3	RTN-TE	1	✗	RTN-TE
	H100	✓	✓	3	RTN-TE	1	✗	RTN-TE
	MI250X	✓	✓	3	RTN-TE	1	N.A.	RTN-TE

New 3-year project on this topic

Project goals (we started about 4 months ago):

- Generalise test expressions through input and output format precisions (**see talk by Faizan Khattak at IEEE HPEC 2025**).
- Check our tests on a selection of simulated models of dot products.
- Design a testsuite that seamlessly works on a variety of architectures and programming languages: Intel, NVIDIA, AMD, ...
- Release a website library of hardware test results.

New 3-year project on this topic

High-level algorithm

1. Develop feature test expressions generalised by p_{in}, p_{out}.
2. Develop N models of inner product, with a variety of features.
3. Deploy tests on the N models, for common input-output format combinations.
4. Deploy tests on NVIDIA, AMD, Intel hardware, to determine which model they follow.
5. If HW behaviour not consistent with any model, add more models and repeat.

The model of dot product unit

We are currently determining the model for:

$$d = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4 + \cdots + a_n b_n + c$$

Rounding points and modes, input/output/internal precision, normalisation points? Need to cover all reasonable hardware implementations.

Part 2: Algorithms for Simulating High-Accuracy Matrix Multiplication via Low Precision

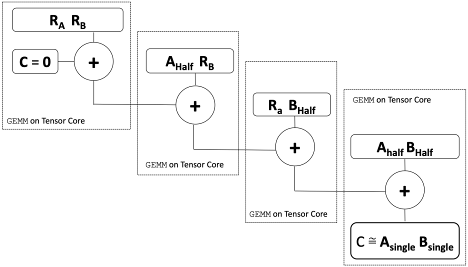
V100 multi-word matrix multiply [Markidis et al. 2018]

Take $R_A = A_{fp32} - A_{fp16}$ and $R_B = B_{fp32} - B_{fp16}$

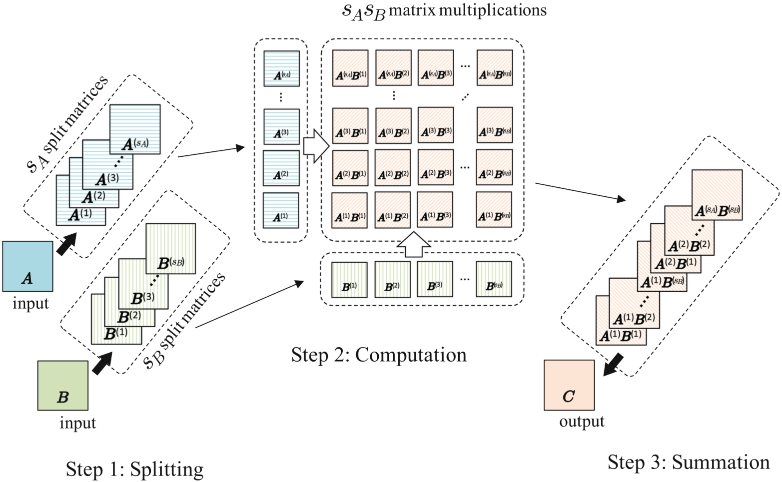
and then

$$A_{fp32}B_{fp32} \approx (A_{fp16} + R_A)(B_{fp16} + R_B) =$$
$$A_{fp16}B_{fp16} + A_{fp16}R_B + R_A B_{fp16} + R_A R_B$$

Use four invocations of fp16-fp32 tensor core.



V100 multi-word matrix multiply [Mukunoki et al. 2020]



Multi-word matrix multiply with 8-bit tensor cores [Mary and Mikaitis, 2025]

$$\begin{array}{ccccccc} D & = & C & + & A & \times & B, \\ \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\substack{\text{binary16 or} \\ \text{binary32}}} & = & \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\substack{\text{binary16 or} \\ \text{binary32}}} & + & \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{8-bit FP}} & \times & \underbrace{\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix}}_{\text{8-bit FP}} \end{array}$$

Multi-word matrix multiply with 8-bit tensor cores [Mary and Mikaitis, 2025]

Goal: Given A and B, matrices in, for example, binary64, multiply them accurately using mixed-precision MMAs.

1. Scale input matrices A and B.
2. Round input matrices to the *input format*.
3. Multiply scaled and rounded A and B in the *accumulation format*.
4. Scale the output matrix.

$$C = \Lambda^{-1}\Big(\mathrm{fl}(\Lambda A)\mathrm{fl}(BM)\Big)M^{-1}$$

- Λ and M are nonsingular diagonal matrices with diagonal coefficients λ_i and μ_i respectively.
- Scale coefficients λ_i and μ_i are powers of two.

Multi-word matrix multiply with 8-bit tensor cores [Mary and Mikaitis, 2025]

Let θ be the maximum value we can afford in the scaled A and B.

Scaling by powers of two means the maximum entry per row of A or column of B is in $(\theta/2, \theta]$.

We should maximise θ to reduce number of underflows, but at the same time remove possibility of overflow.

Choose:

$$\theta = \min(f_{\max}, \sqrt{F_{\max}/n}).$$

which avoids overflow in the input and in the accumulation of n products.

Single-word algorithm: an example

- Take $A \in \mathbb{R}^{4\times 4}$ and $B \in \mathbb{R}^{4\times 4}$.
- Set fp8-E4M3 as the *input format* with $f_{\max} = 448$.
- Set binary16 as the *accumulation format* with $F_{\max} = 65504$.
- No subnormal floating-point numbers.
- This gives $\min(448, \sqrt{65504/4}) = \min(448, 127.9687) \approx 127 = \theta$.

Scaling factors

In this case before rounding matrices to the *input format* we need to scale them such that 127 is the maximum value that appears.

- 127 is lower than $f_{\max} = 448$ - no *input format* overflows.
- $127 \times 127 = 16129$ and if we accumulate four such products we get $64616 < F_{\max} = 65504$. No *accumulation format* overflows.

Single-word algorithm: an example

Take

$$A = \begin{bmatrix} 500 & 1 & 1 & 2^{-6} \\ 128 & 128 & 128 & 128 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}, B = \begin{bmatrix} 1 & 128 & 1 & 1 \\ 1 & 128 & 1 & 1 \\ 1 & 128 & 1 & 1 \\ 1 & 128 & 1 & 1 \end{bmatrix}.$$

We have

$$AB = \begin{bmatrix} 502.015625 & 64258 & 502.015625 & 502.015625 \\ 512 & 65536 & 512 & 512 \\ 4 & 512 & 4 & 4 \\ 4 & 512 & 4 & 4 \end{bmatrix}.$$

Overflows in the above example if no scaling is applied

(Input) $500 > f_{\max} = 448$ and (output) $65536 > F_{\max} = 65504$.

Single-word algorithm: an example

$$C = \Lambda^{-1}\Big(\mathrm{fl}(\Lambda A)\mathrm{fl}(BM)\Big)M^{-1}, \quad \theta = 127$$

Step 1: Scale A and B.

$$\Lambda A = \begin{bmatrix} 2^{-2} & 0 & 0 & 0 \\ 0 & 2^{-1} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 500 & 1 & 1 & 2^{-6} \\ 128 & 128 & 128 & 128 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 125 & 2^{-2} & 2^{-2} & 2^{-8} \\ 64 & 64 & 64 & 64 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$BM = \begin{bmatrix} 1 & 128 & 1 & 1 \\ 1 & 128 & 1 & 1 \\ 1 & 128 & 1 & 1 \\ 1 & 128 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2^{-1} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \end{bmatrix}$$

How the scale coefficients are calculated

For example, take the first row of A. The largest value is 500 and we need to get it below $\theta = 127$. $\lambda_1 = 2^{\lfloor \log_2(127/500) \rfloor} = 2^{-2}$.

Single-word algorithm: an example

$$C = \Lambda^{-1}\Big(\mathrm{fl}(\Lambda A)\mathrm{fl}(BM)\Big)M^{-1}$$

Step 2: Round to the *input format* fp8-E4M3 ($f_{\min} = 2^{-6}$).

$$\mathrm{fl}(\Lambda A) = \mathrm{fl}\left(\begin{bmatrix} 125 & 2^{-2} & 2^{-2} & 2^{-8} \\ 64 & 64 & 64 & 64 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}\right) = \begin{bmatrix} 125 & 2^{-2} & 2^{-2} & \mathbf{0} \\ 64 & 64 & 64 & 64 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\mathrm{fl}(BM) = \mathrm{fl}\left(\begin{bmatrix} 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \end{bmatrix}\right) = \begin{bmatrix} 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \end{bmatrix}$$

Underflow in the above example

Notice that since subnormals are off, numbers $\le f_{\min}/2$ will round to zero, causing underflow. This happened to $\Lambda A(1,4) = 2^{-8}$, which resulted from scaling the first row of A, where originally $A(1,4) = 2^{-6}$.

Single-word algorithm: an example

$$C = \Lambda^{-1}\Big(\mathrm{fl}(\Lambda A)\mathrm{fl}(BM)\Big)M^{-1}$$

Step 3: Perform matrix multiply in the *accumulation format* binary16 ($T = 11$, $F_{\max} = 65504$).

$$\begin{bmatrix} 125 & 2^{-2} & 2^{-2} & 0 \\ 64 & 64 & 64 & 64 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 125.5 & 8032 & 125.5 & 125.5 \\ 256 & 16384 & 256 & 256 \\ 4 & 256 & 4 & 4 \\ 4 & 256 & 4 & 4 \end{bmatrix}$$

Single-word algorithm: an example

$$C = \Lambda^{-1}\Big(\mathrm{fl}(\Lambda A)\mathrm{fl}(BM)\Big)M^{-1}$$

Step 4: Undo the scaling.

$$\begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 125.5 & 8032 & 125.5 & 125.5 \\ 256 & 16384 & 256 & 256 \\ 4 & 256 & 4 & 4 \\ 4 & 256 & 4 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} 502 & 64256 & 502 & 502 \\ 512 & 65536 & 512 & 512 \\ 4 & 512 & 4 & 4 \\ 4 & 512 & 4 & 4 \end{bmatrix}$$

Single-word algorithm: an example

$$C = \Lambda^{-1}\Big(\mathrm{fl}(\Lambda A)\mathrm{fl}(BM)\Big)M^{-1}$$

Comparison. Our result computed with mixed-precision MMA:

$$AB \approx \begin{bmatrix} \mathbf{502} & \mathbf{64256} & \mathbf{502} & \mathbf{502} \\ 512 & 65536 & 512 & 512 \\ 4 & 512 & 4 & 4 \\ 4 & 512 & 4 & 4 \end{bmatrix}$$

And the exact result

$$AB = \begin{bmatrix} \mathbf{502.015625} & \mathbf{64258} & \mathbf{502.015625} & \mathbf{502.015625} \\ 512 & 65536 & 512 & 512 \\ 4 & 512 & 4 & 4 \\ 4 & 512 & 4 & 4 \end{bmatrix}$$

Double-word algorithm: an example

Step 2: Round to the *input format*, in **double-word representation**.

We will round each ΛA and BM to two fp8-E4M3 matrices instead of one.

Compute the first word (first of the two matrices):

$$A^{(0)} = \mathrm{fl}(\Lambda A) = \mathrm{fl}\left(\begin{bmatrix} 125 & 2^{-2} & 2^{-2} & 2^{-8} \\ 64 & 64 & 64 & 64 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}\right) = \begin{bmatrix} 125 & 2^{-2} & 2^{-2} & \mathbf{0} \\ 64 & 64 & 64 & 64 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$B^{(0)} = \mathrm{fl}(BM) = \mathrm{fl}\left(\begin{bmatrix} 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \end{bmatrix}\right) = \begin{bmatrix} 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \end{bmatrix}$$

Double-word algorithm: an example

Step 2: Round to the *input format* fp8-E4M3, in **double-word representation**.

Compute the second word (rounding/underflow error in the first step):

$$A^{(1)} = \mathrm{fl}((\Lambda A - A^{(0)})/u^1) =$$

$$\mathrm{fl}\left(\left(\begin{bmatrix} 125 & 2^{-2} & 2^{-2} & 2^{-8} \\ 64 & 64 & 64 & 64 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 125 & 2^{-2} & 2^{-2} & \mathbf{0} \\ 64 & 64 & 64 & 64 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}\right) ./2^{-4}\right) = \begin{bmatrix} 0 & 0 & 0 & \mathbf{2^{-4}} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Since $B^{(0)} = BM, B^{(1)} = \mathrm{zeros}(4, 4)$.

Extra scaling

Notice the division by $u^1 = 2^{-4}$ before rounding, which is done to reduce underflows in the input format. In general, the multi-word split is $A^{(i)} = \mathrm{fl}\left(\left(\Lambda A - \sum_{k=0}^{i-1} u^k A^{(k)}\right)/u^i\right)$.

Double-word algorithm: an example

Step 3: Perform matrix products and add them in the *accumulation format* binary16.

p-word case

After splitting ΛA and BM into $A^{(0)}, \dots, A^{(p-1)}$ and $B^{(0)}, \dots, B^{(p-1)}$, approximate matrix multiply by $p(p+1)/2$ products

$$C \approx \Lambda^{-1} \left(\sum_{i+j<p} u^{i+j} A^{(i)} B^{(j)} \right) M^{-1}.$$

In our double-word case

$$A^{(0)} B^{(0)} + u A^{(1)} B^{(0)} =$$

$$\begin{bmatrix} 125 & 2^{-2} & 2^{-2} & 0 \\ 64 & 64 & 64 & 64 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \end{bmatrix} + u \begin{bmatrix} 0 & 0 & 0 & \mathbf{2^{-4}} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \\ 1 & 64 & 1 & 1 \end{bmatrix}$$

Double-word algorithm: an example

$$A^{(0)}B^{(0)} + uA^{(1)}B^{(0)} =$$

$$\begin{bmatrix} 125.5 & 8032 & 125.5 & 125.5 \\ 256 & 16384 & 256 & 256 \\ 4 & 256 & 4 & 4 \\ 4 & 256 & 4 & 4 \end{bmatrix} + \begin{bmatrix} \mathbf{2^{-8}} & \mathbf{0.25} & \mathbf{2^{-8}} & \mathbf{2^{-8}} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} =$$

$$\begin{bmatrix} 125.50390625 & 8032.25 & 125.50390625 & 125.50390625 \\ 256 & 16384 & 256 & 256 \\ 4 & 256 & 4 & 4 \\ 4 & 256 & 4 & 4 \end{bmatrix}$$

Double-word algorithm: an example

$$C \approx \Lambda^{-1} \left(\sum_{i+j<p} u^{i+j} A^{(i)} B^{(j)} \right) M^{-1}.$$

Step 4: Undo the scaling.

$$\begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 125.50390625 & 8032.25 & 125.50390625 & 125.50390625 \\ 256 & 16384 & 256 & 256 \\ 4 & 256 & 4 & 4 \\ 4 & 256 & 4 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} 502.015625 & 64258 & 502.015625 & 502.015625 \\ 512 & 65536 & 512 & 512 \\ 4 & 512 & 4 & 4 \\ 4 & 512 & 4 & 4 \end{bmatrix} = AB.$$

Numerical experiments

We generate $A \in \mathbb{R}^{10\times n}$ and $B \in \mathbb{R}^{n\times 10}$ and vary n.

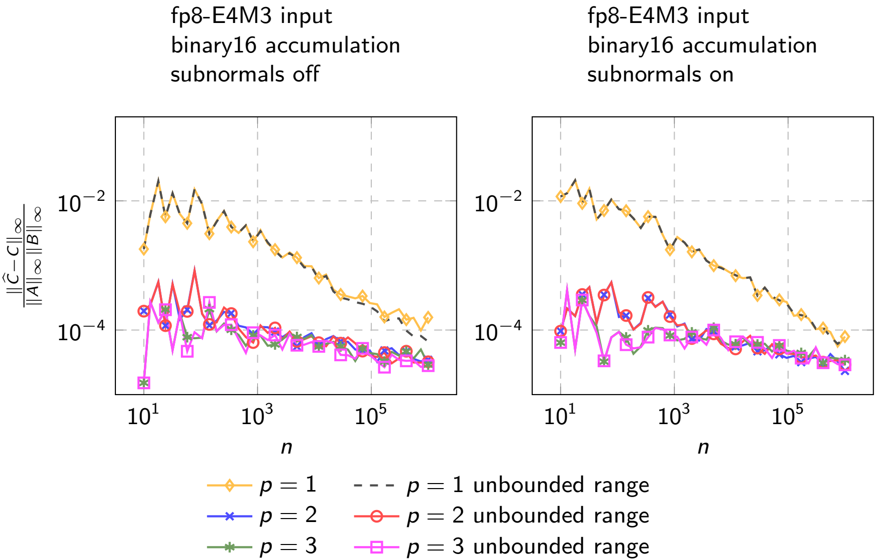
Elements in $[-10^{10}, -10^{-10}] \cup [10^{-10}, 10^{10}]$.

Measure the accuracy with $\frac{\|\widehat{C}-C\|_\infty}{\|A\|_\infty \|B\|_\infty}$ where C is computed in binary64.

We check with subnormals on/off to detect any improvements due to *gradual underflow*.

We also plot the variants of MMA without any range (exponent) limitations.

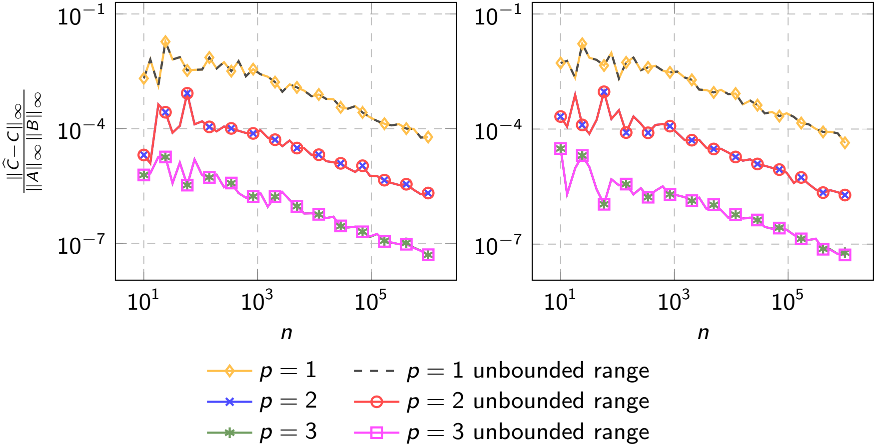
Numerical experiment



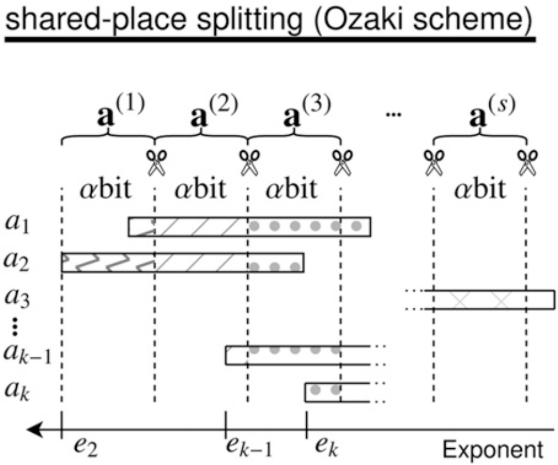
Numerical experiment

fp8-E4M3 input
binary32 accumulation
subnormals off

fp8-E4M3 input
binary32 accumulation
subnormals on



Utilising 8-bit integer tensor cores [Ootomo et al. 2024]



$$A = \begin{bmatrix} 1.5625 & 8 & -3.6875 \end{bmatrix}, \qquad B = \begin{bmatrix} 1.3828125 \\ -7.625 \\ 3.625 \end{bmatrix}$$

Example set up: FP precision 8 bits, 4 slices, integer: 3 bits and a sign

$$\underbrace{\begin{bmatrix} 2^0 \cdot 1.1001000 \\ 2^3 \cdot 1.0000000 \\ -2^1 \cdot 1.1101100 \end{bmatrix}}_{A^T} \Rightarrow 2^4 \cdot \underbrace{\begin{bmatrix} \not{0}.000\ 110\ 010\ 000 \\ \not{0}.100\ 000\ 000\ 000 \\ -\not{0}.001\ 110\ 110\ 000 \end{bmatrix}}_{\text{Block fixed-point}} \Rightarrow 2^1 \cdot \underbrace{\begin{bmatrix} 000 \\ 100 \\ -001 \end{bmatrix}}_{A^T_{(1)}} + 2^{-2} \cdot \underbrace{\begin{bmatrix} 110 \\ 000 \\ -110 \end{bmatrix}}_{A^T_{(2)}} + 2^{-5} \cdot \underbrace{\begin{bmatrix} 010 \\ 000 \\ -110 \end{bmatrix}}_{A^T_{(3)}} + 2^{-8} \cdot \underbrace{\begin{bmatrix} 000 \\ 000 \\ 000 \end{bmatrix}}_{A^T_{(4)}}$$

$$\underbrace{\begin{bmatrix} 2^0 \cdot 1.0110001 \\ -2^2 \cdot 1.1110100 \\ 2^1 \cdot 1.1101000 \end{bmatrix}}_{B} \Rightarrow 2^3 \cdot \underbrace{\begin{bmatrix} \not{0}.001\ 011\ 000\ 100 \\ -\not{0}.111\ 101\ 000\ 000 \\ \not{0}.011\ 101\ 000\ 000 \end{bmatrix}}_{\text{Block fixed-point}} \Rightarrow 2^0 \cdot \underbrace{\begin{bmatrix} 001 \\ -111 \\ 011 \end{bmatrix}}_{B^{(1)}} + 2^{-3} \cdot \underbrace{\begin{bmatrix} 011 \\ -101 \\ 101 \end{bmatrix}}_{B^{(2)}} + 2^{-6} \cdot \underbrace{\begin{bmatrix} 000 \\ 000 \\ 000 \end{bmatrix}}_{B^{(3)}} + 2^{-9} \cdot \underbrace{\begin{bmatrix} 100 \\ 000 \\ 000 \end{bmatrix}}_{B^{(4)}}$$

Our analysis of the method [Abdelfattah et al. 2025]

$A_{(1)}B^{(1)}$	2^{1} ·	−00011111
$A_{(1)}B^{(2)}$	2^{-2} ·	−00011001
$A_{(2)}B^{(1)}$	2^{-2} ·	−00001100
$A_{(1)}B^{(3)}$	2^{-5} ·	00000000
$A_{(2)}B^{(2)}$	2^{-5} ·	−00001100
$A_{(3)}B^{(1)}$	2^{-5} ·	−00010000
$A_{(1)}B^{(4)}$	2^{-8} ·	00000000
$A_{(2)}B^{(3)}$	2^{-8} ·	00000000
$A_{(3)}B^{(2)}$	2^{-8} ·	−00011000
$A_{(4)}B^{(1)}$	2^{-8} ·	00000000
$A_{(2)}B^{(4)}$	2^{-11} ·	00011000
$A_{(3)}B^{(3)}$	2^{-11} ·	00000000
$A_{(4)}B^{(2)}$	2^{-11} ·	00000000
$A_{(3)}B^{(4)}$	2^{-14} ·	00001000
$A_{(4)}B^{(3)}$	2^{-14} ·	00000000
$A_{(4)}B^{(4)}$	2^{-17} ·	00000000
AB	2^{-17} ·	−0010010000011010011100000

Our analysis of the method [Abdelfattah et al. 2025]

As a minimal example, we consider the computation of the inner product $a^T b$, where

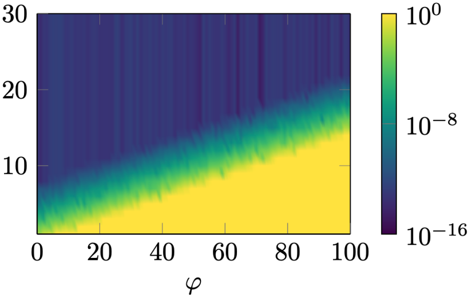
$$a = \begin{bmatrix} 2^{-\varphi} x \\ 1 \end{bmatrix}, \qquad b = \begin{bmatrix} 2^{\varphi} y \\ 1 \end{bmatrix}, \qquad x, y \sim \mathcal{N}(0, 1). \tag{1}$$

$$\frac{|\widehat{c} - c|}{|c|}, \tag{2}$$

where

- $\widehat{c}$ is computed with a variant of the Ozaki scheme with $T = 31$ and $t' = 7$
- c is a reference solution computed using the MATLAB Symbolic Toolbox with 32 decimal digits of accuracy.

Our analysis of the method [Abdelfattah et al. 2025]

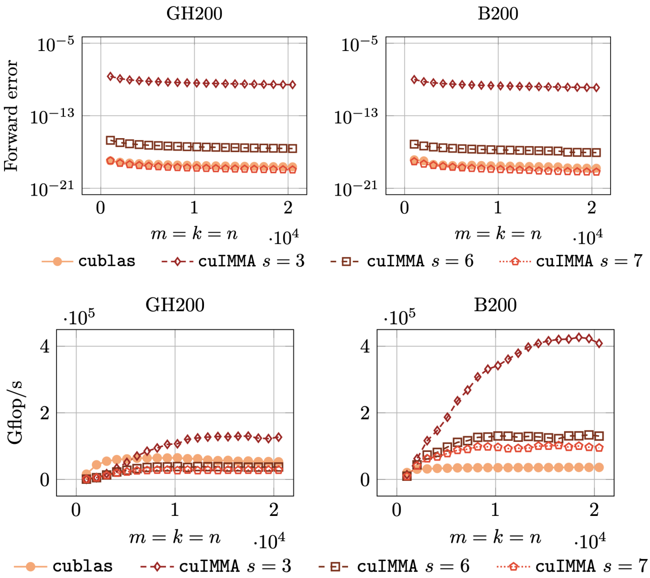


The x-axis denotes the number of slices and the y-axis controls the wideness of the gap between the min and max exponents.

LU factorisation

We did not discover need for large number of slices in many block-LU factorisation experiments. 8 slices sufficient. See paper.

Our analysis of the method [Abdelfattah et al. 2025]



Summary

- Low-precision matrix multipliers are being used for general-purpose computation.
- We are
  - developing software to analyse the properties of such hardware, and
  - developing algorithms for high-accuracy computations with low-precision units, and analysing them.

8-bit integer matrix multiply paper

A. bdelfattah, J. Dongarra, M. Fasi, M. Mikaitis, and F. Tisseur. *Analysis of Floating-Point Matrix Multiplication Computed via Integer Arithmetic* . **Preprint, arXiv:2506.11277 [math.NA]**. Jun. 2025.

Slides at `http://mmikaitis.github.io/talks`

References I

IEEE P3109 Working Group
Interim Report on Binary Floating-point Formats for Machine Learning

https://github.com/P3109/Public

K. Hillesland and A. Lastra
GPU Floating-Point Paranoia
Preprint.

X. Tan, D. Boland, and G. A. Constantinides
FPGA Paranoia: Testing Numerical Properties of FPGA Floating Point IP-Cores
LNCS 7199. 2012.

B. Hickmann and D. Bradford
Experimental Analysis of Matrix Multiplication Functional Units
IEEE 26th Symposium on Computer Arithmetic. 2019.

References II

M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh
Numerical Behavior of NVIDIA Tensor Cores
PeerJ Comp. Sci. Feb. 2021

M. Mikaitis
Monotonicity of Multi-Term Floating-Point Adders
IEEE Trans. Comput., 73. 2024.

X. Li, A. Li, B. Fang, K. Swirydowicz, I. Laguna, G. Gopalakrishnan
FTTN: Feature-Targeted Testing for Numerical Properties of NVIDIA & AMD Matrix Accelerators newblock
Preprint. arXiv:2403.00232. 2024.

S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, J. S. Vetter
NVIDIA Tensor Core Programmability, Performance & Precision
IEEE International Parallel and Distributed Processing Symposium Workshops. 2018.

References III

D. Mukunoki, K. Ozaki, T. Ogita, T. Imamura
DGEMM Using Tensor Cores, and Its Accurate and Reproducible Versions
LNCS 12151. 2020.

T. Mary and M. Mikaitis
Error analysis of matrix multiplication with narrow range floating-point arithmetic
SIAM J. Sci. Comput., 47. 2025.

H. Ootomo, K. Ozaki and R. Yokota
DGEMM on integer matrix multiplication unit
Int. J. High Perf. Comput. Appl., 38. 2024.