Faster dot product and matrix multiplication in arbitrary-precision ball arithmetic

Fredrik Johansson
LFANT – Inria Bordeaux

MPFR / MPC / iRRAM Workshop
University of Trier, Germany
2018-11-21

Hardware floating-point is pretty fast

```
julia> N = 1000;
julia> A = transpose(reshape([sin(k^2)
             for k in 0:N^2-1], (N, N)));
julia> B = ones(N,1);

julia> @time (A \ B)[1,1]
  0.031291 seconds (10 allocations: 7.645 MiB)
133.97836679402695
```

Doubling the precision with `BigFloat` (uses MPFR)

```
julia> setprecision(106);
julia> N = 1000;
julia> A = transpose(reshape([sin(BigFloat(k)^2)
             for k in 0:N^2-1], (N, N)));
julia> B = ones(BigFloat,N,1);

julia> @time (A \ B)[1,1]
372.169355 seconds (1.34 G allocations: 59.791 GiB,
    77.46% gc time)
1.339783667939566095828328309516 45e+02
```

That's 10 000 times slower!

Doubling the precision with `BigFloat` (uses MPFR)

```
julia> setprecision(106);
julia> N = 1000;
julia> A = transpose(reshape([sin(BigFloat(k)^2)
             for k in 0:N^2-1], (N, N)));
julia> B = ones(BigFloat,N,1);

julia> @time (A \ B)[1,1]
372.169355 seconds (1.34 G allocations: 59.791 GiB,
    77.46% gc time)
1.33978366793956609582832830951645e+02
```

That's 10 000 times slower!
mpmath takes an hour – 100 000 times slower!

Arb 2.15 (here via Python-FLINT):

```
>>> from flint import *; from mpmath import timing
>>> ctx.prec = 106
>>> N = 1000
>>> A = arb_mat(N,N,[arb(k**2).sin() for k in range(N*N)])
>>> B = arb_mat(N,1,[1]*N)

>>> t=timing(lambda: print(A.solve(B)[0,0]))
[133.97836679395660958283283 +/- 2.12e-24]
>>> print("%.2g seconds" % t)
21.8 seconds
```

Arb 2.15 (here via Python-FLINT):

```
>>> from flint import *; from mpmath import timing
>>> ctx.prec = 106
>>> N = 1000
>>> A = arb_mat(N,N,[arb(k**2).sin() for k in range(N*N)])
>>> B = arb_mat(N,1,[1]*N)

>>> t=timing(lambda: print(A.solve(B)[0,0]))
[133.97836679395660958283283 +/- 2.12e-24]
>>> print("%.2g seconds" % t)
21.8 seconds
```

Without a rigorous enclosure:

```
>>> t=timing(lambda: print(A.solve(B,algorithm="approx")[0,0]))
[133.97836679395660958328310364 +/- 4.07e-29]
>>> print("%.3g seconds" % t)
3.98 seconds
```

Some requirements for Arb arithmetic

- Rigorous error bounds (+optional nonrigorous versions)
- Both real and complex arithmetic
- True arbitrary precision; inputs and output can have mixed precision; no restrictions on the exponent range
- Multiplying polynomials or matrices gives near-optimal enclosures (like the schoolbook algorithm) *for each entry*, preserving zeros, exact entries, small entries, ...:

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

Dot product

$$\sum_{k=0}^{N-1} a_k b_k, \qquad a_k, b_k \in \mathbb{R} \text{ or } \mathbb{C}$$

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

- Matrix multiplication
- Triangular solving, recursive LU factorization
- Polynomial multiplication, division
- Power series arithmetic, transcendental functions

Dot product in ball arithmetic

For each term:

$$[s \pm s_r] \leftarrow [s \pm s_r] + [a \pm a_r] \cdot [b \pm b_r]$$

Midpoint: $\mathrm{round}(s + a \cdot b)$

- 1 arbitrary-precision multiply-add

Radius: $s_r + |a| b_r + |b| a_r + a_r b_r + \varepsilon_{\text{round}}$

- In Arb, radii have 30-bit precision
- 2 conversions to low-precision bounds $|a|$, $|b|$
- 3 low-precision multiply-adds, 1 low-precision add

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 $s = s_0 + (-1)^c \sum_{k=0}^{N-1} a_{k \cdot \text{astep}} b_{k \cdot \text{bstep}}$)

`arb_dot`, `acb_dot`, `arb_approx_dot`, `acb_approx_dot`

Dot product performance

Cycles/term to compute $\sum_k a_k b_k$ on an i5-4300U (Haswell)

		53 bits (1 word)	106 bits (2 words)	212 bits (4 words)
`double`	`s += a[k] * b[k]`	2		
	Vectorized	1/4?		

Dot product performance

Cycles/term to compute $\sum_k a_k b_k$ on an i5-4300U (Haswell)

		53 bits (1 word)	106 bits (2 words)	212 bits (4 words)
`double`	`s += a[k] * b[k]`	2		
	Vectorized	1/4?		
QD 2.3	`s += a[k] * b[k]`		26	279

Dot product performance

Cycles/term to compute $\sum_k a_k b_k$ on an i5-4300U (Haswell)

		53 bits (1 word)	106 bits (2 words)	212 bits (4 words)
`double`	`s += a[k] * b[k]`	2		
	Vectorized	1/4?		
QD 2.3	`s += a[k] * b[k]`		26	279
MPFR 3.1	`mpfr_mul/mpfr_add`	121	328	403
MPFR 4.0	`mpfr_mul/mpfr_add`	73	91	408

[Lefèvre & Zimmermann, ARITH 2017]

Dot product performance

Cycles/term to compute $\sum_k a_k b_k$ on an i5-4300U (Haswell)

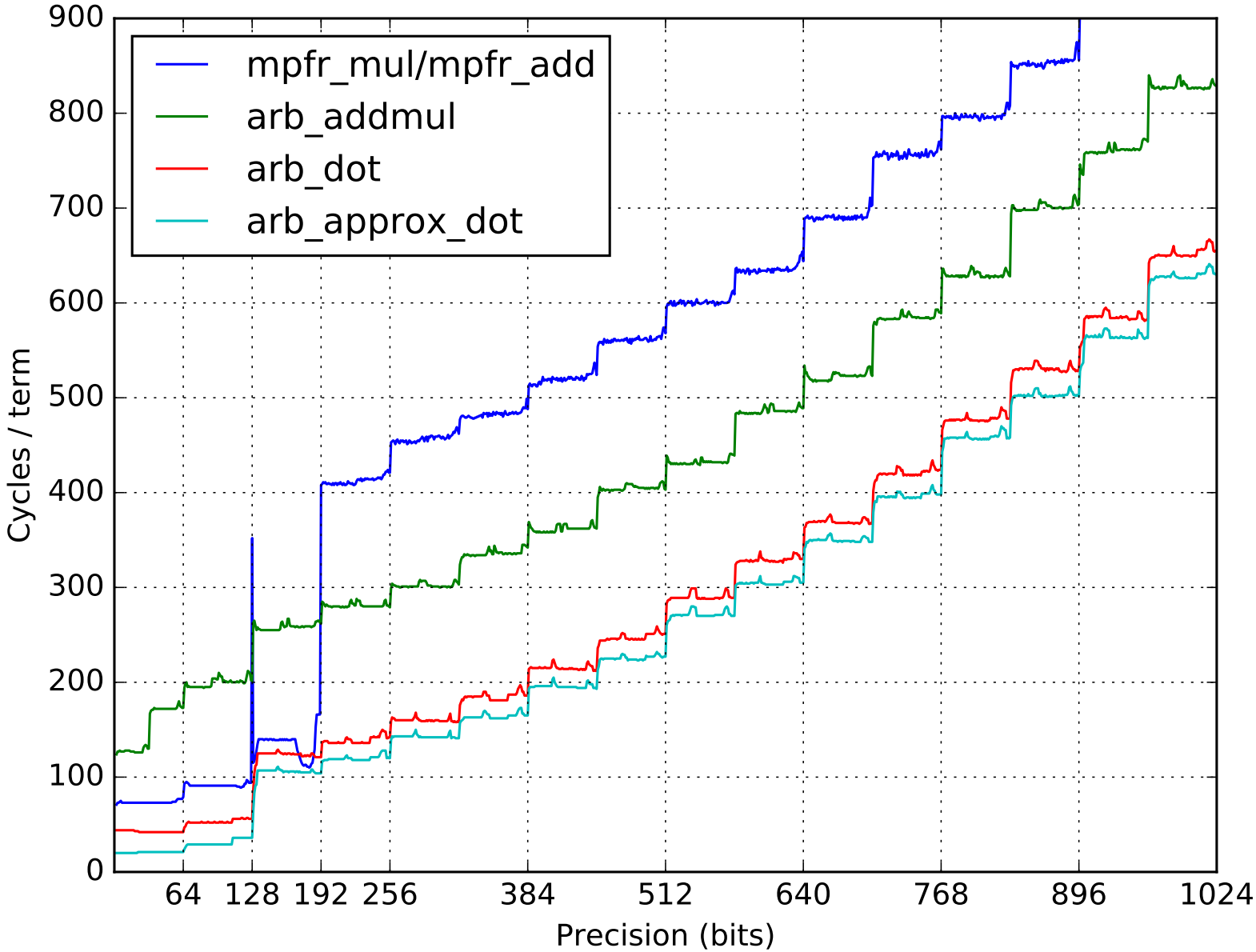
		53 bits (1 word)	106 bits (2 words)	212 bits (4 words)
`double`	`s += a[k] * b[k]`	2		
	Vectorized	1/4?		
QD 2.3	`s += a[k] * b[k]`		26	279
MPFR 3.1	`mpfr_mul/mpfr_add`	121	328	403
MPFR 4.0	`mpfr_mul/mpfr_add`	73	91	408
	[Lefèvre & Zimmermann, ARITH 2017]			
Arb (ball)	`arb_addmul`	172	200	280
Arb (float)	`arf_addmul`	106	136	211

Dot product performance

Cycles/term to compute $\sum_k a_k b_k$ on an i5-4300U (Haswell)

		53 bits (1 word)	106 bits (2 words)	212 bits (4 words)
`double`	`s += a[k] * b[k]`	2		
	Vectorized	1/4?		
QD 2.3	`s += a[k] * b[k]`		26	279
MPFR 3.1	`mpfr_mul/mpfr_add`	121	328	403
MPFR 4.0	`mpfr_mul/mpfr_add`	73	91	408
	[Lefèvre & Zimmermann, ARITH 2017]			
Arb (ball)	`arb_addmul`	172	200	280
Arb (float)	`arf_addmul`	106	136	211
Arb (ball)	`arb_dot`	42	52	136
Arb (float)	`arb_approx_dot`	21	29	119

Dot product performance



Complex dot product performance

Cycles/term to compute $\sum_k a_k b_k$ on an i5-4300U (Haswell)

		53 bits (1 word)	106 bits (2 words)	212 bits (4 words)
MPC 1.1	`mpc_mul/mpc_add`	570	1022	1513
Arb (ball)	`acb_addmul`	406	541	851
Arb (ball)	`acb_dot`	172	214	562
Arb (float)	`acb_approx_dot`	88	123	485

Outline of the algorithm

First pass

- Detect any Inf/NaN or exponents requiring bignums (switch to slow fallback code)
- Count number of nonzero terms in sum
- Determine maximum exponent of any term (separately for midpoints and radii)

Second pass: compute the dot product!

- Using GMP `mpn` arithmetic
- Avoid intermediate normalizations
- Single final rounding and conversion to a ball

Complex dot product $\simeq$ two length-$2N$ real dot products

Arbitrary-precision floating-point addition

$a = 2^e \cdot$ |xxxxxxxx|xxxxxxxx|

$b = 2^f \cdot$ |xxxxxxxx|xxxxxxxx|xxxxxxxx|

Arbitrary-precision floating-point addition

$a = 2^e \cdot$ |xxxxxxxx|xxxxxxxx|
$b = 2^f \cdot$ |xxxxxxxx|xxxxxxxx|xxxxxxxx|

The term with smaller exponent is right-shifted by $|f - e|$ bits, aligning the limb boundaries (several case distinctions)

```
                  |xxxxxxxx|xxxxxxxx|xxxxxxxx|
+                          |0000xxxx|xxxxxxxx|xxxx0000|
```

Arbitrary-precision floating-point addition

$a = 2^e \cdot$ |xxxxxxxx|xxxxxxxx|
$b = 2^f \cdot$ |xxxxxxxx|xxxxxxxx|xxxxxxxx|

The term with smaller exponent is right-shifted by $|f - e|$ bits, aligning the limb boundaries (several case distinctions)

```
                    |xxxxxxxx|xxxxxxxx|xxxxxxxx|
+                             |0000xxxx|xxxxxxxx|xxxx0000|
=  |0000000x|xxxxxxxx|xxxxxxxx|xxxxxxxx|xxxx0000|
```

(Exact sum, with possible carry-out limb)

Arbitrary-precision floating-point addition

$a = 2^e \cdot$ |xxxxxxxx|xxxxxxxx|
$b = 2^f \cdot$ |xxxxxxxx|xxxxxxxx|xxxxxxxx|

The term with smaller exponent is right-shifted by $|f - e|$ bits, aligning the limb boundaries (several case distinctions)

```
                  |xxxxxxxx|xxxxxxxx|xxxxxxxx|
+                            |0000xxxx|xxxxxxxx|xxxx0000|
=  |0000000x|xxxxxxxx|xxxxxxxx|xxxxxxxx|xxxx0000|
```

(Exact sum, with possible carry-out limb)

```
=              |1xxxxxxx|xxxxxxxx|xxxxx000|    + ulp error
```

(Rounded and normalized)

Addition for fast dot product

Terms in dot product:
$t_k = (-1)^{s_k} 2^{e_k} \cdot |\texttt{xxxxxxxx}|\texttt{xxxxxxxx}|, \quad e_k = \exp(a_k) + \exp(b_k)$

Accumulator:
$s = 2^f \cdot |\texttt{xxxxxxxx}|\texttt{xxxxxxxx}|\texttt{xxxxxxxx}|$
$f = \max_k(e_k) + \text{bitlength}(\#\text{nonzero terms}) + 1$
Subtraction uses two's complement

Addition for fast dot product

Terms in dot product:

$t_k = (-1)^{s_k} 2^{e_k} \cdot$ |xxxxxxxx|xxxxxxxx|, $\quad e_k = \exp(a_k) + \exp(b_k)$

Accumulator:

$s = 2^f \cdot$ |xxxxxxxx|xxxxxxxx|xxxxxxxx|

$f = \max_k(e_k) + \text{bitlength}(\#\text{nonzero terms}) + 1$

Subtraction uses two's complement

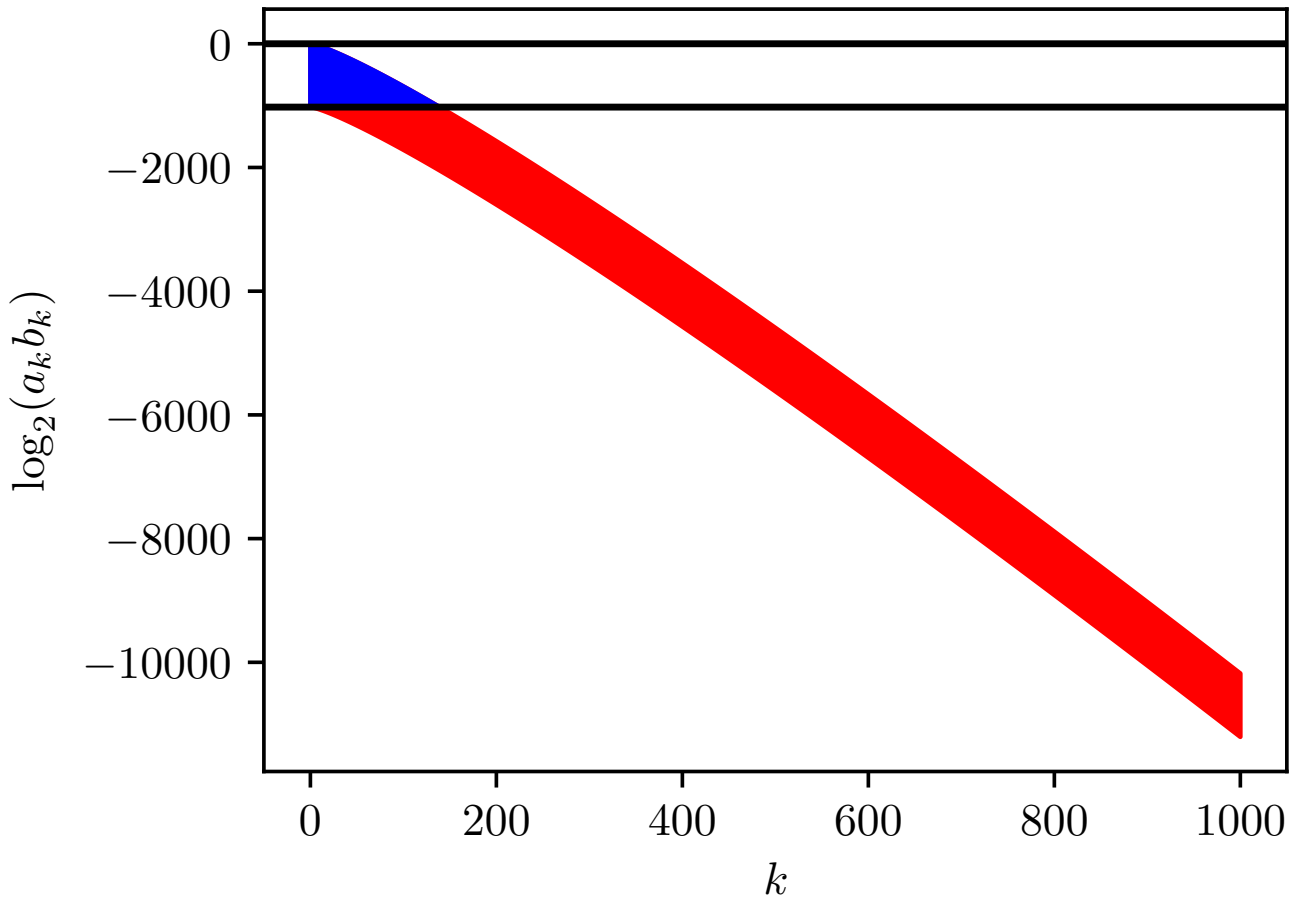
```
                |xxxxxxxx|xxxxxxxx|xxxxxxxx|
+                        |0000xxxx|xxxxxxxx|xxxx0000|
=               |xxxxxxxx|xxxxxxxx|xxxxxxxx|
                                              + ulp error
```

Multiplications in the dot product

- Using GMP's `mpn_mul` in general
- Inline ASM for $\leq 2 \times 2$ limb product, ≤ 3 limb accumulator
- Mulder's mulhigh (via MPFR) for 25 to 10000 limbs
- Gauss formula (3 instead of 4 real multiplications) for complex numbers at ≥ 128 limbs
- Optimal complexity for non-uniform input: insignificant limbs in the input vectors are ignored

Ignoring insignificant limbs

Example: $\sum_{k=0}^{1000} a_k b_k,\ a_k = \dfrac{1}{k!},\ b_k = \dfrac{1}{\pi^k},\ p = 1024$



`arb_addmul`: 0.33 ms `arb_dot`: 0.035 ms

Error bound calculations

Three separate error calculations:

- Sum of propagated errors (ball radius cross-products)
  - One 64-bit scaled fixed-point number: 30 fraction bits plus 34 bits for carry accumulation
  - Exponent is fixed during the whole computation
- Sum of term truncation errors in midpoint sum
  - One 64-bit integer, counting ulps at the working precision
- Final midpoint rounding error
  - 0 or 1 ulp at the output precision

The three terms are added together to form a single 30-bit floating-point number at the end

Idea (easy): correctly rounded dot product for MPFR/MPC

1. Compute fast approximate dot product + power-of-two error bound, using several guard bits of precision

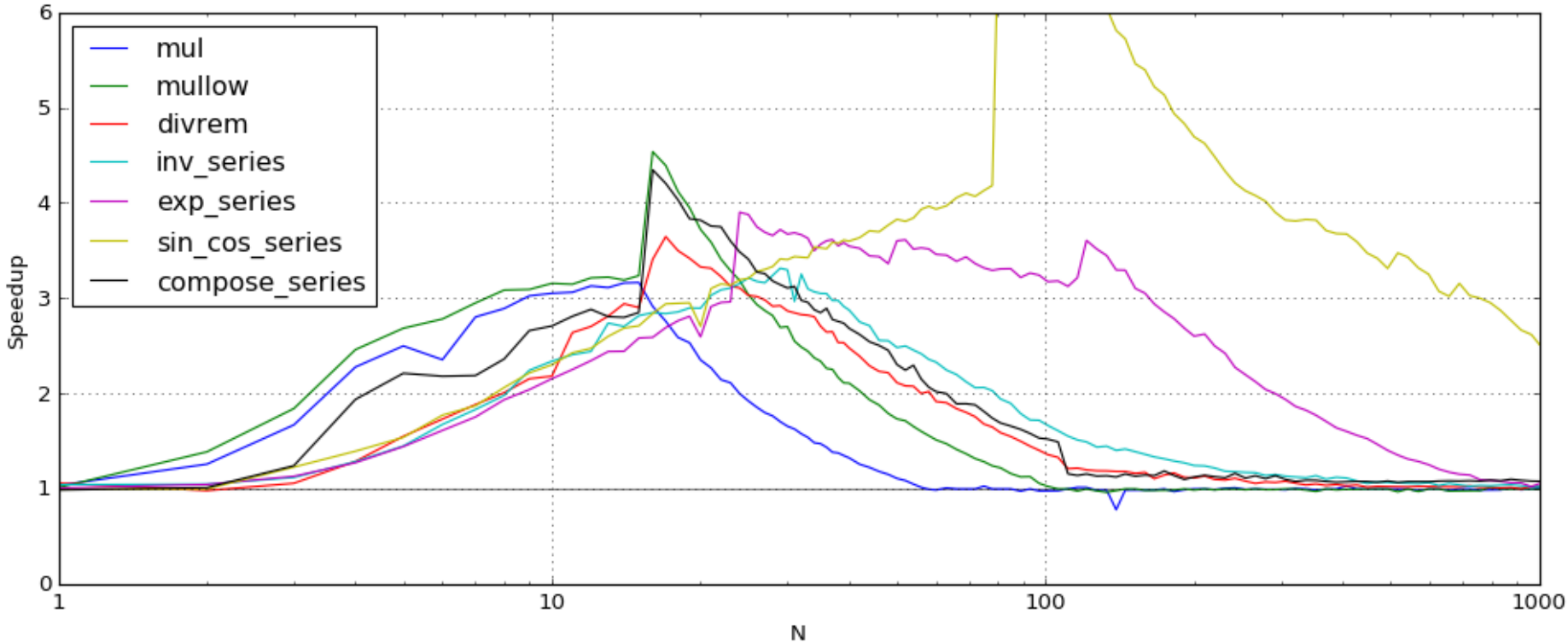
2. If final rounding test fails, use a fallback algorithm (could use `mpfr_sum` + exact products)

Should run about as fast as `arb_approx_dot` on average

Speedup due to dot product: polynomial arithmetic

Speedup between Arb 2.14 and 2.15 due to switching from `arb_addmul`/`acb_addmul` to `arb_dot`/`acb_dot`

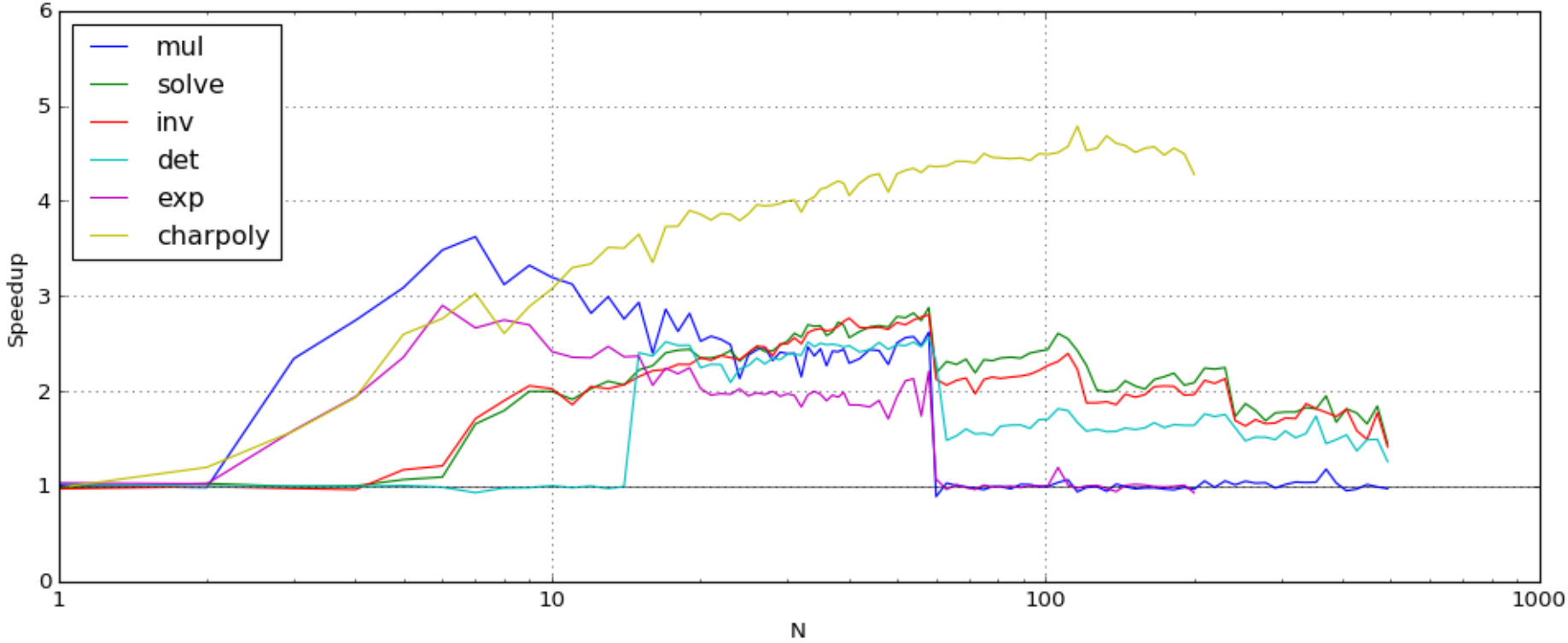
Complex polynomials (`acb_poly`), 64-bit precision



Speedup due to dot product: matrix arithmetic

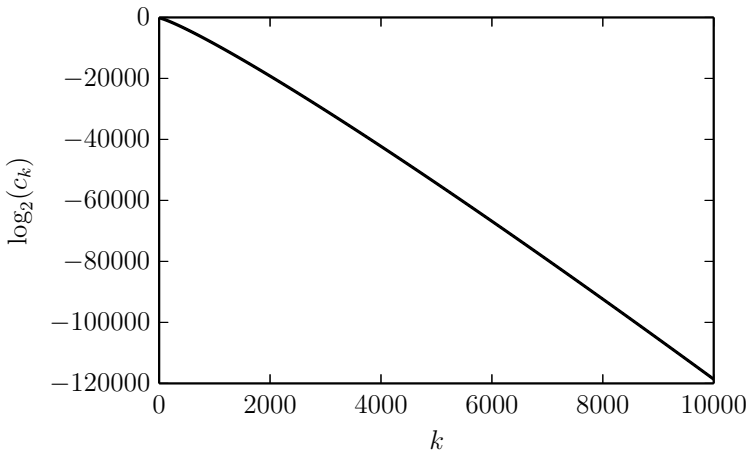
Speedup between Arb 2.14 and 2.15 due to switching from `arb_addmul`/`acb_addmul` to `arb_dot`/`acb_dot`

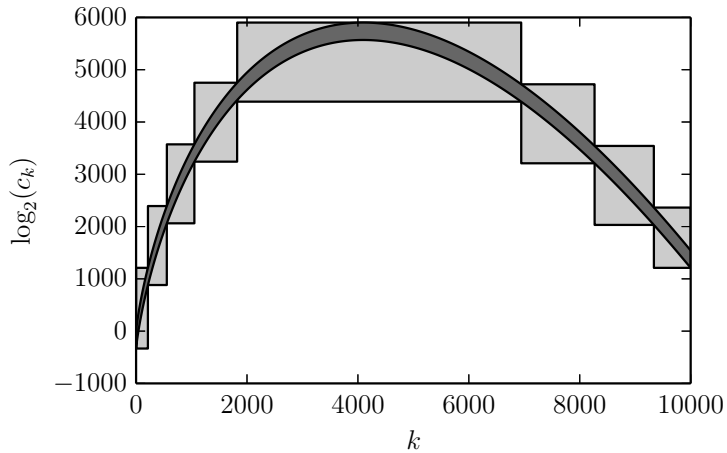
Complex matrices (`acb_mat`), 64-bit precision



BIG polynomial multiplication in Arb

- $(A+a)(B+b)$ via three multiplications AB, $|A|b$, $a(|B|+b)$
- Scaling $x \to 2^e x$, splitting into blocks





Transformation used to square the length-10 000 Taylor polynomial for $\exp(x) = \sum x^k/k!$ *at* 333 *bits precision*

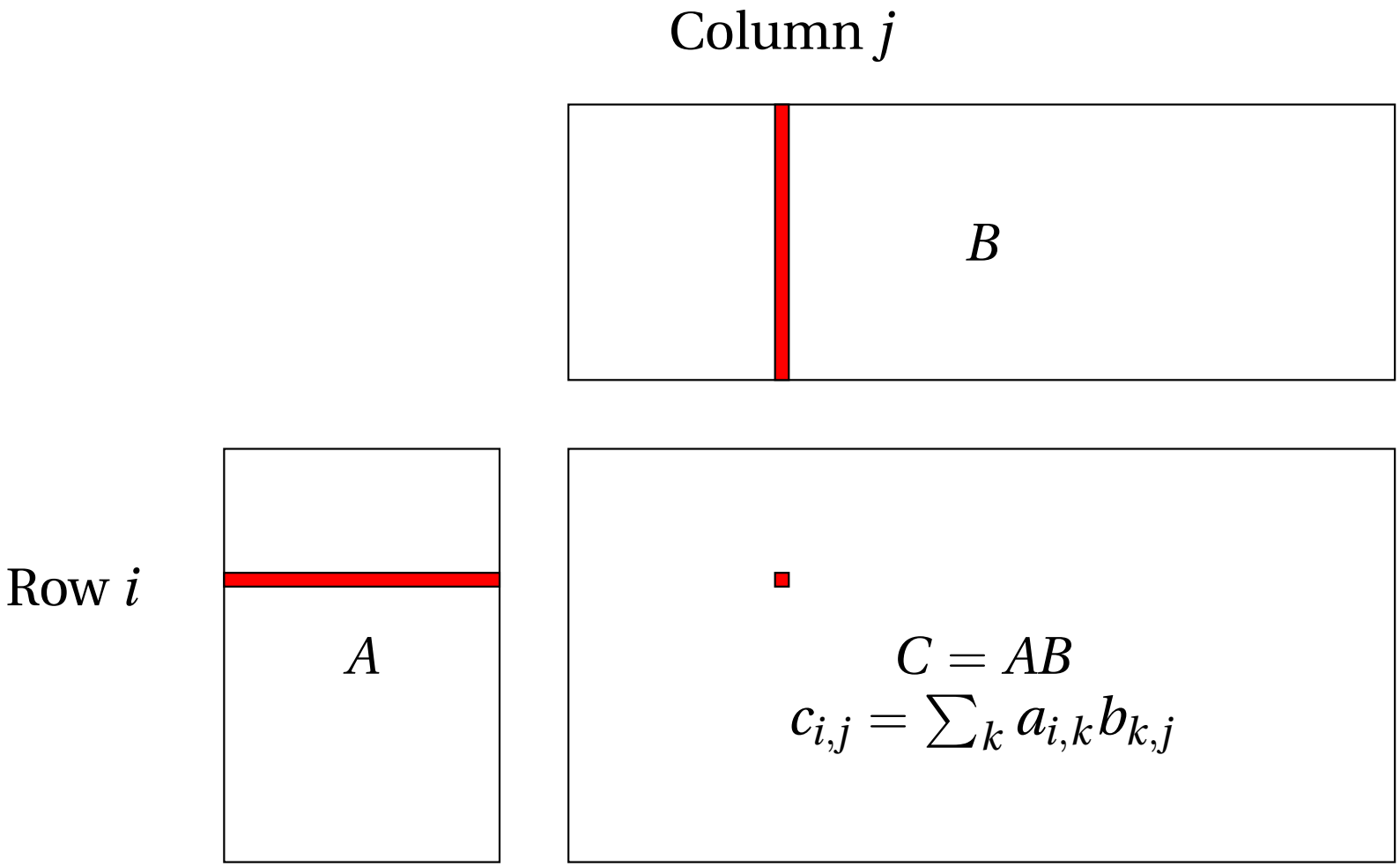
- Blocks multiplied exactly over $\mathbb{Z}$ using FLINT (Karatsuba, Kronecker, Schönhage-Strassen FFT)
- For blocks up to length 1000 in $|A|b$, $a(|B|+b)$, use `double`

[J., IEEE Trans. Comp., 2017], following [van der Hoeven, 2008]

BIG matrix multiplication in Arb (new in Arb 2.14)

- $(A+a)(B+b)$ via three multiplications AB, $|A|b$, $a(|B|+b)$
- Scaling $AB \to (EA)(BF)$, splitting into blocks
- Blocks multiplied exactly over $\mathbb{Z}$ using FLINT (Strassen, small primes multimodular multiplication)
- For blocks in $|A|b$, $a(|B|+b)$, use `double`

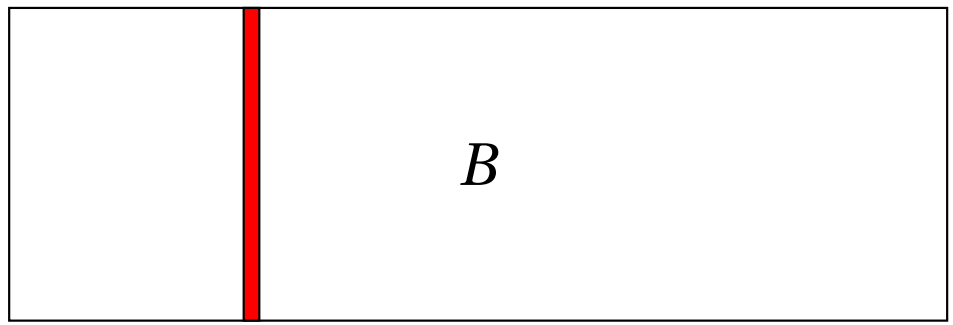
Matrix multiplication



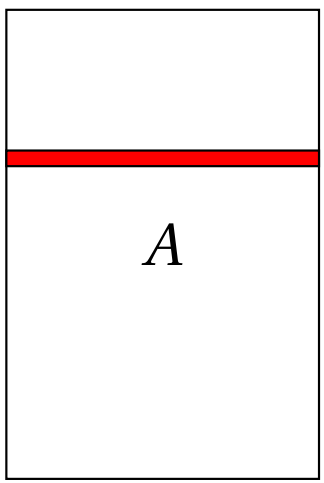
Matrix multiplication, scaled to integers

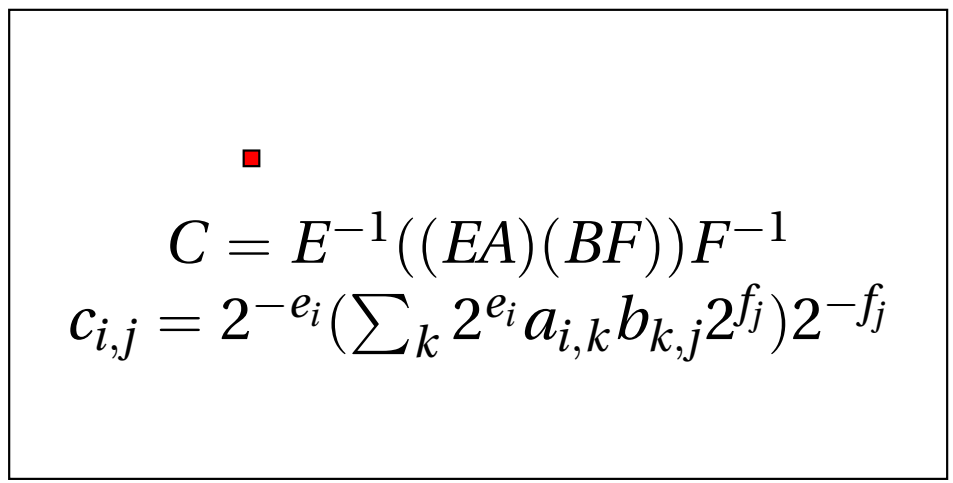
Column j $\times 2^{f_j}$

$E = \mathrm{diag}(2^{e_i})$,
$F = \mathrm{diag}(2^{f_i})$



Row i
$\times 2^{e_i}$

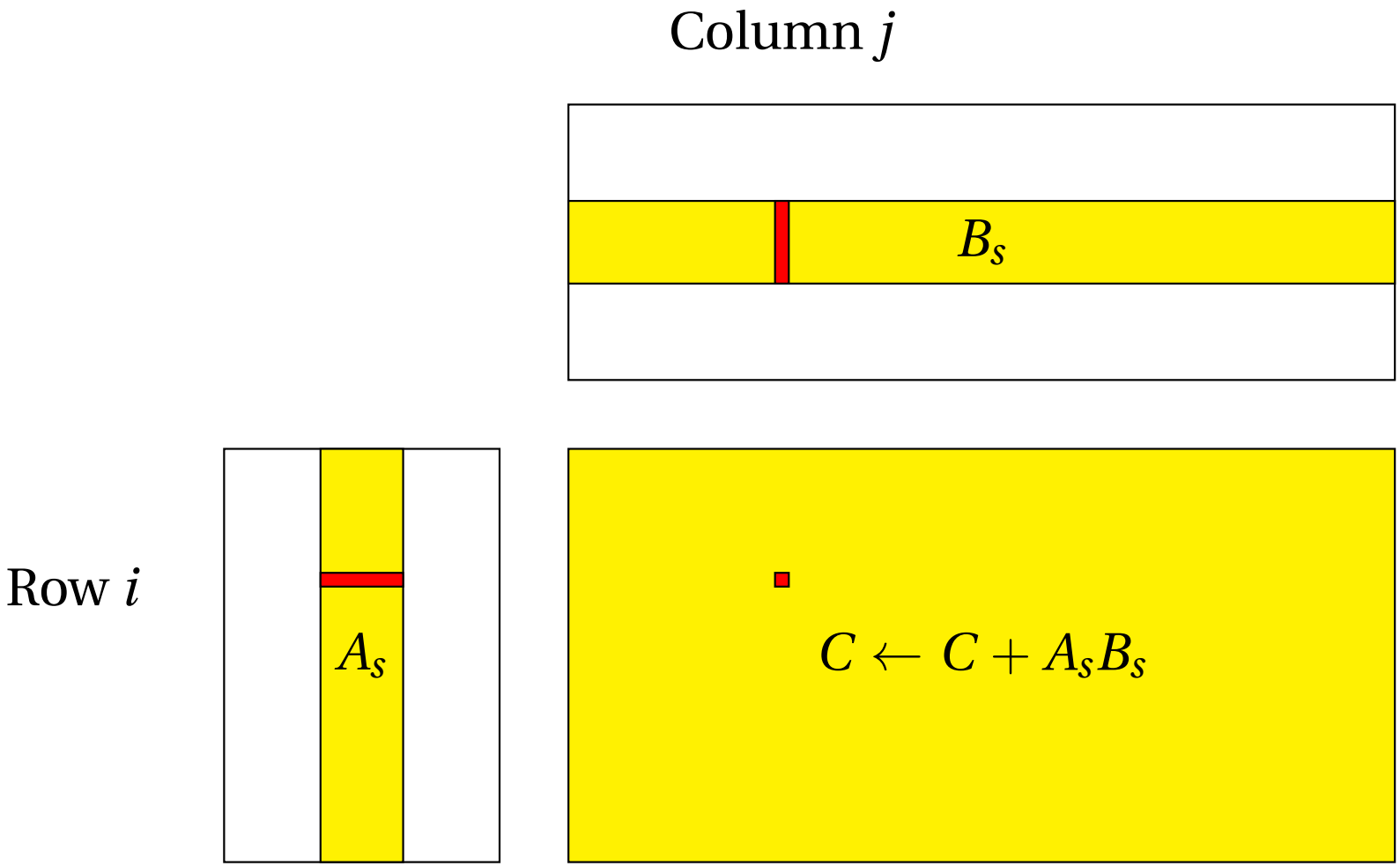




$$C = E^{-1}((EA)(BF))F^{-1}$$
$$c_{i,j} = 2^{-e_i}(\textstyle\sum_k 2^{e_i} a_{i,k} b_{k,j} 2^{f_j})2^{-f_j}$$

Block matrix multiplication

Blocks A_s, B_s chosen (using greedy search) so that each row of A_s and column of B_s has a small internal exponent range



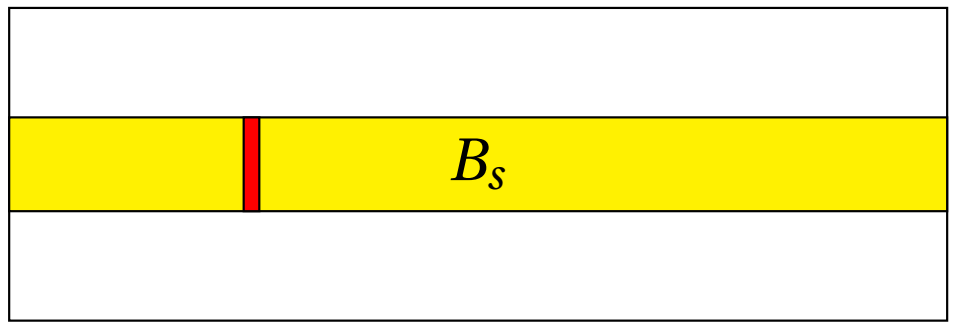
Block matrix multiplication, scaled to integers

Scaling is applied internally to each block A_s, B_s

Column j $\times 2^{f_{j,s}}$

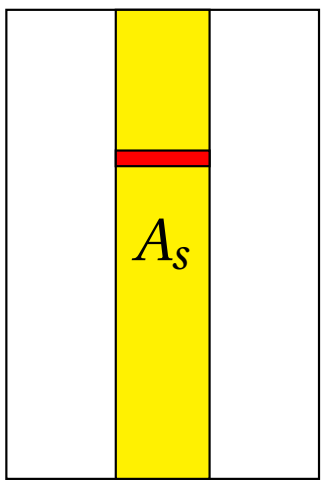
$E_s = \mathrm{diag}(2^{e_{i,s}})$,

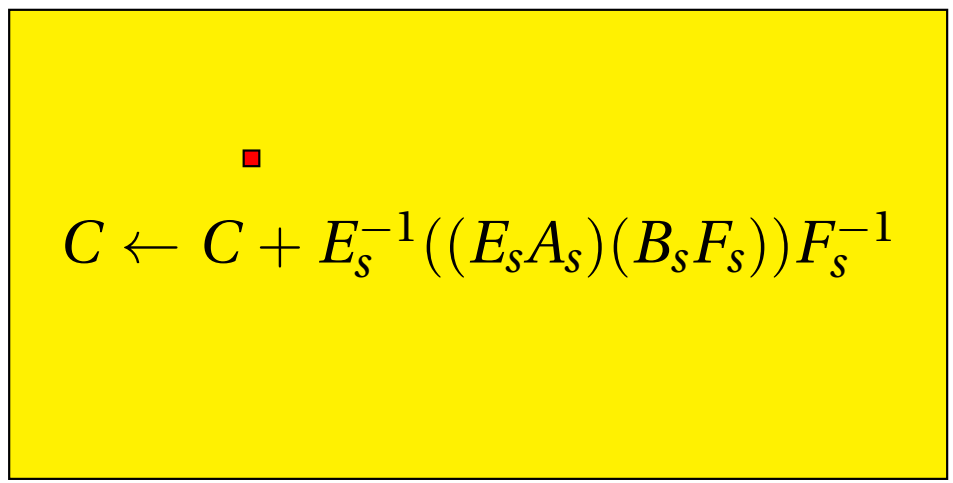
$F_s = \mathrm{diag}(2^{f_{i,s}})$



Row i

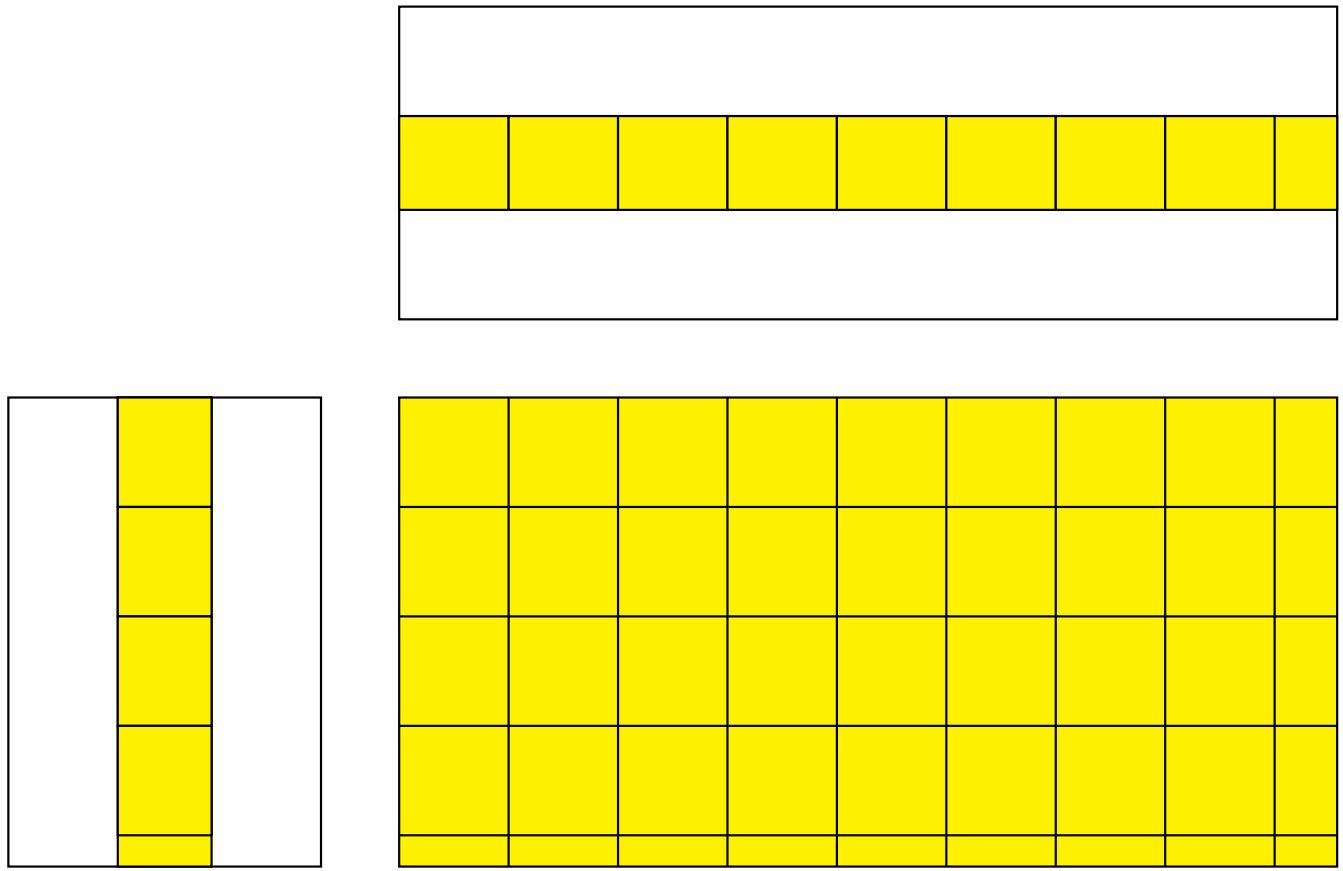
$\times 2^{e_{i,s}}$





Block matrix multiplication, final splitting

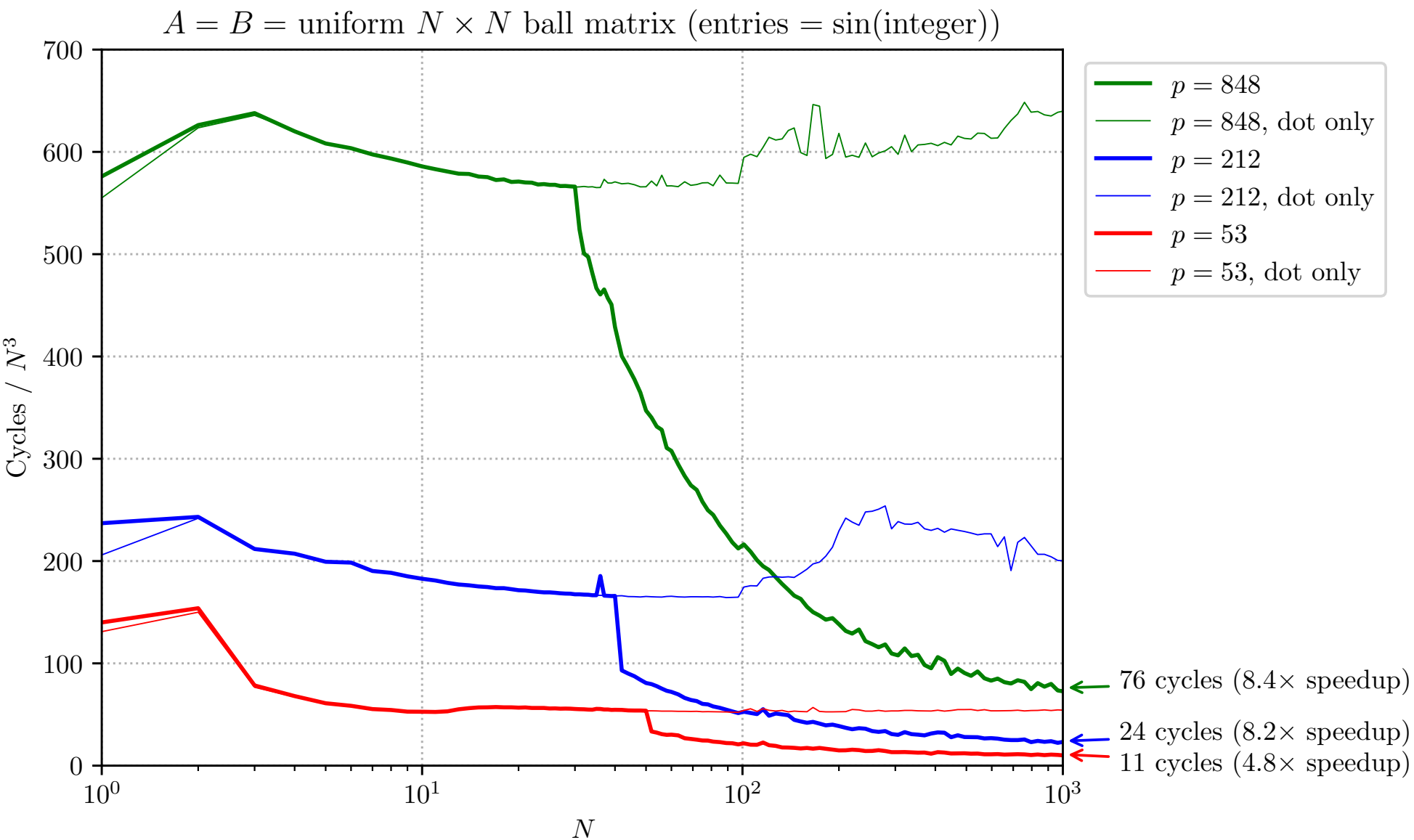
Before actually multiplying over $\mathbb{Z}$, rectangular blocks are split into approximately square sub-blocks



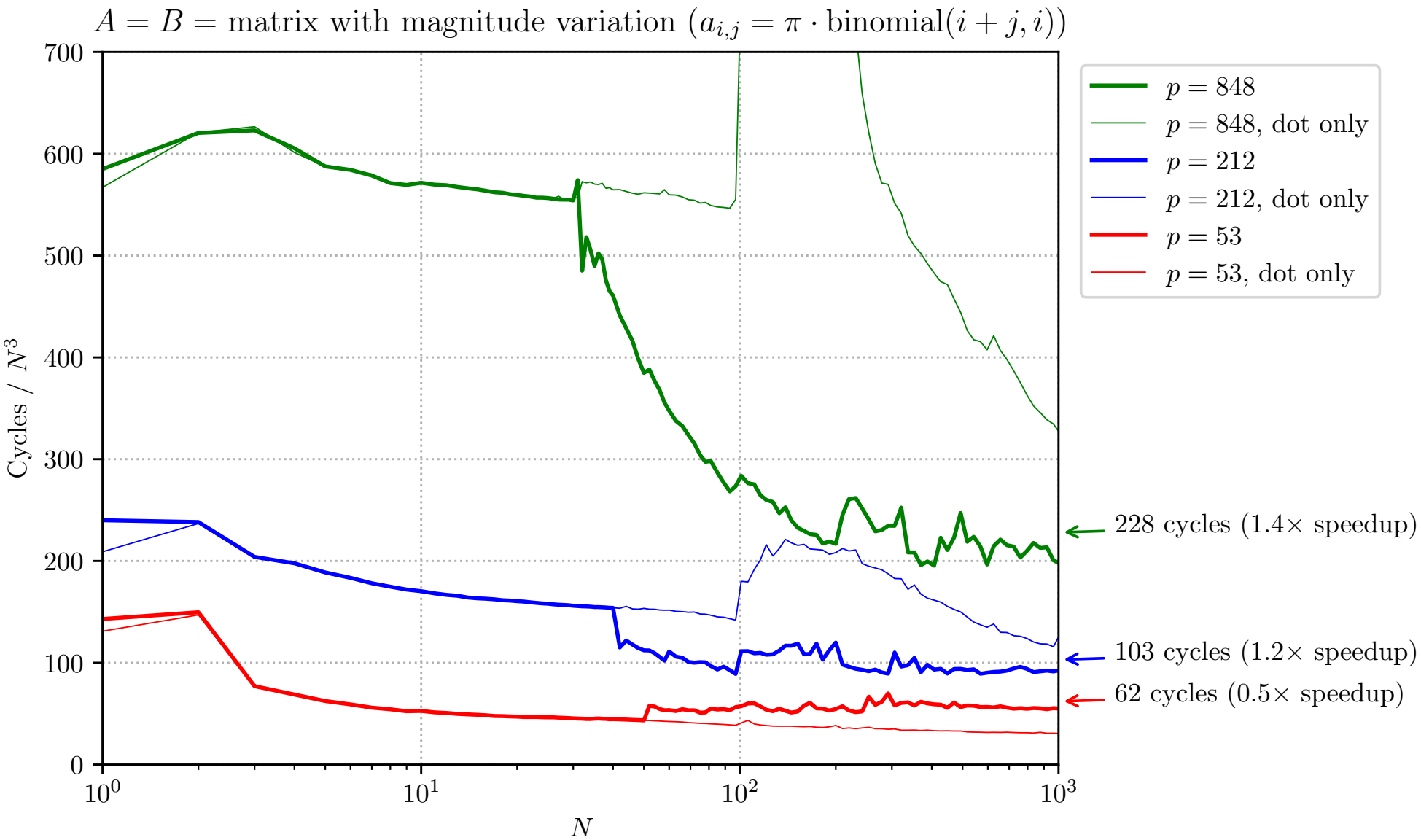
Matrix multiplication TODO

- Faster matrix multiplication in FLINT
  - Use 20-bit primes + BLAS `dgemm`, instead of 60-bit primes
  - ~~Optimize Chinese remaindering~~
  - Optimize for structured matrices
  - Multithreading
- Avoid temporary memory allocations
- Improve cutoffs between basecase (dot) and block algorithms
- Better strategy for splitting into blocks
- Output sensitivity (discard useless bits from input)
- Optimize approximate matrix multiplication (without error bounds)

`arb_mat_mul` performance: dot+block vs dot only



`arb_mat_mul` performance: dot+block vs dot only



Numerically stable ball linear algebra

Gaussian elimination (GE) in ball/interval arithmetic is unstable in general, even for well-conditioned matrices

Numerically stable ball linear algebra

Gaussian elimination (GE) in ball/interval arithmetic is unstable in general, even for well-conditioned matrices

Solving $AX = B$ (Hansen-Smith, 1967):

- Compute $R \approx A^{-1}$ using (nonrigorous) floating-point GE, then solve $(RA)X = RB$
- GE in ball arithmetic applied to $RA \approx I$ is stable (can also use perturbation norm bounds)

Numerically stable ball linear algebra

Gaussian elimination (GE) in ball/interval arithmetic is unstable in general, even for well-conditioned matrices

Solving $AX = B$ (Hansen-Smith, 1967):

- Compute $R \approx A^{-1}$ using (nonrigorous) floating-point GE, then solve $(RA)X = RB$
- GE in ball arithmetic applied to $RA \approx I$ is stable (can also use perturbation norm bounds)

Computing $\det(A)$ (Rump, 2010):

- Compute approximate LU factorization $A \approx PLU$ and approximate inverses $L' \approx L^{-1}$, $U' \approx U^{-1}$, then construct $B = L'P^{-1}AU'$ using ball arithmetic
- The determinant of $B \approx I$ can be enclosed using Gershgorin circles (warning: correctness not obvious!)

Several methods

- `arb_mat_solve_lu` – GE in ball arithmetic; unstable
- `arb_mat_solve_precond` – Hansen-Smith algorithm; 4-5 times slower, stable
- `arb_mat_solve` – automatic choice; GE in ball arithmetic if $n \leq 4$ or $p > 10n$, otherwise Hansen-Smith
- `arb_mat_approx_solve` – GE in floating-point arithmetic (no error bounds); stable; uses `arb_approx_dot` in the basecase

All versions of GE are block recursive to reduce everything to matrix multiplication asymptotically (all algorithms cost $O(1)$ full-size matrix multiplications)

Example: determinant of size-n DFT matrix

Output from `acb_mat_det` at 53-bit precision:

```
n       Arb 2.13
-------------------------------------------------------
10      [-1.0000000000 +/- 5.68e-12] + [+/- 5.67e-12]*I
30      [1.0 +/- 7.08e-3]            +  [+/- 7.08e-3]*I
100
300
1000

n       Arb 2.15
-------------------------------------------------------
10      [-1.0000000000 +/- 8.70e-12] + [+/- 8.69e-12]*I
30      [1.000000000 +/- 1.99e-11]   + [+/- 1.99e-11]*I
100     [+/- 9.43e-10]   +   [1.0000000 +/- 9.43e-10]*I
300     [+/- 2.89e-8]    +     [1.000000 +/- 2.89e-8]*I
1000    [+/- 1.73e-6]    +      [-1.0000 +/- 1.73e-6]*I
```

Example: determinant of size-n DFT matrix

Doubling the precision until `acb_mat_det` succeeds

n	Final precision	Total time	Arb 2.13
10	53	0.00011 s	
30	53	0.0033 s	
100	212	0.43 s	
300	848	27.4 s	
1000	1696	1712 s	

n	Final precision	Total time	Arb 2.15
10	53	0.00011 s	
30	53	0.0060 s	
100	53	0.15 s	
300	53	3.0 s	
1000	53	80.6 s	

Last slide of the talk

Improvements from Arb 2.13 to Arb 2.15:

- $2-5\times$ speedup (at low precision) for basecase polynomial and matrix arithmetic, due to fast dot product
- $20\times$ speedup (at low precision) for multiplying big matrices, due to use of blocks + FLINT
- Numerically stable linear algebra + taking advantage of matrix multiplication

Still a work in progress!