15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations,

Novosibirsk, Institute of Computational Technologies of the Siberian Department of the Russian Academy of Sciences

Automatic Code Transformation to Optimize Accuracy and Speed in Floating-Point Arithmetic

P. Langlois, M. Martel and L. Thévenoux

DALI-LIRMM Research Team, Perpignan, France

September 24, 2012

Outline

Outline

Overview: Automatic Code Transformation...

IEEE754 FP arithmetic may suffer from inaccuracy

- critical matter in scientific computing, embedded systems,...
- existing solutions reserved to experts and implemented manually

Our objective: accurate code synthesis

Allows standard developer to **automatically** transform his/her code

Take into account two opposite criteria

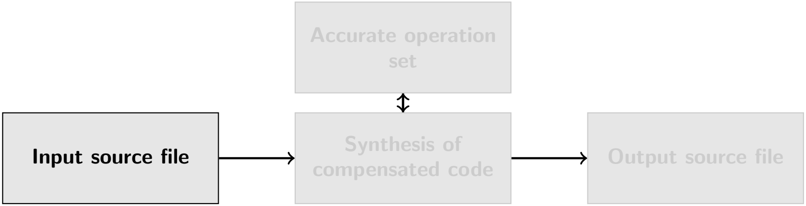
- accuracy
- execution time

We present here a first step towards our final objective →

Synopsis

We propose to automatically introduce at the compile-time...

a compensation step



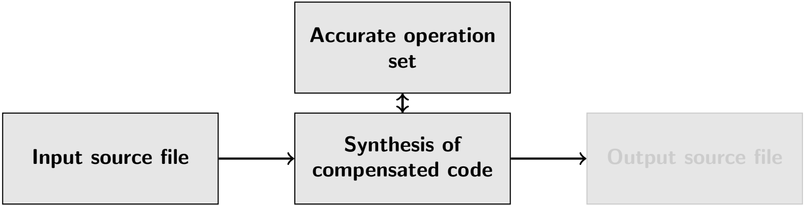
Parse C source code

How we do that? →

Synopsis

We propose to automatically introduce at the compile-time. . .

a compensation step



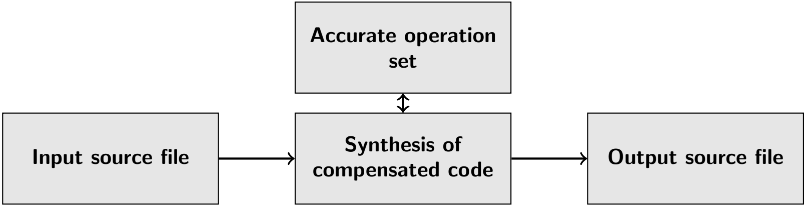
Tool which replace floating point operations by compensated algorithms
Compensated terms are accumulated and added to original computations

How we do that? →

Synopsis

We propose to automatically introduce at the compile-time. . .

a compensation step



Generate new code
Provide a compensated computation that improves the accuracy

How we do that? →

Outline

IEEE754 Floating-Point Arithmetic

Floating-point numbers are approximations of real numbers

Let $x \in \mathbb{R}$, $(-1)^s \cdot b^e \cdot m$ express $x \in \mathbb{F}$

The standard define

- Rounding modes: nearest, toward 0, $+\infty$, $-\infty$
- Several formats: binary32, binary64,...

These errors can cause big human and material damages $\rightarrow$

IEEE754 Floating-Point Arithmetic

Floating-point numbers are approximations of real numbers

Let $x \in \mathbb{R}$, $(-1)^s \cdot b^e \cdot m$ express $x \in \mathbb{F}$

Finite representation implies accuracy variations and losses

- Rounding errors, cancellations, absorptions

$$(a+b)-a=0 \quad \text{if } a \gg b \,^*$$

*absorption example

These errors can cause big human and material damages →

Existing Techniques

Solutions exists to prevent inaccuracy behaviors

- Extending the computing precision size

(software libraries (MPFR), extended arithmetic)

Among these possibilities we choose to generate compensated algorithms $\rightarrow$

Existing Techniques

Solutions exists to prevent inaccuracy behaviors

- Extending the computing precision size

  (software libraries (MPFR), extended arithmetic)

- Rewriting expressions

  (rewriting tools [Ioualalen Martel])

example: $(a+b)-a=0 \quad \rightsquigarrow \quad (a-a)+b=b \quad \text{if } a \gg b$

Among these possibilities we choose to generate compensated algorithms $\rightarrow$

Existing Techniques

Solutions exists to prevent inaccuracy behaviors

- Extending the computing precision size

  (software libraries (MPFR), extended arithmetic)

- Rewriting expressions

  (rewriting tools [Ioualalen Martel])

  example: $(a+b)-a=0 \quad \rightsquigarrow \quad (a-a)+b=b \quad \text{if } a \gg b$

- More accurate algorithms

  (sorting (sum), compensated algorithms,...)

Among these possibilities we choose to generate compensated algorithms $\rightarrow$

Compensated Algorithms – TwoSum EFT

To compensate a sum

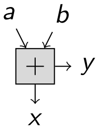
1: $[x, y] = TwoSum(a, b)$
2: $x = fl(a + b)$
3: $z = fl(x - a)$
4: $y = fl((a - (x - z)) + (b - z))$

TwoSum (Knuth)

EFT (Error-Free Transformation:

$$x + y = a + b$$

optimal (cost, time) [Kornerup et al.]



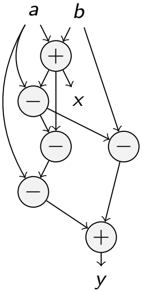


Figure: TwoSum: 6 flops

Compensated Algorithms – TwoProduct EFT

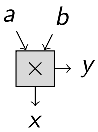
To compensate a product

1: $[x, y] = TwoProduct(a, b)$
2: $x = fl(a \cdot b)$
3: $[a_1, a_2] = Split(a)$
4: $[b_1, b_2] = Split(b)$
5: $y = fl(a_2 \cdot b_2 - (((x - a_1 \cdot b_1) - a_2 \cdot b_1) - a_1 \cdot b_2))$

TwoProduct (Veltkamp)

1: $[x, y] = Split(a)$
2: $factor = 2^{27} + 1$
3: $c = fl(factor \cdot a)$
4: $x = fl(c - (c - a))$
5: $y = fl(a - x)$

Split (Dekker)



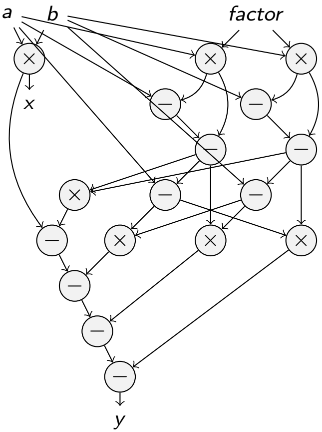


Figure: TwoProduct: 17 flops

Methodology

Principle of the compensation step

Transform each floating-point operations ($\oplus, \ominus, \otimes$) using compensation algorithms (TwoSum, TwoProduct) and accumulate compensate terms in parallel of original computations

Perspectives: keep in mind the execution time criteria

Because these transformations can reduce the execution time...

Methodology

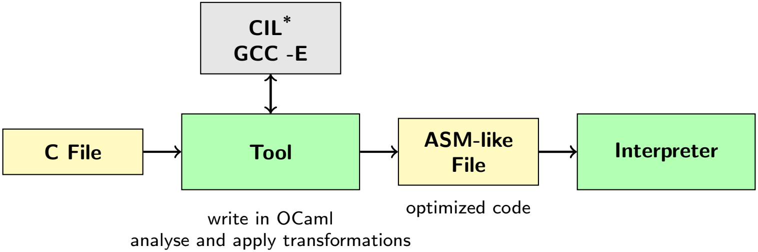


Figure: Tool schematic of our methodology implementation

* [Necula et al.]

Advantages and Drawbacks

Advantages

- Automatic → *fast, don't need to be an expert*
- Compile-time optimization → *data* **independence**

Drawbacks

- Don't treat all the basic operations ($\div, \sqrt{}, \ldots$)

  → *but they're existing solutions (Newton approx.,. . .)*
- Can highly reduce performances

  → *but we have some ideas (developed in the next section)*

Outline

Code Analysis – SSA Conversion

First compilation step

- Static Single Assignment Form

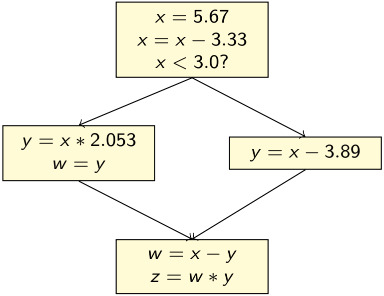


Figure: Control flow graph of an example program

Code Analysis – SSA Conversion

First compilation step

- Static Single Assignment Form
- Each variable is affected only one time *(make optimisation applications easier)*
- Add special information called ϕ nodes *(when variable can take different paths)*

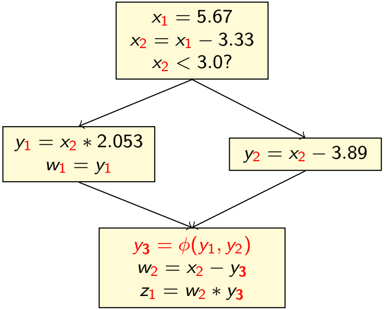


Figure: Control flow graph of an example program in SSA form

Code Analysis – FP Computation Sequence Detection

Second step

- Each FP operation sequences with $\oplus, \ominus, \otimes$ operation inside a basic block
- *(special case: if the sequence contains a single operation and if it not included in a loop: no transformation)*

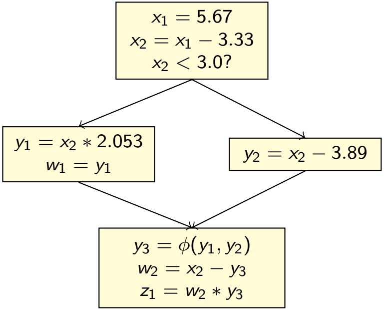


Figure: Sequence Detection

We are ready to compensation transformation $\rightarrow$

Code Analysis – FP Computation Sequence Detection

Second step

- Each FP operation sequences with $\oplus, \ominus, \otimes$ operation inside a basic block
- *(special case: if the sequence contains a single operation and if it not included in a loop: no transformation)*

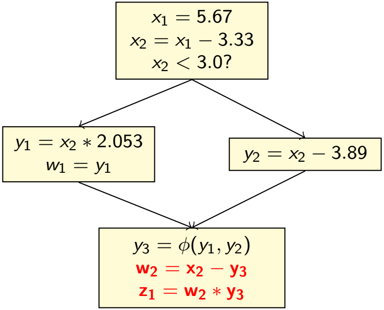


Figure: Sequence Detection

We are ready to compensation transformation $\rightarrow$

Code Transformation – Principle

Transformation step

- Transform $\oplus, \ominus$ in **TwoSum**
- Transform $\otimes$ in **TwoProduct**

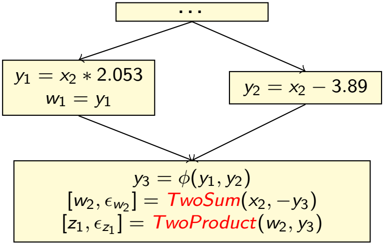


Figure: Compensated code synthesis

Code Transformation – Principle

Transformation step

- Transform $\oplus, \ominus$ in **TwoSum**
- Transform $\otimes$ in **TwoProduct**
- Compensation terms accumulation

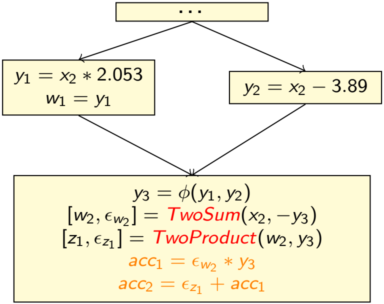


Figure: Compensated code synthesis

Code Transformation – Principle

Transformation step

- Transform $\oplus, \ominus$ in **TwoSum**
- Transform $\otimes$ in **TwoProduct**
- Compensation terms accumulation
- Final compensation

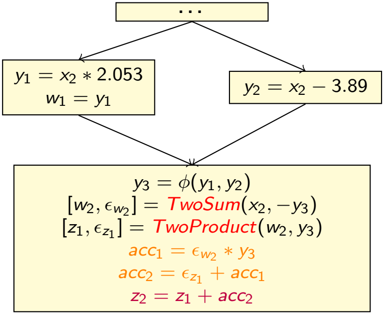


Figure: Compensated code synthesis

Code Transformation – Pattern Introduction

A variable...

A variable x becomes a pair (x, ϵ_x), with:

x, the value of variable
ϵ_x, the initial error *(supposed null here)*

A return of an operator...

A return of an operator $\oplus, \ominus, \otimes$ becomes a pair (x, ϵ_x), with:

x, the result of the operator
ϵ_x, the accumulated compensated value

Code Transformation – Sum Pattern Transformation

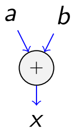


Figure: Pattern A

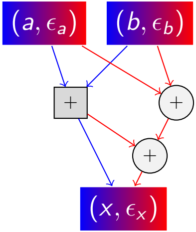


Figure: Transformation

$$x = a + b$$
$$\epsilon_x = (\epsilon_a + \epsilon_b) + \epsilon_{a+b}$$

Code Transformation – Product Pattern Transformation

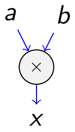


Figure: Pattern B

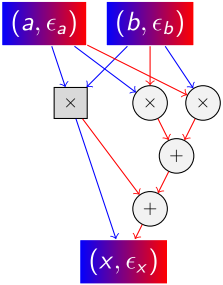
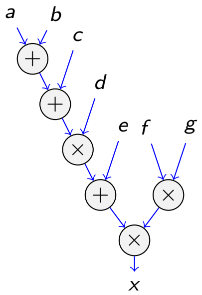


Figure: Transformation

$$x = a \times b$$
$$\epsilon_x = [(\epsilon_a \times b) + (\epsilon_b \times a)] + \epsilon_{a \times b}$$

Our transformations are not EFT: we loose the second order term

Code Transformation – Example

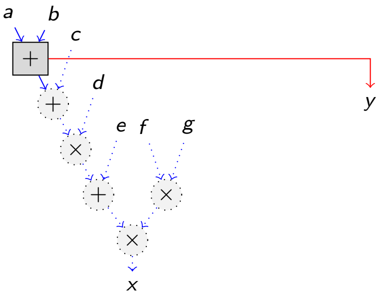


Before transformation

Let the following expression of x

$$x = (((((a + b) + c) \times d) + e) \times (f \times g)$$

Code Transformation – Example



Pattern A transformation

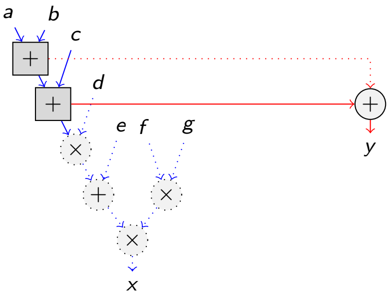
x is the result of $a \oplus b$

$$x = a + b$$

y is defined by the generated error of the TwoSum algorithm

$$y = \epsilon_{a+b}$$

Code Transformation – Example



Pattern A transformation

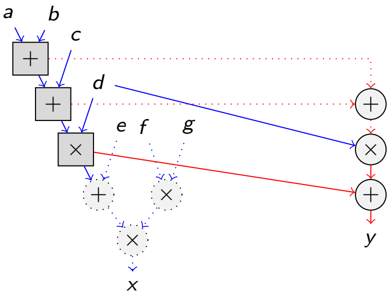
x is the result of $x \oplus c$

$$x = x + c$$

y is defined by the adding of the inherited error and the generated error of the TwoSum algorithm

$$y = y + \epsilon_{x+c}$$

Code Transformation – Example



Pattern B transformation

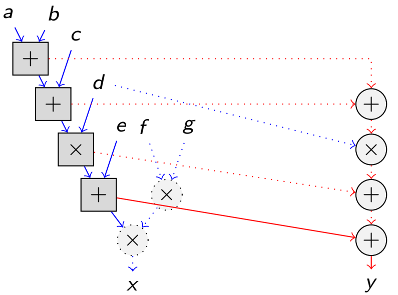
x is the result of $x \otimes d$

$$x = x \times d$$

y is defined by the adding of a function of the inherited error and the generated error of TwoProduct algorithm

$$y = (y \times d) + \epsilon_{x \times d}$$

Code Transformation – Example



Pattern A transformation

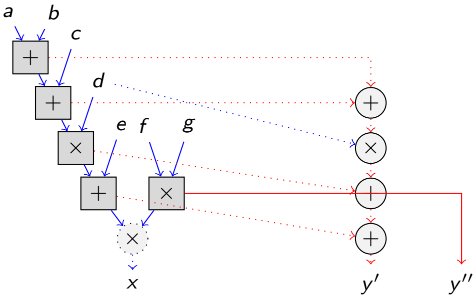
x is the result of $x \oplus e$

$$x = x + e$$

y is defined by the adding of the inherited error and the generated error of the TwoSum algorithm

$$y = y + \epsilon_{x+e}$$

Code Transformation – Example



Pattern B transformation

x' is equal to x and x'' is equal to $f \otimes g$

$$x' = x$$

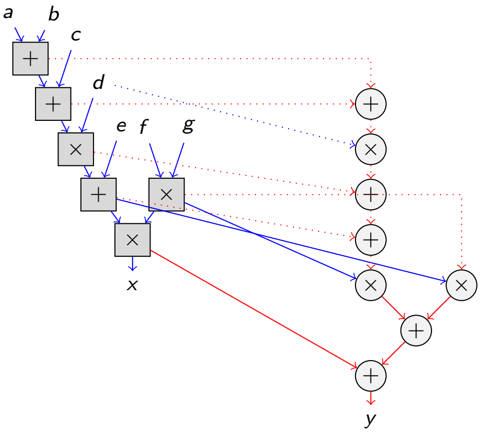
$$x'' = (f \times g)$$

y' is equal to y and y'' is the generated error of the TwoProduct algorithm

$$y' = y$$

$$y'' = \epsilon_{f \times g}$$

Code Transformation – Example



Pattern B transformation

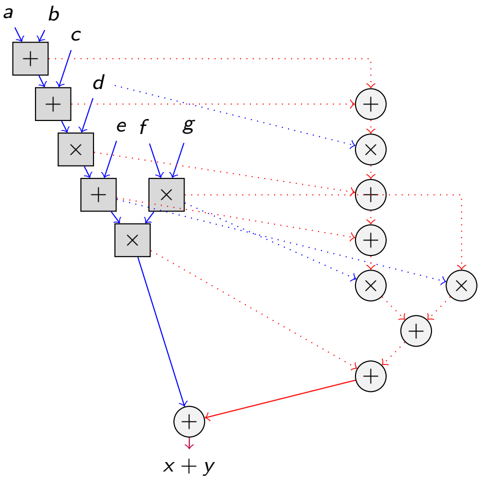
x is the result of $x' \otimes x''$

$$x = x' \times x''$$

y is defined by the adding of a function of the inherited errors and the generated error of the TwoProduct algorithm

$$y = ((y' \times x") + (y'' \times x')) + \epsilon_{x' \times x''}$$

Code Transformation – Example



Final result transformation

x is the result of the adding of the expression and the compensated accumulated terms

$$x = x + y$$

Execution-Time Criteria

In order to save execution-speed, we must add an execution-time criteria!

Ideas to explore...

- Propose trade-offs between accuracy and speed [SCAN10, PASCO10] *(for example: compensate one operation on two/three/...)*

Execution-Time Criteria

In order to save execution-speed, we must add an execution-time criteria!

Ideas to explore...

- Propose trade-offs between accuracy and speed [SCAN10, PASCO10] *(for example: compensate one operation on two/three/...)*
- Use new instructions (`ADD3`, `FMA`) [Ogita et al.]

1: $[x, y] = TwoSumAdd3(a, b)$
2: $x = fl(a + b)$
3: $y = add3(a, b, -x)$

TwoSumADD3

1: $[x, y] = TwoProductFMA(a, b)$
2: $x = fl(a + b)$
3: $y = fma(a, b, -x)$

TwoProductFMA

Execution-Time Criteria

In order to save execution-speed, we must add an execution-time criteria!

Ideas to explore...

- Propose trade-offs between accuracy and speed [SCAN10, PASCO10] *(for example: compensate one operation on two/three/...)*
- Use new instructions (`ADD3`, `FMA`) [Ogita et al.]

1: $[x, y] = TwoSumAdd3(a, b)$
2: $x = fl(a + b)$
3: $y = add3(a, b, -x)$

TwoSumADD3

1: $[x, y] = TwoProductFMA(a, b)$
2: $x = fl(a + b)$
3: $y = fma(a, b, -x)$

TwoProductFMA

- Exploit Instruction Level Parallelism (ILP). *cf. More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms [Langlois Louvet]*

Example – Introduction

Example from [Graillat et al.]

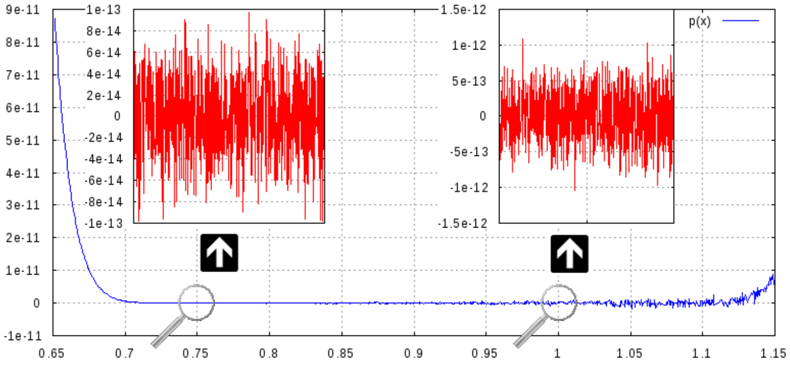
Authors evaluate the Horner form of the polynomial $p(x) = (0.75 - x)^5(1 - x)^{11}$ close to its multiple roots. They show that compensation improves the accuracy

Can we reproduce automatically these results?

We apply our method to this test case aiming to reproduce automatically what experts have done manually

Example – Results

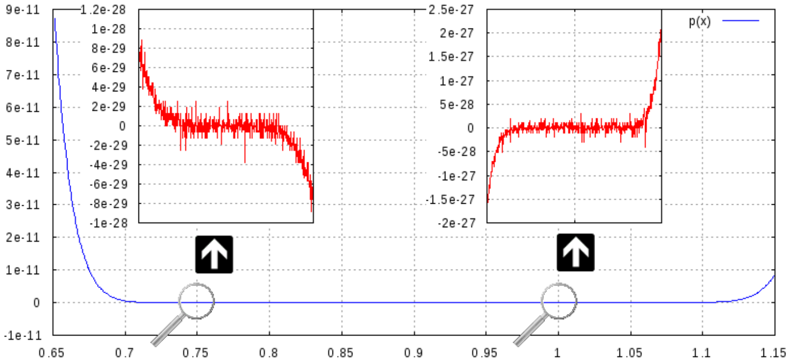
Figure: Results of $p(x)$ and zooms on its roots **before** automatic transformation



As expected original results are meaningless

Example – Results

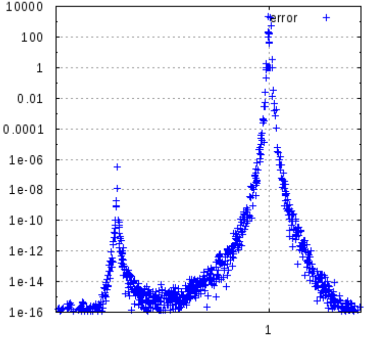
Figure: Results of $p(x)$ and zooms on its roots **after** automatic transformation



The transformed code provides more accuracy and yields a smoother polynomial evaluation

Example – Results

Figure: Relative error computed with CompHorner (left) and with the automatically generated code (right)



Our tool allows non expert user to obtain **automatically**, **quickly** and **easily** such accuracy improvement

Outline

Conclusion & Perspectives

We have...

- a tool able to parse a large subset of C and to apply automatically compensations on basic floating-point operations and to generate optimized code
- similar results to expert manual solution in our test cases

We need...

- to apply our tool on other test cases (Chebyshev, Bernstein...)
- to propose optimizations for execution-time criteria
- to write formal proofs of our transformations (estimate their impact)
- to add other transformations ($\div, \sqrt{}, \ldots$)

Thank You

Questions?

laurent.thevenoux@univ-perp.fr

Bibliography

[Ioualalen Martel] A. Ioualalen, M. Martel. *Sardana: an Automatic Tool for Numerical Accuracy Optimization*. 2012.

[Necula et al.] G. Necula, S. McPeak, S. Rahul, W. Weimer. *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*. 2002.

[Kornerup et al.] P. Kornerup, V. Lefèvre, N. Louvet, J.M. Muller. *On the Computation of Correctly-Rounded Sums*. 19th IEEE Symposium on Computer Arithmetic - Arith'19, 2009.

[Ogita et al.] T. Ogita, S.M. Rump, S. Oishi. *Accurate sum and dot product*. SIAM Journal on Scientific Computing 2005.

[SCAN10] P. Langlois, M. Martel, L. Thévenoux. *Trade-off Between Accuracy and Time for Automatically Generated Summation Algorithms*. SCAN 2010: 14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics, 2010.

[PASCO10] P. Langlois, M. Martel, L. Thévenoux. *Accuracy Versus Time: A Case Study with Summation Algorithms*. PASCO 10: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation, 2010.

[Graillat et al.] S. Graillat, P. Langlois, N. Louvet. *Algorithms for Accurate, Validated and Fast Polynomial Evaluation*. 2009.

[Langlois Louvet] P. Langlois, N. Louvet *More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms, 2007*.