A new multiplication algorithm for extended precision using floating-point expansions

Valentina Popescu, Jean-Michel Muller, Ping Tak Peter Tang

ARITH 23
July 2016
Target applications

1. Need massive parallel computations
   → high performance computing using graphics processors – GPUs;

2. Need more precision than standard available (up to few hundred bits)
   → extend precision using floating-point expansions.
Target applications

1. Need massive parallel computations
   → high performance computing using graphics processors – GPUs;

2. Need more precision than standard available (up to few hundred bits)
   → extend precision using floating-point expansions.

Chaotic dynamical systems:
- bifurcation analysis;
- compute periodic orbits (e.g., finding sinks in the Hénon map, iterating the Lorenz attractor);
- celestial mechanics (e.g., long term stability of the solar system).

Experimental mathematics: ill-posed SDP problems in
- computational geometry (e.g., computation of kissing numbers);
- quantum chemistry/information;
- polynomial optimization etc.
Extended precision

Existing libraries:

- GNU MPFR - not ported on GPU;
- GARPREC & CUMP - tuned for big array operations: data generated on host, operations on device;
- QD & GQD - limited to double-double and quad-double; no correctness proofs.

What we need:

- support for arbitrary precision;
- runs both on CPU and GPU;
- easy to use.

CAMPARY – CUDA MULTIPLE PRECISION ARITHMETIC LIBRARY –
Extended precision

Existing libraries:

- GNU MPFR - not ported on GPU;
- GARPREC & CUMP - tuned for big array operations: data generated on host, operations on device;
- QD & GQD - limited to double-double and quad-double; no correctness proofs.

What we need:

- support for arbitrary precision;
- runs both on CPU and GPU;
- easy to use.

CAMPARY – CudaA Multiple Precision ARithmetic librarY –
Our approach: multiple-term representation
– floating-point expansions –
Our approach: multiple-term representation
  – floating-point expansions –

Pros:
  – use directly available and highly optimized native FP infrastructure;
  – straightforwardly portable to highly parallel architectures, such as GPUs;
  – sufficiently simple and regular algorithms for addition.
CAMPARY (CudaA Multiple Precision ARithmetic librarY)

Our approach: multiple-term representation
– floating-point expansions –

Pros:
– use directly available and highly optimized native FP infrastructure;
– straightforwardly portable to highly parallel architectures, such as GPUs;
– sufficiently simple and regular algorithms for addition.

Cons:
– more than one representation;
– existing multiplication algorithms do not generalize well for an arbitrary number of terms;
– difficult rigorous error analysis → lack of thorough error bounds.
**Non-overlapping expansions**

\( R = 1.11010011e - 1 \) can be represented, using a \( p = 5 \) (in radix 2) system, as:

**Least compact**

\[
R = x_0 + x_1 + x_2:
\begin{align*}
x_0 &= 1.1000e - 1; \\
x_1 &= 1.0010e - 3; \\
x_2 &= 1.0110e - 6.
\end{align*}
\]

**Most compact**

\[
R = z_0 + z_1:
\begin{align*}
z_0 &= 1.1101e - 1; \\
z_1 &= 1.1000e - 8.
\end{align*}
\]

\[
R = y_0 + y_1 + y_2 + y_3 + y_4 + y_5:
\begin{align*}
y_0 &= 1.0000e - 1; \\
y_1 &= 1.0000e - 2; \\
y_2 &= 1.0000e - 3; \\
y_3 &= 1.0000e - 5; \\
y_4 &= 1.0000e - 8; \\
y_5 &= 1.0000e - 9.
\end{align*}
\]
Non-overlapping expansions

\( R = 1.11010011e - 1 \) can be represented, using a \( p = 5 \) (in radix 2) system, as:

\[
R = x_0 + x_1 + x_2:
\]
\[
\begin{align*}
x_0 &= 1.1000e - 1; \\
x_1 &= 1.0010e - 3; \\
x_2 &= 1.0110e - 6.
\end{align*}
\]

Most compact \( R = z_0 + z_1: \)
\[
\begin{align*}
z_0 &= 1.1101e - 1; \\
z_1 &= 1.1000e - 8.
\end{align*}
\]

Less compact \( R = y_0 + y_1 + y_2 + y_3 + y_4 + y_5: \)
\[
\begin{align*}
y_0 &= 1.0000e - 1; \\
y_1 &= 1.0000e - 2; \\
y_2 &= 1.0000e - 3; \\
y_3 &= 1.0000e - 5; \\
y_4 &= 1.0000e - 8; \\
y_5 &= 1.0000e - 9.
\end{align*}
\]

Solution: the FP expansions are required to be non-overlapping.

**Definition: ulp-nonoverlapping.**

For an expansion \( u_0, u_1, \ldots, u_{n-1} \) if for all \( 0 < i < n \), we have \( |u_i| \leq \text{ulp}(u_{i-1}) \).

Example: \( p = 5 \) (in radix 2)
\[
\begin{align*}
x_0 &= 1.1010e - 2; \\
x_1 &= 1.1101e - 7; \\
x_2 &= 1.0000e - 11; \\
x_3 &= 1.1000e - 17.
\end{align*}
\]
Non-overlapping expansions

\( R = 1.11010011e - 1 \) can be represented, using a \( p = 5 \) (in radix 2) system, as:

**Less compact**

\[
R = y_0 + y_1 + y_2 + y_3 + y_4 + y_5:
\]

\[
\begin{align*}
  y_0 &= 1.0000e - 1; \\
  y_1 &= 1.0000e - 2; \\
  y_2 &= 1.0000e - 3; \\
  y_3 &= 1.0000e - 5; \\
  y_4 &= 1.0000e - 8; \\
  y_5 &= 1.0000e - 9.
\end{align*}
\]

**Most compact** \( R = z_0 + z_1 \):

\[
\begin{align*}
  z_0 &= 1.1101e - 1; \\
  z_1 &= 1.1000e - 8.
\end{align*}
\]

Solution: the FP expansions are required to be **non-overlapping**.

**Definition: ulp-nonoverlapping.**

For an expansion \( u_0, u_1, \ldots, u_{n-1} \) if for all \( 0 < i < n \), we have \( |u_i| \leq \text{ulp}(u_{i-1}) \).

Example: \( p = 5 \) (in radix 2)

\[
\begin{align*}
  x_0 &= 1.1010e - 2; \\
  x_1 &= 1.1101e - 7; \\
  x_2 &= 1.0000e - 11; \\
  x_3 &= 1.1000e - 17.
\end{align*}
\]

Restriction: \( n \leq 12 \) for single-precision and \( n \leq 39 \) for double-precision.
Error-Free Transforms: *Fast2Sum & 2MultFMA*

Algorithm 1 (*Fast2Sum* \((a, b)\))

\[
\begin{align*}
{s} & \leftarrow RN(a + b) \\
{z} & \leftarrow RN(s - a) \\
{e} & \leftarrow RN(b - z) \\
\text{return} & \ (s, e)
\end{align*}
\]

Requirement:

\[
e_a \geq e_b;
\]

→ Uses 3 FP operations.

Algorithm 2 (*2MultFMA* \((a, b)\))

\[
\begin{align*}
{p} & \leftarrow RN(a \cdot b) \\
{e} & \leftarrow fma(a, b, -p) \\
\text{return} & \ (p, e)
\end{align*}
\]

Requirement:

\[
e_a + e_b \geq e_{min} + p - 1;
\]

→ Uses 2 FP operations.
Existing multiplication algorithms

1. Priest’s multiplication [Pri91]:
   - very complex and costly;
   - based on scalar products;
   - uses re-normalization after each step;
   - computes the entire result and “truncates” a-posteriori;
   - comes with an error bound and correctness proof.
Existing multiplication algorithms

1. Priest’s multiplication [Pri91]:
   - very complex and costly;
   - based on scalar products;
   - uses re-normalization after each step;
   - computes the entire result and “truncates” a-posteriori;
   - comes with an error bound and correctness proof.

2. Quad-double multiplication in QD library [Hida et.al. 07]:
   - does not straightforwardly generalize;
   - can lead to $O(n^3)$ complexity;
   - worst case error bound is pessimistic;
   - no correctness proof is published.
requires: *ulp-nonoverlapping* FP expansion \( x = (x_0, x_1, \ldots, x_{n-1}) \) and 
\( y = (y_0, y_1, \ldots, y_{m-1}) \);

ensures: *ulp-nonoverlapping* FP expansion \( \pi = (\pi_0, \pi_1, \ldots, \pi_{r-1}) \).

Let me explain it with an example ...
Example: \( n = 4, m = 3 \) and \( r = 4 \)

\[
\begin{array}{cccccc}
  x_0 & x_1 & x_2 & x_3 & *\\
  y_0 & y_1 & y_2 & & & \\
  \hline
  x_0y_2 & x_1y_2 & x_2y_2 & x_3y_2 & & \\
  x_0y_1 & x_1y_1 & x_2y_1 & x_3y_1 & & \\
  x_0y_0 & x_1y_0 & x_2y_0 & x_3y_0 & & \\
\end{array}
\]
Example: $n = 4, m = 3$ and $r = 4$

- paper-and-pencil intuition;
- on-the-fly “truncation”;
Example: \( n = 4, m = 3 \) and \( r = 4 \)

<table>
<thead>
<tr>
<th></th>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( \ast )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( y_0 )</td>
<td>( y_1 )</td>
<td>( y_2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_0 y_2 )</td>
<td>( x_1 y_2 )</td>
<td>( x_2 y_2 )</td>
<td>( x_3 y_2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_0 y_1 )</td>
<td>( x_1 y_1 )</td>
<td>( x_2 y_1 )</td>
<td>( x_3 y_1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_0 y_0 )</td>
<td>( x_1 y_0 )</td>
<td>( x_2 y_0 )</td>
<td>( x_3 y_0 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- paper-and-pencil intuition;
- on-the-fly “truncation”;
- term-times-expansion products, \( x_i \cdot y \);
- error correction term, \( \pi_r \).
Example: $n = 4, m = 3$ and $r = 4$

- $\left\lfloor \frac{r \cdot p}{b} \right\rfloor + 2$ containers of size $b$ (s.t. $3b > 2p$);
- $b + c = p - 1$, s.t. we can add $2^c$ numbers without error ($\text{binary64} \rightarrow b = 45$, $\text{binary32} \rightarrow b = 18$);
- starting exponent $e = e_{x_0} + e_{y_0}$;
- each bin's LSB has a fixed weight;
- bins initialized with $1.5 \cdot 2^{e - (i+1)b+p-1}$.
Example: \( n = 4, m = 3 \) and \( r = 4 \)

- \( \lceil \frac{r \cdot p}{b} \rceil + 2 \) containers of size \( b \) (s.t. \( 3b > 2p \));
- \( b + c = p - 1 \), s.t. we can add \( 2^c \) numbers without error (\( \text{binary64} \rightarrow b = 45, \text{binary32} \rightarrow b = 18 \));
- starting exponent \( e = e_{x_0} + e_{y_0} \);
- each bin's LSB has a fixed weight;
- bins initialized with \( 1.5 \cdot 2^{e-(i+1)b+p-1} \);
- the number of leading bits, \( \ell \);
- accumulation done using a \textit{Fast2Sum} and addition [Rump09].
Example: \( n = 4, m = 3 \) and \( r = 4 \)
Example: \( n = 4, m = 3 \) and \( r = 4 \)

- Subtract initial value;
- Apply renormalization step to \( B \):
  - Fast2Sum and branches;
  - Render the result ulp-nonoverlapping.
Example: \( n = 4, m = 3 \) and \( r = 4 \)

- subtract initial value;
- apply renormalization step to \( B \):
  - \textit{Fast2Sum} and branches;
  - render the result \textit{ulp}-nonoverlapping.
Error bound

Exact result computed as:

\[ \sum_{i=0; j=0}^{n-1; m-1} x_i y_j = \sum_{k=0}^{m+n-2} \sum_{i+j=k} x_i y_j. \]

On-the-fly “truncation” → three error sources:
Error bound

Exact result computed as:

\[
\sum_{i=0; j=0}^{n-1; m-1} x_i y_j = \sum_{k=0}^{m+n-2} \sum_{i+j=k} x_i y_j.
\]

On-the-fly “truncation” → three error sources:

- discarded partial products:
  - we add only the first \( \sum_{k=1}^{r+1} k \):
  \[
  \sum_{k=r+1}^{m+n-2} \sum_{i+j=k} x_i y_j \leq |x_0 y_0| 2^{-(p-1)(r+1)} \left[ \frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2} + \frac{m+n-r-2}{1-2^{-(p-1)}} \right];
  \]
Error bound

Exact result computed as:

\[
\sum_{i=0; j=0}^{n-1; m-1} x_i y_j = \sum_{k=0}^{m+n-2} \sum_{i+j=k} x_i y_j.
\]

On-the-fly “truncation” → three error sources:

- **discarded partial products:**
  - we add only the first \( \sum_{k=1}^{r+1} k \);
  
  \[
  - \sum_{k=r+1}^{m+n-2} \sum_{i+j=k} x_i y_j \leq |x_0 y_0| 2^{-(p-1)(r+1)} \left[ \frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2} + \frac{m+n-r-2}{1-2^{-(p-1)}} \right];
  \]

- **discarded rounding errors:**
  - we compute the last \( r+1 \) partial products using simple FP multiplication;
  - we neglect \( r+1 \) error terms bounded by \( |x_0 y_0| 2^{-(p-1)r} \cdot 2^{-p}; \)
Error bound

Exact result computed as:

$$\sum_{i=0; j=0}^{n-1; m-1} x_i y_j = \sum_{k=0}^{m+n-2} \sum_{i+j=k} x_i y_j.$$ 

On-the-fly “truncation” → three error sources:

- discarded partial products:
  - we add only the first $$\sum_{k=1}^{r+1} k$$; 
  - $$\sum_{k=r+1}^{m+n-2} \sum_{i+j=k} x_i y_j \leq |x_0y_0| 2^{-(p-1)(r+1)} \left[ \frac{-2^{-(p-1)}}{(1-2^{-(p-1)})^2} + \frac{m+n-r-2}{1-2^{-(p-1)}} \right]$$;

- discarded rounding errors:
  - we compute the last $$r+1$$ partial products using simple FP multiplication;
  - we neglect $$r+1$$ error terms bounded by $$|x_0y_0| 2^{-(p-1)r} \cdot 2^{-p}$$;

- error caused by the renormalization step:
  - error less or equal to $$|x_0y_0| 2^{-(p-1)r}.$$
Error bound

Exact result computed as:

\[
\sum_{i=0; j=0}^{n-1;m-1} x_i y_j \leq \sum_{k=0}^{m+n-2} \sum_{i+j=k} x_i y_j.
\]

On-the-fly “truncation” → three error sources:

- **discarded partial products:**
  - we add only the first \( \sum_{k=1}^{r+1} k; \)
  \[
  \sum_{k=r+1}^{m+n-2} \sum_{i+j=k} x_i y_j \leq |x_0 y_0| 2^{-(p-1)(r+1)} \left[ \frac{-2^{-(p-1)}}{1 - 2^{-(p-1)}} + \frac{m+n-r-2}{1 - 2^{-(p-1)}} \right];
  \]

- **discarded rounding errors:**
  - we compute the last \( r + 1 \) partial products using simple FP multiplication;
  - we neglect \( r + 1 \) error terms bounded by \( |x_0 y_0| 2^{-(p-1)r} \cdot 2^{-p}; \)

- **error caused by the renormalization step:**
  - error less or equal to \( |x_0 y_0| 2^{-(p-1)r}. \)

Tight error bound:

\[
|x_0 y_0| 2^{-(p-1)r} \left[ 1 + (r + 1)2^{-p} + 2^{-(p-1)} \left( \frac{-2^{-(p-1)}}{1 - 2^{-(p-1)}} + \frac{m + n - r - 2}{1 - 2^{-(p-1)}} \right) \right].
\]
Comparison

Table: Worst case FP operation count when the input and output expansions are of size $r$.

<table>
<thead>
<tr>
<th>$r$</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>New algorithm</td>
<td>138</td>
<td>261</td>
<td>669</td>
<td>2103</td>
</tr>
<tr>
<td>Priest’s mul. [Pri91]</td>
<td>3174</td>
<td>16212</td>
<td>87432</td>
<td>519312</td>
</tr>
</tbody>
</table>
Comparison

Table: Worst case FP operation count when the input and output expansions are of size $r$.

<table>
<thead>
<tr>
<th>$r$</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>New algorithm</td>
<td>138</td>
<td>261</td>
<td>669</td>
<td>2103</td>
</tr>
<tr>
<td>Priest's mul.[Pri91]</td>
<td>3174</td>
<td>16212</td>
<td>87432</td>
<td>519312</td>
</tr>
</tbody>
</table>

Table: Performance in Mops/s for multiplying two FP expansions on a Tesla K40c GPU, using CUDA 7.5 software architecture, running on a single thread of execution. * precision not supported.

<table>
<thead>
<tr>
<th>$d_x, d_y, d_r$</th>
<th>New algorithm</th>
<th>QD</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 2, 2</td>
<td>0.027</td>
<td>0.1043</td>
</tr>
<tr>
<td>1, 2, 2</td>
<td>0.365</td>
<td>0.1071</td>
</tr>
<tr>
<td>3, 3, 3</td>
<td>0.0149</td>
<td>*</td>
</tr>
<tr>
<td>2, 3, 3</td>
<td>0.0186</td>
<td>*</td>
</tr>
<tr>
<td>4, 4, 4</td>
<td>0.0103</td>
<td>0.0174</td>
</tr>
<tr>
<td>1, 4, 4</td>
<td>0.0215</td>
<td>0.0281</td>
</tr>
<tr>
<td>2, 4, 4</td>
<td>0.0142</td>
<td>*</td>
</tr>
<tr>
<td>8, 8, 8</td>
<td>0.0034</td>
<td>*</td>
</tr>
<tr>
<td>4, 8, 8</td>
<td>0.0048</td>
<td>*</td>
</tr>
<tr>
<td>16, 16, 16</td>
<td>0.001</td>
<td>*</td>
</tr>
</tbody>
</table>
Conclusions

Available online at: http://homepages.laas.fr/mmjoldes/campary/.

- algorithm with strong regularity;
- based on partial products accumulation;
- uses a fixed-point structure that is floating-point friendly;
- thorough error analysis and tight error bound;
- natural fit for GPUs;
- proved to be too complex for small precisions;
- performance gains with increased precision.
Comparison

**Table**: Performance in Mops/s for multiplying two FP expansions on the CPU; \(d_x\) and \(d_y\) represent the number of terms in the input expansions and \(d_r\) is the size of the computed result. * precision not supported

<table>
<thead>
<tr>
<th>(d_x, d_y, d_r)</th>
<th>New algorithm</th>
<th>QD</th>
<th>MPFR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 2, 2</td>
<td>11.69</td>
<td>99.16</td>
<td>18.64</td>
</tr>
<tr>
<td>1, 2, 2</td>
<td>14.96</td>
<td>104.17</td>
<td>19.85</td>
</tr>
<tr>
<td>3, 3, 3</td>
<td>6.97</td>
<td>*</td>
<td>12.1</td>
</tr>
<tr>
<td>2, 3, 3</td>
<td>8.62</td>
<td>*</td>
<td>13.69</td>
</tr>
<tr>
<td>4, 4, 4</td>
<td>4.5</td>
<td>5.87</td>
<td>10.64</td>
</tr>
<tr>
<td>1, 4, 4</td>
<td>8.88</td>
<td>15.11</td>
<td>14.1</td>
</tr>
<tr>
<td>2, 4, 4</td>
<td>6.38</td>
<td>9.49</td>
<td>13.44</td>
</tr>
<tr>
<td>8, 8, 8</td>
<td>1.5</td>
<td>*</td>
<td>6.8</td>
</tr>
<tr>
<td>4, 8, 8</td>
<td>2.04</td>
<td>*</td>
<td>9.15</td>
</tr>
<tr>
<td>16, 16, 16</td>
<td>0.42</td>
<td>*</td>
<td>2.55</td>
</tr>
</tbody>
</table>