Faster arbitrary-precision dot product and matrix multiplication

Fredrik Johansson

Inria Bordeaux

26th IEEE Symposium on Computer Arithmetic (ARITH 26)

Kyoto, Japan

June 10, 2019

Arbitrary-precision arithmetic

Precision: $p \geq 2$ bits (can be thousands or millions)

- Floating-point numbers

$$3.14159265358979323846264338328$$

- Ball arithmetic (mid-rad interval arithmetic)

$$[3.14159265358979323846264338328 \pm 8.65 \cdot 10^{-31}]$$

Why?

- Computational number theory, computer algebra
- Dynamical systems, ill-conditioned problems
- Verifying/testing numerical results/methods

This work: faster arithmetic and linear algebra

CPU time (seconds) to multiply two real 1000×1000 matrices

	$p = 53$	$p = 106$	$p = 212$	$p = 848$
BLAS	0.08			
QD		11	111	
MPFR	36	44	110	293
Arb* (classical)	19	25	76	258
Arb* (block)	3.6	5.6	8.2	27

* With ball coefficients
Arb version 2.16 – `http://arblib.org`

Two important requirements

- True arbitrary precision; inputs and output can have mixed precision; no restrictions on the exponents

- Preserve structure: near-optimal enclosures *for each entry*

$$\begin{pmatrix} [1.23\cdot 10^{100} \pm 10^{80}] & -1.5 & 0 \\ 1 & [2.34 \pm 10^{-20}] & [3.45 \pm 10^{-50}] \\ 0 & 2 & [4.56 \cdot 10^{-100} \pm 10^{-130}] \end{pmatrix}$$

Dot product

$$\sum_{k=1}^{N} a_k b_k, \qquad a_k, b_k \in \mathbb{R} \text{ or } \mathbb{C}$$

Kernel in basecase ($N \lesssim 10$ to 100) algorithms for:

- Matrix multiplication
- Triangular solving, recursive LU factorization
- Polynomial multiplication, division, composition
- Power series operations

Dot product as an atomic operation

The old way:

```
arb_mul(s, a, b, prec);
for (k = 1; k < N; k++)
    arb_addmul(s, a + k, b + k, prec);
```

The new way:

```
arb_dot(s, NULL, 0, a, 1, b, 1, N, prec);
```

(More generally, computes $s = s_0 + (-1)^c \sum_{k=0}^{N-1} a_{k \cdot \mathrm{astep}} b_{k \cdot \mathrm{bstep}}$)

`arb_dot` – ball arithmetic, real
`acb_dot` – ball arithmetic, complex
`arb_approx_dot` – floating-point, real
`acb_approx_dot` – floating-point, complex

Numerical dot product

Approximate (floating-point) dot product:

$$s = \sum_{k=1}^{N} a_k b_k + \varepsilon, \qquad |\varepsilon| \approx 2^{-p} \sum_{k=1}^{N} |a_k b_k|$$

Ball arithmetic dot product:

$$[m \pm r] \supseteq \sum_{k=1}^{N} [m_k \pm r_k][m'_k \pm r'_k]$$

$$m = \sum_{k=1}^{N} m_k m'_k + \varepsilon, \qquad r \geq |\varepsilon| + \sum_{k=1}^{N} |m_k| r'_k + |m'_k| r_k + r_k r'_k$$

Representation of numbers in Arb (like MPFR)

Arbitrary-precision floating-point numbers:

$$(-1)^{\text{sign}} \cdot 2^{\text{exp}} \cdot \sum_{k=0}^{n-1} b_k 2^{64(k-n)}$$

Limbs b_k are 64-bit words, normalized:

$$0 \le b_k < 2^{64}, \quad b_{n-1} \ge 2^{63}, \quad b_0 \ne 0$$

All core arithmetic operations are implemented using word manipulations and low-level GMP (`mpn` layer) function calls

---

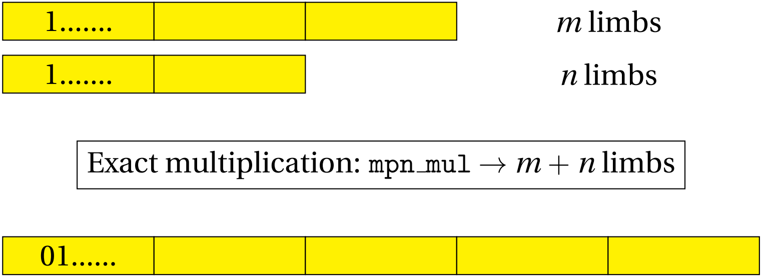
Radius: 30-bit unsigned floating-point

Arbitrary-precision multiplication

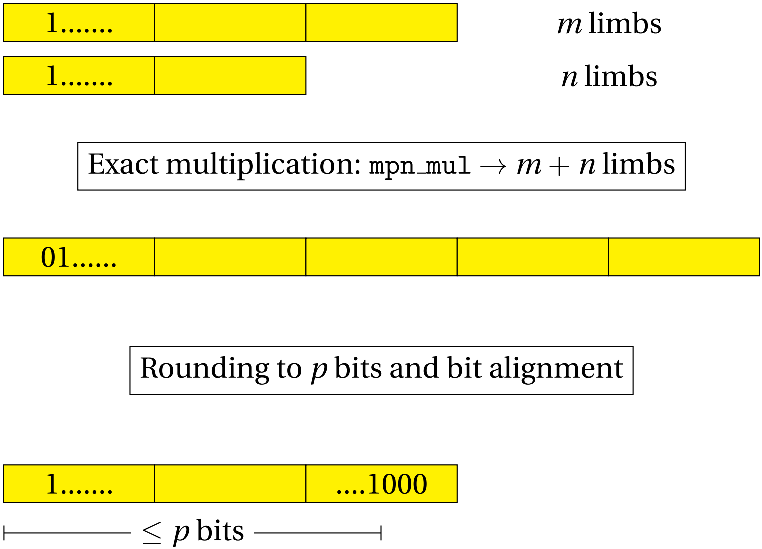
m limbs

n limbs

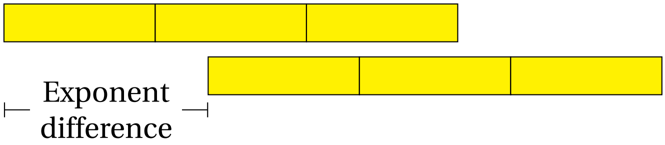
Arbitrary-precision multiplication



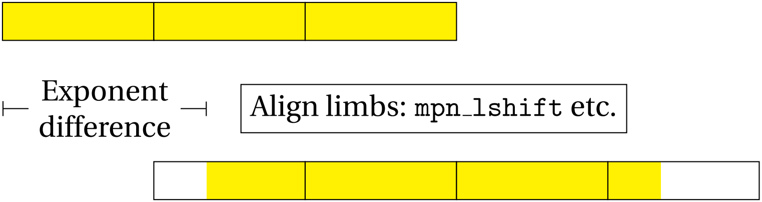
Arbitrary-precision multiplication



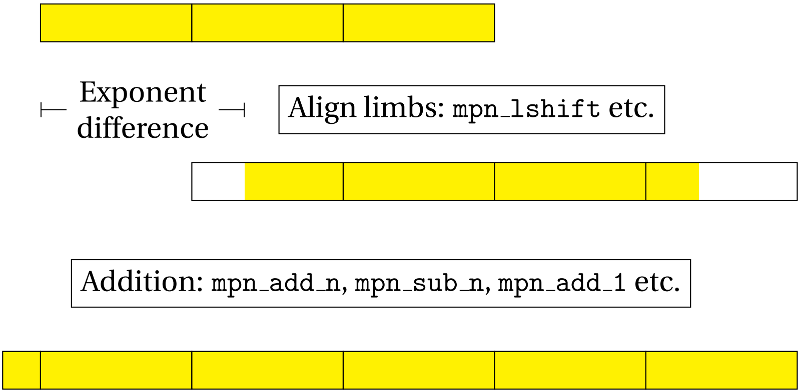
Arbitrary-precision addition



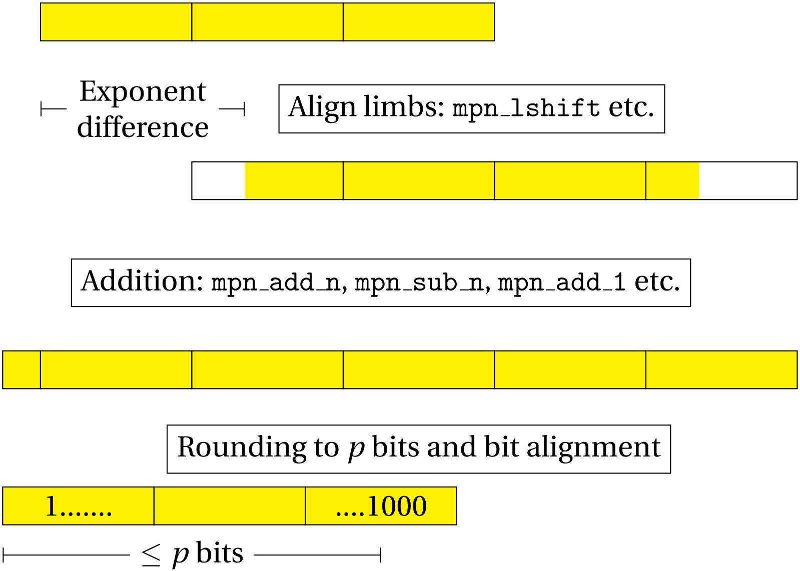
Arbitrary-precision addition



Arbitrary-precision addition



Arbitrary-precision addition



Dot product

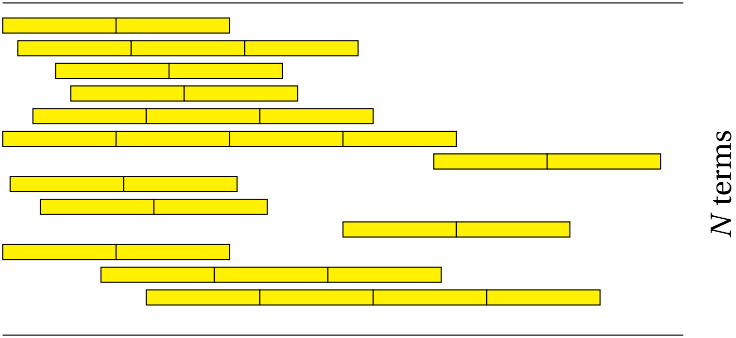
First pass: inspect the terms

- Count nonzero terms
- Bound upper and lower exponents of terms
- Detect Inf/NaN/overflow/underflow (fallback code)

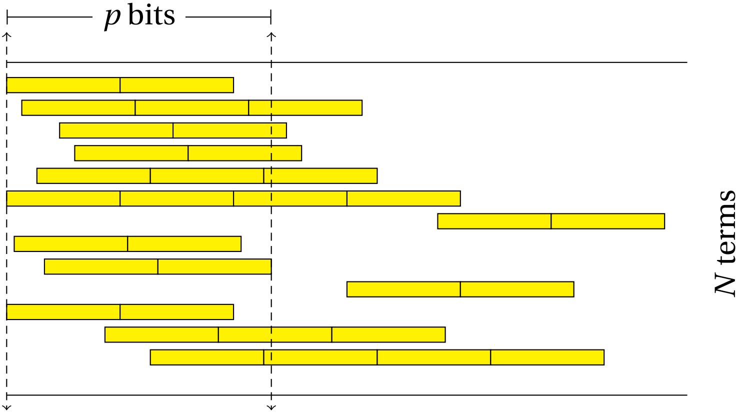
Second pass: compute the dot product!

- Exploit knowledge about exponents
- Single temporary memory allocation
- Single final rounding and normalization

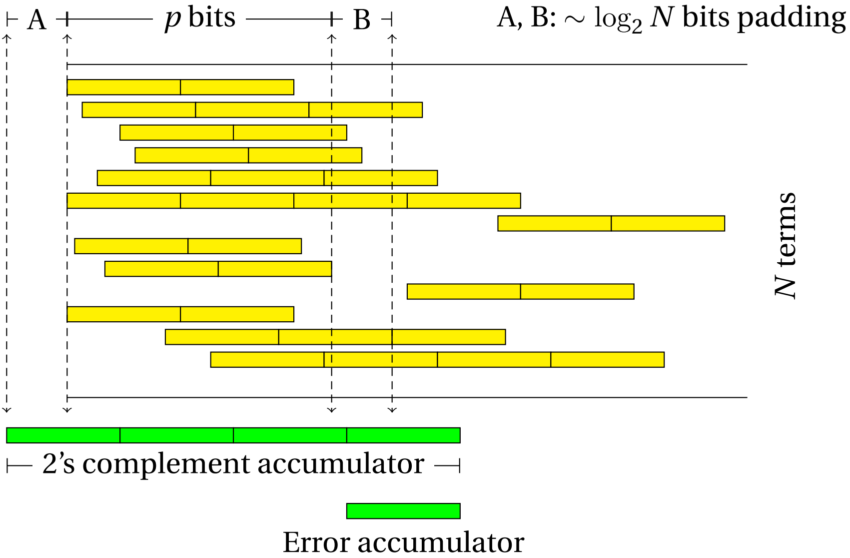
Dot product



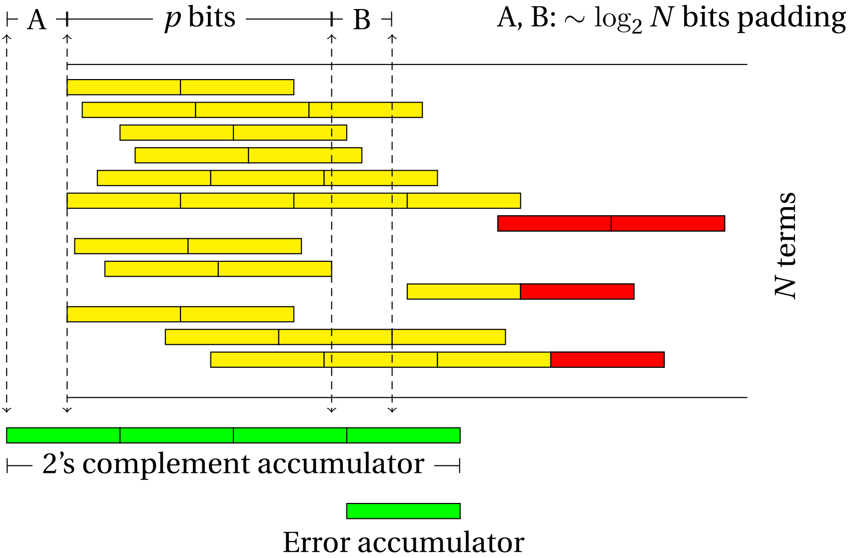
Dot product



Dot product


A
p bits
B
A, B: $\sim \log_2 N$ bits padding
N terms
2's complement accumulator
Error accumulator

Dot product


A
p bits
B
A, B: ∼ log2 N bits padding
N terms
2's complement accumulator
Error accumulator

Technical comments

Radius dot products (for ball arithmetic):

- Dedicated code using 64-bit accumulator

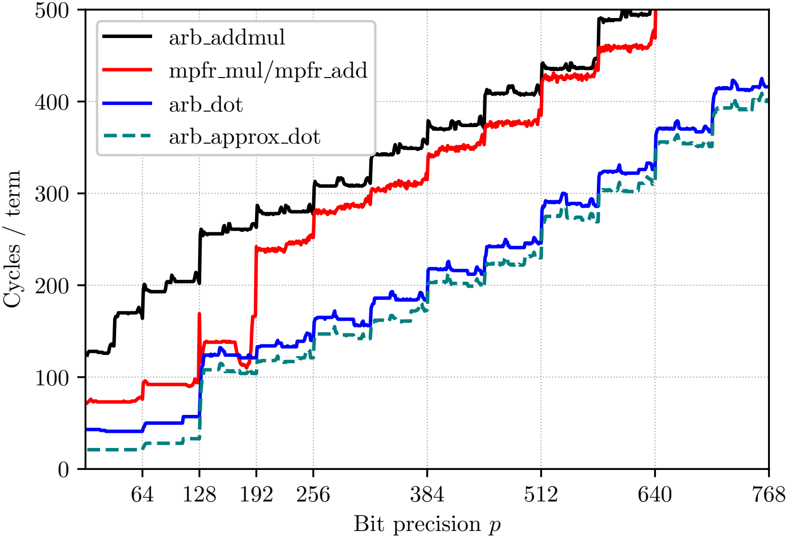
Special sizes:

- Inline ASM instead of GMP function calls for $\leq 2 \times 2$ limb product, ≤ 3 limb accumulator
- Mulder's mulhigh (via MPFR) for 25 to 10000 limbs

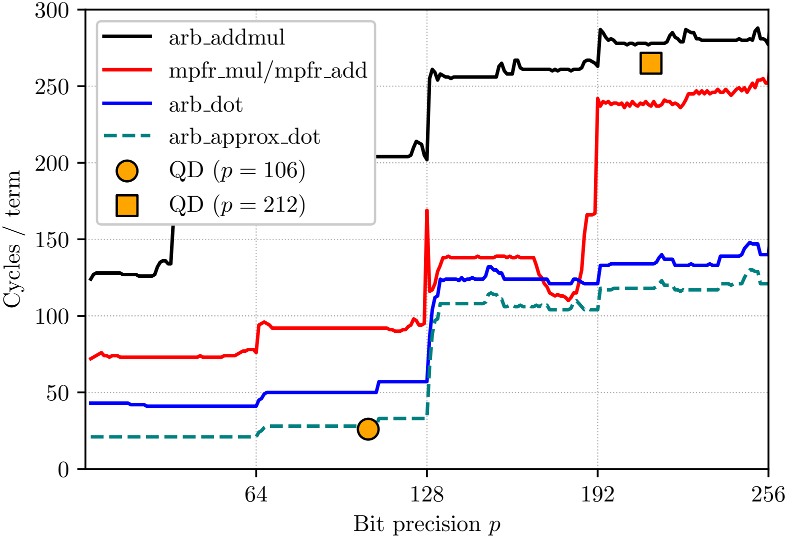
Complex numbers:

- Essentially done as two length-$2N$ real dot products
- Karatsuba-style multiplication (3 instead of 4 real muls) for ≥ 128 limbs

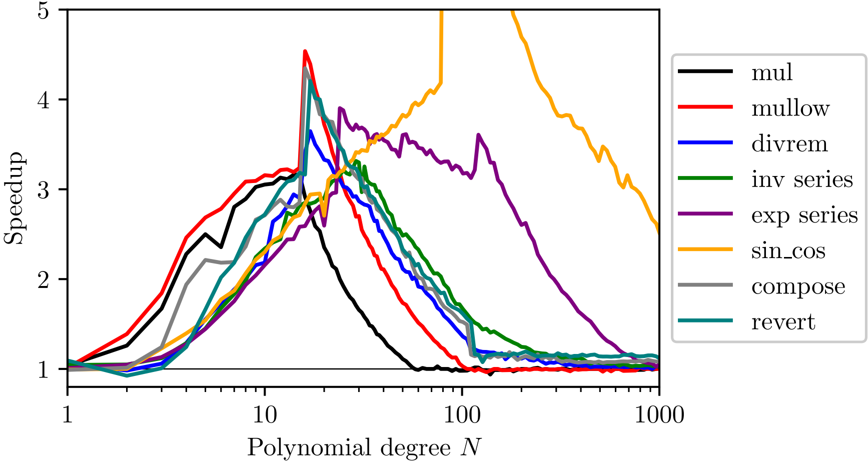
Dot product performance



Dot product performance

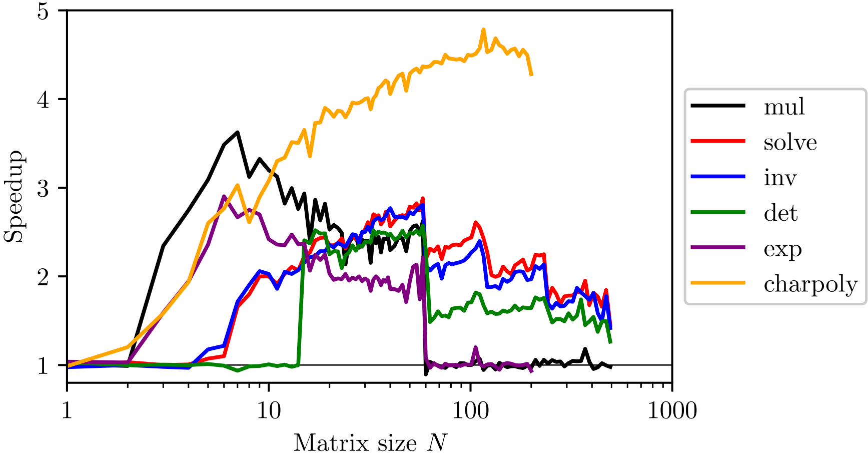


Dot product: polynomial operations speedup in Arb



(Complex coefficients, $p = 64$ bits)

Dot product: matrix operations speedup in Arb



(Complex coefficients, $p = 64$ bits)

Matrix multiplication (large N)

Same ideas as polynomial multiplication in Arb:

1. $[A \pm a][B \pm b]$ via three multiplications AB, $|A|b$, $a(|B|+b)$
2. Split + scale matrices into blocks with uniform magnitude
3. Multiply blocks of A, B exactly over $\mathbb{Z}$ using FLINT
4. Multiply blocks of $|A|$, b, a, $|B|+b$ using hardware FP

Matrix multiplication (large N)

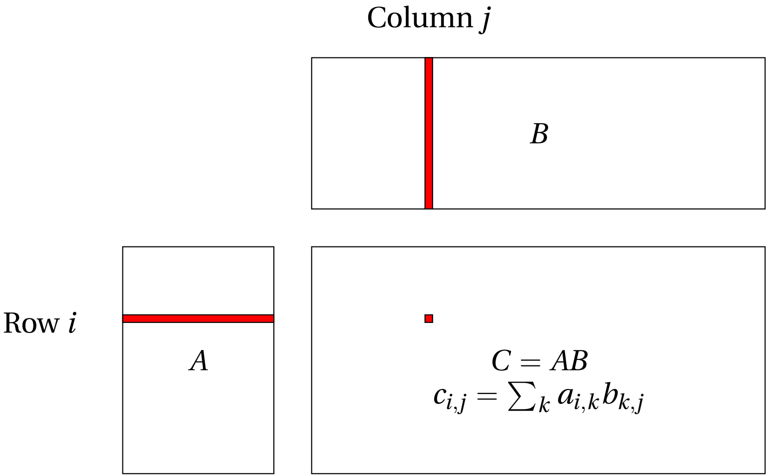
Same ideas as polynomial multiplication in Arb:

1. $[A \pm a][B \pm b]$ via three multiplications AB, $|A|b$, $a(|B|+b)$
2. Split + scale matrices into blocks with uniform magnitude
3. Multiply blocks of A, B exactly over $\mathbb{Z}$ using FLINT
4. Multiply blocks of $|A|$, b, a, $|B|+b$ using hardware FP

Where is the gain?

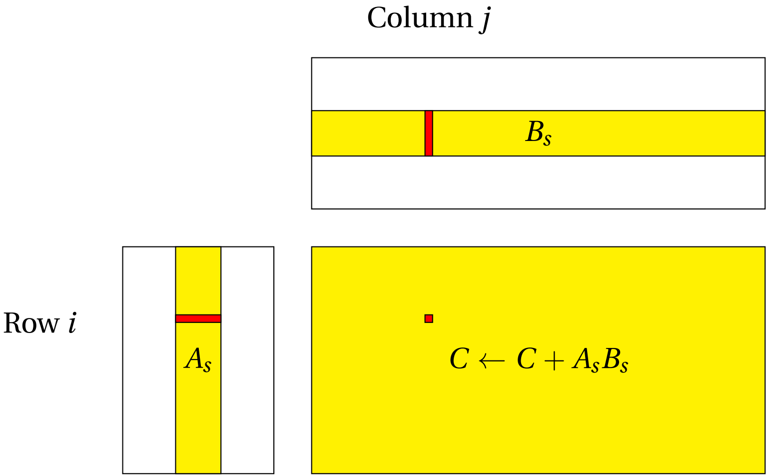
- Integers and hardware FP have less overhead
- Multimodular/RNS arithmetic (60-bit primes in FLINT)
- Strassen $O(N^{2.81})$ matrix multiplication in FLINT

Matrix multiplication



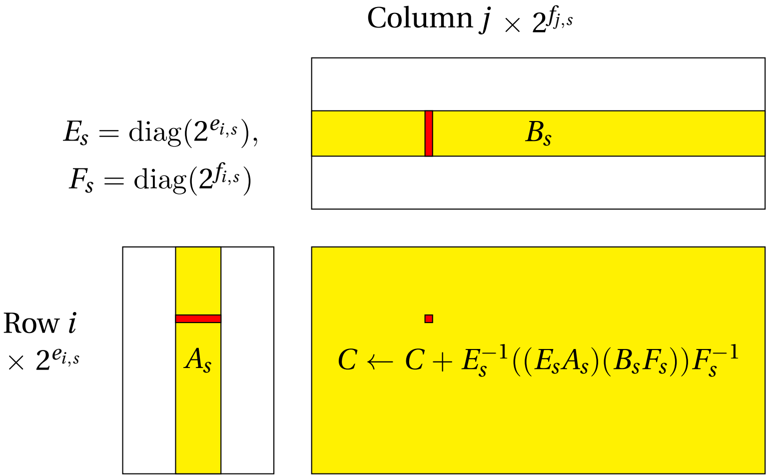
Block matrix multiplication

Choose blocks A_s, B_s (indices $s \subseteq \{1, \ldots, N\}$) so that each row of A_s and column of B_s has a small internal exponent range



Block matrix multiplication, scaled to integers

Scaling is applied internally to each block A_s, B_s



Uniform and non-uniform matrices

Uniform matrix, $N = 1000$

p	Classical	Block	Number of blocks	Speedup
53	19 s	3.6 s	1	5.3
212	76 s	8.2 s	1	9.3
3392	1785 s	115 s	1	15.5

Pascal matrix, $N = 1000$ (entries $A_{i,j} = \pi \cdot \binom{i+j}{j}$)

p	Classical	Block	Number of blocks	Speedup
53	12 s	20 s	10	0.6
212	43 s	35 s	9	1.2
3392	1280 s	226 s	2	5.7

Approximate and certified linear algebra

Three approaches to linear solving $Ax = b$:

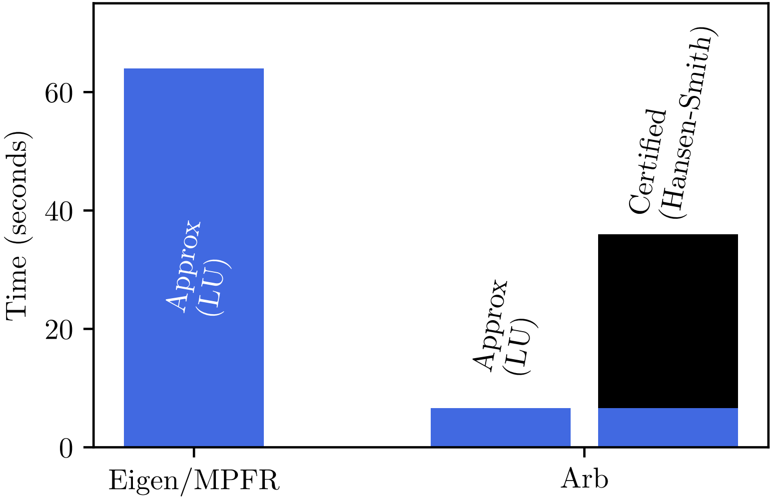
- Gaussian elimination in floating-point arithmetic:
  stable if A is well-conditioned
- Gaussian elimination in interval/ball arithmetic:
  unstable for generic well-conditioned A (lose $O(N)$ digits)
- Approx + certification: $3.141 \to [3.141 \pm 0.001]$

Example: Hansen-Smith algorithm

1. Compute $R \approx A^{-1}$ approximately
2. Solve $(RA)x = Rb$ in interval/ball arithmetic

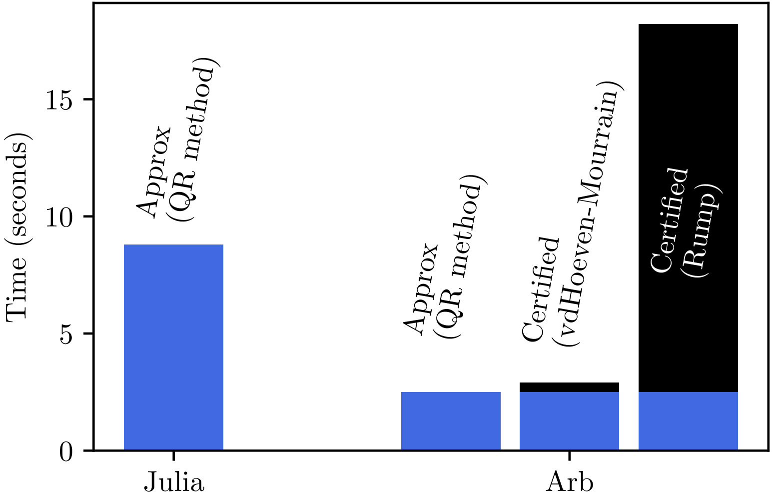
Linear solving

Solving a dense real linear system $Ax = b$ ($N = 1000$, $p = 212$)



Eigenvalues

Computing all eigenvalues and eigenvectors of a nonsymmetric complex matrix ($N = 100$, $p = 128$)



Conclusion

Faster arbitrary-precision arithmetic, linear algebra

- Handle dot product as an atomic operation, use instead of single add/muls where possible ($1 - 5\times$ speedup)
- Accurate and fast large-N matrix multiplication using scaled integer blocks ($\approx 10\times$ speedup)
- Higher operations reduce well to dot product (small N), matrix multiplication (large N)

Future work ideas

- Correctly rounded dot product, for MPFR (easy)
- Horner scheme (in analogy with dot product)
- Better matrix scaling + splitting algorithm