Error Analysis of Matrix Multiplication with Narrow Range Floating-Point Arithmetic

Mantas Mikaitis

School of Computer Science, University of Leeds, Leeds, UK
Joint work with Theo Mary, CNRS, Paris, France

FHNW Campus Brugg-Windisch
The Platform for Advanced Scientific Computing conference (PASC25)
Brugg, Switzerland
18 June, 2025

MS5F: Fast and Accurate Numerical Linear Algebra on Low-Precision Hardware: Algorithms and Error Analysis

Presentations:

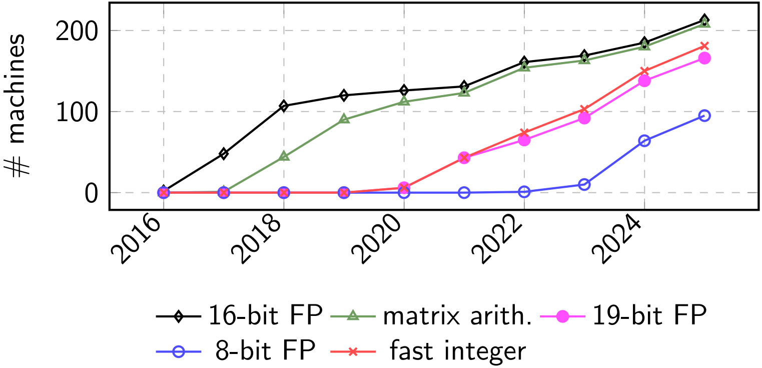
9:00-9:30 *Error Analysis of Matrix Multiplication with Narrow Range Floating-Point Arithmetic*. **Mantas Mikaitis (Univ. Leeds)**

9:30-10:00 *Fast and Accurate Algorithm Efficiently Using FMA for Matrix Multiplication*. **Katsuhisa Ozaki (Shibaura Institute of Technology)**

10:00-10:30 *DGEMM Emulation Using INT8 Matrix Engines and its Rounding Error Analysis*. **Yuki Uchino (RIKEN Center for Computational Science)**

10:30-11:00 *Precision Redefined: Unlocking and Delivering the Full Power of Modern GPUs for Scientific Computing*. **Harun Bayraktar (NVIDIA)**

8-bit 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}).

IEEE P3109: 8-bit format on one slide

C.4 **Value Table: P4,** emin $= -7$, emax $= 7$

0x00 = 0_0000_000 = 0.0	0x40 = 0_1000_000 = +0b1.000×2^0 = 1.0	0x80 = 1_0000_000 = NaN	0xc0 = 1_1000_000 = −0b1.000×2^0 = -1.0
0x01 = 0_0000_001 = +0b0.001×2^-7 = 0.0009765625	0x41 = 0_1000_001 = +0b1.001×2^0 = 1.125	0x81 = 1_0000_001 = −0b0.001×2^-7 = −0.0009765625	0xc1 = 1_1000_001 = −0b1.001×2^0 = -1.125
0x02 = 0_0000_010 = +0b0.010×2^-7 = 0.001953125	0x42 = 0_1000_010 = +0b1.010×2^0 = 1.25	0x82 = 1_0000_010 = −0b0.010×2^-7 = −0.001953125	0xc2 = 1_1000_010 = −0b1.010×2^0 = -1.25
0x03 = 0_0000_011 = +0b0.011×2^-7 = 0.0029296875	0x43 = 0_1000_011 = +0b1.011×2^0 = 1.375	0x83 = 1_0000_011 = −0b0.011×2^-7 = −0.0029296875	0xc3 = 1_1000_011 = −0b1.011×2^0 = -1.375
0x04 = 0_0000_100 = +0b0.100×2^-7 = 0.00390625	0x44 = 0_1000_100 = +0b1.100×2^0 = 1.5	0x84 = 1_0000_100 = −0b0.100×2^-7 = −0.00390625	0xc4 = 1_1000_100 = −0b1.100×2^0 = -1.5
0x05 = 0_0000_101 = +0b0.101×2^-7 = 0.0048828125	0x45 = 0_1000_101 = +0b1.101×2^0 = 1.625	0x85 = 1_0000_101 = −0b0.101×2^-7 = −0.0048828125	0xc5 = 1_1000_101 = −0b1.101×2^0 = -1.625
0x06 = 0_0000_110 = +0b0.110×2^-7 = 0.005859375	0x46 = 0_1000_110 = +0b1.110×2^0 = 1.75	0x86 = 1_0000_110 = −0b0.110×2^-7 = −0.005859375	0xc6 = 1_1000_110 = −0b1.110×2^0 = -1.75
0x07 = 0_0000_111 = +0b0.111×2^-7 = 0.0068359375	0x47 = 0_1000_111 = +0b1.111×2^0 = 1.875	0x87 = 1_0000_111 = −0b0.111×2^-7 = −0.0068359375	0xc7 = 1_1000_111 = −0b1.111×2^0 = -1.875
0x08 = 0_0001_000 = +0b1.000×2^-7 = 0.0078125	0x48 = 0_1001_000 = +0b1.000×2^1 = 2.0	0x88 = 1_0001_000 = −0b1.000×2^-7 = -0.0078125	0xc8 = 1_1001_000 = −0b1.000×2^1 = -2.0
0x09 = 0_0001_001 = +0b1.001×2^-7 = 0.0087890625	0x49 = 0_1001_001 = +0b1.001×2^1 = 2.25	0x89 = 1_0001_001 = −0b1.001×2^-7 = -0.0087890625	0xc9 = 1_1001_001 = −0b1.001×2^1 = -2.25
0x0a = 0_0001_010 = +0b1.010×2^-7 = 0.009765625	0x4a = 0_1001_010 = +0b1.010×2^1 = 2.5	0x8a = 1_0001_010 = −0b1.010×2^-7 = -0.009765625	0xca = 1_1001_010 = −0b1.010×2^1 = -2.5
0x0b = 0_0001_011 = +0b1.011×2^-7 = 0.0107421875	0x4b = 0_1001_011 = +0b1.011×2^1 = 2.75	0x8b = 1_0001_011 = −0b1.011×2^-7 = -0.0107421875	0xcb = 1_1001_011 = −0b1.011×2^1 = -2.75
0x0c = 0_0001_100 = +0b1.100×2^-7 = 0.01171875	0x4c = 0_1001_100 = +0b1.100×2^1 = 3.0	0x8c = 1_0001_100 = −0b1.100×2^-7 = -0.01171875	0xcc = 1_1001_100 = −0b1.100×2^1 = -3.0
0x0d = 0_0001_101 = +0b1.101×2^-7 = 0.0126953125	0x4d = 0_1001_101 = +0b1.101×2^1 = 3.25	0x8d = 1_0001_101 = −0b1.101×2^-7 = -0.0126953125	0xcd = 1_1001_101 = −0b1.101×2^1 = -3.25
0x0e = 0_0001_110 = +0b1.110×2^-7 = 0.013671875	0x4e = 0_1001_110 = +0b1.110×2^1 = 3.5	0x8e = 1_0001_110 = −0b1.110×2^-7 = -0.013671875	0xce = 1_1001_110 = −0b1.110×2^1 = -3.5
0x0f = 0_0001_111 = +0b1.111×2^-7 = 0.0146484375	0x4f = 0_1001_111 = +0b1.111×2^1 = 3.75	0x8f = 1_0001_111 = −0b1.111×2^-7 = -0.0146484375	0xcf = 1_1001_111 = −0b1.111×2^1 = -3.75
0x10 = 0_0010_000 = +0b1.000×2^-6 = 0.015625	0x50 = 0_1010_000 = +0b1.000×2^2 = 4.0	0x90 = 1_0010_000 = −0b1.000×2^-6 = -0.015625	0xd0 = 1_1010_000 = −0b1.000×2^2 = -4.0
0x11 = 0_0010_001 = +0b1.001×2^-6 = 0.017578125	0x51 = 0_1010_001 = +0b1.001×2^2 = 4.5	0x91 = 1_0010_001 = −0b1.001×2^-6 = -0.017578125	0xd1 = 1_1010_001 = −0b1.001×2^2 = -4.5
0x12 = 0_0010_010 = +0b1.010×2^-6 = 0.01953125	0x52 = 0_1010_010 = +0b1.010×2^2 = 5.0	0x92 = 1_0010_010 = −0b1.010×2^-6 = -0.01953125	0xd2 = 1_1010_010 = −0b1.010×2^2 = -5.0
0x13 = 0_0010_011 = +0b1.011×2^-6 = 0.021484375	0x53 = 0_1010_011 = +0b1.011×2^2 = 5.5	0x93 = 1_0010_011 = −0b1.011×2^-6 = -0.021484375	0xd3 = 1_1010_011 = −0b1.011×2^2 = -5.5
0x14 = 0_0010_100 = +0b1.100×2^-6 = 0.0234375	0x54 = 0_1010_100 = +0b1.100×2^2 = 6.0	0x94 = 1_0010_100 = −0b1.100×2^-6 = -0.0234375	0xd4 = 1_1010_100 = −0b1.100×2^2 = -6.0
0x15 = 0_0010_101 = +0b1.101×2^-6 = 0.025390625	0x55 = 0_1010_101 = +0b1.101×2^2 = 6.5	0x95 = 1_0010_101 = −0b1.101×2^-6 = -0.025390625	0xd5 = 1_1010_101 = −0b1.101×2^2 = -6.5
0x16 = 0_0010_110 = +0b1.110×2^-6 = 0.02734375	0x56 = 0_1010_110 = +0b1.110×2^2 = 7.0	0x96 = 1_0010_110 = −0b1.110×2^-6 = -0.02734375	0xd6 = 1_1010_110 = −0b1.110×2^2 = -7.0
0x17 = 0_0010_111 = +0b1.111×2^-6 = 0.029296875	0x57 = 0_1010_111 = +0b1.111×2^2 = 7.5	0x97 = 1_0010_111 = −0b1.111×2^-6 = -0.029296875	0xd7 = 1_1010_111 = −0b1.111×2^2 = -7.5
0x18 = 0_0011_000 = +0b1.000×2^-5 = 0.03125	0x58 = 0_1011_000 = +0b1.000×2^3 = 8.0	0x98 = 1_0011_000 = −0b1.000×2^-5 = -0.03125	0xd8 = 1_1011_000 = −0b1.000×2^3 = -8.0
0x19 = 0_0011_001 = +0b1.001×2^-5 = 0.03515625	0x59 = 0_1011_001 = +0b1.001×2^3 = 9.0	0x99 = 1_0011_001 = −0b1.001×2^-5 = -0.03515625	0xd9 = 1_1011_001 = −0b1.001×2^3 = -9.0
0x1a = 0_0011_010 = +0b1.010×2^-5 = 0.0390625	0x5a = 0_1011_010 = +0b1.010×2^3 = 10.0	0x9a = 1_0011_010 = −0b1.010×2^-5 = -0.0390625	0xda = 1_1011_010 = −0b1.010×2^3 = -10.0
0x1b = 0_0011_011 = +0b1.011×2^-5 = 0.04296875	0x5b = 0_1011_011 = +0b1.011×2^3 = 11.0	0x9b = 1_0011_011 = −0b1.011×2^-5 = -0.04296875	0xdb = 1_1011_011 = −0b1.011×2^3 = -11.0
0x1c = 0_0011_100 = +0b1.100×2^-5 = 0.046875	0x5c = 0_1011_100 = +0b1.100×2^3 = 12.0	0x9c = 1_0011_100 = −0b1.100×2^-5 = -0.046875	0xdc = 1_1011_100 = −0b1.100×2^3 = -12.0
0x1d = 0_0011_101 = +0b1.101×2^-5 = 0.05078125	0x5d = 0_1011_101 = +0b1.101×2^3 = 13.0	0x9d = 1_0011_101 = −0b1.101×2^-5 = -0.05078125	0xdd = 1_1011_101 = −0b1.101×2^3 = -13.0
0x1e = 0_0011_110 = +0b1.110×2^-5 = 0.0546875	0x5e = 0_1011_110 = +0b1.110×2^3 = 14.0	0x9e = 1_0011_110 = −0b1.110×2^-5 = -0.0546875	0xde = 1_1011_110 = −0b1.110×2^3 = -14.0
0x1f = 0_0011_111 = +0b1.111×2^-5 = 0.05859375	0x5f = 0_1011_111 = +0b1.111×2^3 = 15.0	0x9f = 1_0011_111 = −0b1.111×2^-5 = -0.05859375	0xdf = 1_1011_111 = −0b1.111×2^3 = -15.0
0x20 = 0_0100_000 = +0b1.000×2^-4 = 0.0625	0x60 = 0_1100_000 = +0b1.000×2^4 = 16.0	0xa0 = 1_0100_000 = −0b1.000×2^-4 = -0.0625	0xe0 = 1_1100_000 = −0b1.000×2^4 = -16.0
0x21 = 0_0100_001 = +0b1.001×2^-4 = 0.0703125	0x61 = 0_1100_001 = +0b1.001×2^4 = 18.0	0xa1 = 1_0100_001 = −0b1.001×2^-4 = -0.0703125	0xe1 = 1_1100_001 = −0b1.001×2^4 = -18.0
0x22 = 0_0100_010 = +0b1.010×2^-4 = 0.078125	0x62 = 0_1100_010 = +0b1.010×2^4 = 20.0	0xa2 = 1_0100_010 = −0b1.010×2^-4 = -0.078125	0xe2 = 1_1100_010 = −0b1.010×2^4 = -20.0
0x23 = 0_0100_011 = +0b1.011×2^-4 = 0.0859375	0x63 = 0_1100_011 = +0b1.011×2^4 = 22.0	0xa3 = 1_0100_011 = −0b1.011×2^-4 = -0.0859375	0xe3 = 1_1100_011 = −0b1.011×2^4 = -22.0
0x24 = 0_0100_100 = +0b1.100×2^-4 = 0.09375	0x64 = 0_1100_100 = +0b1.100×2^4 = 24.0	0xa4 = 1_0100_100 = −0b1.100×2^-4 = -0.09375	0xe4 = 1_1100_100 = −0b1.100×2^4 = -24.0
0x25 = 0_0100_101 = +0b1.101×2^-4 = 0.1015625	0x65 = 0_1100_101 = +0b1.101×2^4 = 26.0	0xa5 = 1_0100_101 = −0b1.101×2^-4 = -0.1015625	0xe5 = 1_1100_101 = −0b1.101×2^4 = -26.0
0x26 = 0_0100_110 = +0b1.110×2^-4 = 0.109375	0x66 = 0_1100_110 = +0b1.110×2^4 = 28.0	0xa6 = 1_0100_110 = −0b1.110×2^-4 = -0.109375	0xe6 = 1_1100_110 = −0b1.110×2^4 = -28.0
0x27 = 0_0100_111 = +0b1.111×2^-4 = 0.1171875	0x67 = 0_1100_111 = +0b1.111×2^4 = 30.0	0xa7 = 1_0100_111 = −0b1.111×2^-4 = -0.1171875	0xe7 = 1_1100_111 = −0b1.111×2^4 = -30.0
0x28 = 0_0101_000 = +0b1.000×2^-3 = 0.125	0x68 = 0_1101_000 = +0b1.000×2^5 = 32.0	0xa8 = 1_0101_000 = −0b1.000×2^-3 = -0.125	0xe8 = 1_1101_000 = −0b1.000×2^5 = -32.0
0x29 = 0_0101_001 = +0b1.001×2^-3 = 0.140625	0x69 = 0_1101_001 = +0b1.001×2^5 = 36.0	0xa9 = 1_0101_001 = −0b1.001×2^-3 = -0.140625	0xe9 = 1_1101_001 = −0b1.001×2^5 = -36.0
0x2a = 0_0101_010 = +0b1.010×2^-3 = 0.15625	0x6a = 0_1101_010 = +0b1.010×2^5 = 40.0	0xaa = 1_0101_010 = −0b1.010×2^-3 = -0.15625	0xea = 1_1101_010 = −0b1.010×2^5 = -40.0
0x2b = 0_0101_011 = +0b1.011×2^-3 = 0.171875	0x6b = 0_1101_011 = +0b1.011×2^5 = 44.0	0xab = 1_0101_011 = −0b1.011×2^-3 = -0.171875	0xeb = 1_1101_011 = −0b1.011×2^5 = -44.0
0x2c = 0_0101_100 = +0b1.100×2^-3 = 0.1875	0x6c = 0_1101_100 = +0b1.100×2^5 = 48.0	0xac = 1_0101_100 = −0b1.100×2^-3 = -0.1875	0xec = 1_1101_100 = −0b1.100×2^5 = -48.0
0x2d = 0_0101_101 = +0b1.101×2^-3 = 0.203125	0x6d = 0_1101_101 = +0b1.101×2^5 = 52.0	0xad = 1_0101_101 = −0b1.101×2^-3 = -0.203125	0xed = 1_1101_101 = −0b1.101×2^5 = -52.0
0x2e = 0_0101_110 = +0b1.110×2^-3 = 0.21875	0x6e = 0_1101_110 = +0b1.110×2^5 = 56.0	0xae = 1_0101_110 = −0b1.110×2^-3 = -0.21875	0xee = 1_1101_110 = −0b1.110×2^5 = -56.0
0x2f = 0_0101_111 = +0b1.111×2^-3 = 0.234375	0x6f = 0_1101_111 = +0b1.111×2^5 = 60.0	0xaf = 1_0101_111 = −0b1.111×2^-3 = -0.234375	0xef = 1_1101_111 = −0b1.111×2^5 = -60.0
0x30 = 0_0110_000 = +0b1.000×2^-2 = 0.25	0x70 = 0_1110_000 = +0b1.000×2^6 = 64.0	0xb0 = 1_0110_000 = −0b1.000×2^-2 = -0.25	0xf0 = 1_1110_000 = −0b1.000×2^6 = -64.0
0x31 = 0_0110_001 = +0b1.001×2^-2 = 0.28125	0x71 = 0_1110_001 = +0b1.001×2^6 = 72.0	0xb1 = 1_0110_001 = −0b1.001×2^-2 = -0.28125	0xf1 = 1_1110_001 = −0b1.001×2^6 = -72.0
0x32 = 0_0110_010 = +0b1.010×2^-2 = 0.3125	0x72 = 0_1110_010 = +0b1.010×2^6 = 80.0	0xb2 = 1_0110_010 = −0b1.010×2^-2 = -0.3125	0xf2 = 1_1110_010 = −0b1.010×2^6 = -80.0
0x33 = 0_0110_011 = +0b1.011×2^-2 = 0.34375	0x73 = 0_1110_011 = +0b1.011×2^6 = 88.0	0xb3 = 1_0110_011 = −0b1.011×2^-2 = -0.34375	0xf3 = 1_1110_011 = −0b1.011×2^6 = -88.0
0x34 = 0_0110_100 = +0b1.100×2^-2 = 0.375	0x74 = 0_1110_100 = +0b1.100×2^6 = 96.0	0xb4 = 1_0110_100 = −0b1.100×2^-2 = -0.375	0xf4 = 1_1110_100 = −0b1.100×2^6 = -96.0
0x35 = 0_0110_101 = +0b1.101×2^-2 = 0.40625	0x75 = 0_1110_101 = +0b1.101×2^6 = 104.0	0xb5 = 1_0110_101 = −0b1.101×2^-2 = -0.40625	0xf5 = 1_1110_101 = −0b1.101×2^6 = -104.0
0x36 = 0_0110_110 = +0b1.110×2^-2 = 0.4375	0x76 = 0_1110_110 = +0b1.110×2^6 = 112.0	0xb6 = 1_0110_110 = −0b1.110×2^-2 = -0.4375	0xf6 = 1_1110_110 = −0b1.110×2^6 = -112.0
0x37 = 0_0110_111 = +0b1.111×2^-2 = 0.46875	0x77 = 0_1110_111 = +0b1.111×2^6 = 120.0	0xb7 = 1_0110_111 = −0b1.111×2^-2 = -0.46875	0xf7 = 1_1110_111 = −0b1.111×2^6 = -120.0
0x38 = 0_0111_000 = +0b1.000×2^-1 = 0.5	0x78 = 0_1111_000 = +0b1.000×2^7 = 128.0	0xb8 = 1_0111_000 = −0b1.000×2^-1 = -0.5	0xf8 = 1_1111_000 = −0b1.000×2^7 = -128.0
0x39 = 0_0111_001 = +0b1.001×2^-1 = 0.5625	0x79 = 0_1111_001 = +0b1.001×2^7 = 144.0	0xb9 = 1_0111_001 = −0b1.001×2^-1 = -0.5625	0xf9 = 1_1111_001 = −0b1.001×2^7 = -144.0
0x3a = 0_0111_010 = +0b1.010×2^-1 = 0.625	0x7a = 0_1111_010 = +0b1.010×2^7 = 160.0	0xba = 1_0111_010 = −0b1.010×2^-1 = -0.625	0xfa = 1_1111_010 = −0b1.010×2^7 = -160.0
0x3b = 0_0111_011 = +0b1.011×2^-1 = 0.6875	0x7b = 0_1111_011 = +0b1.011×2^7 = 176.0	0xbb = 1_0111_011 = −0b1.011×2^-1 = -0.6875	0xfb = 1_1111_011 = −0b1.011×2^7 = -176.0
0x3c = 0_0111_100 = +0b1.100×2^-1 = 0.75	0x7c = 0_1111_100 = +0b1.100×2^7 = 192.0	0xbc = 1_0111_100 = −0b1.100×2^-1 = -0.75	0xfc = 1_1111_100 = −0b1.100×2^7 = -192.0
0x3d = 0_0111_101 = +0b1.101×2^-1 = 0.8125	0x7d = 0_1111_101 = +0b1.101×2^7 = 208.0	0xbd = 1_0111_101 = −0b1.101×2^-1 = -0.8125	0xfd = 1_1111_101 = −0b1.101×2^7 = -208.0
0x3e = 0_0111_110 = +0b1.110×2^-1 = 0.875	0x7e = 0_1111_110 = +0b1.110×2^7 = 224.0	0xbe = 1_0111_110 = −0b1.110×2^-1 = -0.875	0xfe = 1_1111_110 = −0b1.110×2^7 = -224.0
0x3f = 0_0111_111 = +0b1.111×2^-1 = 0.9375	0x7f = 0_1111_111 = +Inf	0xbf = 1_0111_111 = −0b1.111×2^-1 = -0.9375	0xff = 1_1111_111 = −Inf

4/6/8/16-bit floating point formats have narrow ranges

Format	precision	min pos.	max pos.	u
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	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}

Mixed-precision matrix multipliers

Formats with narrow ranges are available in matrix multiply operation.

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

Hardware matrix multipliers in mixed precision

- Example above is 4×4, but dimensions differ across architectures.
- Internal dot product precision, rounding, and subnormal support.

While the input formats have narrow ranges, the output is less constrained on both the precision and range.

Part 1: Basic single-word algorithm

Single-word algorithm

Goal: Given A and B, matrices in 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.

Single-word algorithm

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 $\leq f_{\min}/2$ 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

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

Part 2: Multi-word algorithm

Double-word algorithm: example

Again, for a step-by-step example, take

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

Double-word algorithm: an example

Step 1: Scale A and B (same as before).

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

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

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 $A^{(0)}, \ldots, A^{(p-1)}$ and $B^{(0)}, \ldots, B^{(p-1)}$, 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

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

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

Double-word algorithm: an example

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

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

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

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

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.

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

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

Part 3: Numerical experiments

Numerical experiments

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

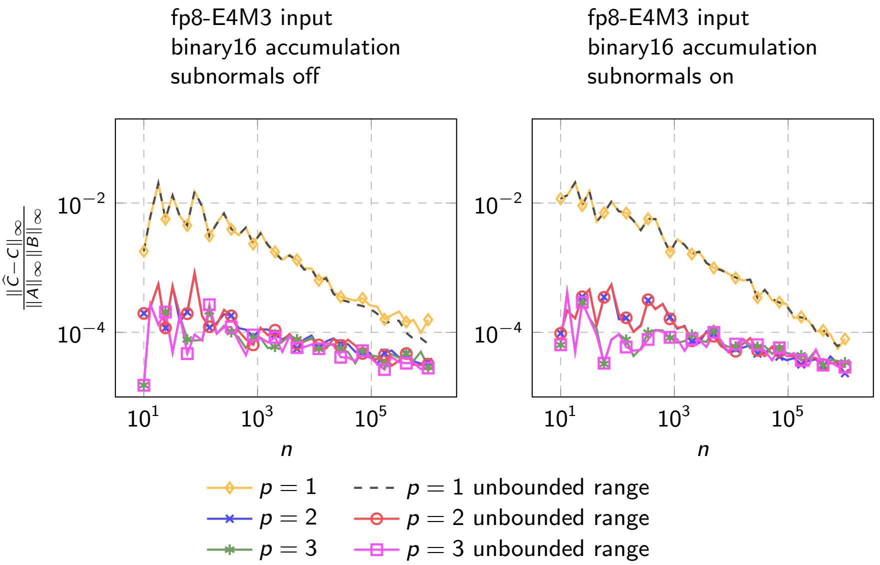
Elements in $[-10^{10}, -10^{-10}] \cup [10^{-10}, 10^{10}]$.

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

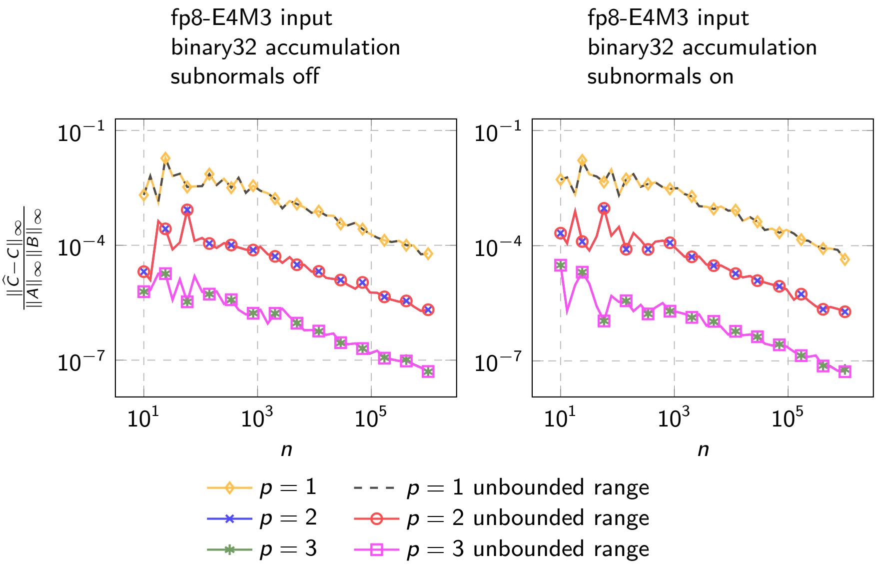
We check with subnormals on/off to detect any improvements due to *gradual underflow*.

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

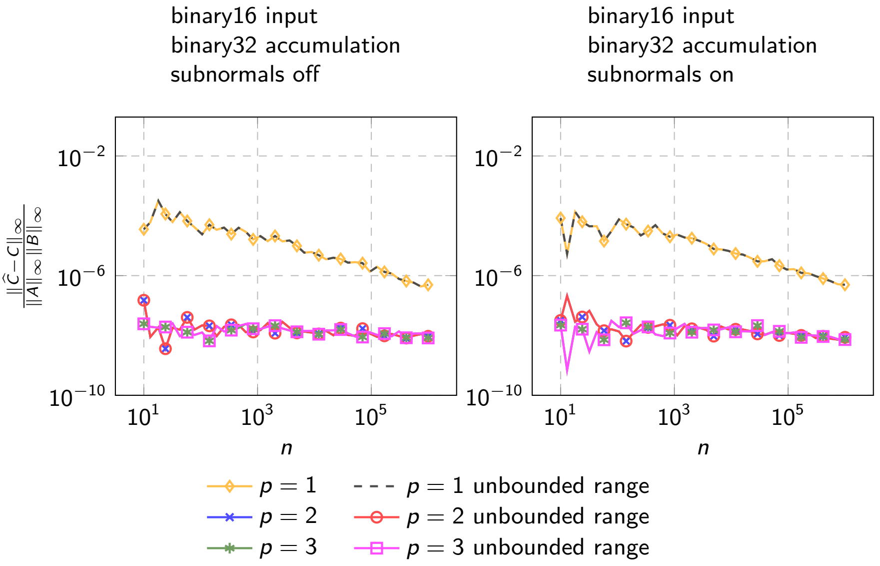
Numerical experiments I



Numerical experiments II



Numerical experiments III



Part 4: Error analysis

Matrix Multiply-Accumulate (MMA)

Model 1

The following model describes a mixed-precision MMA operation to compute $C = AB$, assuming round-to-nearest ties-to-even is used. We have two FP formats:

- *Input format* with precision t, unit roundoff $u = 2^{-t}$, exponent in $[e_{min}, e_{max}]$, range of normalized values $\pm[f_{\min}, f_{\max}]$. The maximum distance between any number in $[-f_{\min}, f_{\min}]$ and the nearest FP number is

$$g_{\min} = \begin{cases} f_{\min}/2 & \text{if subnormals are not available} \\ uf_{\min} & \text{with subnormals (gradual underflow)} \end{cases}$$

- *Accumulation format* with $T \geq t$, $U = 2^{-T}$, exponent in $[E_{\min}, E_{\max}] \supseteq [e_{\min}, e_{\max}]$, and range of norm. numbers $\pm[F_{\min}, F_{\max}]$. The maximum distance between any number in $[-F_{\min}, F_{\min}]$ and the nearest FP number is

$$G_{\min} = \begin{cases} F_{\min}/2 & \text{if subnormals are not available} \\ UF_{\min} & \text{with subnormals (gradual underflow)} \end{cases}$$

Models of worst-case rounding errors

Rounding error model based on [Demmel, 1984]

Take $x, y \in \mathbb{R}$. Assuming no overflows occur, the rounding operator to the *input format* is described as

$$\mathsf{fl}(x) = x(1+\delta) + \eta, \quad |\delta| \le u, \quad |\eta| \le g_{\min}, \quad \eta\delta = 0,$$

and arithmetic operations in the *accumulation format* as

$$\mathsf{FL}(x \operatorname{op} y) = (x \operatorname{op} y)(1+\delta) + \eta, \quad |\delta| \le U, \quad |\eta| \le G_{\min}, \quad \eta\delta = 0,$$

with $\operatorname{op} \in \{+, -, \times, \div\}$.

Here $\eta\delta = 0$ accounts for only one type of error, rounding or overflow.

Error analysis: summary

Single-word algorithm:

$$\|\widehat{C} - AB\|_\infty \lesssim \Big(2u + nU + 4n^2\theta^{-1}g_{\min} + 4n^2\theta^{-2}G_{\min}\Big)\|A\|_\infty\|B\|_\infty.$$

p-word algorithm:

$$\|\widehat{C} - AB\|_\infty \lesssim \Big((p+1)u^p + 4nu^{p-1}\theta^{-1}g_{\min} + (n+p^2)U + 2p(p+1)n^2\theta^{-2}G_{\min}\Big)\|A\|_\infty\|B\|_\infty.$$

Summary

- Underflows in narrow-range FP formats not a problem, provided three types of scaling are used.
- Shown algorithms require minimal bit-level manipulations.
- Can be used to obtain binary32 accuracy in high performance.
- If higher accuracy is needed, MMA can still be used in conjunction with binary64—see the next talks.

SIAM SISC paper

T. Mary, and M. Mikaitis. *Error Analysis of Matrix Multiplication with Narrow Range Floating-Point Arithmetic*. **Preprint. Accepted for SIAM SISC**. Mar. 2025. `https://bit.ly/42dqpkn`.

Slides and more info at `http://mmikaitis.github.io`

References I

J. Demmel
Underflow and the reliability of numerical software
SIAM J. Sci. Comput., 5:4. Dec. 1984.

P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh
Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores
SIAM J. Sci. Comput., 42:3. Jan. 2020.

M. Fasi, N. J. Higham, F. Lopez, T. Mary, and M. Mikaitis
Matrix Multiplication in Multiword Arithmetic: Error Analysis and Application to GPU Tensor Cores
SIAM J. Sci. Comput., 45:1. Feb. 2023.

M. Fasi and M. Mikaitis
CPFloat: A C library for Simulating Low-Precision Arithmetic
ACM Trans. Math. Software, 49. 2023