Valentina Popescu

Joined work with:
Mioara Joldes, Jean-Michel Muller

April 2015
When do we need more precision?

**Youtube view counter:**

- 32-bit integer register (limit 2,147,483,647)

When do we need more precision?

**Youtube view counter:**

- 32-bit integer register (limit 2,147,483,647)
- in 2014 the video **Psy - Gangnam Style** touched the limit

When do we need more precision?

**Youtube view counter:**

- 32-bit integer register (limit 2, 147, 483, 647)
- In 2014 the video **Psy - Gangnam Style** touched the limit
- Update to 64-bit with a limit of 9, 223, 372, 036, 854, 775, 808 (more than 9 quintillion)

*statement:*

*We never thought a video would be watched in numbers greater than a 32-bit integer... but that was before we met Psy.*

*courtesy of http://www.reddit.com/r/ProgrammerHumor/*
Dynamical systems:

- bifurcation analysis,
- compute periodic orbits (finding sinks in the Hénon map, iterating the Lorenz attractor),
- long term stability of the solar system.
Dynamical systems:
- bifurcation analysis,
- compute periodic orbits (finding sinks in the Hénon map, iterating the Lorenz attractor),
- long term stability of the solar system.

- Need more precision –few hundred bits– than standard available
- Need massive parallel computations:
  –high performance computing (HPC)–
- SIMD (Single Instruction Multiple Data) Architecture with N-multiprocessors, each with M cores
- the cores on each multiprocessor share an Instruction Unit
GPUs - Overview

- 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

- threads grouped into warps (32 threads), which are grouped into blocks and these ones into grids
GPUs - Overview

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

- threads scheduled by a hardware-based scheduler to be executed on thread processors
- blocks map on multiprocessors and do not migrate
- the grid is executed on the entire device
- on-chip shared memory that allows threads in the same block to cooperate.
- every thread has an exact amount of local memory that resides in the device's DRAM.
- the global memory is also in the DRAM and can be read and written by the host.
GPUs - Overview

- on-chip shared memory that allows threads in the same block to cooperate.
- every thread has an exact amount of local memory that resides in the device’s DRAM.
- the global memory is also in the DRAM and can be read and written by the host.

Example:

- Fermi Architecture: up to 512 cores in 16 SMs (32 cores/SM)
- Kepler Architecture: up to 1536 cores in 8 SMXs (192 cores/SMX)
- Maxwell Architecture: up to 2048 cores in 16 SMMs (128 cores/SMM)
Floating point arithmetics

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 ($e_{\text{min}} \leq e_x \leq e_{\text{max}}$).
Floating point arithmetics

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

**Concepts:**

- **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 $M_x$. 
Most common formats

- Single precision format \((p = 24)\):

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>e</td>
<td>m</td>
</tr>
</tbody>
</table>

- Double precision format \((p = 53)\):

<table>
<thead>
<tr>
<th></th>
<th>11</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>e</td>
<td>m</td>
</tr>
</tbody>
</table>

→ Implicit bit that is not stored.
### Most common formats

- **Single precision format** ($p = 24$):
  
<table>
<thead>
<tr>
<th>s</th>
<th>e</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>23</td>
</tr>
</tbody>
</table>

- **Double precision format** ($p = 53$):
  
<table>
<thead>
<tr>
<th>s</th>
<th>e</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>52</td>
</tr>
</tbody>
</table>

  → Implicit bit that is not stored.

### Rounding modes

- 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).
- Portability, determinism.
Reminder: IEEE 754-2008 standard

Most common formats

- Single precision format \((p = 24)\):

<table>
<thead>
<tr>
<th>1</th>
<th>8</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>e</td>
<td>m</td>
</tr>
</tbody>
</table>

- Double precision format \((p = 53)\):

<table>
<thead>
<tr>
<th>1</th>
<th>11</th>
<th>52</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>e</td>
<td>m</td>
</tr>
</tbody>
</table>

→ Implicit bit that is not stored.

Rounding modes

- 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).
- Portability, determinism.

→ GPUs are conform with the standard and support both standard formats.
Two ways of representing numbers in extended precision

- **multiple-digit representation** - a number is represented by a sequence of digits coupled with a single exponent (Ex. GNU MPFR, ARPREC, GARPREC, CUMP);

```
 s       M       e
```

- **multiple-term representation** - a number is expressed as the unevaluated sum of several FP numbers (also called a FP expansion) (Ex. QD, GQD).

Need for another multiple precision library:

- GNU MPFR - not ported on GPU
- GARPREC & CUMP - tuned for big array operations where the data is generated on the host and only the operations are performed on the device
- QD & GQD - offer only double-double and quad-double precision; the results are not correctly rounded
Multiple precision arithmetic libraries

Two ways of representing numbers in extended precision

- **multiple-digit representation** - a number is represented by a sequence of digits coupled with a single exponent (Ex. GNU MPFR, ARPREC, GARPREC, CUMP);

![Multiple-digit representation diagram](image)

- **multiple-term representation** - a number is expressed as the unevaluated sum of several FP numbers (also called a FP expansion) (Ex. QD, GQD).

![Multiple-term representation diagram](image)
Multiple precision arithmetic libraries

Two ways of representing numbers in extended precision

- **multiple-digit representation** - a number is represented by a sequence of digits coupled with a single exponent (Ex. GNU MPFR, ARPREC, GARPREC, CUMP);

  \[
  \begin{array}{c}
  \text{s} \\
  \hline \\
  \text{M} \\
  \hline \\
  \text{e}
  \end{array}
  \]

- **multiple-term representation** - a number is expressed as the unevaluated sum of several FP numbers (also called a FP expansion) (Ex. QD, GQD).

  \[
  \begin{array}{c}
  \text{u}_0 \\
  \hline \\
  \text{u}_1 \\
  \hline \\
  \cdots \\
  \hline \\
  \text{u}_{n-1}
  \end{array}
  \]

Need for another multiple precision library:

- GNU MPFR - not ported on GPU
- GARPREC & CUMP - tuned for big array operations where the data is generated on the host and only the operations are performed on the device
- QD & GQD - offer only double-double and quad-double precision; the results are not correctly rounded
Our approach: multiple-term representation

Drawback: more than one representation.

Example:

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

$$\mathbf{R} = x_0 + x_1 + x_2$$

- $x_0 = 1.1000 \times 2^{-1}$
- $x_1 = 1.0010 \times 2^{-3}$
- $x_2 = 1.0110 \times 2^{-6}$

Most compact $\mathbf{R} = z_0 + z_1$:

- $z_0 = 1.1101 \times 2^{-1}$
- $z_1 = 1.1000 \times 2^{-8}$

Least compact $\mathbf{R} = y_0 + y_1 + y_2 + y_3 + y_4 + y_5$:

- $y_0 = 1.0000 \times 2^{-1}$
- $y_1 = 1.0000 \times 2^{-2}$
- $y_2 = 1.0000 \times 2^{-3}$
- $y_3 = 1.0000 \times 2^{-5}$
- $y_4 = 1.0000 \times 2^{-8}$
- $y_5 = 1.0000 \times 2^{-9}$

Solution

To ensure that an expansion carries significantly more information than one FP number only, it is required to be non-overlapping (re-)normalization algorithms.
Our approach: multiple-term representation

Drawback: more than one representation.

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

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

$$R = x_0 + x_1 + x_2:
\begin{align*}
x_0 &= 1.1000e - 1; \\
x_1 &= 1.0010e - 3; \\
x_2 &= 1.0110e - 6.
\end{align*}$$
Our approach: multiple-term representation

**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*}
```
Our approach: multiple-term representation

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 $R = y_0 + y_1 + y_2 + y_3 + y_4 + y_5$:
\[
\begin{align*}
y_0 &= 1.0000e - 1; \\
y_1 &= 1.0000e - 2; \\
y_2 &= 1.0000e - 3; \\
y_3 &= 1.0000e - 5; \\
y_4 &= 1.0000e - 8; \\
y_5 &= 1.0000e - 9; \\
\end{align*}
\]
Our approach: multiple-term representation

Drawback: more than one representation.

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

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

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

$$x_0 = 1.1000e - 1;$$
$$x_1 = 1.0010e - 3;$$
$$x_2 = 1.0110e - 6.$$  

**Most compact** $R = z_0 + z_1$:

$$z_0 = 1.1101e - 1;$$
$$z_1 = 1.1000e - 8.$$

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

$$y_0 = 1.0000e - 1;$$
$$y_1 = 1.0000e - 2;$$
$$y_2 = 1.0000e - 3;$$
$$y_3 = 1.0000e - 5;$$
$$y_4 = 1.0000e - 8;$$
$$y_5 = 1.0000e - 9;$$

Solution

To ensure that an expansion carries significantly more information than one FP number only, it is required to be *non-overlapping*.

→ (re-)normalization algorithms
Nonoverlapping expansions

**Definition 1:** $\mathcal{P}$-nonoverlapping (according to Priest's definition).

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

Example:

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

**Definition1**: \( P \)-nonoverlapping (according to Priest's definition).

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

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

**Definition2** (nonzero-overlapping): \( S \)-nonoverlapping (according to Shewchuk's definition).

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

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

**Definition 1:** \( \mathcal{P} \)-nonoverlapping (according to Priest's definition).
For an expansion \( u_0, u_1, \ldots, u_{n-1} \) if for all \( 0 < i < n \), we have \( |u_i| < \text{ulp}(u_{i-1}) \).  

Example:

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

**Definition 2 (nonzero-overlapping):** \( S \)-nonoverlapping (according to Shewchuk's definition).
For an expansion \( u_0, u_1, \ldots, u_{n-1} \) if for all \( 0 < i < n \), we have \( |u_i| < \text{uls}(u_{i-1}) \).  

Example:

\[
\begin{align*}
x_0 &= 1.1000e - 1; \\
x_1 &= 1.0100e - 3; \\
x_2 &= 1.1001e - 7; \\
x_3 &= 1.1010e - 12;
\end{align*}
\]
Error-Free Transforms: 2Sum & 2ProdFMA

Theorem 1 (2Sum algorithm)

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;
- \(s = RN(a + b)\).

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

Algorithm 1 (2Sum \((a, b)\))

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

\(\rightarrow 6\) FP operations (proved to be optimal unless we have information on the ordering of \(|a|\) and \(|b|\))
Error-Free Transforms: 2Sum & 2ProdFMA

Theorem 2 (Fast2Sum algorithm)

Let $a$ and $b$ be FP numbers that satisfy $|a| \geq |b|$. Algorithm Fast2Sum computes two FP numbers $s$ and $e$ that satisfy the following:

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

Algorithm 2 (Fast2Sum $(a, b)$)

```
\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*}
```

$\rightarrow$ 3 FP operations
Theorem 3 (2ProdFMA algorithm)

Let $a$ and $b$ be FP numbers, $e_a + e_b \geq e_{\text{min}} + p - 1$. Algorithm 2ProdFMA computes two FP numbers $p$ and $e$ that satisfy the following:

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

Algorithm 3 (2ProdFMA $(a, b)$)

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

$\rightarrow$ 2 FP operations

$\rightarrow$ GPUs offer hardware implemented FMA.
Distillation Algorithms: \textit{VecSum}

**Algorithm 4** (\textit{VecSum} \((x_0, \ldots, x_{n-1})\))

\[
\begin{align*}
\text{for } i &\leftarrow n - 1 \text{ to } 1 \text{ do} \\
(s_{i-1}, e_i) &\leftarrow 2\text{Sum}(x_i, x_{i-1}) \\
\text{end for} \\
\end{align*}
\]

\[
\begin{align*}
e_0 &\leftarrow s_0 \\
\text{return } &\quad e_0, \ldots, e_{n-1}
\end{align*}
\]

Recently proven property:

If \(x_0, \ldots, x_{n-1}\) overlap by at most \(d \leq p - 1\) digits, then the sequence \(e_0, \ldots, e_{n-1}\) is \textit{S-non-overlapping}.

Restriction: \(n \leq 12\) for single precision and \(n \leq 39\) for double precision.
Priest's renormalization algorithm [Priest'91]

Drawback: many conditional branches → no pipelined operations → slow in practice
Renormalization algorithms

Priest's renormalization algorithm [Priest'91]

Schematic drawing for \( n = 5 \).

**Drawback:** many conditional branches → no pipelined operations → slow in practice
New renormalization algorithm

Based on chained levels of 2Sum and Fast2Sum.
New renormalization algorithm

Input:
- $x_0, \ldots, x_{n-1}$, FP numbers that satisfy one of the following cases:
  (i) overlap by at most $d \leq p-1$ digits;
  (ii) overlap by at most $d \leq p-2$ digits and may contain pairs of at most 2 consecutive terms that overlap by $p$ digits;
 Remark: in both cases we allow interleaving 0s;
- $m$, with $1 \leq m \leq n$, required number of output terms;

Output: "truncation" to $m$ terms of a $P$-nonoverlapping FP expansion $f = f_0, \ldots, f_{n-1}$ such that $x_0 + \ldots + x_{n-1} = f$ and $f_{i+1} \leq (\frac{1}{2} + 2^{-p+2} + 2^{-p}) \text{ulp}(f_i)$, for all $0 \leq i < m-1$.

* Arithmetic algorithms for extended precision using floating-point expansions, joint work with M. Joldes, O. Marty and, J.-M. Muller. Submitted to the IEEE Transactions on Computers, January 2015
Renormalization algorithm - First level (VecSum)

**Input:** \( x_0, \ldots, x_{n-1} \), FP numbers that overlap by at most \( d \leq p - 2 \) digits and can contain pairs of at most 2 consecutive terms that overlap by \( p \) digits and may also contain interleaving 0s;

**Output:** \( e = (e_0, \ldots, e_{n-1}) \) that satisfies:
\[
|e_0| > |e_1| \geq \ldots > |e_{i-1}| \geq |e_i| > |e_{i+1}| \geq |e_{i+2}| > \ldots,
\]
where:
- \( |e_i| > |e_{i+1}| \) implies they are \( S\)-nonoverlapping;
- \( |e_i| \geq |e_{i+1}| \) implies they are \( S\)-nonoverlapping for strict inequality or they are equal to a power of 2.
Renormalization algorithm - First level (\textit{VecSum})

\textbf{Example: } \( p = 5; \ n = 6; \)

\textbf{Input:} \( x_0, \ldots, x_{n-1}, \) FP numbers that overlap by at most \( d \leq p - 2 \) digits and can contain pairs of at most 2 consecutive terms that overlap by \( p \) digits and may also contain interleaving 0s;

\textbf{Output:} \( e = (e_0, \ldots, e_{n-1}) \) that satisfies:

\(|e_0| > |e_1| \geq \ldots \geq |e_{i-1}| \geq |e_i| > |e_{i+1}| \geq |e_{i+2}| > \ldots, \) where:

- \(|e_i| > |e_{i+1}|\) implies they are \textit{S-nonoverlapping};
- \(|e_i| \geq |e_{i+1}|\) implies they are \textit{S-nonoverlapping} for strict inequality or they are equal to a power of 2.
Input:

- \( e = (e_0, \ldots, e_{n-1}) \) that satisfies:
  \[ |e_0| > |e_1| \geq \ldots \geq |e_{i-1}| \geq |e_i| > |e_{i+1}| \geq |e_{i+2}| > \ldots, \]
  where:
  - \( |e_i| > |e_{i+1}| \) implies they are \( S \)-nonoverlapping;
  - \( |e_i| \geq |e_{i+1}| \) implies they are \( S \)-nonoverlapping for strict inequality or they are equal to a power of 2

- \( m + 1, \) with \( 1 \leq m < n \), the required number of output terms;

Output: \( f = (f_0, \ldots, f_m) \), that satisfies \( |f_{i+1}| \leq \text{ulp}(f_i) \) for all \( 0 \leq i < m \).
Renormalization algorithm - Second level (VecSumErrBranch)

Example: $p = 5; n = 6; m = 3$

Input:
- $e = (e_0, \ldots, e_{n-1})$ that satisfies:
  $$|e_0| > |e_1| \geq \ldots \geq |e_{i-1}| \geq |e_i| > |e_{i+1}| \geq |e_{i+2}| > \ldots,$$
  where:
  - $|e_i| > |e_{i+1}|$ implies they are $S$-nonoverlapping;
  - $|e_i| \geq |e_{i+1}|$ implies they are $S$-nonoverlapping for strict inequality or they are equal to a power of 2
- $m + 1$, with $1 \leq m < n$, the required number of output terms;

Output: $f = (f_0, \ldots, f_m)$, that satisfies $|f_{i+1}| \leq \text{ulp}(f_i)$ for all $0 \leq i < m$. 
Input: $f = (f_0, \ldots, f_m)$, with $|f_{i+1}| \leq \text{ulp}(f_i)$, for all $0 \leq i < m$;

Output: $g = (g_0, \ldots, g_m)$ satisfies $|g_1| \leq \left(\frac{1}{2} + 2^{-p+2}\right)\text{ulp}(g_0)$ and $|g_{i+1}| \leq \text{ulp}(g_i)$, for $0 < i < m$.

Remark: we can use $\text{Fast2Sum}(\rho_i, f_{i+1})$, because we either have $\rho_i = 0$ or $|f_{i+1}| \leq |\rho_i|$. 
Example: \( p = 5; m = 3 \)

\[
\begin{align*}
1.001000 & \quad -1.100000 \\
\Rightarrow & \\
\Rightarrow & \\
\Rightarrow & \\
2Sum & \quad 2Sum & \quad 2Sum \\
S & \quad S & \quad S \\
\downarrow & \quad \downarrow & \quad \downarrow \\
1.001000 & \quad -1.100000 & \quad 1.000000 \\
\end{align*}
\]

**Input:** \( f = (f_0, \ldots, f_m) \), with \( |f_{i+1}| \leq \text{ulp}(f_i) \), for all \( 0 \leq i < m \);

**Output:** \( g = (g_0, \ldots, g_m) \) satisfies \( |g_1| \leq \left( \frac{1}{2} + 2^{-p+2} \right) \text{ulp}(g_0) \) and \( |g_{i+1}| \leq \text{ulp}(g_i) \), for \( 0 < i < m \).

**Remark:** we can use \( \text{Fast2Sum}(\rho_i, f_{i+1}) \), because we either have \( \rho_i = 0 \) or \( |f_{i+1}| \leq |\rho_i| \).
Renormalization algorithm - Subsequent levels

After third level:

- $P$-nonoverlapping condition for the first two elements of $g$;
- the rest of $g$ keeps the existing bound, $|g_{i+1}| \leq \text{ulp}(g_i)$ for all $0 < i < m$.

**Advantage:** if zeros appear, they are pushed at the end.

**Solution:** continue applying $m - 1$ subsequent levels of $\text{VecSumErr}$ on the remaining elements that overlap.
Example: $p = 5; n = 6; m = 3$. 
Main algorithm for renormalization of FP expansion

Algorithm 5 (Renormalization algorithm.)

**Require:** FP expansion \( x = x_0 + \ldots + x_{n-1} \) consisting of FP numbers that overlap by at most \( d \) digits, with \( d \leq p - 1 \) or \( d \leq p - 2 \) and may contain pairs of at most 2 consecutive terms that overlap by \( p \) digits; \( m \) length of output FP expansion.

**Ensure:** FP expansion \( f = f_0 + \ldots + f_{m-1} \) s.t.

\[
    f_{i+1} \leq \left( \frac{1}{2} + 2^{-p+2} + 2^{-p} \right) \text{ulp}(f_i), \text{ for all } 0 \leq i < m - 1.
\]

1: \( e[0 : n - 1] \leftarrow \text{VecSum}(x[0 : n - 1]) \)
2: \( f^{(0)}[0 : m] \leftarrow \text{VecSumErrBranch}(e[0 : n - 1], m + 1) \)
3: for \( i \leftarrow 0 \) to \( m - 2 \) do
4: \( f^{(i+1)}[i : m] \leftarrow \text{VecSumErr}(f^{(i)}[i : m]) \)
5: end for
6: return FP expansion \( f = f^{(1)}_0 + \ldots + f^{(m-1)}_{m-2} + f^{(m-1)}_{m-1} \).
Operation count

- $R_{Priest}(n) = 20(n - 1)$ FP operations;
- $R_{new}(n, m) = 13n + \frac{3}{2}m^2 + \frac{3}{2}m - 22$ FP operations.

**Table:** FP operation count for the new renormalization algorithm vs. Priest’s one [Priest’91]. We consider that both algorithms compute $n - 1$ terms in the output expansion.

<table>
<thead>
<tr>
<th>$n$</th>
<th>2</th>
<th>4</th>
<th>7</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alg. 5</td>
<td>13</td>
<td>60</td>
<td>153</td>
<td>190</td>
<td>273</td>
<td>368</td>
<td>594</td>
</tr>
<tr>
<td>Priest’s algorithm</td>
<td>20</td>
<td>60</td>
<td>120</td>
<td>140</td>
<td>180</td>
<td>220</td>
<td>300</td>
</tr>
</tbody>
</table>

Observe that for $n > 4$ Priest’s algorithm performs better. **But**, in practice, the last $m - 1$ levels will take advantage of the computers pipeline → our algorithm performs better
Addition

Input:
- \( x = (x_0, \ldots, x_{N-1}) \) and \( y = (y_0, \ldots, y_{M-1}) \), \( \mathcal{P} \)-nonoverlapping FP expansions
- \( R \), number of required output terms;

Output: \( r = (r_0, \ldots, r_{R-1}) \) approximation to \( R \) terms of \( x + y \).
Addition

**Input:**
- $x = (x_0, \ldots, x_{N-1})$ and $y = (y_0, \ldots, y_{M-1})$, $P$-nonoverlapping FP expansions
- $R$, number of required output terms;

**Output:** $r = (r_0, \ldots, r_{R-1})$ approximation to $R$ terms of $x + y$.

**Solution:** merge the input expansions; renormalize the resulted array.

Full length

Consider all the terms in the input expansions. Error bound:

$$|x + y - r| < \text{ulp}(r_{R-1})$$

Truncated

Consider only the first $R$ significant terms of each input expansion. Error bound:

$$||x + y - R - 1 \sum_{i=0}^{R-1} r_i|| \leq (|x_0| + |y_0|)^2 - (p-1)R - 2 - (p-1)^2.$$
Addition

Input:
- \( x = (x_0, \ldots, x_{N-1}) \) and \( y = (y_0, \ldots, y_{M-1}) \), \( \mathcal{P} \)-nonoverlapping FP expansions
- \( R \), number of required output terms;

Output: \( r = (r_0, \ldots, r_{R-1}) \) approximation to \( R \) terms of \( x + y \).

Solution: merge the input expansions; renormalize the resulted array.

“Full length”

Consider all the terms in the input expansions. Error bound:

\[ |x + y - r| < \text{ulp}(r_{R-1}). \]
Addition

Input:
- \( x = (x_0, \ldots, x_{N-1}) \) and \( y = (y_0, \ldots, y_{M-1}) \), \( P\)-nonoverlapping FP expansions
- \( R \), number of required output terms;

Output: \( r = (r_0, \ldots, r_{R-1}) \) approximation to \( R \) terms of \( x + y \).

Solution: merge the input expansions; renormalize the resulted array.

“Full length”
Consider all the terms in the input expansions. Error bound:

\[
|x + y - r| < \text{ulp}(r_{R-1}).
\]

“Truncated”
Consider only the first \( R \) significant terms of each input expansion. Error bound:

\[
\left| x + y - \sum_{i=0}^{R-1} r_i \right| \leq (|x_0| + |y_0|) \frac{2^{-(p-1)R}}{1 - 2^{-(p-1)}}.
\]
Performance measures for addition algorithm

Table: Performance† in MFlops/s for “full length” and “truncated” addition vs. QD and MPFR implementation. N and M represent the number of terms in the input expansions and R is the size of the computed result.

<table>
<thead>
<tr>
<th>N, M, R</th>
<th>CAMPARY</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>“full length”</td>
<td>“truncated”</td>
<td>QD</td>
<td>MPFR</td>
<td></td>
</tr>
<tr>
<td>2, 2, 2</td>
<td>39.75</td>
<td>39.75</td>
<td>339</td>
<td>37.11</td>
<td></td>
</tr>
<tr>
<td>4, 4, 4</td>
<td>14.79</td>
<td>14.79</td>
<td>28.66</td>
<td>34.82</td>
<td></td>
</tr>
<tr>
<td>8, 8, 8</td>
<td>5.88</td>
<td>5.88</td>
<td>*</td>
<td>28.74</td>
<td></td>
</tr>
<tr>
<td>16, 16, 16</td>
<td>2.1</td>
<td>2.1</td>
<td>*</td>
<td>23.09</td>
<td></td>
</tr>
<tr>
<td>2, 4, 4</td>
<td>20.39</td>
<td>20.39</td>
<td>32.51</td>
<td>24.17</td>
<td></td>
</tr>
<tr>
<td>4, 2, 2</td>
<td>32.54</td>
<td>40.22</td>
<td>*</td>
<td>26.55</td>
<td></td>
</tr>
<tr>
<td>2, 8, 8</td>
<td>9.14</td>
<td>9.14</td>
<td>*</td>
<td>23.01</td>
<td></td>
</tr>
<tr>
<td>8, 2, 2</td>
<td>24.30</td>
<td>39.82</td>
<td>*</td>
<td>32.52</td>
<td></td>
</tr>
<tr>
<td>4, 8, 8</td>
<td>7.62</td>
<td>7.62</td>
<td>*</td>
<td>21.94</td>
<td></td>
</tr>
<tr>
<td>8, 4, 4</td>
<td>12.74</td>
<td>14.79</td>
<td>*</td>
<td>31.01</td>
<td></td>
</tr>
<tr>
<td>4, 16, 16</td>
<td>2.92</td>
<td>2.92</td>
<td>*</td>
<td>20.39</td>
<td></td>
</tr>
<tr>
<td>16, 4, 4</td>
<td>10.1</td>
<td>14.94</td>
<td>*</td>
<td>31.06</td>
<td></td>
</tr>
<tr>
<td>8, 16, 16</td>
<td>2.55</td>
<td>2.55</td>
<td>*</td>
<td>19.90</td>
<td></td>
</tr>
<tr>
<td>16, 8, 8</td>
<td>24.30</td>
<td>39.83</td>
<td>*</td>
<td>25.32</td>
<td></td>
</tr>
</tbody>
</table>

† Intel(R) Core(TM) i7 CPU 3820, 3.6GHz computer
* precision not supported
Solution: Adaptation of Priest’s multiplication algorithm [Priest'91], by computing and adding scalar products.
Solution: Adaption of Priest’s multiplication algorithm [Priest'91], by computing and adding scalar products.

Algorithm 6 ("Full length")

Require: $x = (x_0, \ldots, x_{N-1})$ and $y = (y_0, \ldots, y_{M-1})$, $\mathcal{P}$-nonoverlapping FP expansions; $R$, number of required output terms

Ensure: $r = (r_0, \ldots, r_{R-1})$ approximation to $R$ terms of $xy$ that satisfies

$$|xy - r| < \text{ulp}(r_{R-1}).$$

1: newSize ← 0
2: for $i \leftarrow 0$ to $n - 1$ do
3: for $j \leftarrow 0$ to $m - 1$ do
4: $(p_j, e_j) \leftarrow 2\text{ProdFMA}(x_i, y_j)$
5: end for
6: $p \leftarrow \text{Renorm}(p, M)$
7: $e \leftarrow \text{Renorm}(e, M)$
8: aux $\leftarrow \text{Add}(p, e)$
9: $(r, \text{newSize}) \leftarrow \text{Add}(r, \text{newSize}, \text{aux}, 2M)$
10: end for
11: return FP expansion $r = r_0 + \ldots + r_{R-1}$. 
"Full length"

We compute and add all scalar products. Error bound:

\[ |xy - \sum_{i=0}^{R-1} r_i| < ulp(r_{R-1}). \]
Multiplication

"Full length"

We compute and add all scalar products. Error bound:

\[ |xy - \sum_{i=0}^{R-1} r_i| < \text{ulp}(r_{R-1}). \]

"Truncated"

We "truncate" the operations to obtain the first \( R \) significant components in the result \( r \) by not computing \( \sum_{k=R}^{M+N-2} \sum_{i+j=k} x_i y_j \). Error bound:

\[ |xy - \sum_{i=0}^{R-1} r_i| \leq |x_0 y_0| 2^{-(p-1)}R \frac{2^{-2(p-1)} - 2^{-(p-1)} + N + M - R - 1}{(1 - 2^{-(p-1)})^2}. \]
Two available methods:

- based on classical "paper-and-pencil" algorithm [Bailey’01, Priest’91, Daumas’99]
- based on the Newton-Raphson iteration

\[
\alpha_{n+1} = \alpha_n - \frac{\alpha_n^2 - f}{f'}(x_n), \quad x_0 \text{ close to } \alpha, \quad f'(\alpha) \neq 0 \Rightarrow \text{quadratic convergence.}
\]
Reciprocal/division and square root algorithms

<table>
<thead>
<tr>
<th>Two available methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- based on classical &quot;paper-and-pencil&quot; algorithm [Bailey’01, Priest’91, Daumas’99]</td>
</tr>
<tr>
<td>- based on the Newton-Raphson iteration</td>
</tr>
</tbody>
</table>

Newton-Raphson iteration for root $\alpha$ of $f$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

$x_0$ close to $\alpha$, $f'(\alpha) \neq 0 \rightarrow$ quadratic convergence.
Reciprocal/division algorithm

Newton-Raphson iteration for reciprocal

I.e., root $1/a$ of \( f(x) = 1/x - a \)

\[
x_{n+1} = x_n(2 - ax_n),
\]

\(x_0 \) close to $1/a \rightarrow$ quadratic convergence, \( x_{n+1} - \frac{1}{a} = -a(x_n - \frac{1}{a})^2 \).
Newton-Raphson iteration for reciprocal

I.e., root $1/a$ of $f(x) = 1/x - a$

$$x_{n+1} = x_n(2 - ax_n),$$

$x_0$ close to $1/a \rightarrow$ quadratic convergence, $x_{n+1} - 1/a = -a(x_n - 1/a)^2$.

Adapted Newton-Raphson iteration for reciprocal of FP expansion:

$$x_{n+1} = x_n \cdot (2 - a \cdot x_n).$$

→ quadratic convergence
Adapted Newton-Raphson iteration on FP expansions

FP expansion: \( a = a_0 + a_1 + \cdots + a_{n-1} \)

General procedure: \( x_{n+1} = x_n(2 - ax_n) \)

iter 0 \( x_0 = \text{RN}\left(\frac{1}{a_0}\right) \)
Adapted Newton-Raphson iteration on FP expansions

FP expansion: \( a = a_0 + a_1 + \cdots + a_{n-1} \)

General procedure: \( x_{n+1} = x_n(2 - ax_n) \)

iter 0

\[
x_0 = \text{RN}\left(\frac{1}{a_0}\right)
\]

iter 1

\[
x_0 x_1 = x_0 \cdot \left(2 - a_0 a_1 x_0\right),
\]
Adapted Newton-Raphson iteration on FP expansions

FP expansion: \( a = a_0 + a_1 + \cdots + a_{n-1} \)

General procedure: \( x_{n+1} = x_n(2 - ax_n) \)

iter 0
\[
\begin{align*}
  x_0 &= \text{RN}(\frac{1}{a_0}), \\
\end{align*}
\]

iter 1
\[
\begin{align*}
  x_0 x_1 &= x_0 \cdot \left( 2 - a_0 a_1 x_0 \right), \\
\end{align*}
\]

iter 2
\[
\begin{align*}
  x_0 x_1 x_2 x_3 &= x_0 x_1 \cdot \left( 2 - a_0 a_1 a_2 a_3 x_0 x_1 \right), \\
\end{align*}
\]

\vdots
Adapted Newton-Raphson iteration on FP expansions

FP expansion: \( a = a_0 + a_1 + \cdots + a_{n-1} \)

General procedure: \( x_{n+1} = x_n(2 - ax_n) \)

iter 0 \( x_0 = \text{RN} \left( \frac{1}{a_0} \right) \)

iter 1 \( x_0 \cdot x_1 = x_0 \cdot \left( \frac{2 - a_0}{a_1} \right) \)

iter 2 \( x_0 \cdot x_1 \cdot x_2 \cdot x_3 = x_0 \cdot x_1 \cdot \left( \frac{2 - a_0}{a_1} \cdot \frac{2 - a_0}{a_2} \cdot \frac{2 - a_0}{a_3} \cdot \cdots \right) \)

\ldots

But... when using Error Free Transforms,

\[
\begin{pmatrix}
    u_0 & u_1 & \cdots & u_{n-1}
\end{pmatrix}
\cdot
\begin{pmatrix}
    v_0 & v_1 & \cdots & v_{m-1}
\end{pmatrix}
= 
\begin{pmatrix}
    w_0 & w_1 & \cdots & w_{2mn-1}
\end{pmatrix}
\]
Adapted Newton-Raphson iteration on FP expansions

FP expansion: $a = a_0 + a_1 + \cdots + a_{n-1}$

General procedure: $x_{n+1} = x_n (2 - a x_n)$

iter 0

$$x_0 = \text{RN} \left( \frac{1}{a_0} \right)$$

iter 1

$$x_0 \quad x_1 = x_0 \cdot \left( \frac{2}{2 - a_0 a_1 x_0} \right)$$

iter 2

$$x_0 \quad x_1 \quad x_2 \quad x_3 = x_0 \quad x_1 \cdot \left( \frac{2}{2 - a_0 a_1 a_2 a_3 x_0 x_1} \right)$$

But... when using Error Free Transforms,

$$\begin{pmatrix} u_0 & u_1 & \cdots & u_{n-1} \\ v_0 & v_1 & \cdots & v_{m-1} \end{pmatrix} \cdot \begin{pmatrix} w_0 & w_1 & \cdots & w_{2mn-1} \end{pmatrix} = \begin{pmatrix} w_0 & w_1 & \cdots & w_{2mn-1} \end{pmatrix}$$

Solution: use truncated addition/multiplication.
Truncations and Error Analysis

FP expansion: \( a = a_0 + a_1 + \cdots + a_{n-1} \)

General procedure: \( x_{n+1} = x_n(2 - ax_n) \)

iter 0 \( x_0 = \text{RN}\left(\frac{1}{a_0}\right) \)
Truncations and Error Analysis

FP expansion: \[ a = a_0 + a_1 + \cdots + a_{n-1} \]

General procedure: \[ x_{n+1} = x_n (2 - ax_n) \]

iter 0

\[
\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 2 \\ a_0 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ a_1 \end{pmatrix} + \begin{pmatrix} \hat{v}_0 \\ \hat{v}_1 \end{pmatrix},
\]

iter 1

\[
\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 2 \\ a_0 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ a_1 \end{pmatrix} + \begin{pmatrix} \hat{v}_0 \\ \hat{v}_1 \end{pmatrix},
\]

Error analysis based on triangular inequalities. Let

\[
\eta = \sum_{j=0}^{\infty} \left( -\frac{j-1}{2} \right)^p = 2 - p_1 - 2 - p_2, \\
\gamma_i = 2 - (2i + 1 - 1)p \eta_1 - \eta,
\]

\[
| x_i+1 - \tau_i | \leq \gamma_i | x_i \cdot \hat{w}_i |,
\]

\[
| w_i - \hat{w}_i | \leq \gamma_i | w_i | \leq \gamma_i | 2 - \hat{v}_i |,
\]

\[
| v_i - \hat{v}_i | \leq \gamma_i | a(f_i) \cdot x_i |,
\]

And finally,

\[
| a - a(f_i) | \leq \gamma_i | a |.
\]
Truncations and Error Analysis

FP expansion:  \( a = a_0 + a_1 + \cdots + a_{n-1} \)

General procedure:  \( x_{n+1} = x_n(2 - ax_n) \)

iter 0  \( x_0 = \text{RN}\left(\frac{1}{a_0}\right) \)

iter 1  \[
\begin{pmatrix}
\hat{v}_0 \hat{v}_1 \\
\hat{w}_1 \hat{w}_0 \\
\end{pmatrix}
\begin{pmatrix}
x_0 & x_1 \\
2 & a_0 & a_1 & x_0 \\
\end{pmatrix}
\begin{pmatrix}
\hat{v}_0 \hat{v}_1 \\
\hat{w}_1 \hat{w}_0 \\
\end{pmatrix}
\]

Error analysis based on triangular inequalities. Let
\[
\eta = \sum_{j=0}^{\infty} 2^{(-j-1)p} = \frac{2^{-p}}{1-2^{-p}},
\]
\[
\gamma_i = 2^{-(2^{i+1}-1)p} \frac{\eta}{1-\eta},
\]
\[
|x_{i+1} - \tau_i| \leq \gamma_i |x_i \cdot \hat{w}_i|,
\]
\[
|w_i - \hat{w}_i| \leq \gamma_i |w_i| \leq \gamma_i |2 - \hat{v}_i|,
\]
\[
|v_i - \hat{v}_i| \leq \gamma_i |a^{(f_i)} \cdot x_i|,
\]
\[
|a - a^{(f_i)}| \leq \gamma_i |a|.
\]
FP expansion:  \( a = a_0 + a_1 + \cdots + a_{n-1} \)

General procedure:  \( x_{n+1} = x_n(2 - ax_n) \)

**Error analysis based on triangular inequalities.** Let

\[
\eta = \sum_{j=0}^{\infty} 2^{-j-1} p = \frac{2^{-p}}{1-2^{-p}} ,
\]

\[
\gamma_i = 2^{-2^{i+1}-1} p \frac{\eta}{1-\eta} ,
\]

\[
|x_{i+1} - \tau_i| \leq \gamma_i |x_i \cdot \hat{w}_i| , \quad (1a)
\]

\[
|w_i - \hat{w}_i| \leq \gamma_i |w_i| \leq \gamma_i |2 - \hat{v}_i| , \quad (1b)
\]

\[
|v_i - \hat{v}_i| \leq \gamma_i |a^{(f_i)} \cdot x_i| , \quad (1c)
\]

\[
|a - a^{(f_i)}| \leq \gamma_i |a| . \quad (1d)
\]

And finally,

\[
\left| \frac{x_i - a^{-1}}{a^{-1}} \right| \leq 2^{-2^i (p-3)-1}
\]
Main algorithm for reciprocal

Algorithm 7 (Truncated Newton iteration based algorithm for reciprocal of a FP expansion.)

**Require:** FP expansion \( a = a_0 + \ldots + a_{2^k-1} \); length of output FP expansion \( 2^q \).

**Ensure:** FP expansion \( x = x_0 + \ldots + x_{2^q-1} \) s.t.

\[
\left| x - \frac{1}{a} \right| \leq \frac{2^{-2^q(p-3)-1}}{|a|}. \tag{2}
\]

1: \( x_0 = \text{RN}(1/a_0) \)
2: **for** \( i \leftarrow 0 \) **to** \( q - 1 \) **do**
3: \( \hat{v}[0 : 2^{i+1} - 1] \leftarrow \text{TruncMulE}(x[0 : 2^i - 1], a[0 : 2^{i+1} - 1], 2^{i+1}) \)
4: \( \hat{w}[0 : 2^{i+1} - 1] \leftarrow \text{TruncSubE}(2, \hat{v}[0 : 2^{i+1} - 1], 2^{i+1}) \)
5: \( x[0 : 2^{i+1} - 1] \leftarrow \text{TruncMulE}(x[0 : 2^i - 1], \hat{w}[0 : 2^{i+1} - 1], 2^{i+1}) \)
6: **end for**
7: **return** FP expansion \( x = x_0 + \ldots + x_{2^q-1} \).
Comparison of reciprocal/division algorithms

**Table:** Error bounds values for Priest’s formula [Priest’91] vs. Daumas [Daumas] vs. our analysis (2). $\beta$ represents the largest errors given by Algorithm 7 using the standard FP formats *double* and *single*.

<table>
<thead>
<tr>
<th>Prec, iteration</th>
<th>Eq. Priest</th>
<th>Eq. Daumas</th>
<th>Eq. (2)</th>
<th>$\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p = 53, q = 0$</td>
<td>2</td>
<td>$2^{-49}$</td>
<td>$2^{-51}$</td>
<td>$2^{-52}$</td>
</tr>
<tr>
<td>$p = 53, q = 1$</td>
<td>1</td>
<td>$2^{-98}$</td>
<td>$2^{-101}$</td>
<td>$2^{-104}$</td>
</tr>
<tr>
<td>$p = 53, q = 2$</td>
<td>$2^{-2}$</td>
<td>$2^{-195}$</td>
<td>$2^{-201}$</td>
<td>$2^{-208}$</td>
</tr>
<tr>
<td>$p = 53, q = 3$</td>
<td>$2^{-6}$</td>
<td>$2^{-387}$</td>
<td>$2^{-401}$</td>
<td>$2^{-416}$</td>
</tr>
<tr>
<td>$p = 53, q = 4$</td>
<td>$2^{-13}$</td>
<td>$2^{-764}$</td>
<td>$2^{-801}$</td>
<td>$2^{-833}$</td>
</tr>
<tr>
<td>$p = 24, q = 0$</td>
<td>2</td>
<td>$2^{-20}$</td>
<td>$2^{-22}$</td>
<td>$2^{-23}$</td>
</tr>
<tr>
<td>$p = 24, q = 1$</td>
<td>1</td>
<td>$2^{-40}$</td>
<td>$2^{-43}$</td>
<td>$2^{-46}$</td>
</tr>
<tr>
<td>$p = 24, q = 2$</td>
<td>$2^{-2}$</td>
<td>$2^{-79}$</td>
<td>$2^{-85}$</td>
<td>$2^{-92}$</td>
</tr>
<tr>
<td>$p = 24, q = 3$</td>
<td>$2^{-5}$</td>
<td>$2^{-155}$</td>
<td>$2^{-169}$</td>
<td>*</td>
</tr>
<tr>
<td>$p = 24, q = 4$</td>
<td>$2^{-12}$</td>
<td>$2^{-300}$</td>
<td>$2^{-337}$</td>
<td>*</td>
</tr>
</tbody>
</table>

* underflow occurs
Performance measures for reciprocal algorithm

**Table:** Performance† in MFlops/s for Alg. 7 vs. QD and MPFR implementation for reciprocal (A) and division (B); the numerator, denominator and quotient have respectively $d_n$, $d_i$ and $d_o$ terms.

![Table](image)

(B) Division

†Intel(R) Core(TM) i7 CPU 3820, 3.6GHz computer

* precision not supported
Conclusions

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


Conclusions

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

- Use multiple-term format for FP multiple-precision numbers → FP expansions;

---


Conclusions

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

- Use multiple-term format for FP multiple-precision numbers → FP expansions;
- Support for addition, multiplication, reciprocal/division and square root of FP expansions on both CPU and GPU;


Conclusions

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

- Use multiple-term format for FP multiple-precision numbers $\rightarrow$ FP expansions;
- Support for addition, multiplication, reciprocal/division and square root of FP expansions on both CPU and GPU;
- Thorough error analysis and explicit error bounds;

---


Conclusions

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

- Use multiple-term format for FP multiple-precision numbers $\rightarrow$ FP expansions;
- Support for addition, multiplication, reciprocal/division and square root of FP expansions on both CPU and GPU;
- Thorough error analysis and explicit error bounds;
- Renormalization algorithm that guarantees to render the result nonoverlapping.


Conclusions

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

- Use multiple-term format for FP multiple-precision numbers → FP expansions;
- Support for addition, multiplication, reciprocal/division and square root of FP expansions on both CPU and GPU;
- Thorough error analysis and explicit error bounds;
- Renormalization algorithm that guaranties to render the result nonoverlapping.

On going work

- efficient way for implementing the multiplication


Newton-Raphson iteration for square root (Heron’s iteration)

I.e., root $\sqrt{a}$ of $f(x) = x^2 - a$

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n}),$$

when $x_0 > 0 \rightarrow$ quadratic convergence.

**Drawback:** one division at each step.

Newton-Raphson iteration for reciprocal of square root

I.e., root $\frac{1}{\sqrt{a}}$ of $f(x) = \frac{1}{x^2} - a$

$$x_{n+1} = \frac{1}{2}x_n(3 - ax_n^2),$$

when $x_0$ is close to $\frac{1}{\sqrt{a}} \rightarrow$ quadratic convergence.
Algorithm 8 (Truncated Newton iteration based algorithm for square root of a FP expansion.)

Require:  *FP expansion* $a = a_0 + \ldots + a_{2^k-1}$; *length of output FP expansion* $2^q$.
Ensure:  *FP expansion* $x = x_0 + \ldots + x_{2^q-1}$ s.t.

$$\left| x - \frac{1}{\sqrt{a}} \right| \leq \frac{2^{-2^q(p-3)-1}}{\sqrt{a}}. \quad (3)$$

1: $x_0 = \text{RN}(1/\sqrt{a_0})$
2: for $i \leftarrow 0$ to $q - 1$ do
3: $\hat{v}[0 : 2^{i+1} - 1] \leftarrow \text{MulRoundE}(x[0 : 2^i - 1], a[0 : 2^{i+1} - 1], 2^{i+1})$
4: $\hat{w}[0 : 2^{i+1} - 1] \leftarrow \text{MulRoundE}(x[0 : 2^i - 1], \hat{v}[0 : 2^{i+1} - 1], 2^{i+1})$
5: $\hat{y}[0 : 2^{i+1} - 1] \leftarrow \text{SubRoundE}(3, \hat{w}[0 : 2^{i+1} - 1], 2^{i+1})$
6: $\hat{z}[0 : 2^{i+1} - 1] \leftarrow \text{MulRoundE}(x[0 : 2^i - 1], \hat{y}[0 : 2^{i+1} - 1], 2^{i+1})$
7: $x[0 : 2^{i+1} - 1] \leftarrow \hat{z}[0 : 2^{i+1} - 1] \ast 0.5$
8: end for
9: return *FP expansion* $x = x_0 + \ldots + x_{2^q-1}$.

For the 'Heron iteration' we got:

$$|x - \sqrt{a}| \leq 3 \sqrt{a} \cdot 2^{-2^q(p-3)-2}. \quad (4)$$
Table: Error bounds values for equation (3) vs. equation (4). $\beta$ represents the largest errors given by Algorithm 8 using the standard FP formats *double* and *single*.

<table>
<thead>
<tr>
<th>Prec, iteration</th>
<th>Eq. (3)</th>
<th>Eq. (4)</th>
<th>$\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p = 53, q = 0$</td>
<td>$2^{-51}$</td>
<td>$3 \cdot 2^{-52}$</td>
<td>$2^{-52}$</td>
</tr>
<tr>
<td>$p = 53, q = 1$</td>
<td>$2^{-101}$</td>
<td>$3 \cdot 2^{-102}$</td>
<td>$2^{-103}$</td>
</tr>
<tr>
<td>$p = 53, q = 2$</td>
<td>$2^{-201}$</td>
<td>$3 \cdot 2^{-202}$</td>
<td>$2^{-206}$</td>
</tr>
<tr>
<td>$p = 53, q = 3$</td>
<td>$2^{-401}$</td>
<td>$3 \cdot 2^{-402}$</td>
<td>$2^{-412}$</td>
</tr>
<tr>
<td>$p = 53, q = 4$</td>
<td>$2^{-801}$</td>
<td>$3 \cdot 2^{-802}$</td>
<td>$2^{-823}$</td>
</tr>
<tr>
<td>$p = 24, q = 0$</td>
<td>$2^{-22}$</td>
<td>$3 \cdot 2^{-23}$</td>
<td>$2^{-23}$</td>
</tr>
<tr>
<td>$p = 24, q = 1$</td>
<td>$2^{-43}$</td>
<td>$3 \cdot 2^{-44}$</td>
<td>$2^{-45}$</td>
</tr>
<tr>
<td>$p = 24, q = 2$</td>
<td>$2^{-85}$</td>
<td>$3 \cdot 2^{-86}$</td>
<td>$2^{-90}$</td>
</tr>
<tr>
<td>$p = 24, q = 3$</td>
<td>$2^{-169}$</td>
<td>$3 \cdot 2^{-170}$</td>
<td>*</td>
</tr>
<tr>
<td>$p = 24, q = 4$</td>
<td>$2^{-337}$</td>
<td>$3 \cdot 2^{-338}$</td>
<td>*</td>
</tr>
</tbody>
</table>

* underflow occurs
Table: Performance in MFlops/s for Alg. 8 for computing the square root of expansions; $d_i$ represents the number of terms in the input and $d_o$ is the size of the computed result.

<table>
<thead>
<tr>
<th>$d_i, d_o$</th>
<th>CAMPARY</th>
<th>QD</th>
<th>MPFR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 2</td>
<td>7.95</td>
<td>37.04</td>
<td>8.56</td>
</tr>
<tr>
<td>4, 4</td>
<td>2.21</td>
<td>1.21</td>
<td>4.9</td>
</tr>
<tr>
<td>8, 8</td>
<td>0.3</td>
<td>*</td>
<td>3.13</td>
</tr>
<tr>
<td>16, 16</td>
<td>0.043</td>
<td>*</td>
<td>1.73</td>
</tr>
<tr>
<td>1, 2</td>
<td>9.64</td>
<td>1687.88</td>
<td>8.78</td>
</tr>
<tr>
<td>2, 4</td>
<td>2.32</td>
<td>37.14</td>
<td>5.35</td>
</tr>
<tr>
<td>1, 4</td>
<td>2.59</td>
<td>1684.57</td>
<td>5.15</td>
</tr>
<tr>
<td>4, 2</td>
<td>7.94</td>
<td>*</td>
<td>8.33</td>
</tr>
<tr>
<td>2, 8</td>
<td>0.41</td>
<td>*</td>
<td>3.25</td>
</tr>
<tr>
<td>4, 8</td>
<td>0.34</td>
<td>*</td>
<td>3.15</td>
</tr>
<tr>
<td>4, 16</td>
<td>0.053</td>
<td>*</td>
<td>1.83</td>
</tr>
<tr>
<td>8, 16</td>
<td>0.041</td>
<td>*</td>
<td>1.75</td>
</tr>
</tbody>
</table>

†Intel(R) Core(TM) i7 CPU 3820, 3.6GHz computer
Available online at: http://homepages.laas.fr/mmjoldes/campary/.

### Implementation details

- templated class implemented in CUDA C;
- templates for both the number of terms in the expansion and the native type of the terms (allows static generation of any input-output precision combinations);
- functions defined using `__host__ __device__` specifiers;
- in-place algorithms to avoid registers spill/loads;
- fully customized algorithms for computations using expansions of various sizes;
- support for all basic operations
- interval arithmetic supported in a similar class;