Vector-friendly numbers with *n*-word precision

Fredrik Johansson

Inria Bordeaux

2025-07-17
30th Applications of Computer Algebra (ACA 2025)
Heraklion (Crete), Greece

Motivation

I will present ongoing work to speed up $\mathbb{R}$-vectors in FLINT.

We focus on "medium" precision, say 20 - 1000 digits.

Applications in computer algebra (e.g. differential equations):

- Direct numerical computation
- Arithmetic in algebraic structures over $\mathbb{R}$ like

$$\mathrm{Mat}_{d\times d}(\mathbb{C}[[x]]/\langle x^r\rangle) \quad \leftrightarrow \quad \mathbb{R}^{2rd^2}$$

Representing vectors of real numbers

$$x \in \mathbb{R} \qquad \rightarrow \qquad x \subseteq [m \pm r], \qquad m, r \in \mathbb{Z}[\tfrac{1}{2}]$$

- Vector of balls

$$[x_0, x_1, \ldots] \subseteq [[m_0 \pm r_0], [m_1 \pm r_1], \ldots]$$

- Vector with shared error bound

$$[x_0, x_1, \ldots] \subseteq [m_0, m_1, \ldots] \pm r$$

- Approximation (e.g. with posteriori bounds)

$$[x_0, x_1, \ldots] \approx [m_0, m_1, \ldots]$$

Flexible

Fast

Representing vectors of dyadic numbers

$$x = (-1)^{sgn} \cdot 2^{exp} \cdot [d_{n-1}\beta^{-1} + d_{n-2}\beta^{-2} + \ldots + d_0\beta^{-n}], \quad \beta = 2^{64}$$

- Variable precision floating-point

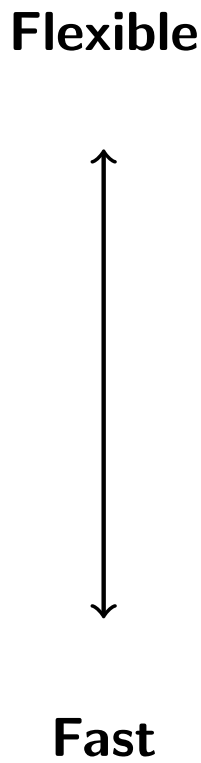
  $[x_0, x_1, \ldots]$, each x_i has its own precision n

- Uniform precision (shared n)

- Fixed-point (shared or implicit exponent)

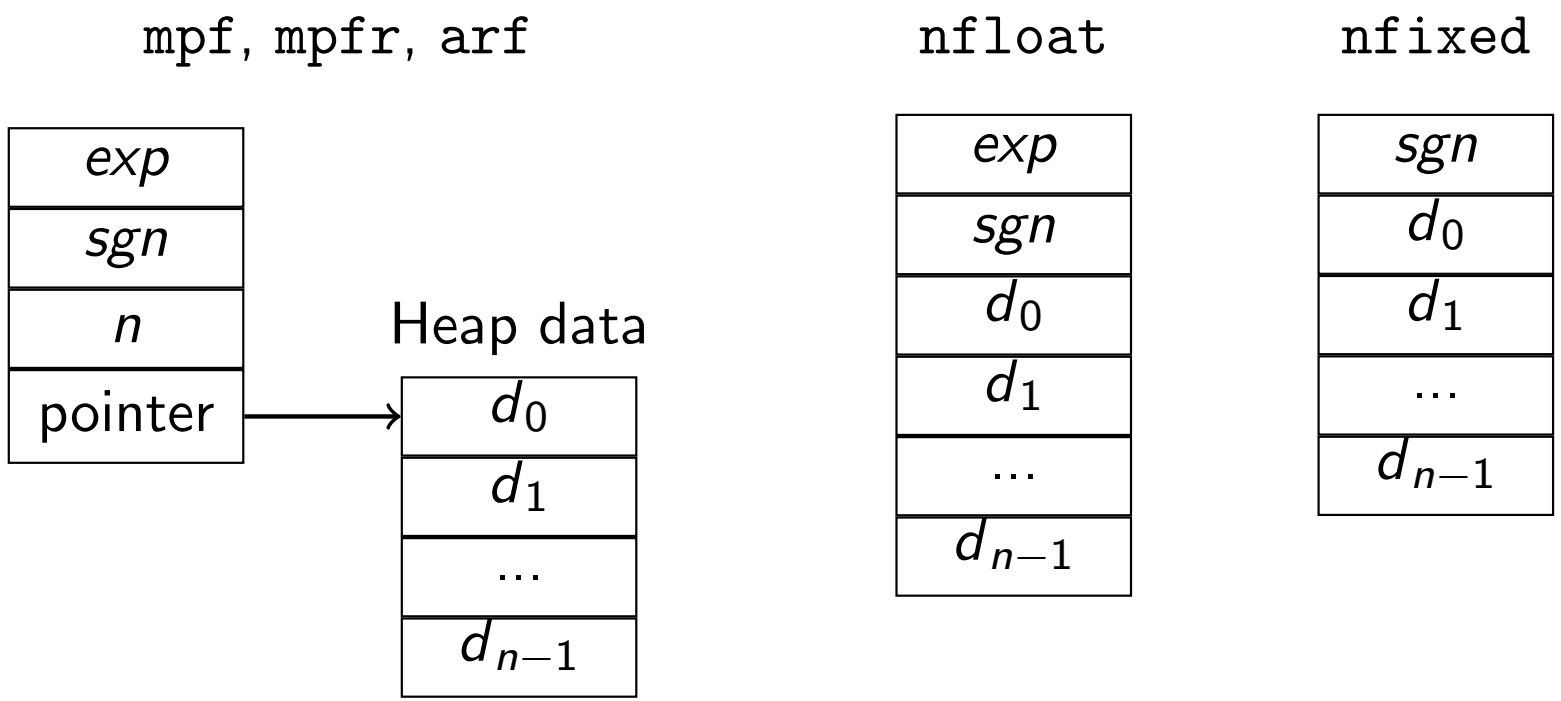
  $[x_0, x_1, \ldots] = 2^{exp}[\tilde{x}_0, \tilde{x}_1, \ldots]$, $\tilde{x}_i \in (-1, 1)$

- Redundant / carry-free encoding ($\beta < 2^{64}$)
- Multimodular encoding



Packing elements efficiently

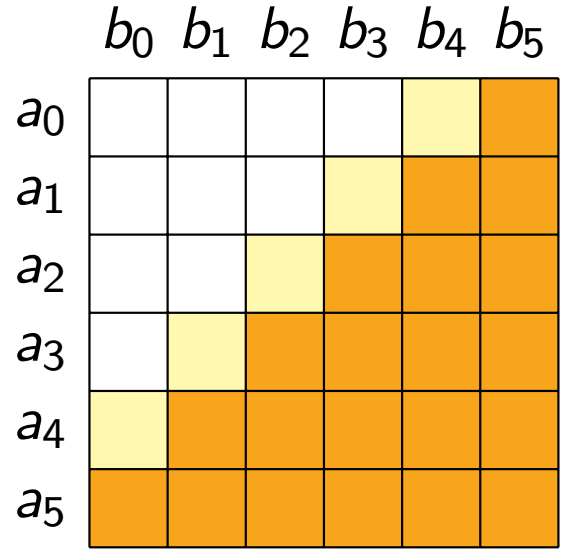
https://flintlib.org/doc/nfloat.html : alternative to `mpfr` or FLINT's `arf`, optimized for uniform precision vectors



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.

`nfixed` : low-level fixed-point arithmetic without overflow checking

High multiplication

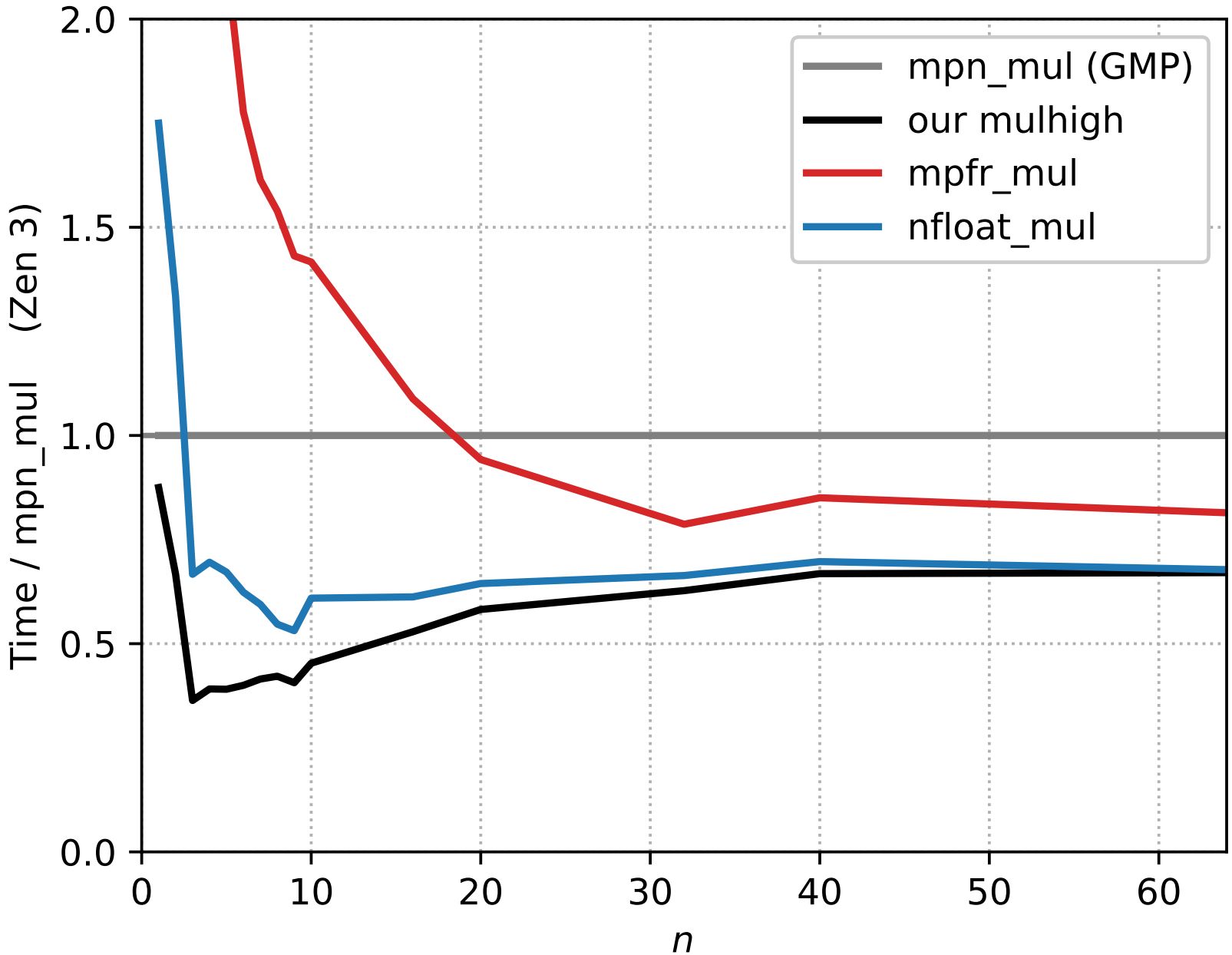


Terms to approximate the high n words of $(\sum_{i=0}^{n-1} a_i\beta^i) \cdot (\sum_{j=0}^{n-1} b_j\beta^j)$

Joint work with Albin Ahlbäck (ARITH '25)

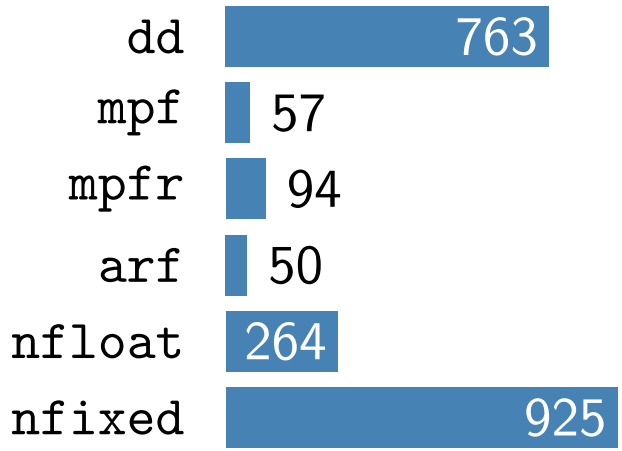
- $n(n+1)/2$ terms for $O(n)$ ulp error
- $n-1$ correction terms for $O(n/\beta)$ ulp error
- Hardcoded basecases for x86-64 and arm64 ($n \lesssim 9$)
- Mulders-style splitting for Karatsuba-size n

High multiplication

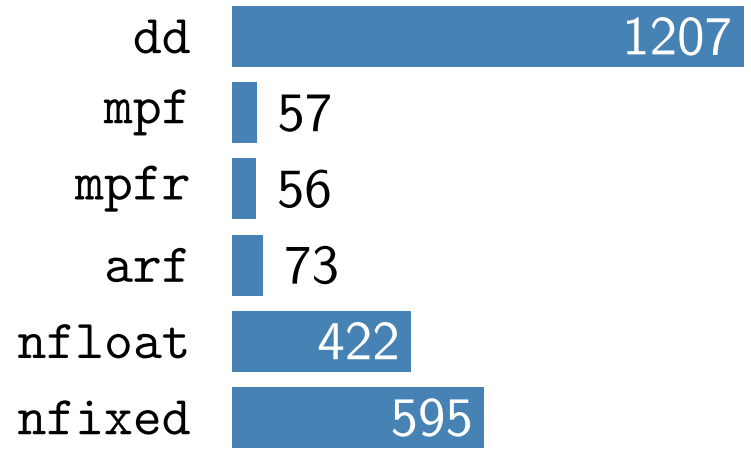


Mflop/s $n = 2$ 128-bit precision

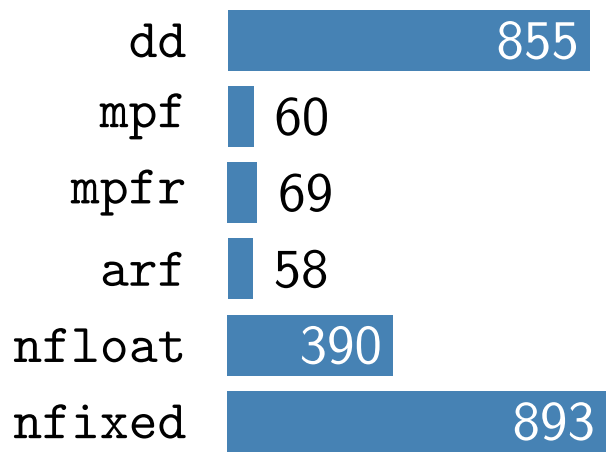
add $((x_i + y_i)_{i<100})$



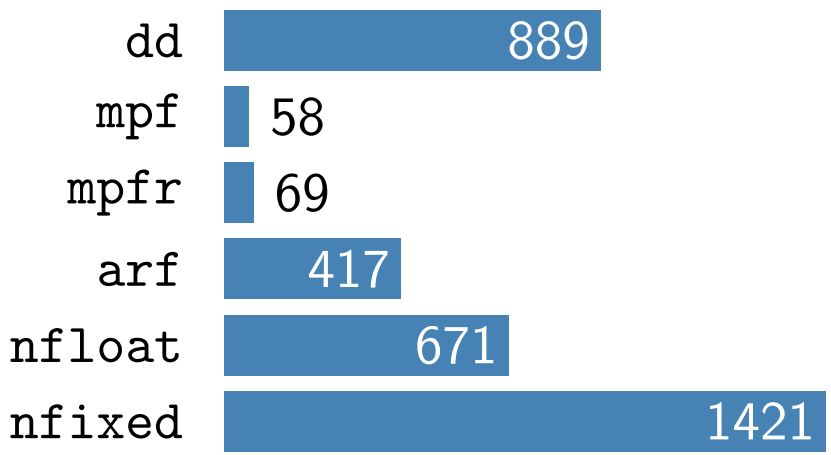
mul $((x_i \cdot y_i)_{i<100})$



fma $((y_i + x_i \cdot c)_{i<100})$

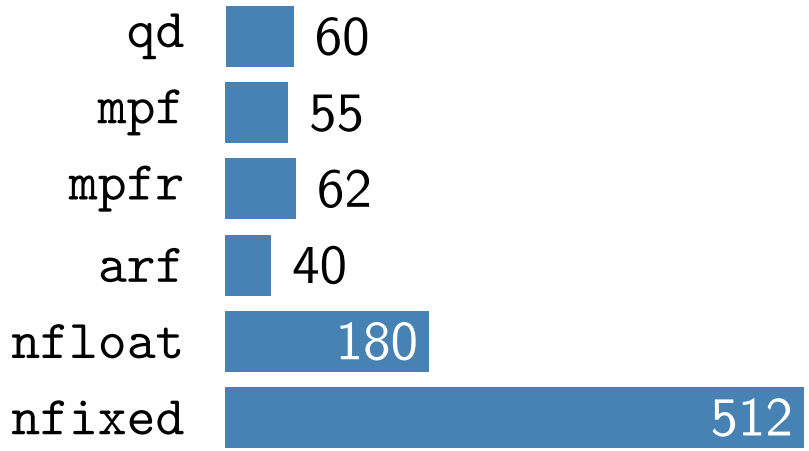


dot $(\sum_{i<100} x_i \cdot y_i)$

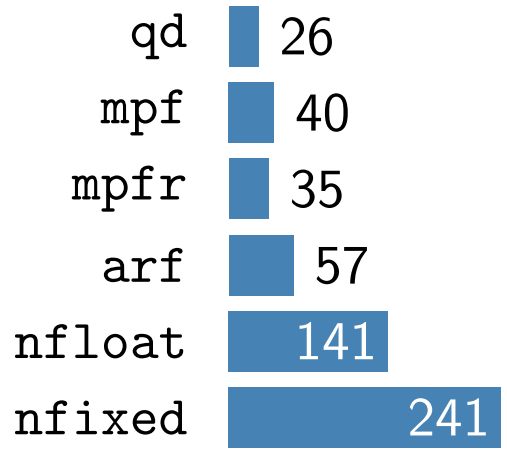


Mflop/s $n = 4$ 256-bit precision

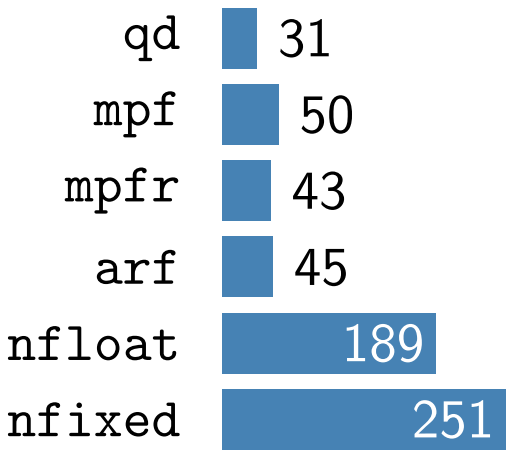
add $((x_i + y_i)_{i<100})$



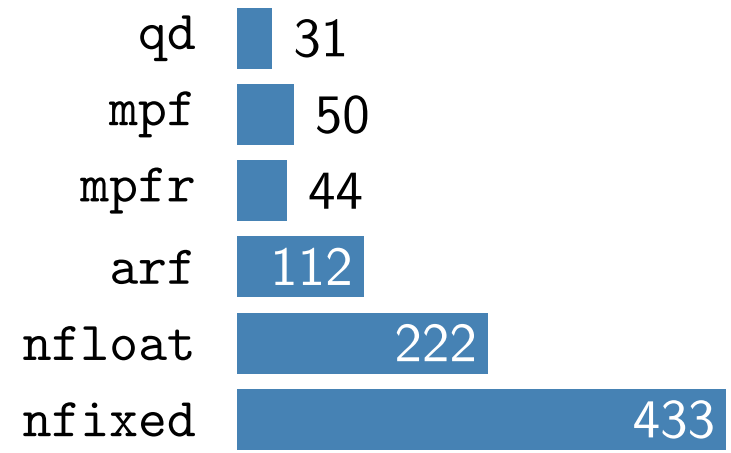
mul $((x_i \cdot y_i)_{i<100})$



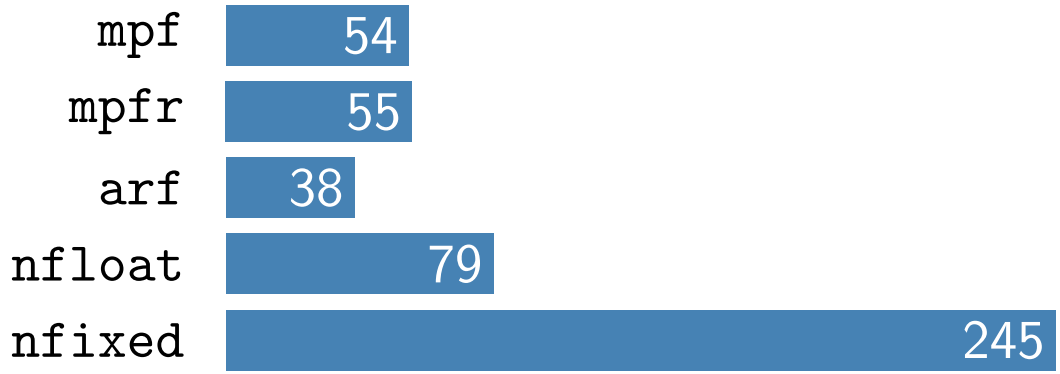
fma $((y_i + x_i \cdot c)_{i<100})$



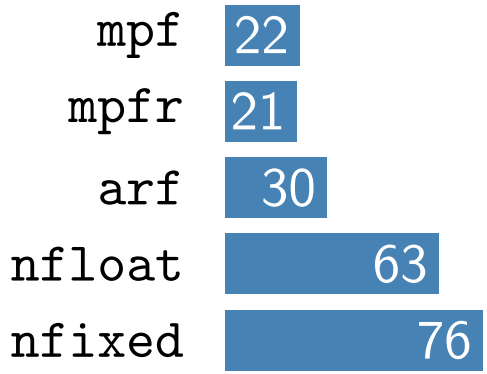
dot $(\sum_{i<100} x_i \cdot y_i)$



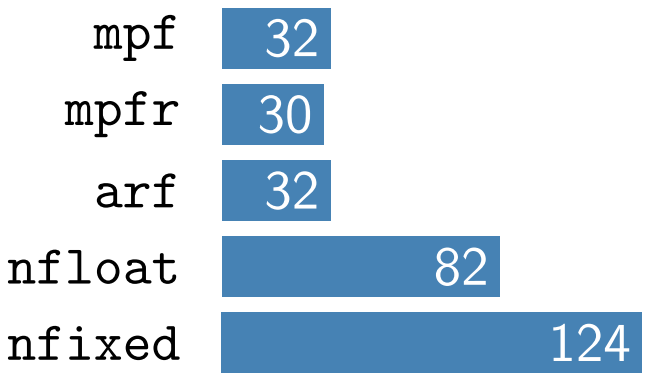
add $((x_i + y_i)_{i<100})$



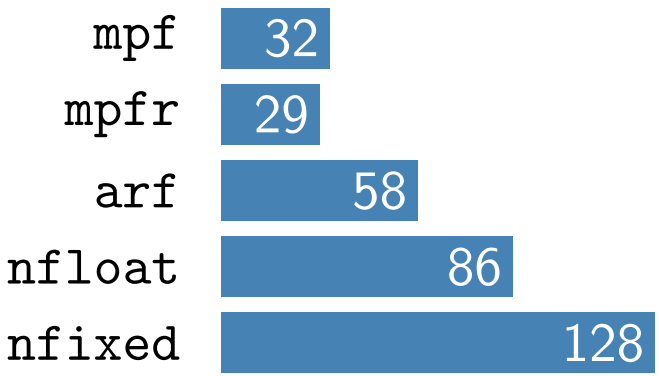
mul $((x_i \cdot y_i)_{i<100})$



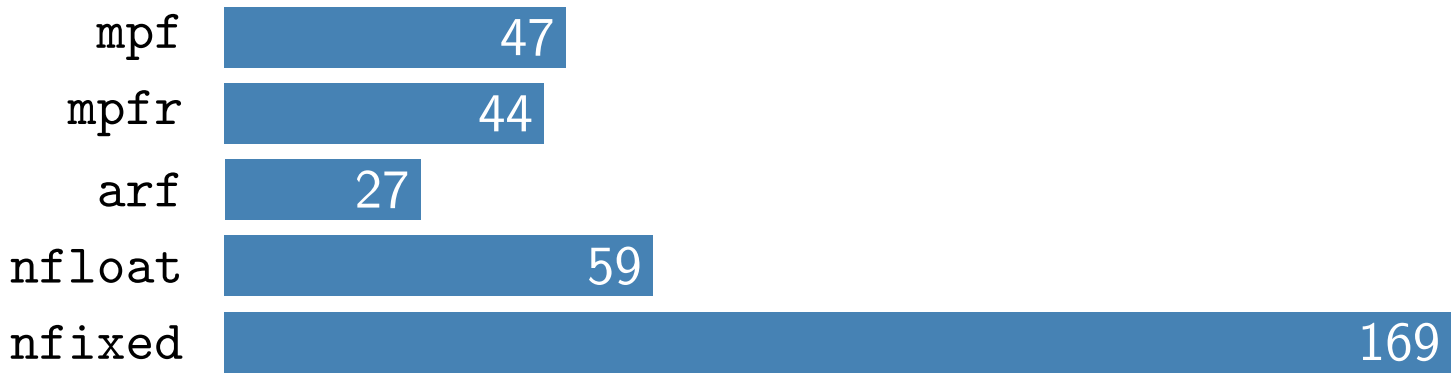
fma $((y_i + x_i \cdot c)_{i<100})$



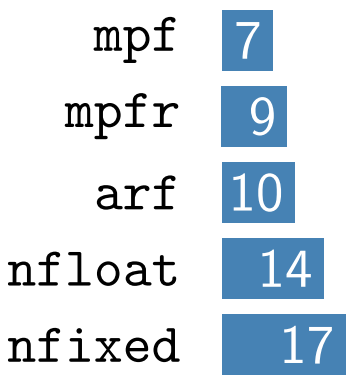
dot $(\sum_{i<100} x_i \cdot y_i)$



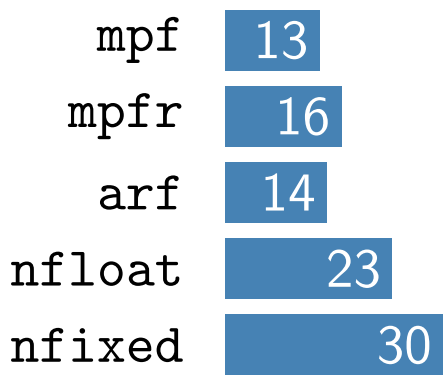
add $((x_i + y_i)_{i<100})$



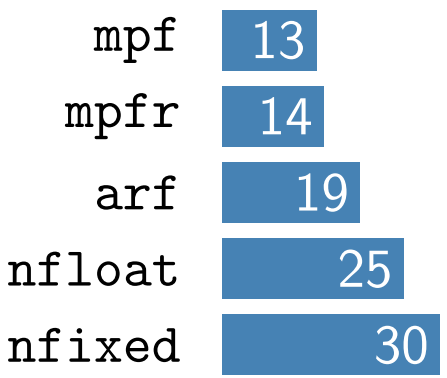
mul $((x_i \cdot y_i)_{i<100})$



fma $((y_i + x_i \cdot c)_{i<100})$

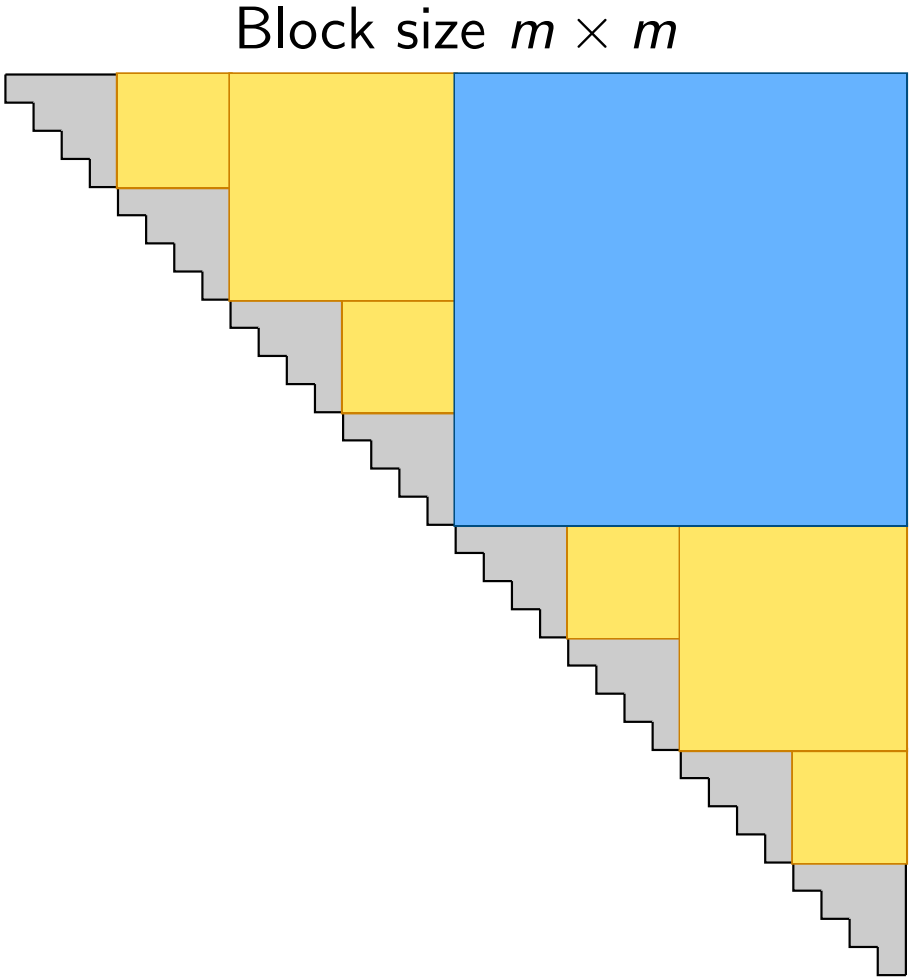


dot $(\sum_{i<100} x_i \cdot y_i)$



Block recursive linear algebra for `nfloat`

Typical recursion (e.g. for LU factorization):



$m \gtrsim 10^2$

Multimodular matrix multiplication

$m \gtrsim 10^1$

Fixed-point matrix multiplication

Direct floating-point arithmetic

Emulating floating-point using fixed-point arithmetic

If we want the entrywise error in AB to satisfy

$$|\varepsilon_{ij}| \lesssim (|A||B|)_{ij} \cdot 2^{-p}, \tag{1}$$

we can use $(p + \Delta + O(\log m))$-bit fixed-point arithmetic if

$$2^e \le |A_{ij}| \le 2^{e+\Delta}, \quad 2^f \le |B_{ij}| \le 2^{f+\Delta}, \quad \text{all } i, j.$$

Can we do better when Δ is large (e.g. $\Delta > 0.1p$)?

- Rescaling by row and column

$$AB = S^{-1}(SAS')(S'^{-1}BS'')S''^{-1}, \quad S, S', S'' = \begin{pmatrix} 2^{\square} & & \\ & \ddots & \\ & & 2^{\square} \end{pmatrix}$$

- Splitting A, B into smaller blocks
- Opportunistic evaluation + backtracking
- Choosing applications with weaker requirements than (1)

Fast fixed-point matrix multiplication

- Classical matrix multiplication

  m^2 dot products $= m^3$ multiplications $+ m^3$ additions

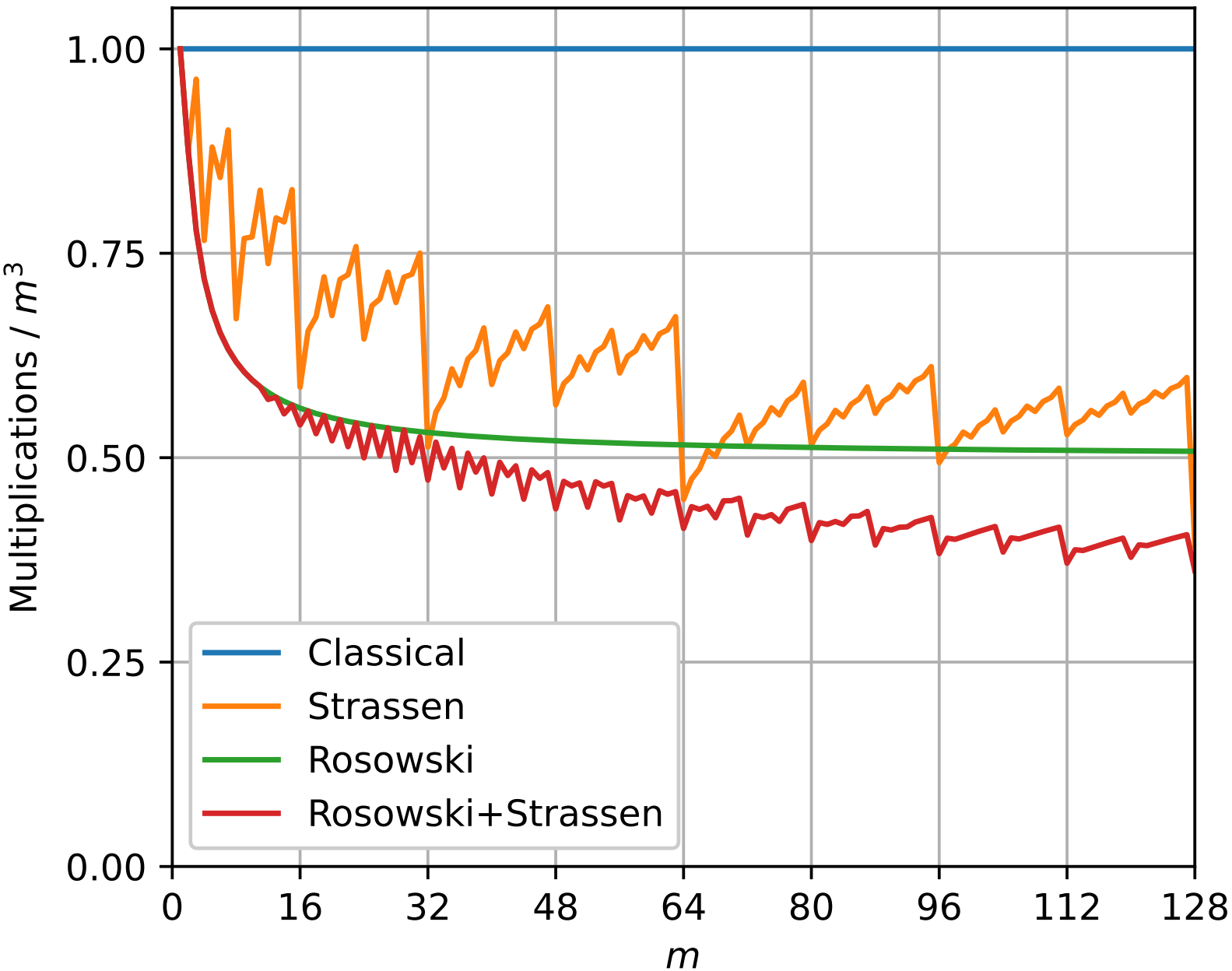
- Strassen: $O(m^{2.81})$ multiplications and additions

- Winograd's commutative algorithm (later refined by Waksman and Rosowski):

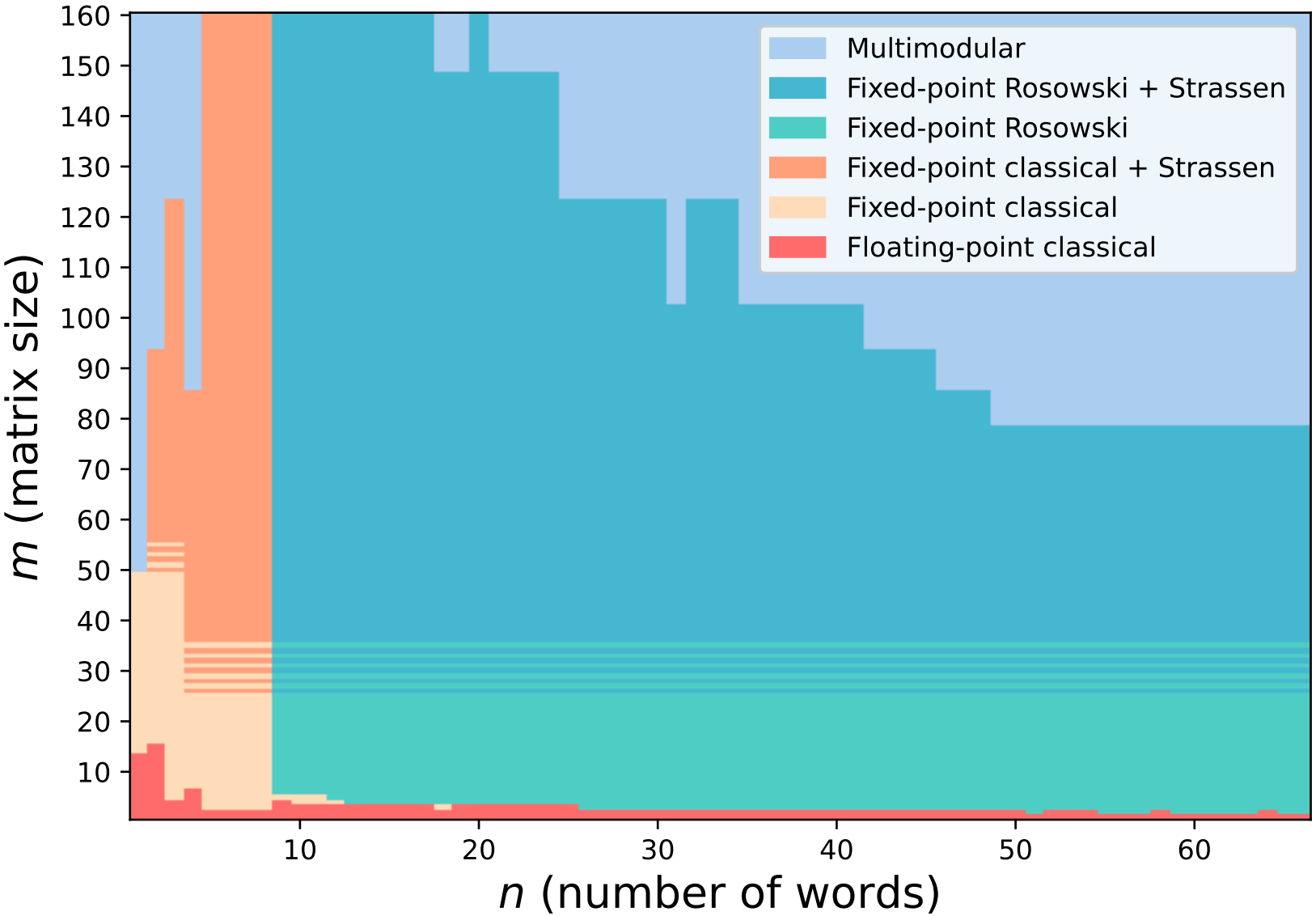
$$\underbrace{(a_1 + b_2)(a_2 + b_1)}_{\text{One multiplication}} = \underbrace{[a_1 b_1 + a_2 b_2]}_{\text{Two dot product terms}} + \underbrace{\{a_1 a_2 + b_1 b_2\}}_{\textit{Correction}}$$

$$\sim 0.5m^3 \text{ multiplications} + \sim 1.5m^3 \text{ additions}$$

Multiplication count in matrix multiplication



Algorithm selection for `nfloat` matrix multiplication



Solving a 100×100 system $Ax = b$

Time in seconds

Real

Bits	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

Bits	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
- Various algebraic structures
- Code generation, compilation (e.g. for straight-line programs)
- SIMD optimization