Towards fast and certified multiple-precision libraries

Valentina Popescu

under the supervision of:
Mioara Joldes and Jean-Michel Muller

École Normale Supérieure de Lyon, AriC Team, Laboratoire de l’Informatique du Parallélisme
Do we actually need more precision?
Do we actually need more precision?

Most watched video in the world.
Do we actually need more precision?

Breaking the Youtube counter:
Do we actually need more precision?

Breaking the Youtube counter:
- 32-bit integer register $\rightarrow$ counts up to $2,147,483,647$;
Do we actually need more precision?

**Breaking the Youtube counter:**

- 32-bit integer register → counts up to 2,147,483,647;
- in 2014 the video **Psy - Gangnam Style** touched the limit;

*courtesy of http://www.reddit.com/r/ProgrammerHumor/*
Do we actually need more precision?

Breaking the Youtube counter:

- 32-bit integer register → counts up to 2,147,483,647;
- in 2014 the video Psy - Gangnam Style touched the limit;
- update to 64-bit register → counts up to 9,223,372,036,854,775,808 (more than 9 quintillion).

*courtesy of http://www.reddit.com/r/ProgrammerHumor/
Floating-point (FP) arithmetic

A real number $X$ is approximated in a machine by a rational

$$x = M_x \cdot 2^{e_x - p + 1},$$

where

- $M_x$ is the **significand**, a $p$-digit signed integer in radix 2 s.t.

  $$2^{p-1} \leq |M_x| \leq 2^p - 1;$$

- $e_x$ is the **exponent**, a signed integer that satisfies

  $$e_{\text{min}} \leq e_x \leq e_{\text{max}}.$$
A real number $X$ is approximated in a machine by a rational

$$x = M_x \cdot 2^{e_x-p+1},$$

where

- $M_x$ is the **significand**, a $p$-digit signed integer in radix $2$ s.t.

  $$2^{p-1} \leq |M_x| \leq 2^p - 1;$$

- $e_x$ is the **exponent**, a signed integer that satisfies

  $$e_{min} \leq e_x \leq e_{max}.$$ 

### Single-precision format (*binary32*):

\[
\begin{array}{c|cc}
1 & 8 & 23 \\
\hline
s & e & m \\
\end{array}
\]

### Double-precision format (*binary64*):

\[
\begin{array}{c|cc}
1 & 11 & 52 \\
\hline
s & e & m \\
\end{array}
\]

→ Implicit bit that is not stored.
IEEE-754 2008 standard

- 4 rounding modes: RD, RU, RZ, RN;
- Correct rounding for: $+, -, \times, \div, \sqrt{}$ (return what we would get by infinitely precise operations followed by rounding);
- Fused Multiply-Add (FMA):
  $\circ(a \times b + c)$.

\[
\begin{align*}
RD(x) & \\
RN(x) & \\
RZ(x) & \\
\text{RU}(x) & \quad x
\end{align*}
\]
A serious example: sine function in CRLibm

\[
\sin(y) \downarrow \\
\text{reduced argument} \quad x \in \left[ -\frac{\pi}{512}, \frac{\pi}{512} \right] \subset \left[ -\frac{2}{7}, \frac{2}{7} \right] \\
\downarrow \text{represent } x \text{ as } x_h + x_l \\
\downarrow \text{evaluate } P(x) = x + x^3 \cdot (s_3 + x^2 \cdot (s_5 + x^2 \cdot s_7)))
\]

where \( s_1, s_3 \) and \( s_5 \) are floating-point numbers
A serious example: sine function in CRLibm

Using polynomial approximations.
A serious example: sine function in CRLibm

Using polynomial approximations.

\[ \sin(y) \]
Using polynomial approximations.

\[ \sin(y) \]

\[
\downarrow
\]

reduced argument
\[ x \in \left[ -\frac{\pi}{512}, \frac{\pi}{512} \right] \subset \left[ -2^{-7}, 2^{-7} \right] \]
A serious example: sine function in CRLibm

Using polynomial approximations.

\[
\sin(y) \\
\downarrow \\
\text{reduced argument}
\]

\[x \in [-\pi/512, \pi/512] \subset [-2^{-7}, 2^{-7}]\]

\[x \text{ is irrational} \Rightarrow \text{the range reduction step needs to return a number more accurate than a binary64, such that the intermediary output accuracy for } P(x) \text{ allows for subsequent correct rounding of } \sin(x).\]
A serious example: sine function in CRLibm

Using polynomial approximations.

\[
\sin(y) \Downarrow \text{reduced argument} \\
x \in [-\pi/512, \pi/512] \subset [-2^{-7}, 2^{-7}] \Downarrow \text{represent } x \text{ as } x_h + x_l
\]
A serious example: sine function in CRLibm

Using polynomial approximations.

\[ \sin(y) \]

↓

Reduced argument

\[ x \in [-\pi/512, \pi/512] \subset [-2^{-7}, 2^{-7}] \]

↓

Represent \( x \) as \( x_h + x_l \)

↓

Evaluate

\[ P(x) = x + x^3 \cdot (s_3 + x^2 \cdot (s_5 + x^2 \cdot s_7)) \],

where \( s_1, s_3 \) and \( s_5 \) are floating-point numbers
A serious example: sine function in CRLibm

Numerical example

\[
\sin(0.5) \\
\downarrow \\
x = \frac{1}{2} - \frac{41}{256}
\]

\[
\downarrow \\
x_h = \frac{-7253486725817229}{261} \\
x_l = \frac{-508039184604813}{2112}
\]

\[
\downarrow \\
P_{\text{eval}}(x_h + x_l) = (x_h + x_l) + (x_h + x_l)(3 \cdot (s_3 + (x_h + x_l)(2 \cdot (s_5 + (x_h + x_l)(2 \cdot s_7)))))
\]
Numerical example

\[ \sin(0.5) \]
A serious example: sine function in CRLibm

Numerical example

\[
\sin(0.5) \\
\downarrow \\
x = \frac{1}{2} - 41 \frac{\pi}{256}
\]
A serious example: sine function in CRLibm

Numerical example

\[
\sin(0.5)
\]

\[
\downarrow
\]

\[
x = \frac{1}{2} - 41 \frac{\pi}{256}
\]

\[
\downarrow
\]

\[
x_h = -7253486725817229/2^{61}
\]
\[
x_l = -508039184604813/2^{112}
\]
A serious example: sine function in CRLibm

Numerical example

\[
sin(0.5)
\]

\[
\downarrow
\]

\[
x = \frac{1}{2} - 41 \frac{\pi}{256}
\]

\[
\downarrow
\]

\[
x_h = -7253486725817229/2^{61}
\]

\[
x_l = -508039184604813/2^{112}
\]

\[
\downarrow
\]

\[
P_{eval}(x_h + x_l) = (x_h + x_l) + (x_h + x_l)^3 \cdot (s_3 + (x_h + x_l)^2 \cdot (s_5 + (x_h + x_l)^2 \cdot s_7))
\]
A serious example: sine function in CRLibm

Numerical example

\[
\sin(0.5)
\]

\[
\downarrow
\]

\[
x = \frac{1}{2} - 41 \frac{\pi}{256}
\]

\[
\downarrow
\]

\[
x_h = -7253486725817229/2^{61}
\]

\[
x_l = -508039184604813/2^{112}
\]

\[
\downarrow
\]

\[
P_{eval}(x_h + x_l) = (x_h + x_l) + (x_h + x_l)^3 \cdot (s_3 + (x_h + x_l)^2 \cdot (s_5 + (x_h + x_l)^2 \cdot s_7))
\]

Poor accuracy! With this order of operations, the addition \(x_h + x_l\) returns \(x_h\) ⇒ the information held by \(x_l\) is lost.
A serious example: sine function in CRLibm

\[ P_{eval}(x_h + x_l) = (x_h + x_l) + (x_h + x_l)^3 \cdot (s_3 + (x_h + x_l)^2 \cdot (s_5 + (x_h + x_l)^2 \cdot s_7)) \]
A serious example: sine function in CRLibm

\[ P_{\text{eval}}(x_h + x_l) = (x_h + x_l) + (x_h + x_l)^3 \cdot (s_3 + (x_h + x_l)^2 \cdot (s_5 + (x_h + x_l)^2 \cdot s_7)) \]
A serious example: sine function in CRLibm

\[ P_{\text{eval}}(x_h + x_l) = (x_h + x_l) + (x_h + x_l)^3 \cdot (s_3 + (x_h + x_l)^2 \cdot (s_5 + (x_h + x_l)^2 \cdot s_7)) \]

\[ \downarrow \]

\[ s = x_l + (x_h \cdot x_h \cdot x_h \cdot (s_3 + (x_h \cdot x_h \cdot (s_5 + (x_h \cdot x_h \cdot s_7)))))) \]

\[ P'_{\text{eval}}(x_h + x_l) = \text{Fast2Sum}(x_h, s) * \]

*computes the exact sum of two floating-point numbers.*
A serious example: sine function in CRLibm

\[
P_{\text{eval}}(x_h + x_l) = (x_h + x_l) + (x_h + x_l)^3 \cdot (s_3 + (x_h + x_l)^2 \cdot (s_5 + (x_h + x_l)^2 \cdot s_7))
\]

\[
P_{\text{eval}} = -3626737381554291/2^{60} \rightarrow \text{54 bits of precision}
\]

\[
s = x_l + (x_h \cdot x_h \cdot x_h \cdot (s_3 + (x_h \cdot x_h \cdot (s_5 + (x_h \cdot x_h \cdot s_7))))))
\]

\[
P'_{\text{eval}}(x_h + x_l) = \text{Fast2Sum}(x_h, s)^*
\]

\[
P'_{\text{eval}} = -7253474763108583/2^{61} + 82031/2^{79} \rightarrow \text{72 bits of precision}
\]

*computes the exact sum of two floating-point numbers.
Dynamical systems field:
- 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).

Optimization problems in experimental mathematics:
- computation of kissing numbers,
- bounds for binary codes,
- problems in control theory and structural design (e.g., the wing of Airbus A380).
- problems in quantum chemistry/information, etc.
What we need

1. Need more precision than standard available
   –few hundred bits–
What we need

1. Need more precision than standard available
   – few hundred bits –

2. Need massive parallel computations: high performance computing (HPC)
   – Graphics Processing Units –
What we need

1. Need more precision than standard available
   - few hundred bits –

2. Need massive parallel computations: high performance computing (HPC)
   - Graphics Processing Units –

![Logo](CudA_Multiple_Precision_Arithmetic_Library)
Table of contents

- Graphics Processing Units
- Extending the precision
- A look into CAMPARY
- Applications
- Conclusions
Graphics Processing Units - GPUs

- increasingly used for scientific computing

  –General Purpose GPU Computing–

- easily programmable (usually): CUDA, OpenCL, etc.
Graphics Processing Units - GPUs

- increasingly used for scientific computing
  - General Purpose GPU Computing–
- easily programmable (usually): CUDA, OpenCL, etc.

- SIMD (Single Instruction Multiple Data) Architecture;
  - with N multiprocessors, each with M cores;
  - the cores on each multiprocessor share an Instruction Unit.
a kernel is a piece of code executed on the device by a single thread;

the threads are grouped into warps (32 threads), which are grouped into blocks and these ones into grids;

each kernel has access to variables that define its position: (gridDim, blockIdx, blockDim, threadIdx);

threads communicate through lockstep, warp vote, shuffle instructions and shared/global memory,
GPUs - floating-point characteristics

- conform to IEEE-754 standard;
GPUs - floating-point characteristics

- conform to IEEE-754 standard;
- (+, −, *, /, √) support the four rounding modes;
- dynamic rounding mode change is supported:

  __dadd_rn(a,b)=RN(a+b);
GPUs - floating-point characteristics

- conform to IEEE-754 standard;
- \((+,-,\ast,/,\sqrt{\cdot})\) support the four rounding modes;
- dynamic rounding mode change is supported:
  \[
  \text{__dadd\_rn}(a,b) = \text{RN}(a+b);
  \]
- support for FMA:
  \[
  \text{__dfma\_rn}\ (a,b,c) = \text{RN}(A \ast B + C);
  \]
GPUs - floating-point characteristics

- conform to IEEE-754 standard;
- (+, −, *, /, √) support the four rounding modes;
- dynamic rounding mode change is supported:
  \[
  \_\_dadd\_rn(a,b) = RN(a+b);
  \]
- support for FMA:
  \[
  \_\_dfma\_rn(a,b,c) = RN(A \ast B + C);
  \]
- hardware acceleration for some elementary functions.
GPUs - floating-point characteristics

- conform to IEEE-754 standard;
- \((+, -, *, /, \sqrt{ })\) support the four rounding modes;
- dynamic rounding mode change is supported:
  \[\text{__dadd\_rn}(a,b) = \text{RN}(a+b)\];
- support for FMA:
  \[\text{__dfma\_rn}(a,b,c) = \text{RN}(A \times B + C)\];
- hardware acceleration for some elementary functions.

\[\rightarrow\] Implement single- and double-precision.
<table>
<thead>
<tr>
<th>Table of contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Graphics Processing Units</td>
</tr>
<tr>
<td>• <strong>Extending the precision</strong></td>
</tr>
<tr>
<td>• A look into CAMPARY</td>
</tr>
<tr>
<td>• Applications</td>
</tr>
<tr>
<td>• Conclusions</td>
</tr>
</tbody>
</table>
Extending the precision

- **multiple-digit representation**: a number is represented by a sequence of digits coupled with a single exponent;

\[ M \hat{t} \]
Extending the precision

**multiple-digit representation**: a number is represented by a sequence of digits coupled with a single exponent:

```
M   t
```

**multiple-term representation**: a number is expressed as the unevaluated sum of several FP numbers: double-double (DD), triple-double (TD), quad-double (QD), etc. or FP expansion when made up with arbitrary precision:

```
u_0   u_1   \ldots   u_{n-1}
```
**Extending the precision**

- **multiple-digit representation**: a number is represented by a sequence of digits coupled with a single exponent;

  
  \[
  \begin{array}{c}
  M \\
  \hline
  \hline
  t
  \end{array}
  \]

  - millions of bits of precision
  - heavy alternative for moderate precision

- **multiple-term representation**: a number is expressed as the unevaluated sum of several FP numbers: double-double (DD), triple-double (TD), quad-double (QD), etc. or FP expansion when made up with arbitrary precision:

  \[
  \begin{array}{c}
  u_0 \\
  \hline
  \hline
  u_1 \\
  \hline
  \hline
  \cdots
  \hline
  \hline
  u_{n-1}
  \end{array}
  \]

  - uses optimized floating-point units
  - "tricky" error analysis
multiple-digit representation: a number is represented by a sequence of digits coupled with a single exponent;

\[ M \times 10^t \]

+ millions of bits of precision
- heavy alternative for moderate precision

- GNU MPFR (based on GMP) → not ported on GPU
- ARPREC/GARPREC and CUMP → tuned for big array operations

multiple-term representation: a number is expressed as the unevaluated sum of several FP numbers: double-double (DD), triple-double (TD), quad-double (QD), etc. or FP expansion when made up with arbitrary precision:

\[ u_0 + u_1 + \cdots + u_{n-1} \]

+ uses optimized floating-point units
- “tricky” error analysis

- doubledouble → no longer maintained
- QD/GQD → limited to DD and QD
CAMPARY (CudA Multiple Precision ARithmetic librarY)

Available online at: http://homepages.laas.fr/mmjoldes/campary/.
CAMPARY (CudA Multiple Precision ARithmetic librarY)

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

- moderate arbitrary precision –few hundred bits–
CAMPARY (CudA Multiple Precision ARithmetic librarY)

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

- moderate arbitrary precision –few hundred bits–
- targets both CPU and GPU (compilers: GCC, NVCC)
- underlying FP format: binary32 (up to 12 terms) or binary64 (up to 39 terms)
CAMPARY (CudA Multiple Precision ARithmetic librarY)

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

- moderate arbitrary precision –few hundred bits–
- targets both CPU and GPU (compilers: GCC, NVCC)
- underlying FP format: binary32 (up to 12 terms) or binary64 (up to 39 terms)
- sequential algorithms: all basic operations (+/−, ×, ÷, √)
  - accurate algorithms - tight error bound
  - “quick-and-dirty” algorithms - does not consider corner cases
  * optimized algorithms for double-word arithmetic
CAMPARY (CudA Multiple Precision ARithmetic librarY)

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

- moderate arbitrary precision –few hundred bits–
- targets both CPU and GPU (compilers: GCC, NVCC)
- underlying FP format: binary32 (up to 12 terms) or binary64 (up to 39 terms)
- sequential algorithms: all basic operations (+/−, ×, ÷, √)
  - accurate algorithms - tight error bound
  - “quick-and-dirty” algorithms - does not consider corner cases
  * optimized algorithms for double-word arithmetic
- GPU-tuned parallel algorithms: +/−, ×
CAMPARY (CudA Multiple Precision ARithmetic librarY)

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

- moderate arbitrary precision –few hundred bits–
- targets both CPU and GPU (compilers: GCC, NVCC)
- underlying FP format: binary32 (up to 12 terms) or binary64 (up to 39 terms)
- sequential algorithms: all basic operations (+, −, ×, ÷, √)
  - accurate algorithms - tight error bound
  - “quick-and-dirty” algorithms - does not consider corner cases
  * optimized algorithms for double-word arithmetic
- GPU-tuned parallel algorithms: +, −, ×
- thorough correctness proofs and error analysis
Joint work with:

Sylvie Boldo
Sylvain Collange
Mioara Joldes
Olivier Marty
Jean-Michel Muller
Peter Tang
Warwick Tucker
Table of contents

- Graphics Processing Units
- Extending the precision
- A look into CAMPARY
- Applications
- Conclusions
- **unit in the last place** (Goldberg’s definition):

\[ \text{ulp}(x) = 2^{e_x - p + 1}. \]

- **unit in the last significant place:**

\[ \text{uls}(x) = \text{ulp}(x) \cdot 2^{z_x}, \]

where \( z_x \) is the number of trailing zeros at the end of \( M_x \).

→ Allow for error handling.
Let \( a \) and \( b \) be FP numbers. Algorithm 2Sum computes two FP numbers \( s \) and \( e \) that satisfy the following:

- \( s + e = a + b \) exactly;
- \( s = \text{RN} \,(a + b) \).

( \text{RN} \text{ stands for performing the operation in rounding to nearest rounding mode.} )

\[
\begin{align*}
\text{Algorithm 1 (2Sum} \,(a, b)\,\text{)} \\
\text{s} & \leftarrow \text{RN} \,(a + b) \\
t & \leftarrow \text{RN} \,(s - b) \\
e & \leftarrow \text{RN} \,(\text{RN} \,(a - t) + \text{RN} \,(b - \text{RN} \,(s - t))) \\
\text{return} \,(s, e)
\end{align*}
\]

→ 6 FP operations (proved to be optimal unless we have information on the ordering of \(|a|\) and \(|b|\))
Let $a$ and $b$ be FP numbers that satisfy $e_a \geq e_b$, where $e_a$ and $e_b$ are the exponents of $a$ and $b$, respectively. Algorithm Fast2Sum computes two FP numbers $s$ and $e$ that satisfy the following:

- $s + e = a + b$ exactly;
- $s = \text{RN}(a + b)$.

(RN stands for performing the operation in rounding to nearest rounding mode.)

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

Algorithm 3 (VecSum \((x_0, \ldots, x_{n-1})\))

\[
\text{for } i \leftarrow n - 1 \text{ to } 1 \text{ do} \\
\quad (s_{i-1}, e_i) \leftarrow \text{2Sum}(x_i, x_{i-1}) \\
\text{end for} \\
\quad e_0 \leftarrow s_0 \\
\text{return } e_0, \ldots, e_{n-1}
\]

Properties:
- \(x_0 + \cdots + x_{n-1} = e_0 + \cdots + e_{n-1}\)
- \(e_0 = \text{RN}(x_0 + \text{RN}(x_1 + \text{RN}(\cdots + \text{RN}(x_{n-2} + x_{n-1})))))\)
Addition of double-word numbers

**Input:** $x = (x_h, x_\ell)$ and $y = (y_h, y_\ell)$ two DW numbers.

**Output:** $z = (z_h, z_\ell)$, their sum as a DW number.
Addition of double-word numbers

**Input:** $x = (x_h, x_\ell)$ and $y = (y_h, y_\ell)$ two DW numbers.

**Output:** $z = (z_h, z_\ell)$, their sum as a DW number.

**AccurateDWPlusDW**

Tight and rigorous error bounds for basic building blocks of double-word arithmetic, joint work with M. Joldes, and, J.-M. Muller. Accepted for publication in *ACM Transactions on Mathematical Software*.
Addition of double-word numbers

**Input:** \( x = (x_h, x_\ell) \) and \( y = (y_h, y_\ell) \) two DW numbers.

**Output:** \( z = (z_h, z_\ell) \), their sum as a DW number.

---

**AccurateDWPlusDW**

---

**Tight and rigorous error bounds for basic building blocks of double-word arithmetic**, joint work with M. Joldes, and, J.-M. Muller. Accepted for publication in *ACM Transactions on Mathematical Software*. 
Addition of double-word numbers

**Input:** \(x = (x_h, x_\ell)\) and \(y = (y_h, y_\ell)\) two DW numbers.

**Output:** \(z = (z_h, z_\ell)\), their sum as a DW number.

---

**Tight and rigorous error bounds for basic building blocks of double-word arithmetic**, joint work with M. Joldes, and J.-M. Muller. Accepted for publication in *ACM Transactions on Mathematical Software*. 

---

**AccurateDWPlusDW**
Addition of double-word numbers

**Input:** $x = (x_h, x_\ell)$ and $y = (y_h, y_\ell)$ two DW numbers.

**Output:** $z = (z_h, z_\ell)$, their sum as a DW number.

**AccurateDWPlusDW**

**SloppyDWPlusDW**

---

**Tight and rigorous error bounds for basic building blocks of double-word arithmetic**, joint work with M. Joldes, and, J.-M. Muller. Accepted for publication in *ACM Transactions on Mathematical Software*. 
Addition of double-word numbers

Input: \( x = (x_h, x_\ell) \) and \( y = (y_h, y_\ell) \) two DW numbers.

Output: \( z = (z_h, z_\ell) \), their sum as a DW number.

**AccurateDWPlusDW**

**SloppyDWPlusDW**

Tight and rigorous error bounds for basic building blocks of double-word arithmetic, joint work with M. Joldes, and, J.-M. Muller. Accepted for publication in *ACM Transactions on Mathematical Software*. 
Addition of double-word numbers

Input: \( x = (x_h, x_\ell) \) and \( y = (y_h, y_\ell) \) two DW numbers.
Output: \( z = (z_h, z_\ell) \), their sum as a DW number.

AccurateDWPlusDW

SloppyDWPlusDW

Tight and rigorous error bounds for basic building blocks of double-word arithmetic, joint work with M. Joldes, and, J.-M. Muller. Accepted for publication in ACM Transactions on Mathematical Software.
Addition of double-word numbers

**Input:** $x = (x_h, x_\ell)$ and $y = (y_h, y_\ell)$ two DW numbers.

**Output:** $z = (z_h, z_\ell)$, their sum as a DW number.

---

**AccurateDWPlusDW**

---

**SloppyDWPlusDW**

---

Tight and rigorous error bounds for basic building blocks of double-word arithmetic, joint work with M. Joldes, and, J.-M. Muller. Accepted for publication in *ACM Transactions on Mathematical Software.*
Addition of double-word numbers

**Input:** \( x = (x_h, x_\ell) \) and \( y = (y_h, y_\ell) \) two DW numbers.

**Output:** \( z = (z_h, z_\ell) \), their sum as a DW number.

**AccurateDWPlusDW**

**SloppyDWPlusDW**

\[ \varepsilon = \frac{|x+y-z|}{|x+y|} \]

- conjectured: \( \varepsilon \leq 2 \times 2^{-2p} \) (incorrect)
- largest observed: \( \varepsilon = 2.25 \times 2^{-2p} \)
- proved: \( \varepsilon \leq 3 \times 2^{-2p} + 13 \times 2^{-3p} \)

- largest observed: \( \varepsilon = 1 \)
- proved: N/A
Addition of double-word numbers

**Input:** \( x = (x_h, x_\ell) \) and \( y = (y_h, y_\ell) \) two DW numbers.

**Output:** \( z = (z_h, z_\ell) \), their sum as a DW number.

---

**AccurateDWPlusDW**

- **Computation:**
  - Fast 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum

**SloppyDWPlusDW**

- **Computation:**
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum
  - 2Sum

**Error Bound:**

\[ \varepsilon = \frac{|x+y-z|}{|x+y|} \]

- Conjectured: \( \varepsilon \leq 2 \times 2^{-2p} \) (incorrect)
- Largest observed: \( \varepsilon = 2.25 \times 2^{-2p} \)
- Proved: \( \varepsilon \leq 3 \times 2^{-2p} + 13 \times 2^{-3p} \)

- Computed sum: \( z_h + z_\ell = 0 \)
- Exact sum: \( x + y = 2^{-2p} \)

- Largest observed: \( \varepsilon = 1 \)
- Proved: N/A
Floating-point expansions

Drawback: more than one representation.

\[ p = 5 \text{ (in radix 2)} \]

The real number \( R = 1.11010011 \times 2^{-1} \) can be represented as:

\[
R = x_0 + x_1 + x_2:
\]

\[
\begin{align*}
  x_0 &= 1.1000 \times 2^{-1}; \\
  x_1 &= 1.0010 \times 2^{-3}; \\
  x_2 &= 1.0110 \times 2^{-6}.
\end{align*}
\]

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

\[
\begin{align*}
  z_0 &= 1.1101 \times 2^{-1}; \\
  z_1 &= 1.1000 \times 2^{-8}.
\end{align*}
\]

Least compact \( R = y_0 + y_1 + y_2 + y_3 + y_4 + y_5: \)

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

\[ \rightarrow \text{non-overlapping expansions} \]
Floating-point expansions

Drawback: more than one representation.

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

The real number $R = 1.11010011e - 1$ can be represented as:

$$R = x_0 + x_1 + x_2:$$

$$x_0 = 1.1000e - 1;$$

$$x_1 = 1.0010e - 3;$$

$$x_2 = 1.0110e - 6.$$
Floating-point expansions

**Drawback:** more than one representation.

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

The real number \( R = 1.11010011e - 1 \) can be represented 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** \( R = z_0 + z_1: \)
\[
\begin{align*}
z_0 &= 1.1101e - 1; \\
z_1 &= 1.1000e - 8.
\end{align*}
\]
Floating-point expansions

Drawback: more than one representation.

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

The real number $R = 1.11010011e - 1$ can be represented as:

Most compact $R = z_0 + z_1$:

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

Least compact $R = y_0 + y_1 + y_2 + y_3 + y_4 + y_5$:

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

Drawback: more than one representation.

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

The real number $R = 1.11010011e - 1$ can be represented 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 $R = z_0 + z_1$:
$$\begin{align*}
z_0 &= 1.1101e - 1; \\
z_1 &= 1.1000e - 8. \\
\end{align*}$$

Least compact $R = y_0 + y_1 + y_2 + y_3 + y_4 + y_5$:
$$\begin{align*}
y_0 &= 1.0000e - 1; \\
y_1 &= 1.0000e - 2; \\
y_2 &= 1.0000e - 3; \\
y_3 &= 1.0000e - 5; \\
y_4 &= 1.0000e - 8; \\
y_5 &= 1.0000e - 9; \\
\end{align*}$$

$\rightarrow$ non-overlapping expansions
Non-overlapping expansions

Considering a sequence of FP numbers $x_1, x_2, \ldots, x_n$, with $|x_1| > |x_2| > \ldots > |x_n|:$

- $\mathcal{P}$-nonoverlapping expansions (defined by Priest):
  
  $|x_{k+1}| < \text{ulp}(x_k)$ for every $1 \leq k \leq n - 1$

Example:

- $x_0 = 1.1010e - 2$
- $x_1 = 1.1101e - 7$
- $x_2 = 1.0100e - 12$
- $x_3 = 1.1000e - 18$
Non-overlapping expansions

Considering a sequence of FP numbers \( x_1, x_2, \ldots, x_n \), with \( |x_1| > |x_2| > \ldots > |x_n| \):

- \( P \)-nonoverlapping expansions (defined by Priest):
  \[ |x_{k+1}| < \text{ulp}(x_k) \] for every \( 1 \leq k \leq n - 1 \)

  Example:
  \[
  \begin{align*}
  x_0 &= 1.1010e - 2; \\
  x_1 &= 1.1101e - 7; \\
  x_2 &= 1.0100e - 12; \\
  x_3 &= 1.1000e - 18.
  \end{align*}
  \]

- \( S \)-nonoverlapping expansions (defined by Shewchuk):
  \[ |x_{k+1}| < \text{uls}(x_k) \] for every \( 1 \leq k \leq n - 1 \)

  Example:
  \[
  \begin{align*}
  x_0 &= 1.1000e - 1; \\
  x_1 &= 1.0100e - 3; \\
  x_2 &= 1.1001e - 7; \\
  x_3 &= 1.1010e - 12.
  \end{align*}
  \]
Non-overlapping expansions

Considering a sequence of FP numbers \( x_1, x_2, \ldots, x_n \), with \(|x_1| > |x_2| > \ldots > |x_n|\):

- \( \mathcal{P} \)-nonoverlapping expansions (defined by Priest):
  \[ |x_{k+1}| < \text{ulp}(x_k) \text{ for every } 1 \leq k \leq n - 1 \]

  Example:
  \[
  \begin{cases}
  x_0 = 1.1010e - 2; \\
  x_1 = 1.1101e - 7; \\
  x_2 = 1.0100e - 12; \\
  x_3 = 1.1000e - 18.
  \end{cases}
  \]

- \( S \)-nonoverlapping expansions (defined by Shewchuk):
  \[ |x_{k+1}| < \text{uls}(x_k) \text{ for every } 1 \leq k \leq n - 1 \]

  Example:
  \[
  \begin{cases}
  x_0 = 1.1000e - 1; \\
  x_1 = 1.0100e - 3; \\
  x_2 = 1.1001e - 7; \\
  x_3 = 1.1010e - 12;
  \end{cases}
  \]

- ulp-nonoverlapping expansions:
  \[ |x_{k+1}| \leq \text{ulp}(x_k) \text{ for every } 1 \leq k \leq n - 1 \]

  Example:
  \[
  \begin{cases}
  x_0 = 1.0010e - 1; \\
  x_1 = 1.0111e - 6; \\
  x_2 = 1.0000e - 10; \\
  x_3 = 1.0110e - 17;
  \end{cases}
  \]
Non-overlapping expansions

**Problem:** broken property after each operation
Problem: broken property after each operation
Non-overlapping expansions

**Problem:** broken property after each operation
Non-overlapping expansions

Problem: broken property after each operation
Non-overlapping expansions

Problem: broken property after each operation

(re-)normalize
Renormalization algorithm


Renormalization algorithm

**Input:**
- \( x = (x_0, \ldots, x_{n-1}) \), an array of FP numbers that overlap by at most \( d \leq p - 2 \) digits and that may contain interleaving 0s;
- \( m \), with \( 1 \leq m \leq n \), required number of output terms.

**Output:** "truncation" to \( m \) terms of a ulp-nonoverlapping FP expansion \( r = r_0, \ldots, r_{n-1} \) such that \( x_0 + \ldots + x_{n-1} = r \) and \( r_{i+1} \leq \text{ulp}(r_i) \), for all \( 0 \leq i < m - 1 \).

---


Formal verification of a floating-point expansion renormalization algorithm, joint work with S. Boldo, M. Joldes and, J.-M. Muller. In *Proceedings of the 8th International Conference on Interactive Theorem Proving (ITP 2017)*.
Input: $x = (x_0, \ldots, x_{n-1})$, an array of FP numbers that overlap by at most $d \leq p - 2$ digits and that may contain interleaving 0s.

Output: $e = (e_0, \ldots, e_{n-1})$, an $S$-nonoverlapping expansion that may contain interleaving 0s.

\[ \rightarrow \text{provided that } \frac{2^d}{1-2d-p} (1 + (n - 2)2^{-p}) \leq 2^{p-1} \]

Remark: we can use Fast2Sum($x_i, s_{i+1}$), because we have $|s_{i+1}| \leq 2^{p-1} \text{ulp} (x_i)$ (always holds in practice) and $\text{ulp} (x_i) \leq 2^{-p+1} |x_i|$. As a deduction $|s_{i+1}| \leq |x_i|$. 
First level (VecSum)

**Input:** \( x = (x_0, \ldots, x_{n-1}) \), an array of FP numbers that overlap by at most \( d \leq p - 2 \) digits and that may contain interleaving 0s.

**Output:** \( e = (e_0, \ldots, e_{n-1}) \), an \( S \)-nonoverlapping expansion that may contain interleaving 0s.

\[ \Rightarrow \text{provided that } \frac{2^d}{1-2d-p} (1 + (n-2)2^{-p}) \leq 2^{p-1} \]

**Remark:** we can use Fast2Sum\((x_i, s_{i+1})\), because we have \( |s_{i+1}| \leq 2^{p-1} \ulp (x_i) \) (always holds in practice) and \( \ulp (x_i) \leq 2^{-p+1} |x_i| \). As a deduction \( |s_{i+1}| \leq |x_i| \).
Second level (VecSumErrBranch)

Input:
- \(e = (e_0, \ldots, e_{n-1})\), an \(S\)-nonoverlapping expansion that may contain interleaving 0s;
- \(m + 1\), with \(1 \leq m < n\), the required number of output terms.

Output: \(r = (r_0, \ldots, r_{m-1})\), an ulp-nonoverlapping expansion, i.e., it satisfies
\[|r_{i+1}| \leq \text{ulp}(r_i)\] for all \(0 \leq i < m - 1\)

Remark: we can use Fast2Sum\((\varepsilon_i, e_{i+1})\), because either \(|e_{i+1}| < |\varepsilon_i|\), or \(\varepsilon_i = 0\), in which case we replace \(\varepsilon_i = r_{j-1}\), which is a multiple of \(2^{k_i}\) with \(|e_{i+1}| < 2^{k_i}\).
Second level (VecSumErrBranch)

**Input:**
- $e = (e_0, \ldots, e_{n-1})$, an $S$-nonoverlapping expansion that may contain interleaving 0s;
- $m + 1$, with $1 \leq m < n$, the required number of output terms.

**Output:** $r = (r_0, \ldots, r_{m-1})$, an ulp-nonoverlapping expansion, i.e., it satisfies $|r_{i+1}| \leq \text{ulp}(r_i)$ for all $0 \leq i < m - 1$

**Remark:** we can use $\text{Fast2Sum}(e_i, e_{i+1})$, because either $|e_{i+1}| < |e_i|$, or $e_i = 0$, in which case we replace $e_i = r_j - 1$, which is a multiple of $2^{k_i}$ with $|e_{i+1}| < 2^{k_i}$. 
Renormalization algorithm - example

VecSum

VecSumErrBranch
Arithmetic algorithms in CAMPARY

- 7 algorithms for all basic operations (+, −, ×, ÷, √) with arbitrary precision FP expansions;
- coupled with 2 re-normalization algorithms;
- 14 algorithms for +, −, ×, ÷ with DW numbers;
- 3 GPU-tuned parallel algorithms for +, −, × with arbitrary precision FP expansions.
Performance

CPU performance in Mop/s for the addition algorithms.

<table>
<thead>
<tr>
<th>$n, m, r$</th>
<th>CAMPARY accurate</th>
<th>“quick-and-dirty” $2^2$</th>
<th>Fast “quick-and-dirty” $2^2$</th>
<th>MPFR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 2, 2</td>
<td>114.9</td>
<td>208.9</td>
<td>208.9</td>
<td>40.5</td>
</tr>
<tr>
<td>2, 1, 2</td>
<td>229.8</td>
<td>229.8</td>
<td>229.8</td>
<td>42.2</td>
</tr>
<tr>
<td>3, 3, 3</td>
<td>34</td>
<td>27.3</td>
<td>40</td>
<td>31.5</td>
</tr>
<tr>
<td>3, 1, 3</td>
<td>75.3</td>
<td>75.3</td>
<td>75.3</td>
<td>22.2</td>
</tr>
<tr>
<td>4, 4, 4</td>
<td>19.3</td>
<td>16.9</td>
<td>23.2</td>
<td>28.2</td>
</tr>
<tr>
<td>4, 2, 4</td>
<td>33.5</td>
<td>19.9</td>
<td>28.1</td>
<td>20.2</td>
</tr>
<tr>
<td>4, 1, 4</td>
<td>59.2</td>
<td>59.2</td>
<td>59.2</td>
<td>20.7</td>
</tr>
<tr>
<td>8, 8, 8</td>
<td>8.15</td>
<td>4.48</td>
<td>7.07</td>
<td>24.5</td>
</tr>
<tr>
<td>8, 4, 8</td>
<td>11.2</td>
<td>5</td>
<td>7.36</td>
<td>19.7</td>
</tr>
<tr>
<td>8, 2, 8</td>
<td>15</td>
<td>6.32</td>
<td>10.2</td>
<td>19.1</td>
</tr>
<tr>
<td>16, 16, 16</td>
<td>3.82</td>
<td>1.12</td>
<td>1.73</td>
<td>17.1</td>
</tr>
<tr>
<td>16, 8, 16</td>
<td>5.46</td>
<td>1.2</td>
<td>2.5</td>
<td>15.3</td>
</tr>
</tbody>
</table>
Performance

GPU performance in Kop/s for the addition algorithms.

<table>
<thead>
<tr>
<th>n, m, r</th>
<th>accurate</th>
<th>“quick-and-dirty”</th>
<th>Fast “quick-and-dirty”</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 2, 2</td>
<td>3,810</td>
<td>5,379</td>
<td>5,379</td>
</tr>
<tr>
<td>2, 1, 2</td>
<td>5,379</td>
<td>5,379</td>
<td>5,379</td>
</tr>
<tr>
<td>3, 3, 3</td>
<td>396</td>
<td>310</td>
<td>765</td>
</tr>
<tr>
<td>3, 1, 3</td>
<td>1,244</td>
<td>1,244</td>
<td>1,244</td>
</tr>
<tr>
<td>4, 4, 4</td>
<td>270</td>
<td>189</td>
<td>566</td>
</tr>
<tr>
<td>4, 2, 4</td>
<td>365</td>
<td>217</td>
<td>752</td>
</tr>
<tr>
<td>4, 1, 4</td>
<td>972</td>
<td>972</td>
<td>972</td>
</tr>
<tr>
<td>8, 8, 8</td>
<td>135</td>
<td>60.8</td>
<td>274</td>
</tr>
<tr>
<td>8, 4, 8</td>
<td>170</td>
<td>72.2</td>
<td>162</td>
</tr>
<tr>
<td>8, 2, 8</td>
<td>205</td>
<td>78.9</td>
<td>235</td>
</tr>
<tr>
<td>16, 16, 16</td>
<td>55.8</td>
<td>17.4</td>
<td>35.3</td>
</tr>
<tr>
<td>16, 8, 16</td>
<td>81.3</td>
<td>22.1</td>
<td>46</td>
</tr>
</tbody>
</table>
Table of contents

- Graphics Processing Units
- Floating-point arithmetic
- Extending the precision
- A look into CAMPARY
- Applications
- Conclusions
Applications

Testing the accuracy and performance:

- dynamical systems: the Hénon attractor *

Applications

Testing the accuracy and performance:

- dynamical systems: the Hénon attractor *

- experimental mathematics: Semi-Definite Programing (SDP) Solver †

---


SDP Solver - what we did

1. integrated CAMPARY with MPACK (multiple-precision linear algebra package):
   - replaced the underlying arithmetic for all CPU routines in 2D;
   - re-implemented the GPU tuned matrix multiplication using CAMPARY;
   - integrated CAMPARY with SDPA (multiple-precision SDP solver)
     - replaced the underlying arithmetic in the SDPA-DD package;
     - linked the CAMPARY version of MPACK with it;
     - test performance and accuracy using classical problems from SDPLIB
1. integrated CAMPARY with MPACK (multiple-precision linear algebra package):
   - replaced the underlying arithmetic for all CPU routines in 2D;
   - re-implemented the GPU tuned matrix multiplication using CAMPARY;

2. integrated CAMPARY with SDPA (multiple-precision SDP solver)
   - replaced the underlying arithmetic in the SDPA-DD package;
   - linked the CAMPARY version of MPACK with it;
   - test performance and accuracy using classical problems from SDPLIB
### Some results

<table>
<thead>
<tr>
<th>Problem</th>
<th>SDPA-DD</th>
<th>SDPA-QD</th>
<th>SDPA-CAMPARY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>GPU</td>
<td>CPU</td>
</tr>
<tr>
<td>gpp124-1</td>
<td>7e-04</td>
<td>7e-04</td>
<td>6e-13</td>
</tr>
<tr>
<td>rel. gap</td>
<td>24</td>
<td>24</td>
<td>40</td>
</tr>
<tr>
<td>iteration</td>
<td>1.32</td>
<td>1.96</td>
<td>23.4</td>
</tr>
<tr>
<td>time (s)</td>
<td>1.32</td>
<td>1.96</td>
<td>23.4</td>
</tr>
<tr>
<td>gpp250-1</td>
<td>5e-04</td>
<td>5e-04</td>
<td>4e-13</td>
</tr>
<tr>
<td>rel. gap</td>
<td>25</td>
<td>25</td>
<td>41</td>
</tr>
<tr>
<td>iteration</td>
<td>7.86</td>
<td>9.43</td>
<td>151.7</td>
</tr>
<tr>
<td>time (s)</td>
<td>7.86</td>
<td>9.43</td>
<td>151.7</td>
</tr>
<tr>
<td>gpp500-1</td>
<td>1e-03</td>
<td>1e-03</td>
<td>4e-13</td>
</tr>
<tr>
<td>rel. gap</td>
<td>25</td>
<td>25</td>
<td>42</td>
</tr>
<tr>
<td>iteration</td>
<td>50.4</td>
<td>54.2</td>
<td>1,053</td>
</tr>
<tr>
<td>time (s)</td>
<td>50.4</td>
<td>54.2</td>
<td>1,053</td>
</tr>
<tr>
<td>gap10</td>
<td>1e-04</td>
<td>3e-05</td>
<td>1e-14</td>
</tr>
<tr>
<td>rel. gap</td>
<td>19</td>
<td>20</td>
<td>37</td>
</tr>
<tr>
<td>iteration</td>
<td>12.87</td>
<td>16.4</td>
<td>288.9</td>
</tr>
<tr>
<td>time (s)</td>
<td>12.87</td>
<td>16.4</td>
<td>288.9</td>
</tr>
<tr>
<td>theta5</td>
<td>6e-25</td>
<td>1e-23</td>
<td>2e-46</td>
</tr>
<tr>
<td>rel. gap</td>
<td>70</td>
<td>54</td>
<td>58</td>
</tr>
<tr>
<td>iteration</td>
<td>470.2</td>
<td>401.25</td>
<td>4,687</td>
</tr>
<tr>
<td>time (s)</td>
<td>470.2</td>
<td>401.25</td>
<td>4,687</td>
</tr>
</tbody>
</table>

† **SDPA-DD & SDPA-QD** - multiple-precision SDPA with Bailey’s QD library.

† **SDPA-CAMPARY** - multiple-precision SDPA with CAMPARY.
<table>
<thead>
<tr>
<th>Graphics Processing Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating-point arithmetic</td>
</tr>
<tr>
<td>Extending the precision</td>
</tr>
<tr>
<td>A look into CAMPARY</td>
</tr>
<tr>
<td>Applications</td>
</tr>
<tr>
<td><strong>Conclusions</strong></td>
</tr>
</tbody>
</table>
Conclusions

- algorithms for all basic operations using arbitrary precision;
- parallel algorithms for arbitrary precision $+$ and $\times$;
- specialized algorithms for $+$, $\times$ and $\div$ using double-word numbers;
- two practical applications.
Conclusions

- algorithms for all basic operations using arbitrary precision;
- parallel algorithms for arbitrary precision $+$ and $\times$;
- specialized algorithms for $+$, $\times$ and $\div$ using double-word numbers;
- two practical applications.

Perspectives

- look into specific triple-word arithmetic algorithms;
- continue developing CAMPARY - elementary functions;
- provide formal proofs;
- endless applications.