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 $1000 \times 1000$ matrices

<table>
<thead>
<tr>
<th></th>
<th>$p = 53$</th>
<th>$p = 106$</th>
<th>$p = 212$</th>
<th>$p = 848$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS</td>
<td>0.08</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QD</td>
<td></td>
<td>11</td>
<td>111</td>
<td></td>
</tr>
<tr>
<td>MPFR</td>
<td>36</td>
<td>44</td>
<td>110</td>
<td>293</td>
</tr>
<tr>
<td>Arb* (classical)</td>
<td>19</td>
<td>25</td>
<td>76</td>
<td>258</td>
</tr>
<tr>
<td>Arb* (block)</td>
<td>3.6</td>
<td>5.6</td>
<td>8.2</td>
<td>27</td>
</tr>
</tbody>
</table>

* With ball coefficients
Arb version 2.16 – [http://arblib.org](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

\[ \sum_{k=1}^{N} a_k b_k, \quad a_k, b_k \in \mathbb{R} \text{ or } \mathbb{C} \]

Kernel in basecase \((N \lesssim 10 \text{ to } 100)\) 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:

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

The new way:

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

(More generally, computes \( s = s_0 + (-1)^c \sum_{k=0}^{N-1} a_k \cdot a_{\text{step}} b_k \cdot b_{\text{step}} \))

- `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:

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

Ball arithmetic dot product:

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

\[ m = \sum_{k=1}^{N} m_k m'_k + \varepsilon, \quad r \geq |\varepsilon| + \sum_{k=1}^{N} |m_k| r'_k + |m'_k| r_k + r_k r'_k \]
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:

\[ 0 \leq b_k < 2^{64}, \quad b_{n-1} \geq 2^{63}, \quad b_0 \neq 0 \]

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

Radius: 30-bit unsigned floating-point
Arbitrary-precision multiplication

\[ \begin{array}{c|c}
1 & \cdots & m \text{ limbs} \\
1 & \cdots & n \text{ limbs} \\
\end{array} \]
Arbitrary-precision multiplication

\[
\begin{align*}
1 & \ldots \quad m \text{ limbs} \\
1 & \ldots \quad n \text{ limbs} \\
\end{align*}
\]

Exact multiplication: \( mn \text{ mul } \rightarrow m + n \text{ limbs} \)
Arbitrary-precision multiplication

\[
\begin{array}{c|c}
1 \ldots & \multicolumn{1}{c}{m \text{ limbs}} \\
1 \ldots & \multicolumn{1}{c}{n \text{ limbs}} \\
\end{array}
\]

Exact multiplication: \( \text{mpn} \text{ mul} \rightarrow m + n \text{ limbs} \)

\[
\begin{array}{c|c|c|c|c|c}
 & & & & & \\
01 \ldots & & & & & \\
\end{array}
\]

Rounding to \( p \) bits and bit alignment

\[
\begin{array}{c|c}
1 \ldots & \multicolumn{1}{c}{\ldots1000} \\
\hline
\text{\_\_\_\_} & \leq p \text{ bits} \\
\end{array}
\]
Arbitrary-precision addition

Exponent difference
Arbitrary-precision addition

Align limbs: mpn_lshift etc.

Exponent difference

10 / 26
Arbitrary-precision addition

Exponent difference

Align limbs: mpn_lshift etc.

Addition: mpn_add_n, mpn_sub_n, mpn_add_1 etc.
Arbitrary-precision addition

Exponent difference → Align limbs: mpn_lshift etc.

Addition: mpn_add_n, mpn_sub_n, mpn_add_1 etc.

Rounding to \( p \) bits and bit alignment

\[
\begin{array}{c}
1 \ldots \ldots \ldots 1000 \\
\hline
\leq p \text{ bits}
\end{array}
\]
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

N terms
Dot product

$p$ bits

$N$ terms

2's complement accumulator

A, B: $\sim \log_2 N$ bits padding
Dot product

\[ A \quad p \text{ bits} \quad B \]

\[ A, B: \sim \log_2 N \text{ bits paddling} \]

\[ \text{N terms} \]

\[ \rightarrow \text{2’s complement accumulator} \rightarrow \]

\[ \text{Error accumulator} \]
Dot product

A, B: $\sim \log_2 N$ bits padding

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, $\leq 3$ 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 $\geq 128$ limbs
Dot product performance

![Graph showing dot product performance with different bit precisions and cycles per term for various operations: arb_addmul, mpfr_mul/mpfr_add, arb_dot, and arb_approx_dot.](image)
Dot product performance

- **arb_addmul**
- **mpfr_mul/mpfr_add**
- **arb_dot**
- **arb_approx_dot**
- **QD (p = 106)**
- **QD (p = 212)**

**Cycles / term**

**Bit precision p**
Dot product: polynomial operations speedup in Arb

(Complex coefficients, $p = 64$ bits)
## Dot product: matrix operations speedup in Arb

<table>
<thead>
<tr>
<th>Matrix size $N$</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

(Complex coefficients, $p = 64$ bits)

![Graph showing speedup for various matrix operations as matrix size increases.](image)
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

\[ C = AB \]
\[ c_{i,j} = \sum_k a_{i,k} b_{k,j} \]
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.

$C \leftarrow C + A_s B_s$
Block matrix multiplication, scaled to integers

Scaling is applied internally to each block $A_s, B_s$

\[
E_s = \text{diag}(2^{e_{i,s}}), \quad F_s = \text{diag}(2^{f_{i,s}})
\]

\[
C \leftarrow C + E_s^{-1}((E_sA_s)(B_sF_s))F_s^{-1}
\]
# Uniform and non-uniform matrices

**Uniform matrix, \( N = 1000 \)**

<table>
<thead>
<tr>
<th>( p )</th>
<th>Classical</th>
<th>Block</th>
<th>Number of blocks</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>19 s</td>
<td>3.6 s</td>
<td>1</td>
<td>5.3</td>
</tr>
<tr>
<td>212</td>
<td>76 s</td>
<td>8.2 s</td>
<td>1</td>
<td>9.3</td>
</tr>
<tr>
<td>3392</td>
<td>1785 s</td>
<td>115 s</td>
<td>1</td>
<td>15.5</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>( p )</th>
<th>Classical</th>
<th>Block</th>
<th>Number of blocks</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>12 s</td>
<td>20 s</td>
<td>10</td>
<td>0.6</td>
</tr>
<tr>
<td>212</td>
<td>43 s</td>
<td>35 s</td>
<td>9</td>
<td>1.2</td>
</tr>
<tr>
<td>3392</td>
<td>1280 s</td>
<td>226 s</td>
<td>2</td>
<td>5.7</td>
</tr>
</tbody>
</table>
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: $3.141 \rightarrow [3.141 \pm 0.001]$

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 $Ax = b$ ($N = 1000$, $p = 212$)

![Bar chart showing comparison of algorithms for solving linear systems.

- **Eigen/MPFR**
- **Arb**
- **Certified (Hansen-Smith)**

The chart illustrates the time (in seconds) required for each algorithm to solve the linear system.
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× speedup)

- Accurate and fast large-$N$ matrix multiplication using scaled integer blocks (≈ 10× 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