Vector-friendly floats with $64n$-bit precision

Fredrik Johansson

Inria Bordeaux

2025-07-10
FPTalks 2025 (online)

Multiple precision floating-point numbers

$$(-1)^{sgn} \cdot 2^{exp} \cdot [d_{n-1}\beta^{-1} + d_{n-2}\beta^{-2} + \ldots + d_0\beta^{-n}]$$

$$d_i \, : \, \text{64-bit digit}, \quad \beta = 2^{64}$$

Goal: drop-in replacement for `mpfr` or FLINT's `arf` for vectors with the same ($64n$-bit) precision for all operands.

https://flintlib.org/doc/nfloat.html

Requirements

- 64-bit exponents
- Normalized significand (MSB of d_{n-1})
- Bounded relative error (e.g. 2 ulp)
- Error handling (over/underflow/domain)

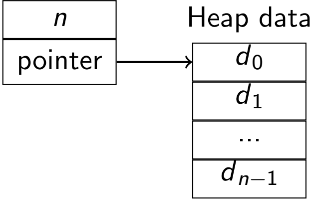
Non-requirements

- Correct rounding
- NaNs and Infs

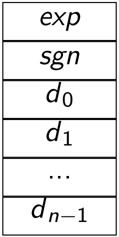
Packing things efficiently

`mpf`, `mpfr`, `arf`

(Rough layout)

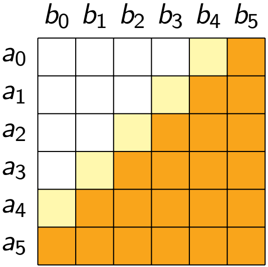


`nfloat`



We restrict `nfloat` to $1 \leq n \leq 66$ (64 to 4224 bits) so that temporary variables can be created safely on the C stack.

High multiplication

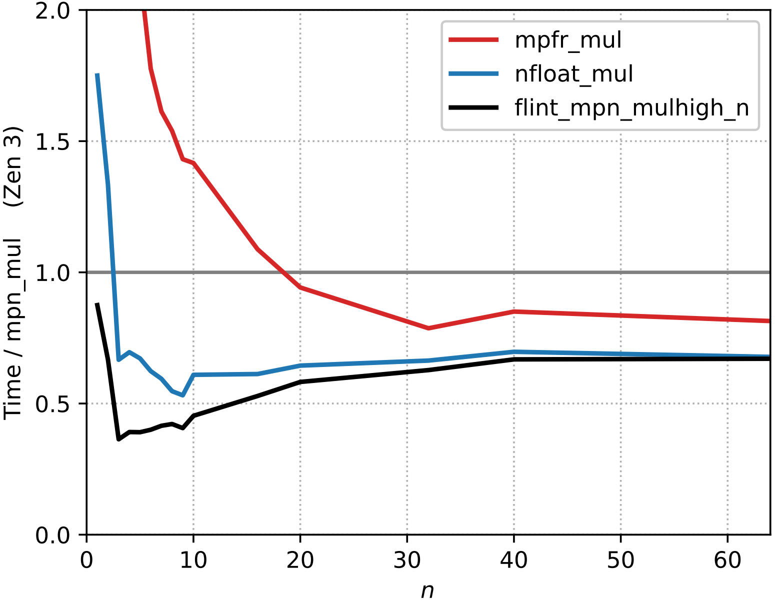


Terms to approximate the high n words of $\left(\sum_{i=0}^{n-1} a_i \beta^i\right) \cdot \left(\sum_{j=0}^{n-1} b_j \beta^j\right)$

Joint work with Albin Ahlbäck (ARITH '25)

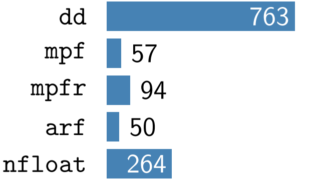
- $n(n+1)/2$ terms for $O(n)$ ulp error
- $n-1$ correction terms for $O(n/\beta)$ ulp error
- Hardcoded basecases for x86-64 and arm64 ($n \lesssim 9$)
- Mulders-style splitting for Karatsuba-size n

High multiplication

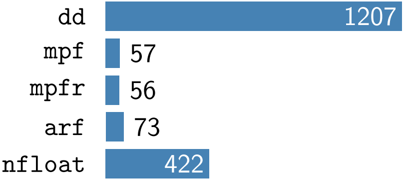


Mflop/s[1] $n = 2$ 128-bit precision[2]

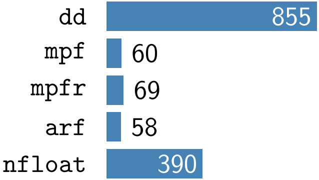
add ($z_i \leftarrow x_i + y_i$)



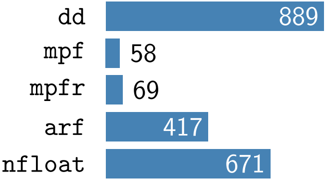
mul ($z_i \leftarrow x_i \cdot y_i$)



addmul_scalar ($z_i \leftarrow z_i + x_i \cdot y$)



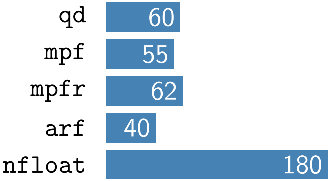
dot ($z \leftarrow \sum_i x_i y_i$)



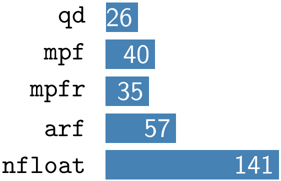
[1] AMD Ryzen 7 PRO 5850U (Zen 3), GCC 11, length-100 vectors

[2] 106-bit double-double (dd) arithmetic included for comparison

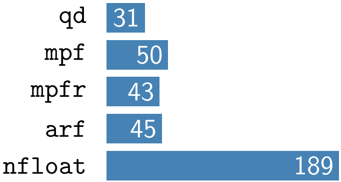
add $(z_i \leftarrow x_i + y_i)$



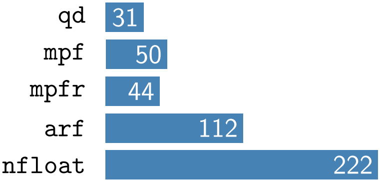
mul $(z_i \leftarrow x_i \cdot y_i)$



addmul_scalar $(z_i \leftarrow z_i + x_i \cdot y)$



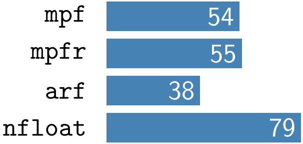
dot $(z \leftarrow \sum_i x_i y_i)$



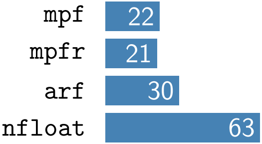
[3] 212-bit quad-double (qd) arithmetic from the QD library included for comparison

Mflop/s $n = 8$ 512-bit precision

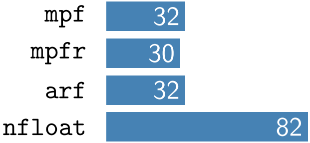
add $(z_i \leftarrow x_i + y_i)$



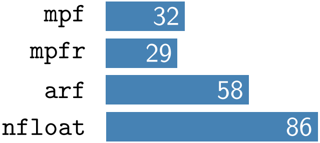
mul $(z_i \leftarrow x_i \cdot y_i)$



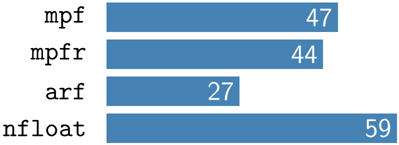
addmul_scalar $(z_i \leftarrow z_i + x_i \cdot y)$



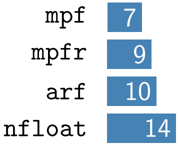
dot $(z \leftarrow \sum_i x_i y_i)$



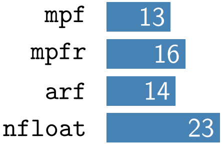
add $(z_i \leftarrow x_i + y_i)$



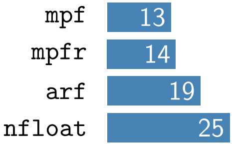
mul $(z_i \leftarrow x_i \cdot y_i)$



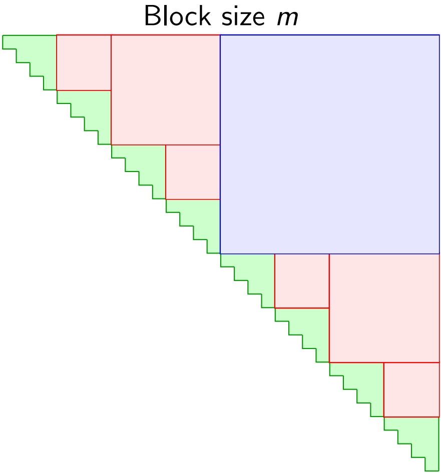
addmul_scalar $(z_i \leftarrow z_i + x_i \cdot y)$



dot $(z \leftarrow \sum_i x_i y_i)$



Block recursive linear algebra (e.g. LU factorization)



$m \gtrsim 10^2$

RNS matrix multiplication

$m \gtrsim 10^1$

Fixed-point matrix multiplication

- Faster additions
- $\lesssim 0.5m^3$ muls (Waksman+Strassen)

Direct floating-point arithmetic

RNS and fixed-point matrix multiplications need higher internal precision when entries are not uniformly scaled

Solving 100×100 system $Ax = b$

Time in seconds (ratio to ours)

Real

Precision	mpf	mpfr	arf	nfloat
128	0.0158 (5.9x)	0.0189 (7.1x)	0.00438 (1.6x)	0.00266
256	0.0178 (3.6x)	0.025 (5.0x)	0.0103 (2.1x)	0.00496
512	0.0256 (3.1x)	0.032 (3.9x)	0.0163 (2.0x)	0.00815
1024	0.0555 (2.9x)	0.0557 (2.9x)	0.0452 (2.4x)	0.0189
2048	0.149 (2.6x)	0.116 (2.0x)	0.100 (1.7x)	0.0578
4096	0.433 (2.3x)	0.345 (1.8x)	0.295 (1.6x)	0.187

Complex

Precision	mpc	acf	nfloat_complex
128	0.0778 (9.9x)	0.0161 (2.1x)	0.00785
256	0.100 (6.7x)	0.0365 (2.4x)	0.0149
512	0.135 (5.2x)	0.0607 (2.3x)	0.0260
1024	0.259 (4.3x)	0.176 (3.0x)	0.0596
2048	0.748 (4.0x)	0.392 (2.1x)	0.186
4096	1.70 (3.1x)	1.40 (2.5x)	0.556

Outlook

- Ball arithmetic
- Combining with SIMD
  - AVX-512 IFMA for multi-operand or multi-word multiplication
  - Conversion to fixed-point
  - Conversion to floating-point expansions
  - Conversion to RNS