Advancements in (cr)libm development

Presentation at Intel - Portland

Christoph Quirin Lauter

Arénaire team
Laboratoire de l’Informatique et du Parallélisme
École Normale Supérieure de Lyon

Portland, 10 October 2007
Introduction

Correct rounding of $x^y$

Automatic implementation of libm functions

Conclusion
crlibm\(^1\): correctly rounded elementary function library

\(^1\)http://lipforge.ens-lyon.fr/www/crlibm/
crlibm\(^1\): correctly rounded elementary function library

- Elementary functions as in an usual libm:
  - exp
  - sin
  - cos
  - ...

\(^1\)http://lipforge.ens-lyon.fr/www/crlibm/
crlibm\(^1\): correctly rounded elementary function library

- Elementary functions as in an usual libm:
  - exp
  - sin
  - cos
  - ...

- Bit-exact, correctly rounded results \( f(x) = \circ(f(x)) \)

\(^1\)http://lipforge.ens-lyon.fr/www/crlibm/
crlibm\(^1\): correctly rounded elementary function library

- Elementary functions as in an usual libm:
  - exp
  - sin
  - cos
  - ...
- Bit-exact, correctly rounded results \( f(x) = \circ(f(x)) \)
- No important impact on average performance

\(^1\)http://lipforge.ens-lyon.fr/www/crlibm/
crlibm\(^1\): correctly rounded elementary function library

- **Elementary functions** as in an usual libm:
  - \(\exp\)
  - \(\sin\)
  - \(\cos\)
  - ... 

- **Bit-exact, correctly rounded** results \(f(x) = o(f(x))\)

- No important impact on average performance

- **Guaranteed worst case** performance

---

\(^1\)http://lipforge.ens-lyon.fr/www/crlibm/
crlibm\(^1\): correctly rounded elementary function library

- Elementary functions **as in an usual libm**:
  - exp
  - sin
  - cos
  - ...
- Bit-exact, correctly rounded results \( f(x) = o(f(x)) \)
- No important impact on average performance
- Guaranteed worst case performance
- Challenge: Correct rounding requires high accuracy and complete proofs

\(^1\)http://lipforge.ens-lyon.fr/www/crlibm/
Advancements in the correct rounding of $x^y$

Techniques for automatic implementation of libm functions.
Correct rounding of $x^y$

Introduction

Correct rounding of $x^y$

Automatic implementation of libm functions

Conclusion
Correct rounding must overcome the Table Maker’s Dilemma

\[ \circ(f(x) \cdot (1 + \varepsilon)) \equiv \circ(f(x)) \]
Worst-case computations

- **Correct rounding** must overcome the Table Maker’s Dilemma

\[ \circ (f(x) \cdot (1 + \varepsilon)) \overset{?}{=} \circ (f(x)) \]

- **Finite domain**, \( x \) is FP number, \( x \in \mathbb{F} \): worst-case \( \bar{\varepsilon} \) exists

\[ \exists \bar{\varepsilon} > 0. \forall \varepsilon, |\varepsilon| \leq \bar{\varepsilon}. \forall x \in \mathbb{F}. \circ (f(x) \cdot (1 + \varepsilon)) = f(x) \]
Worst-case computations

- Correct rounding must overcome the Table Maker’s Dilemma

\[ \circ(f(x) \cdot (1 + \varepsilon)) \equiv \circ(f(x)) \]

- Finite domain, \( x \) is FP number, \( x \in \mathbb{F} \): worst-case \( \varepsilon \) exists

\[ \exists \varepsilon > 0. \forall \varepsilon, |\varepsilon| \leq \varepsilon. \forall x \in \mathbb{F}. \circ (f(x) \cdot (1 + \varepsilon)) = f(x) \]

- Univariate functions implemented in double precision:
  - Computation of \( \varepsilon \) actually possible (Lefèvre, Stehlé et al.)
Worst-case computations

- Correct rounding must overcome the Table Maker’s Dilemma

\[ \circ(f(x) \cdot (1 + \varepsilon)) = \circ(f(x)) \]

- Finite domain, \( x \) is FP number, \( x \in \mathbb{F} \): worst-case \( \varepsilon \) exists

\[ \exists \varepsilon > 0. \forall \varepsilon, |\varepsilon| \leq \varepsilon. \forall x \in \mathbb{F}. \circ(f(x) \cdot (1 + \varepsilon)) = f(x) \]

- Univariate functions implemented in double precision:
  - Computation of \( \varepsilon \) actually possible (Lefèvre, Stehlé et al.)
  - Computation of \( \varepsilon \) is a smart exhaustive search
Correct rounding must overcome the Table Maker’s Dilemma

\[ o(f(x) \cdot (1 + \varepsilon)) \not= o(f(x)) \]

Finite domain, \( x \) is FP number, \( x \in \mathbb{F} \): worst-case \( \varepsilon \) exists

\[ \exists \varepsilon > 0. \forall \varepsilon, |\varepsilon| \leq \varepsilon. \forall x \in \mathbb{F}. \circ (f(x) \cdot (1 + \varepsilon)) \not= f(x) \]

Univariate functions implemented in double precision:
- Computation of \( \varepsilon \) actually possible (Lefèvre, Stehlé et al.)
- Computation of \( \varepsilon \) is a smart exhaustive search

Bivariate function \( x^y : \mathbb{F}^2 \rightarrow \mathbb{F} \)
Worst-case computations

- **Correct rounding** must overcome the Table Maker’s Dilemma

\[ \circ(f(x) \cdot (1 + \varepsilon)) \neq \circ(f(x)) \]

- **Finite domain**, \( x \) is FP number, \( x \in \mathbb{F} \): worst-case \( \varepsilon \) exists

\[ \exists \varepsilon > 0. \forall \varepsilon, |\varepsilon| \leq \varepsilon. \forall x \in \mathbb{F}. \circ(f(x) \cdot (1 + \varepsilon)) = f(x) \]

- **Univariate functions** implemented in double precision:
  - Computation of \( \varepsilon \) actually possible (Lefèvre, Stehlé et al.)
  - Computation of \( \varepsilon \) is a smart exhaustive search

- **Bivariate function** \( x^y : \mathbb{F}^2 \to \mathbb{F} \)
  - roughly \( 2^{112} \) valid inputs
Worst-case computations

- Correct rounding must overcome the Table Maker’s Dilemma

\[ \circ(f(x) \cdot (1 + \varepsilon)) \neq \circ(f(x)) \]

- Finite domain, \( x \) is FP number, \( x \in \mathbb{F} \): worst-case \( \varepsilon \) exists

\[ \exists \varepsilon > 0. \, \forall \varepsilon, |\varepsilon| \leq \varepsilon . \forall x \in \mathbb{F} . \circ (f(x) \cdot (1 + \varepsilon)) = f(x) \]

- Univariate functions implemented in double precision:
  - Computation of \( \varepsilon \) actually possible (Lefêvre, Stehlé et al.)
  - Computation of \( \varepsilon \) is a smart exhaustive search
- Bivariate function \( x^y : \mathbb{F}^2 \rightarrow \mathbb{F} \)
  - roughly \( 2^{112} \) valid inputs
  - Worst-case search of \( \varepsilon \) currently untractable
Correct rounding of $x^n$

- Consider $x^n$, $x \in F$, $n \in \mathbb{N}$, $n$ small
- Lefèvre: traditional worst-case search is possible
  - Consider each $n$ separately
  - Current range achieved: $n \leq 255$
  - Worst case $\bar{\varepsilon} = 2^{-117}$ comparable to other double precision functions
- Correctly rounded $\text{power}(x,n) = o(x^n)$
Consider \( x^n, x \in \mathbb{F}, n \in \mathbb{N}, n \) small

Lefèvre: traditional worst-case search is possible

- Consider each \( n \) separately
- Current range achieved: \( n \leq 255 \)
- Worst case \( \varepsilon = 2^{-117} \) comparable to other double precision functions

Correctly rounded \( \text{power}(x,n) = o(x^n) \)

- Guaranteed worst-case performance for small \( n \)
Correct rounding of \( x^n \)

- Consider \( x^n, x \in \mathbb{F}, n \in \mathbb{N}, n \) small
- Lefèvre: traditional worst-case search is possible
  - Consider each \( n \) separately
  - Current range achieved: \( n \leq 255 \)
  - Worst case \( \varepsilon = 2^{-117} \) comparable to other double precision functions
- Correctly rounded \( \text{power}(x,n) = o(x^n) \)
  - Guaranteed worst-case performance for small \( n \)
  - Situation comparable to \( \sin \) and \( \cos \):
Correct rounding of $x^n$

- Consider $x^n$, $x \in \mathbb{F}$, $n \in \mathbb{N}$, $n$ small
- Lefèvre: traditional worst-case search is possible
  - Consider each $n$ separately
  - Current range achieved: $n \leq 255$
  - Worst case $\varepsilon = 2^{-117}$ comparable to other double precision functions
- Correctly rounded $\text{power}(x,n) = o(x^n)$
  - Guaranteed worst-case performance for small $n$
  - Situation comparable to sin and cos:
    - small values of $n$ (resp. $x$ for sin) are the most interesting
    - Ziv’s rounding technique allows for correct rounding outside the known domain
Correct rounding of $x^n$

- Consider $x^n$, $x \in \mathbb{F}$, $n \in \mathbb{N}$, $n$ small
- Lefèvre: traditional worst-case search is possible
  - Consider each $n$ separately
  - Current range achieved: $n \leq 255$
  - Worst case $\varepsilon = 2^{-117}$ comparable to other double precision functions
- Correctly rounded $\text{power}(x,n) = \circ (x^n)$
  - Guaranteed worst-case performance for small $n$
  - Situation comparable to sin and cos:
    - small values of $n$ (resp. $x$ for sin) are the most interesting
    - Ziv’s rounding technique allows for correct rounding outside the known domain
- This research paves the road for $x^y$
Ziv's rounding technique for $x^y$

- **Ziv's rounding technique:**
  Decrease error $\varepsilon$ of approximation $x^y \cdot (1 + \varepsilon)$ until rounding becomes possible

$$\circ(x^y \cdot (1 + \varepsilon)) = \circ(x^y)$$
Ziv’s rounding technique:
Decrease error \( \varepsilon \) of approximation \( x^y \cdot (1 + \varepsilon) \) until rounding becomes possible

\[
\circ(x^y \cdot (1 + \varepsilon)) = \circ(x^y)
\]

**Issue:**
For ensuring termination, rounding boundary cases must be filtered out

![Diagram showing rounding boundary, exact value, approximation, and interval Z]
Ziv's rounding technique for $x^y$

- **Ziv's rounding technique:**
  Decrease error $\varepsilon$ of approximation $x^y \cdot (1 + \varepsilon)$ until rounding becomes possible

  $$\circ(x^y \cdot (1 + \varepsilon)) = \circ(x^y)$$

- **Issue:**
  For ensuring termination, rounding boundary cases must be filtered out

- **Rounding boundary cases:**
  Complex set for $x^y$:

  $$RB = \{ x^y = z \mid x, y \in F_{53}, z \in F_{54} \}$$
Previous approaches:

- Rewrite

\[ RB = \{ x^y = z \mid x, y \in \mathbb{F}_{53}, z \in \mathbb{F}_{54} \} \]

as

\[ x = 2^E \cdot m, \quad y = 2^F \cdot n, \quad z = 2^G \cdot k \]

\[ E \cdot 2^F \cdot n = G, \quad (m^n)^{2^F \cdot n} = k \]
Previous approaches:

- Rewrite

\[ RB = \{ x^y = z \mid x, y \in \mathbb{F}_{53}, z \in \mathbb{F}_{54} \} \]

as

\[ x = 2^E \cdot m, \quad y = 2^F \cdot n, \quad z = 2^G \cdot k \]

\[ E \cdot 2^F \cdot n = G, \quad (m^n)^{2^F \cdot n} = k \]

- Mainly test whether

\[ (m^n)^{2^F} = k \]
Previous approaches:

- Rewrite

\[ RB = \{ x^y = z \mid x, y \in \mathbb{F}_{53}, z \in \mathbb{F}_{54} \} \]

as

\[ x = 2^E \cdot m, \quad y = 2^F \cdot n, \quad z = 2^G \cdot k \]

\[ E \cdot 2^F \cdot n = G, \quad (m^n)^{2^F \cdot n} = k \]

- Mainly test whether

\[ (m^n)^{2^F} = k \]

- Cost of the test in double precision:
  - up to 5 square root extractions
  - up to 10 doubled precision multiplies
  - pipeline broken by many ifs
An efficient rounding boundary test for $x^y - 1$

Use worst-case information for rounding boundary testing
An efficient rounding boundary test for $x^y - 2$

- **Worst-case actually unknown for** $x^y$!
An efficient rounding boundary test for $x^y - 2$

- Worst-case actually unknown for $x^y$!
- All rounding boundary cases for $x^y$ in double precision lie in a subset

$$S = \{(x, y) \in \mathbb{F}_{53}^2 \mid y \in \mathbb{N}, \ 2 \leq y \leq 35\}$$
$$\cup \quad \{(m, 2^F n) \in \mathbb{F}_{53}^2 \mid F \in \mathbb{Z}, \ -5 \leq F < 0, \ n \in 2\mathbb{N} + 1, \ 3 \leq n \leq 35, \ m \in 2\mathbb{N} + 1\}$$

Worst-case search is tractable for $(x, y) \in S$
Testing if $(x, y) \in S$ is easy: straightforward comparisons
Experimental results: 39% speed-up on average w.r.t. previous implementations
Overhead of RB detection decreased from 50% to 9%
Still more optimization: 99.1% of RB cases imply $y = 3$
An efficient rounding boundary test for $x^y - 2$

- Worst-case actually unknown for $x^y$!
- All rounding boundary cases for $x^y$ in double precision lie in a subset

$$S = \{(x, y) \in \mathbb{F}_{53}^2 \mid y \in \mathbb{N}, 2 \leq y \leq 35\} \cup \{(m, 2^F n) \in \mathbb{F}_{53}^2 \mid F \in \mathbb{Z}, -5 \leq F < 0, n \in 2\mathbb{N} + 1, 3 \leq n \leq 35, m \in 2\mathbb{N} + 1\}$$

- Worst-case search is tractable for $(x, y) \in S$
An efficient rounding boundary test for $x^y - 2$

- Worst-case actually unknown for $x^y$!
- All rounding boundary cases for $x^y$ in double precision lie in a subset

\[ S = \left\{ (x, y) \in \mathbb{F}_{53}^2 \mid y \in \mathbb{N}, \ 2 \leq y \leq 35 \right\} \]
\[
\cup \left\{ (m, 2^F n) \in \mathbb{F}_{53}^2 \mid F \in \mathbb{Z}, \ -5 \leq F < 0, \ n \in 2\mathbb{N} + 1, \ 3 \leq n \leq 35, \ m \in 2\mathbb{N} + 1 \right\}
\]

- Worst-case search is tractable for $(x, y) \in S$
- Testing if $(x, y) \in S$ is easy: straightforward comparisons
An efficient rounding boundary test for $x^y - 2$

- Worst-case actually unknown for $x^y$!
- All rounding boundary cases for $x^y$ in double precision lie in a subset

$$
S = \{ (x, y) \in \mathbb{F}_{53}^2 \mid y \in \mathbb{N}, \ 2 \leq y \leq 35 \}
\cup
\{ (m, 2^F n) \in \mathbb{F}_{53}^2 \mid F \in \mathbb{Z}, \ -5 \leq F < 0, \ n \in 2\mathbb{N} + 1, \\
3 \leq n \leq 35, \ m \in 2\mathbb{N} + 1 \}
$$

- Worst-case search is tractable for $(x, y) \in S$
- Testing if $(x, y) \in S$ is easy: straightforward comparisons
- Experimental results:
An efficient rounding boundary test for $x^y - 2$

- Worst-case actually unknown for $x^y$!
- All rounding boundary cases for $x^y$ in double precision lie in a subset

$$S = \{(x, y) \in \mathbb{F}^2_{53} \mid y \in \mathbb{N}, \ 2 \leq y \leq 35\} \cup \{(m, 2^F n) \in \mathbb{F}^2_{53} \mid F \in \mathbb{Z}, \ -5 \leq F < 0, \ n \in 2\mathbb{N} + 1, \ 3 \leq n \leq 35, \ m \in 2\mathbb{N} + 1\}$$

- Worst-case search is tractable for $(x, y) \in S$
- Testing if $(x, y) \in S$ is easy: straightforward comparisons
- Experimental results:
  - 39% speed-up on average w.r.t. previous implementations
An efficient rounding boundary test for $x^y - 2$

- Worst-case actually unknown for $x^y$!
- All rounding boundary cases for $x^y$ in double precision lie in a subset

$$
S = \left\{ (x, y) \in \mathbb{F}^{2}_{53} \mid y \in \mathbb{N}, \ 2 \leq y \leq 35 \right\}
\cup \left\{ (m, 2^F n) \in \mathbb{F}^{2}_{53} \mid F \in \mathbb{Z}, \ -5 \leq F < 0, \ n \in 2\mathbb{N} + 1, \ 3 \leq n \leq 35, \ m \in 2\mathbb{N} + 1 \right\}
$$

- Worst-case search is tractable for $(x, y) \in S$
- Testing if $(x, y) \in S$ is easy: straightforward comparisons
- Experimental results:
  - 39% speed-up on average w.r.t. previous implementations
  - Overhead of RB detection decreased from 50% to 9%
An efficient rounding boundary test for $x^y - 2$

- Worst-case actually unknown for $x^y$!
- All rounding boundary cases for $x^y$ in double precision lie in a subset

$$\mathcal{S} = \{(x, y) \in \mathbb{F}_{53}^2 \mid y \in \mathbb{N}, \ 2 \leq y \leq 35\}$$

$$\cup \{(m, 2^F n) \in \mathbb{F}_{53}^2 \mid F \in \mathbb{Z}, \ -5 \leq F < 0, \ n \in 2\mathbb{N} + 1, \ 3 \leq n \leq 35, \ m \in 2\mathbb{N} + 1\}$$

- Worst-case search is tractable for $(x, y) \in \mathcal{S}$
- Testing if $(x, y) \in \mathcal{S}$ is easy: straightforward comparisons
- Experimental results:
  - 39% speed-up on average w.r.t. previous implementations
  - Overhead of RB detection decreased from 50% to 9%
  - Still more optimization: 99.1% of RB cases imply $y = \frac{3}{2}$
An efficient rounding boundary test for $x^y - 3$

Details can be found at

http://prunel.ccsd.cnrs.fr/ensl-00169409/
Automatic implementation of libm functions

Introduction

Correct rounding of $x^y$

Automatic implementation of libm functions

Conclusion
First function in crlibm
First function in crlibm

- \( \exp(x) \) by David Defour
First function in crlibm

- \( \exp(x) \) by David Defour
- correctly rounded in two approximation steps
First function in crlibm

- exp(x) by David Defour
- correctly rounded in two approximation steps
- portable C code
- integer library for second step
First function in crlibm

- $\exp(x)$ by David Defour
- correctly rounded in two approximation steps
- portable C code
- integer library for second step
- complex, hand-written proof
First function in crlibm

- \( \exp(x) \) by David Defour
- correctly rounded in two approximation steps
- portable C code
- integer library for second step
- complex, hand-written proof
- duration: a Ph.D. thesis
An alternative implementation
An alternative implementation

- \( \exp(x) \) by myself
An alternative implementation

- $\exp(x)$ by myself
- correctly rounded in one approximation step
An alternative implementation

- \( \exp(x) \) by myself
- correctly rounded in one approximation step
- usage of Itanium specific features through assembler
An alternative implementation

- \( \exp(x) \) by myself
- correctly rounded in one approximation step
- usage of Itanium specific features through assembler
- complex, hand-written, \textit{wrong proof}
An alternative implementation

- $\exp(x)$ by myself
- correctly rounded in one approximation step
- usage of Itanium specific features through assembler
- complex, hand-written, wrong proof
- duration: a summer internship at Intel Nizhny Novgorod
Further functions in crlibm: $\text{atan}(x)$, $\log(x)$...
Further functions in crlibm: $\tan(x)$, $\log(x)$...

- Maple scripts generating header files
Further functions in crlibm: \( \text{atan}(x), \log(x) \)...

- Maple scripts generating header files
- Computation of infinite norms in Maple
Further functions in crlibm: \( \text{atan}(x) \), \( \text{log}(x) \)...

- Maple scripts generating header files
- Computation of infinite norms in Maple
- Hand-written Gappa proofs
Further functions in crlibm: \( \text{atan}(x) \), \( \text{log}(x) \)...

- Maple scripts generating header files
- Computation of infinite norms in Maple
- Hand-written Gappa proofs
- duration: about 1 month per function
And at Intel?

How many man-hours are accounted per libm function?
What is the issue?

Why is the Arénaire development process so slow?
What is the issue?

Why is the Arénaire development process so slow?

Actually, I thought we were always doing the same things...
Task: implement $f$ in a domain $[a, b]$ with an accuracy of $k$ bits
Task: implement $f$ in a domain $[a, b]$ with an accuracy of $k$ bits

- Analyze the behaviour of $f$ in $[a, b]$
Steps in the implementation of a function

**Task**: implement $f$ in a domain $[a, b]$ with an accuracy of $k$ bits

- Analyze the behaviour of $f$ in $[a, b]$
- **Find an appropriate range reduction**
Task: implement $f$ in a domain $[a, b]$ with an accuracy of $k$ bits

- Analyze the behaviour of $f$ in $[a, b]$
- Find an appropriate range reduction
- Compute an approximation polynomial $p^*$

Steps in the implementation of a function
Steps in the implementation of a function

Task: implement $f$ in a domain $[a, b]$ with an accuracy of $k$ bits

- Analyze the behaviour of $f$ in $[a, b]$
- Find an appropriate range reduction
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
Steps in the implementation of a function

Task: implement $f$ in a domain $[a, b]$ with an accuracy of $k$ bits

- Analyze the behaviour of $f$ in $[a, b]$
- Find an appropriate range reduction
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
- Implement $p$ in floating-point arithmetic
Steps in the implementation of a function

Task: implement $f$ in a domain $[a, b]$ with an accuracy of $k$ bits

- Analyze the behaviour of $f$ in $[a, b]$
- Find an appropriate range reduction
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
- Implement $p$ in floating-point arithmetic
- Bound round-off errors, write a proof
Task: implement $f$ in a domain $[a, b]$ with an accuracy of $k$ bits

- Analyze the behaviour of $f$ in $[a, b]$
- Find an appropriate range reduction
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
- Implement $p$ in floating-point arithmetic
- Bound round-off errors, write a proof
- Check the proof for mistakes
Task: implement $f$ in a domain $[a, b]$ with an accuracy of $k$ bits

- Analyze the behaviour of $f$ in $[a, b]$
- Find an appropriate range reduction
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
- Implement $p$ in floating-point arithmetic
- Bound round-off errors, write a proof
- Check the proof for mistakes
- Bound and proof the approximation error: $\| \frac{p-f}{f} \|_\infty$
Steps in the implementation of a function

Task: implement $f$ in a domain $[a, b]$ with an accuracy of $k$ bits

- Analyze the behaviour of $f$ in $[a, b]$
- Find an appropriate range reduction
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
- Implement $p$ in floating-point arithmetic
- Bound round-off errors, write a proof
- Check the proof for mistakes
- Bound and proof the approximation error: $\| \frac{p-f}{f} \|_\infty$
- Integrate everything
A prototype, automatic toolchain for the implementation process
A prototype toolchain – 1

A prototype, automatic toolchain for the implementation process

Joint work by
- S. Chevillard (floating-point polynomial approximation part)
- Ch. Lauter (implementation and proof part)
- G. Melquiond (Gappa)
- and other Arénaire members
A prototype toolchain – 1

A prototype, automatic toolchain for the implementation process

- Joint work by
  - S. Chevillard (floating-point polynomial approximation part)
  - Ch. Lauter (implementation and proof part)
  - G. Melquiond (Gappa)
  - and other Arénaire members

- Written in
  - Pari/GP
  - C, C++
  - Shell scripts
  - an internal language: arenaireplot
A prototype toolchain – 1

A prototype, automatic toolchain for the implementation process

- Joint work by
  - S. Chevillard (floating-point polynomial approximation part)
  - Ch. Lauter (implementation and proof part)
  - G. Melquiond (Gappa)
  - and other Arénaire members

- Written in
  - Pari/GP
  - C, C++
  - Shell scripts
  - an internal language: arenaireplot

- Targetted to
  - portable C implementations
  - using double, double-double and triple-double arithmetic
  - with easy-to-handle Horner evaluation
A prototype toolchain – 2

Automatic handling of the following sub-problems:

- Find an appropriate range translation
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form:
- Implement $p$ in floating-point arithmetic
- Bound round-off errors, write a proof
- Check the proof for errors
- Bound and proof the approximation error: $\| p - f \|_{\infty}$
- Analyze the behaviour of $f$ in $[a, b]$
- Find a range reduction using tables etc.
- Integrate everything
A prototype toolchain – 2

Automatic handling of the following sub-problems:

- Find an appropriate range translation
A prototype toolchain – 2

Automatic handling of the following sub-problems:

- Find an appropriate range translation
- Compute an approximation polynomial $p^*$
A prototype toolchain – 2

Automatic handling of the following sub-problems:

- Find an appropriate range translation
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
Automatic handling of the following sub-problems:

- Find an appropriate range translation
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
- Implement $p$ in floating-point arithmetic
Automatic handling of the following sub-problems:

- Find an appropriate range translation
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
- Implement $p$ in floating-point arithmetic
- Bound round-off errors, write a proof
Automatic handling of the following sub-problems:

- Find an appropriate range translation
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
- Implement $p$ in floating-point arithmetic
- Bound round-off errors, write a proof
- Check the proof for errors
Automatic handling of the following sub-problems:

- Find an appropriate range translation
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
- Implement $p$ in floating-point arithmetic
- Bound round-off errors, write a proof
- Check the proof for errors
- Bound and proof the approximation error: $\| \frac{p-f}{f} \|_\infty$
Automatic handling of the following sub-problems:

- Find an appropriate range translation
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
- Implement $p$ in floating-point arithmetic
- Bound round-off errors, write a proof
- Check the proof for errors
- Bound and proof the approximation error: $\| \frac{p-f}{f} \|_\infty$

Missing parts:
- Analyze the behaviour of $f$ in $[a, b]$
A prototype toolchain – 2

Automatic handling of the following sub-problems:

- Find an appropriate range translation
- Compute an approximation polynomial $p^*$
- Bring the coefficients of $p^*$ into floating-point form: $p$
- Implement $p$ in floating-point arithmetic
- Bound round-off errors, write a proof
- Check the proof for errors
- Bound and proof the approximation error: $\| \frac{p-f}{f} \|_\infty$

Missing parts:

- Analyze the behaviour of $f$ in $[a, b]$
- Find a range reduction using tables etc.
A prototype toolchain – 2

Automatic handling of the following sub-problems:

- Find an appropriate range translation
- Compute an approximation polynomial \( p^* \)
- Bring the coefficients of \( p^* \) into floating-point form: \( p \)
- Implement \( p \) in floating-point arithmetic
- Bound round-off errors, write a proof
- Check the proof for errors
- Bound and proof the approximation error: \( \| \frac{p-f}{f} \|_\infty \)

Missing parts:

- Analyze the behaviour of \( f \) in \([a, b]\)
- Find a range reduction using tables etc.
- Integrate everything
Task: Implement

\[ f(x) = e^{\cos x^2 + 1} \]

in the interval

\[ I = [-2^{-8}; 2^{-5}] \]

with at least 66 bits of accuracy
Task: Implement

\[ f(x) = e^{\cos x^2 + 1} \]

in the interval

\[ I = [-2^{-8}; 2^{-5}] \]

with at least 66 bits of accuracy

Let’s try it out...
Results on new functions

Last functions in crlibm
Results on new functions

Last functions in crlibm

- sinpi(x), cospi(x), tanpi(x)
Results on new functions

Last functions in crlibm

- \( \sin \pi(x) \), \( \cos \pi(x) \), \( \tan \pi(x) \)
- correctly rounded in two approximation steps
Last functions in crlibm

- \texttt{sinpi(x)}, \texttt{cospi(x)}, \texttt{tanpi(x)}
- correctly rounded in two approximation steps
- both evaluation codes generated automatically
Results on new functions

Last functions in crlibm

- \( \sin \pi(x), \cos \pi(x), \tan \pi(x) \)
- correctly rounded in two approximation steps
- both evaluation codes generated automatically
- duration: two days
And Intel’s customers?

Could this be interesting for Intel’s customers?

- Faster-to-market and cheaper implementations?
And Intel’s customers?

Could this be interesting for Intel’s customers?

- Faster-to-market and cheaper implementations?
- Easier approach to Gappa usage?
Could this be interesting for Intel’s customers?

- Faster-to-market and cheaper implementations?
- Easier approach to Gappa usage?
- Better maintainability of some code parts?
Could this be interesting for Intel’s customers?

- Faster-to-market and cheaper implementations?
- Easier approach to Gappa usage?
- Better maintainability of some code parts?
- Compilers that inline composite functions like $e^{\cos x^2 + 1}$?
Conclusion

Introduction

Correct rounding of $x^y$

Automatic implementation of libm functions

Conclusion
More correctly rounded functions:
More correctly rounded functions:

- High performance on average can be achieved for $o(x^y)$

Worst-case bounding might become feasible for $x^y$:

A certificate that 2500 bits suffice for double seems to cost about 500 machine-years.

Attacking double-extended precision:

Worst-case search would be possible for univariate functions.

We have tools for simplifying the implementation process.

More numerical knowledge inside high-level compilers.

Remove the numerical burden from low-level C/Fortran.

Numerical algorithms described in a high-level language.

Highly investigated by Arénaire.

Need: more and more computational power.

Advancements in elementary function development - Intel Portland - 10 October 2007
More correctly rounded functions:
- High performance on average can be achieved for \( o(x^y) \)
- Worst case bounding might become feasible for \( x^y \):
  a certificate that 2500 bits suffice for double seems to cost about 500 machine-years
More correctly rounded functions:
- High performance on average can be achieved for $o(x^y)$
- Worst case bounding might become feasible for $x^y$:
  a certificate that 2500 bits suffice for double seems to cost about 500 machine-years

Attacking double-extended precision:
More correctly rounded functions:
- High performance on average can be achieved for $\circ (x^y)$
- Worst case bounding might become feasible for $x^y$:
  a certificate that 2500 bits suffice for double seems to cost about 500 machine-years

Attacking double-extended precision:
- Worst-case search would be possible for univariate functions
More correctly rounded functions:

- High performance on average can be achieved for $\circ(x^y)$
- Worst case bounding might become feasible for $x^y$:
  a certificate that 2500 bits suffice for double seems to cost about 500 machine-years

Attacking double-extended precision:

- Worst-case search would be possible for univariate functions
- We have tools for simplifying the implementation process
Conclusion and outlooks...

- More correctly rounded functions:
  - High performance on average can be achieved for $o(x^y)$
  - Worst case bounding might become feasible for $x^y$:
    a certificate that 2500 bits suffice for double seems to cost about 500 machine-years
- Attacking double-extended precision:
  - Worst-case search would be possible for univariate functions
  - We have tools for simplifying the implementation process
- More numerical knowledge inside high-level compilers
More correctly rounded functions:

- High performance on average can be achieved for $o(x^y)$
- Worst case bounding might become feasible for $x^y$:
  - a certificate that 2500 bits suffice for double seems to cost about 500 machine-years

Attacking double-extended precision:

- Worst-case search would be possible for univariate functions
- We have tools for simplifying the implementation process

More numerical knowledge inside high-level compilers

- Remove the numerical burden from low-level C/Fortran
More correctly rounded functions:
- High performance on average can be achieved for $\circ(x^y)$
- Worst case bounding might become feasible for $x^y$: a certificate that 2500 bits suffice for double seems to cost about 500 machine-years

Attacking double-extended precision:
- Worst-case search would be possible for univariate functions
- We have tools for simplifying the implementation process

More numerical knowlegde inside high-level compilers
- Remove the numerical burden from low-level C/Fortran
- Numerical algorithms described in a high-level language
Conclusion and outlooks...

- More correctly rounded functions:
  - High performance on average can be achieved for $o(x^y)$
  - Worst case bounding might become feasible for $x^y$:
    a certificate that 2500 bits suffice for double seems to cost about 500 machine-years

- Attacking double-extended precision:
  - Worst-case search would be possible for univariate functions
  - We have tools for simplifying the implementation process

- More numerical knowledge inside high-level compilers
  - Remove the numerical burden from low-level C/Fortran
  - Numerical algorithms described in a high-level language
  - Highly investigated by Arénaire

- Need: more and more computational power
Thank you for your attention!

Questions?