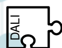
Soutenance de thèse – 4 Juillet 2014
Université de Perpignan Via Domitia

Synthèse de code avec compromis entre performance et précision

en arithmétique flottante IEEE 754

Laurent THÉVENOUX

Membres du jury :
Philippe LANGLOIS et Matthieu MARTEL .... (directeurs)
Jean-Marie CHESNEAUX et Sylvie PUTOT .... (rapporteurs)
Jean-Michel MULLER .... (examinateur)



Équipe-projet DALI, Univ. Perpignan Via Domitia. LIRMM, CNRS: UMR 5506 - Univ. Montpellier 2

Deux des **préoccupations majeures** de l'informatique

Précision

données
modèle
calculs

Performance

taille du code
consommation
vitesse/temps d'exécution

Deux des **préoccupations majeures** de l'informatique

Précision

données
modèle
calculs

Performance

taille du code
consommation
vitesse/temps d'exécution

Deux des **préoccupations majeures** de l'informatique

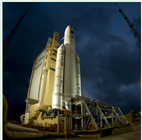
Précision

données
modèle
calculs

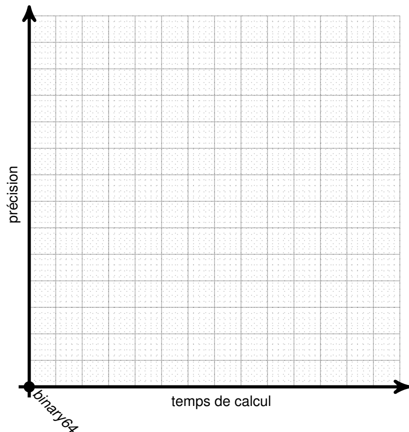
Performance

taille du code
consommation
vitesse/temps d'exécution

Critiques dans de nombreux systèmes



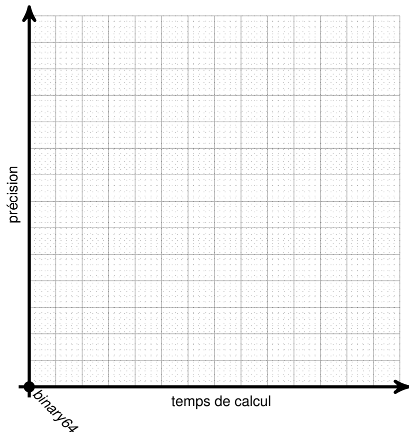
Les relations entre le temps et la précision



Améliorer la précision

- Motivation : certains résultats imprécis
  - Ariane 5, Patriot

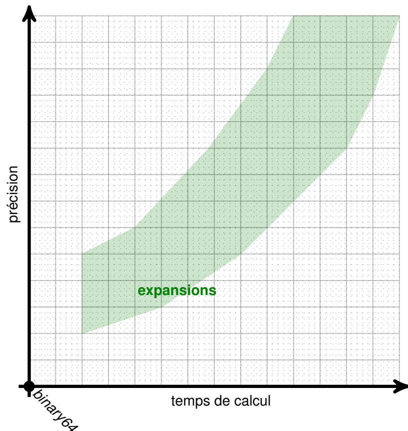
Les relations entre le temps et la précision



Améliorer la précision

- Motivation : certains résultats imprécis
  - Ariane 5, Patriot
- Précision étendue :

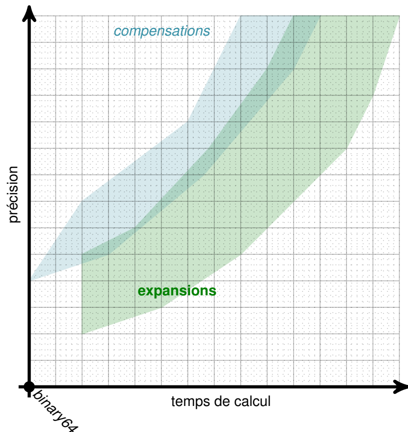
Les relations entre le temps et la précision



Améliorer la précision

- Motivation : certains résultats imprécis
  - Ariane 5, Patriot
- Précision étendue :
  - expansions [She97, HBL01]

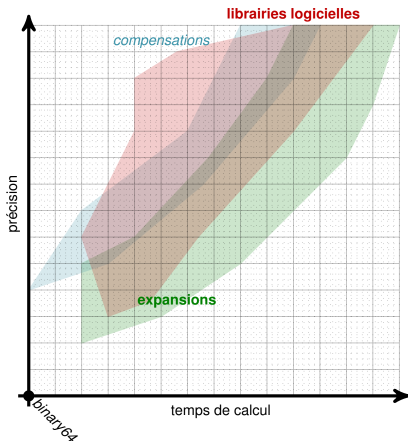
Les relations entre le temps et la précision



Améliorer la précision

- Motivation : certains résultats imprécis
  - Ariane 5, Patriot
- Précision étendue :
  - expansions [She97, HBL01]
  - compensations [ROO05, GLL09]

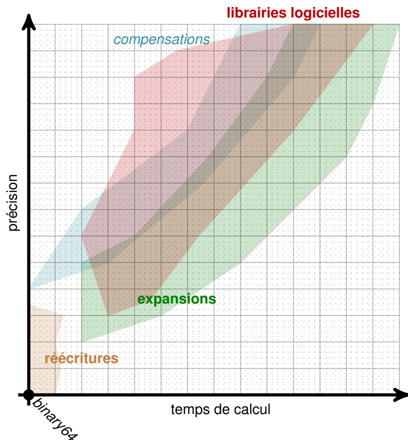
Les relations entre le temps et la précision



Améliorer la précision

- Motivation : certains résultats imprécis
  - Ariane 5, Patriot
- Précision étendue :
  - expansions [She97, HBL01]
  - compensations [ROO05, GLL09]
  - librairies logicielles [MPFR]

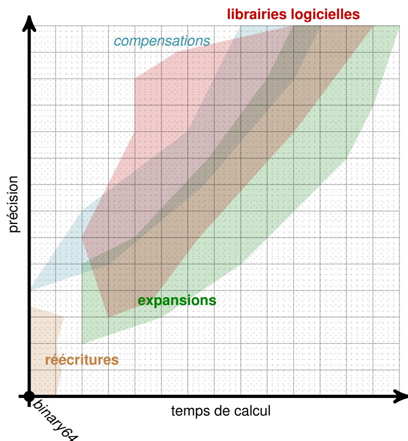
Les relations entre le temps et la précision



Améliorer la précision

- Motivation : certains résultats imprécis
  - Ariane 5, Patriot
- Précision étendue :
  - expansions [She97, HBL01]
  - compensations [ROO05, GLL09]
  - librairies logicielles [MPFR]
  - réécritures [Iou13]

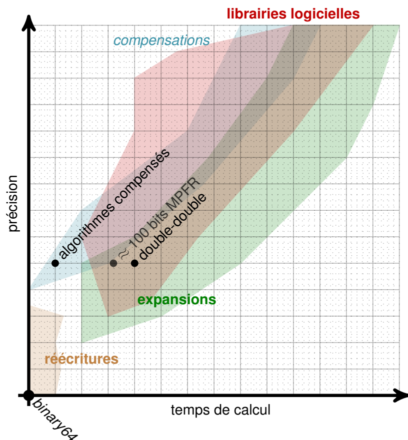
Les relations entre le temps et la précision



Améliorer la précision

- Motivation : certains résultats imprécis
  - Ariane 5, Patriot
- Précision étendue :
  - **expansions** [She97, HBL01]
  - *compensations* [ROO05, GLL09]
  - **librairies logicielles** [MPFR]
  - **réécritures** [Iou13]
- Distinguer le **générique** du *spécifique*

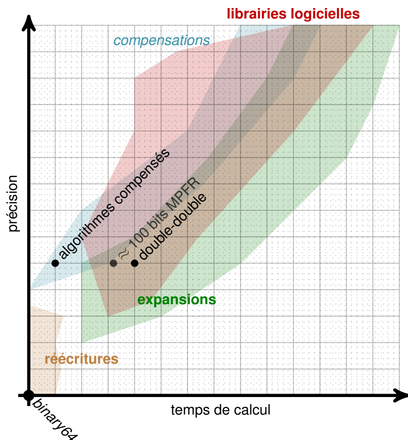
Les relations entre le temps et la précision



Améliorer la précision

- Motivation : certains résultats imprécis
  - Ariane 5, Patriot
- Précision étendue :
  - **expansions** [She97, HBL01]
  - *compensations* [ROO05, GLL09]
  - **librairies logicielles** [MPFR]
  - **réécritures** [Iou13]
- Distinguer le **générique** du *spécifique*
- Précision arbitraire : doublement
  - double-double
  - algorithmes compensés

Les relations entre le temps et la précision



Améliorer la précision

- Motivation : certains résultats imprécis
  - Ariane 5, Patriot
- Précision étendue :
  - **expansions** [She97, HBL01]
  - *compensations* [ROO05, GLL09]
  - **librairies logicielles** [MPFR]
  - **réécritures** [Iou13]
- Distinguer le **générique** du *spécifique*
- Précision arbitraire : doublement
  - double-double
  - algorithmes compensés
- Difficulté supplémentaire : le temps et la précision ont tendance à s'opposer

Vue d’ensemble des travaux de la thèse

Soit la problématique :

1. **Améliorer** la précision
2. En **réduisant la dégradation** de la **performance**
3. Avec de plus l’objectif : processus **automatique**

Vue d'ensemble des travaux de la thèse

Soit la problématique :

1. **Améliorer** la précision
2. En **réduisant la dégradation** de la **performance**
3. Avec de plus l'objectif : processus **automatique**

Aujourd'hui il est facile de doubler la précision presque automatiquement

Peut-on faire plus **efficace** et aussi **précis** ?

Allons plus loin, peut-on être encore plus **efficace** quitte à sacrifier une partie de la **précision** ?

Vue d'ensemble des travaux de la thèse

Soit la problématique :

1. **Améliorer** la précision
   Transformation à base d'EFT (à l'image de la compensation)
2. En **réduisant la dégradation** de la **performance**
   Des stratégies de transformation avec compromis
3. Avec de plus l'objectif : processus **automatique**
   Synthèse de code

Aujourd'hui il est facile de doubler la précision presque automatiquement

Peut-on faire plus **efficace** et aussi **précis** ?

Allons plus loin, peut-on être encore plus **efficace** quitte à sacrifier une partie de la **précision** ?

Plan de l'exposé

1. Notions d'arithmétique flottante et d'amélioration de la précision
   - Arithmétique flottante IEEE 754
   - Transformations sans erreur
   - Arithmétique double-double et algorithmes compensés
2. Transformation automatique de programme
   - Mise en place : l'outil CoHD
   - Méthodologie : correction de la précision
   - Les transformations de la somme et du produit
   - Résultats expérimentaux avec comparaisons à l'existant
3. Synthèse de code pour des compromis entre précision et temps d'exécution
   - Mise en place : l'outil SyHD
   - Des stratégies de transformation de programme avec compromis
   - Ensemble de programme et critères de sélection
   - Résultats expérimentaux
4. Conclusions et perspectives

Plan de l'exposé

1. Notions d'arithmétique flottante et d'amélioration de la précision
   - Arithmétique flottante IEEE 754
   - Transformations sans erreur
   - Arithmétique double-double et algorithmes compensés

2. Transformation automatique de programme
   - Mise en place : l'outil CoHD
   - Méthodologie : correction de la précision
   - Les transformations de la somme et du produit
   - Résultats expérimentaux avec comparaisons à l'existant

3. Synthèse de code pour des compromis entre précision et temps d'exécution
   - Mise en place : l'outil SyHD
   - Des stratégies de transformation de programme avec compromis
   - Ensemble de programme et critères de sélection
   - Résultats expérimentaux

4. Conclusions et perspectives

Arithmétique flottante IEEE 754

Permet la représentation des nombres réels en machine [IEEE754]

- Ou plutôt leurs approximations : impossible de les représenter exactement en machine
- Un standard qui définit :

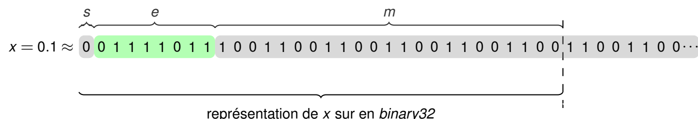
une représentation	$(-1)^s \times m \times 2^e$
des arrondis	au plus près (RN), vers 0 (RZ), $+\infty$ (RU), $-\infty$ (RD)
des formats	*binary32*, *binary64*, etc

Arithmétique flottante IEEE 754

Permet la représentation des nombres réels en machine [IEEE754]

- Ou plutôt leurs approximations : impossible de les représenter exactement en machine
- Un standard qui définit :

une représentation	$(-1)^s \times m \times 2^e$
des arrondis	au plus près (RN), vers 0 (RZ), $+\infty$ (RU), $-\infty$ (RD)
des formats	*binary32*, *binary64*, etc

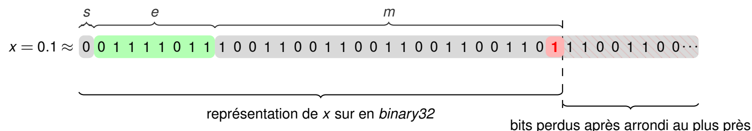


Arithmétique flottante IEEE 754

Permet la représentation des nombres réels en machine [IEEE754]

- Ou plutôt leurs approximations : impossible de les représenter exactement en machine
- Un standard qui définit :

une représentation	$(-1)^s \times m \times 2^e$
des arrondis	au plus près (RN), vers 0 (RZ), $+\infty$ (RU), $-\infty$ (RD)
des formats	*binary32*, *binary64*, etc



Une représentation finie implique des pertes de précision

- Les erreurs d'arrondis mènent à d'autres phénomènes : élimination, absorption

par exemple (absorption) : $RN((a+b)-a) = 0 \qquad$ si $a \gg b$ et $b \neq 0$

Blocs de base : les transformations sans erreur (EFT)

Définition [MBdD10]

Soit $\circ \in \{+, -, \times\}$, si $x = RN(a \circ b)$, alors l'**erreur** de l'opération flottante $y = (a \circ b) - x$ **est un flottant**

Blocs de base : les transformations sans erreur (EFT)

Définition [MBdD10]

Soit $\circ \in \{+, -, \times\}$, si $x = RN(a \circ b)$, alors l'**erreur** de l'opération flottante $y = (a \circ b) - x$ **est un flottant**

Comment calculer y avec les opérations de l'arithmétique flottante ?

Blocs de base : les transformations sans erreur (EFT)

Définition [MBdD10]

Soit $\circ \in \{+, -, \times\}$, si $x = RN(a \circ b)$, alors l'**erreur** de l'opération flottante $y = (a \circ b) - x$ **est un flottant**

Comment calculer y avec les opérations de l'arithmétique flottante : les EFT

Blocs de base : les transformations sans erreur (EFT)

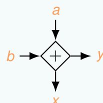
Définition [MBdD10]

Soit $\circ \in \{+,-,\times\}$, si $x = RN(a \circ b)$, alors l'**erreur** de l'opération flottante $y = (a \circ b) - x$ **est un flottant**

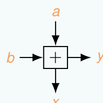
Comment calculer y avec les opérations de l'arithmétique flottante : les EFT

La somme

```
function FASTTWOSUM(a, b)            ▷ [DEKKER, 71]
    x ← RN(a + b)                    ▷ |a| ≥ |b|
    y ← RN((a − x) + b)
    return (x, y)
end function
```




```
function TWOSUM(a, b)                ▷ [KNUTH, 69]
    x ← RN(a + b)
    z ← RN(x − a)
    y ← RN((a − (x − z)) + (b − z))
    return (x, y)
end function
```



	flop	Latence
FASTTWOSUM	3	3
TWOSUM	6	5

Blocs de base : les transformations sans erreur (EFT)

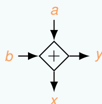
Définition [MBdD10]

Soit $\circ \in \{+,-,\times\}$, si $x = RN(a \circ b)$, alors l'**erreur** de l'opération flottante $y = (a \circ b) - x$ **est un flottant**

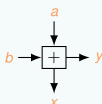
Comment calculer y avec les opérations de l'arithmétique flottante : les EFT

La somme

```
function FastTwoSum(a, b)          ▷ [Dekker, 71]
    x ← RN(a + b)                  ▷ |a| ≥ |b|
    y ← RN((a − x) + b)
    return (x, y)
end function
```




```
function TwoSum(a, b)              ▷ [Knuth, 69]
    x ← RN(a + b)
    z ← RN(x − a)
    y ← RN((a − (x − z)) + (b − z))
    return (x, y)
end function
```

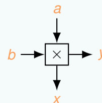


	flop	Latence
FastTwoSum	3	3
TwoSum	6	5

Le produit

```
function TwoProduct(a, b)          ▷ [Dekker, 71]
    x ← RN(a × b)
    [a_H, a_L] = Split(a)
    [b_H, b_L] = Split(b)
    y ← RN(a_L × b_L − (((x − a_H × b_H) − a_L × b_H) − a_H × b_L))
    return (x, y)
end function
```

```
function Split(a)                  ▷ [Veltkamp, 68]
    c ← RN(f × a)                  ▷ f = 2^⌈p/2⌉ + 1
    a_H ← RN(c − (c − a))
    a_L ← RN(a − a_H)
    return (a_H, a_L)
end function
```



	flop	Latence
TwoProduct	17	8
TwoProductFma	2	2

Arithmétique *double-double* et algorithmes compensés 1/2

Deux méthodes à base d'EFT pour améliorer la précision

- arithmétique *double-double* : DEKKER en 1971 avec la somme et le produit [Dek71]
- algorithmes compensés : années 2000 par RUMP, LOUVET, etc [ROO05, Lou07]

Arithmétique *double-double* et algorithmes compensés 1/2

Deux méthodes à base d'EFT pour améliorer la précision

- arithmétique *double-double* : DEKKER en 1971 avec la somme et le produit [Dek71]
- algorithmes compensés : années 2000 par RUMP, LOUVET, etc [ROO05, Lou07]

double-double (DD)

- - DEKKER, 1971
  - BAILEY et al. (QD Lib), 2000
  - SAITO, 2010 (Scilab Toolbox : QuPAT)
- Générique, algorithmes à appliquer sur chaque opération élémentaire
- Application presque automatique : surcharge

Arithmétique *double-double* et algorithmes compensés 1/2

Deux méthodes à base d'EFT pour améliorer la précision

- arithmétique *double-double* : DEKKER en 1971 avec la somme et le produit [Dek71]
- algorithmes compensés : années 2000 par RUMP, LOUVET, etc [ROO05, Lou07]

double-double (DD)

- - DEKKER, 1971
  - BAILEY et al. (QD Lib), 2000
  - SAITO, 2010 (Scilab Toolbox : QuPAT)
- Générique, algorithmes à appliquer sur chaque opération élémentaire
- Application presque automatique : surcharge

algorithmes compensés

- - RUMP et al. (SUM2, DOT2), 2005
  - LOUVET (COMPHORNER), 2007
  - GRAILLAT et al. (COMPHORNERDER), 2013
- Spécifique, travail d'expert : une thèse ou article par algorithme
- Aujourd'hui : somme, produit scalaire, évaluations polynômiales

Arithmétique *double-double* et algorithmes compensés 1/2

Deux méthodes à base d'EFT pour améliorer la précision

- arithmétique *double-double* : DEKKER en 1971 avec la somme et le produit [Dek71]
- algorithmes compensés : années 2000 par RUMP, LOUVET, etc [ROO05, Lou07]

double-double (DD)

- - DEKKER, 1971
  - BAILEY et al. (QD Lib), 2000
  - SAITO, 2010 (Scilab Toolbox : QuPAT)
- Générique, algorithmes à appliquer sur chaque opération élémentaire
- Application presque automatique : surcharge

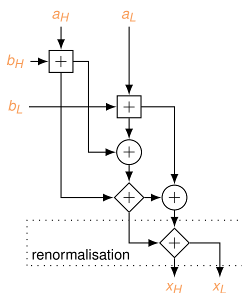
algorithmes compensés

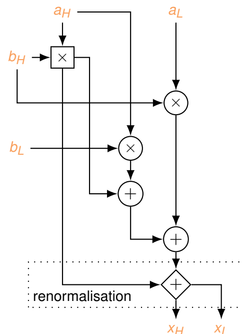
- - RUMP et al. (SUM2, DOT2), 2005
  - LOUVET (COMPHORNER), 2007
  - GRAILLAT et al. (COMPHORNERDER), 2013
- Spécifique, travail d'expert : une thèse ou article par algorithme
- Aujourd'hui : somme, produit scalaire, évaluations polynômiales

double-double générique **mais** très fort impact sur le temps de calcul

algo. compensés permet de meilleurs performances : un parallélisme d'instruction (ILP) exploité facilement par le processeur [LL07] **mais** très spécifique

Des opérateurs *double-double* : QDLib [HBL01]





function QD_TwoSum(a_H, a_L, b_H, b_L)
  $[r_H, r_L] = \text{TwoSum}(a_H, b_H)$
  $[s_H, s_L] = \text{TwoSum}(a_L, b_L)$
  $c \leftarrow RN(r_L + s_H)$
  $[u_H, u_L] = \text{FastTwoSum}(r_H, c)$
  $w \leftarrow RN(s_L + u_L)$
  return $[x_H, x_L] = \text{FastTwoSum}(u_H, w)$
end function

function QD_TwoProduct(a_H, a_L, b_H, b_L)
  $[r_H, r_L] = \text{TwoProduct}(a_H, b_H)$
  $r_L \leftarrow RN(r_L + (a_H \times b_L))$
  $r_L \leftarrow RN(r_L + (a_L \times b_H))$
  return $[x_H, x_L] = \text{FastTwoSum}(r_H, r_L)$
end function

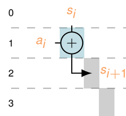
	flop	Latence
QD_TwoSum	20	13
QD_TwoProduct	24	13

Arithmétique *double-double* et algorithmes compensés 2/2

function SUM($a_1, a_2, \dots, a_n$)
  for $i = 2 : n$ **do**
    $s \leftarrow RN(s + a_i)$
  end for
  return s
end function

function SUMDD($a_1, a_2, \dots, a_n$)
  $s_H \leftarrow a_1$
  $s_L \leftarrow 0$
  for $i = 2 : n$ **do**
    $[s_H, s_L] = \text{QD_TWOSUM}(s_H, s_L, a_i, \varnothing)$
  end for
  return s_H
end function

function SUM2($a_1, a_2, \dots, a_n$)
  $s \leftarrow a_1$
  $e \leftarrow 0$
  for $i = 2 : n$ **do**
    $[s, \varepsilon] = \text{TWOSUM}(s, a_i)$
    $e \leftarrow RN(e + \varepsilon)$
  end for
  return $RN(s + e)$
end function

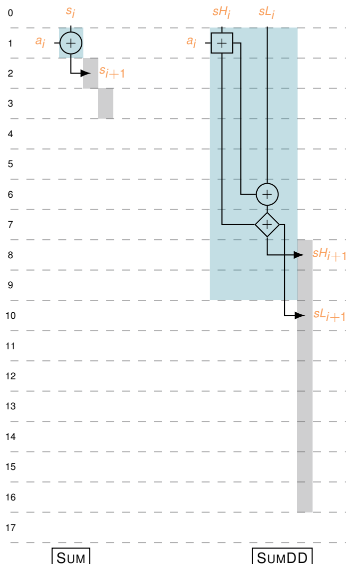


Arithmétique *double-double* et algorithmes compensés 2/2

```
function SUM(a_1, a_2, ..., a_n)
    for i = 2 : n do
        s ← RN(s + a_i)
    end for
    return s
end function
```

```
function SUMDD(a_1, a_2, ..., a_n)
    s_H ← a_1
    s_L ← 0
    for i = 2 : n do
        [s_H, s_L] = QD_TWOSUM(s_H, s_L, a_i, ∅)
    end for
    return s_H
end function
```

```
function SUM2(a_1, a_2, ..., a_n)
    s ← a_1
    e ← 0
    for i = 2 : n do
        [s, ε] = TWOSUM(s, a_i)
        e ← RN(e + ε)
    end for
    return RN(s + e)
end function
```

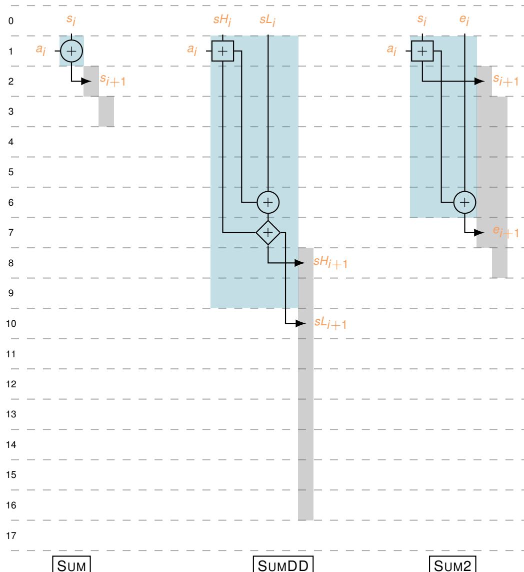


Arithmétique *double-double* et algorithmes compensés 2/2

```
function Sum(a_1, a_2, ..., a_n)
    for i = 2 : n do
        s ← RN(s + a_i)
    end for
    return s
end function
```

```
function SumDD(a_1, a_2, ..., a_n)
    s_H ← a_1
    s_L ← 0
    for i = 2 : n do
        [s_H, s_L] = QD_TwoSum(s_H, s_L, a_i, ∅)
    end for
    return s_H
end function
```

```
function Sum2(a_1, a_2, ..., a_n)
    s ← a_1
    e ← 0
    for i = 2 : n do
        [s, ε] = TwoSum(s, a_i)
        e ← RN(e + ε)
    end for
    return RN(s + e)
end function
```



Plan de l'exposé

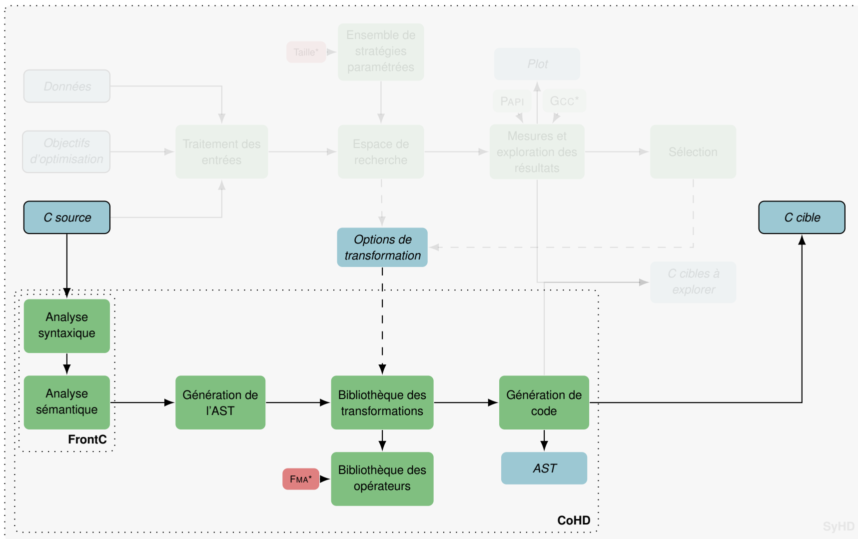
1. Notions d'arithmétique flottante et d'amélioration de la précision
- Arithmétique flottante IEEE 754
- Transformations sans erreur
- Arithmétique double-double et algorithmes compensés

2. Transformation automatique de programme
- Mise en place : l'outil CoHD
- Méthodologie : correction de la précision
- Les transformations de la somme et du produit
- Résultats expérimentaux avec comparaisons à l'existant

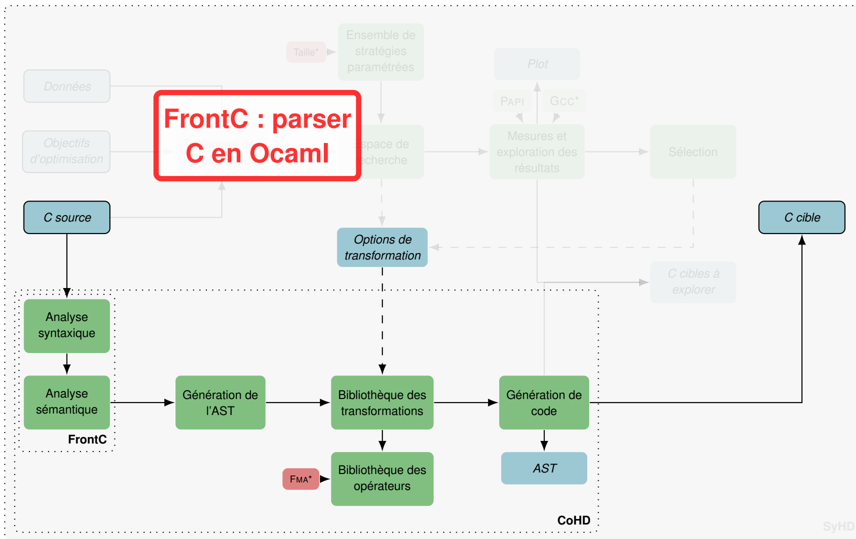
3. Synthèse de code pour des compromis entre précision et temps d'exécution
- Mise en place : l'outil SyHD
- Des stratégies de transformation de programme avec compromis
- Ensemble de programme et critères de sélection
- Résultats expérimentaux

4. Conclusions et perspectives

Un outil pour corriger la précision : **CoHD**

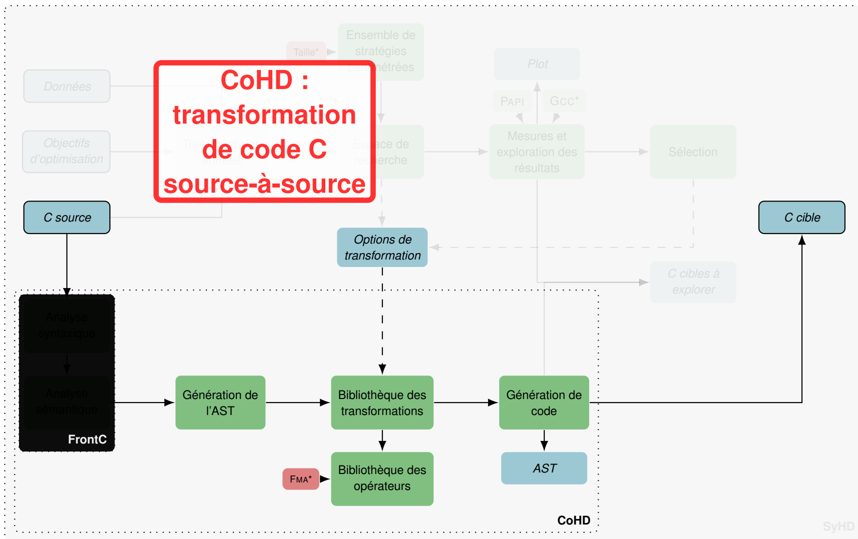


Un outil pour corriger la précision : **CoHD**

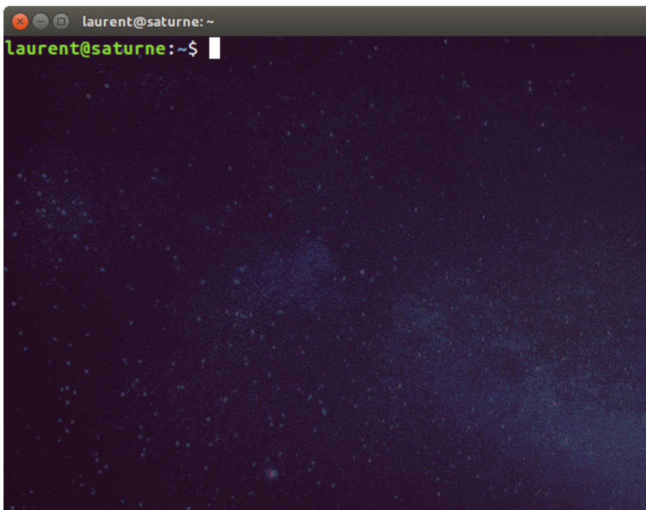


Modules | Outils externes | Options (*) | Entrées/sorties

Un outil pour corriger la précision : **CoHD**



Démonstration : **transformation automatique** source-à-source



Méthodologie de la transformation : correction de la précision

Trois étapes pour la transformation

1. Détecter les **séquences** d'opérations flottantes

Méthodologie de la transformation : correction de la précision

Trois étapes pour la transformation

1. Détecter les **séquences** d'opérations flottantes
2. Pour chaque séquence : **calculer les erreurs** puis les **accumuler** tout au long de la suite d'opérations

Méthodologie de la transformation : correction de la précision

Trois étapes pour la transformation

1. Détecter les **séquences** d'opérations flottantes
2. Pour chaque séquence : **calculer les erreurs** puis les **accumuler** tout au long de la suite d'opérations
3. **Fermer** les séquences

Méthodologie de la transformation : correction de la précision

Trois étapes pour la transformation

1. Détecter les **séquences** d'opérations flottantes
   - Une séquence est l'ensemble $\mathscr{E}$ de toutes les opérations flottantes dépendantes requises à l'obtention d'un ou plusieurs résultats : *calcul des dépendances à partir de l'AST*
2. Pour chaque séquence : **calculer les erreurs** puis les **accumuler** tout au long de la suite d'opérations
3. **Fermer** les séquences

Méthodologie de la transformation : correction de la précision

Trois étapes pour la transformation

1. Détecter les **séquences** d'opérations flottantes
   - Une séquence est l'ensemble $\mathscr{E}$ de toutes les opérations flottantes dépendantes requises à l'obtention d'un ou plusieurs résultats : *calcul des dépendances à partir de l'AST*

2. Pour chaque séquence : **calculer les erreurs** puis les **accumuler** tout au long de la suite d'opérations
   - remplacer les opérations *élémentaires* par des EFT
   - accumuler les termes d'erreurs
     un nombre flottant x correspond alors à un nombre « automatiquement compensé » (auto-comp) : la paire (x, δ_x) *où* $\delta_x \in \mathbb{F}$ *est l'approximation des erreurs accumulées résultant du calcul de x*

3. **Fermer** les séquences

Méthodologie de la transformation : correction de la précision

Trois étapes pour la transformation

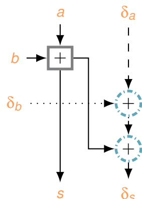
1. Détecter les **séquences** d'opérations flottantes
   - Une séquence est l'ensemble $\mathscr{E}$ de toutes les opérations flottantes dépendantes requises à l'obtention d'un ou plusieurs résultats : *calcul des dépendances à partir de l'AST*
2. Pour chaque séquence : **calculer les erreurs** puis les **accumuler** tout au long de la suite d'opérations
   - remplacer les opérations *élémentaires* par des EFT
   - accumuler les termes d'erreurs
     un nombre flottant x correspond alors à un nombre « automatiquement compensé » (auto-comp) : la paire (x, δ_x) où $\delta_x \in \mathbb{F}$ est l'approximation des erreurs accumulées résultant du calcul de x
3. **Fermer** les séquences
   - c'est l'étape de correction proprement dite : pour fermer une séquence, sommer x et δ_x pour chacun des résultats de la séquence

$$\mathit{Fermeture}(\mathscr{E}) \Leftrightarrow \forall x \text{ un résultat de la séquence } : x \leftarrow \mathit{RN}(x + \delta_x)$$

Les transformations de la somme et du produit

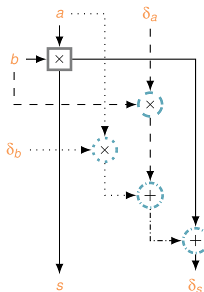
function AUTOCOMP_TWOSUM($(a,\delta_a),(b,\delta_b)$)
  $[s,\ \delta_+] = \text{TwoSum}(a,\ b)$
  $\delta_s \leftarrow RN((\delta_a+\delta_b)+\delta_+)$
  return (s,δ_s)
end function

function AUTOCOMP_TWOPRODUCT($(a,\delta_a),(b,\delta_b)$)
  $[s,\ \delta_\times] = \text{TwoProduct}(a,\ b)$
  $\delta_s \leftarrow RN(((a\times\delta_b)+(b\times\delta_a))+\delta_\times)$
  return (s,δ_s)
end function



AUTOCOMP_TWOSUM

	flop	Latence
AC_TWOSUM	8	6



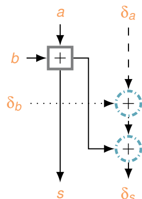
AUTOCOMP_TWOPRODUCT

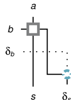
	flop	Latence
AC_TWOPRODUCT	21	9

Les transformations de la somme et du produit

function AUTOCOMP_TWOSUM($(a,\delta_a),(b,\delta_b)$)
  $[s,\ \delta_+] = \text{TWOSUM}(a,\ b)$
  $\delta_s \leftarrow RN(\delta_a + \delta_+)$
  return (s, δ_s)
end function

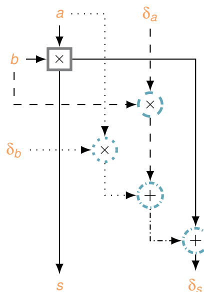
function AUTOCOMP_TWOPRODUCT($(a,\delta_a),(b,\delta_b)$)
  $[s,\ \delta_\times] = \text{TWOPRODUCT}(a,\ b)$
  $\delta_s \leftarrow RN((a \times \delta_b) + \delta_\times)$
  return (s, δ_s)
end function

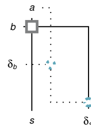




AUTOCOMP_TWOSUM

	flop	Latence
AC_TWOSUM	7	6





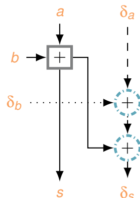
AUTOCOMP_TWOPRODUCT

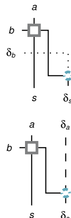
	flop	Latence
AC_TWOPRODUCT	19	9

Les transformations de la somme et du produit

function AUTOCOMP_TWOSUM($(a,\delta_a),(b,\delta_b)$)
  $[s,\ \delta_+] = \text{TWOSUM}(a,\ b)$
  $\delta_s \leftarrow RN(\delta_b + \delta_+)$
  return (s,δ_s)
end function

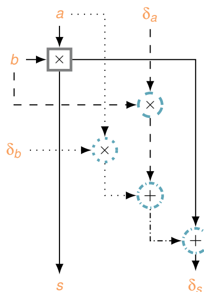
function AUTOCOMP_TWOPRODUCT($(a,\delta_a),(b,\delta_b)$)
  $[s,\ \delta_\times] = \text{TWOPRODUCT}(a,\ b)$
  $\delta_s \leftarrow RN((b\times\delta_a) + \delta_\times)$
  return (s,δ_s)
end function

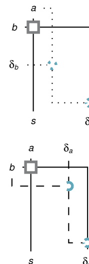




AUTOCOMP_TWOSUM

	flop	Latence
AC_TWOSUM	7	6





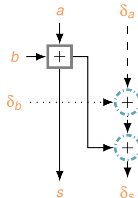
AUTOCOMP_TWOPRODUCT

	flop	Latence
AC_TWOPRODUCT	19	9

Les transformations de la somme et du produit

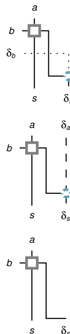
function AUTOCOMP_TWOSUM($(a,\delta_a),(b,\delta_b)$)
  $[s,\ \delta_+] = \text{TwoSum}(a,\ b)$
  $\delta_s \leftarrow \delta_+$
  return (s, δ_s)
end function

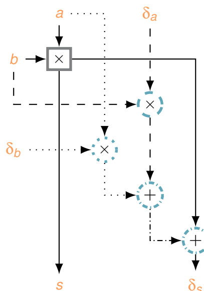
function AUTOCOMP_TWOPRODUCT($(a,\delta_a),(b,\delta_b)$)
  $[s,\ \delta_\times] = \text{TwoProduct}(a,\ b)$
  $\delta_s \leftarrow \delta_\times$
  return (s, δ_s)
end function



AUTOCOMP_TWOSUM

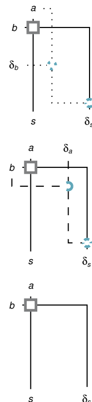
	flop	Latence
AC_TWOSUM	6	5





AUTOCOMP_TWOPRODUCT

	flop	Latence
AC_TWOPRODUCT	17	8



Résultats expérimentaux avec comparaisons à l'existant

Ces transformations permettent-elles de retrouver les résultats de la littérature ?

Algorithmes DD et compensés :

- Sommation : SUM2 [ROO05]
- Évaluation polynomiale

monômes	HORNER [GLL09] HORNERDER [JGH13]
Bernstein	DECASTELJAU, DECASTELJAUDER [JLCS10]
Tchebychev	CLENSHAWI, CLENSHAWII [JBL11]

Illustration du processus de transformation avec l'algorithme HORNER

Analyse à l'image des études similaires faites dans la littérature [Lou07]

```
function HORNER(p, x)
    r_n ← a_n
    for i = n−1 : −1 : 0 do
        r_i ← RN(r_{i+1} × x + a_i)
    end for
    return r_0
end function
```

polynôme	$p_H(x) = (x - 0,75)^5 (x - 1,0)^{11}$
données	512 points $x \in \{0,68 : 1,15\}$
mesures	bits corrects, cycles, instructions, IPC

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r ;
  int i ;

  r = P[n] ;
  for(i = n-1; i >= 0; i--) {
    r = r * x + P[i] ;
  }

  return r ;
}
```

Première transformation

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r, tmp ;
  int i ;

  r = P[n] ;
  for(i = n-1; i >= 0; i--) {
    tmp = r * x ;
    r = tmp + P[i] ;
  }

  return r ;
}
```

Première transformation → passage au format trois adresses

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r, tmp ;
  int i ;

  r = P[n] ;
  for(i = n-1; i >= 0; i--) {
    tmp = r * x ;
    r = tmp + P[i] ;
  }

  return r ;
}
```

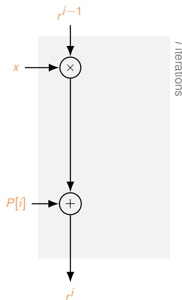
Détection des séquences d'opérations

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r, tmp ;
  int i ;

  r = P[n] ;
  for(i = n-1; i >= 0; i--) {
    tmp = r * x ;
    r = tmp + P[i] ;
  }

  return r ;
}
```



Détection des séquences d'opérations → 1 séquence détectée

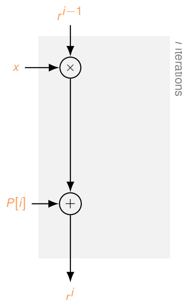
Compteurs (d'opérations flottantes)		
nombre total	latence par itération	latence totale
$2n$	2	$2n$

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r, tmp ;
  int i ;

  r = P[n] ;
  for(i = n-1; i >= 0; i--) {
    tmp = r * x ;
    r = tmp + P[i] ;
  }

  return r ;
}
```



Appliquer les transformations pour chaque opération de la séquence

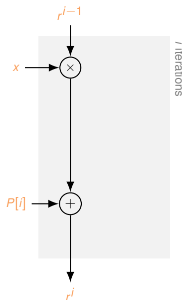
Compteurs (d'opérations flottantes)		
nombre total	latence par itération	latence totale
$2n$	2	$2n$

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r, tmp, d_tmp, d_r ;
  int i ;

  r = P[n] ;
  d_r = 0.0 ;
  for(i = n-1; i >= 0; i--) {
    tmp = r * x ;
    r = tmp + P[i] ;
  }

  return r ;
}
```



Appliquer les transformations pour chaque opération de la séquence → mais avant déterminer les données étendues (i.e. *auto-comp*)

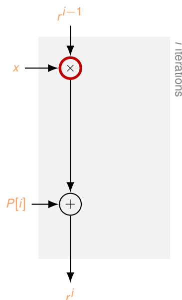
Compteurs (d'opérations flottantes)		
nombre total	latence par itération	latence totale
$2n$	2	$2n$

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r, tmp, d_tmp, d_r ;
  int i ;

  r = P[n] ;
  d_r = 0.0 ;
  for(i = n-1; i >= 0; i--) {
    tmp = r * x ;
    r = tmp + P[i] ;
  }

  return r ;
}
```



Transformation de la multiplication `tmp = r * x`

$$[tmp, d_{tmp}] = \text{AUTOCOMP_TWOPRODUCT}(r, d_r, x, \varnothing)$$

coût : 19 (latence de 9) opérations flottantes

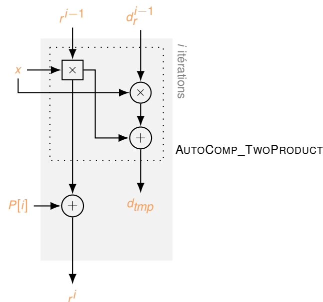
Compteurs (d'opérations flottantes)		
nombre total	latence par itération	latence totale
$2n$	2	$2n$

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r, tmp, d_tmp, d_r, c, rh, rl, xh, xl, d_2p ;
  int i ;

  r = P[n] ;
  d_r = 0.0 ;
  for(i = n-1; i >= 0; i--) {
    tmp = r * x ;
    c = r * 134217729 ;
    rh = c - (c - r) ;
    rl = r - rh ;
    c = x * 134217729 ;
    xh = c - (c - x) ;
    xl = x - xh ;
    d_2p = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl) ;
    d_tmp = d_2p + d_r * x ;
    r = tmp + P[i] ;
  }

  return r ;
}
```



Transformation de la multiplication `tmp = r * x`

$$[tmp, d_{tmp}] = \text{AUTOCOMP_TWOPRODUCT}(r, d_r, x, \varnothing)$$

coût : 19 (latence de 9) opérations flottantes

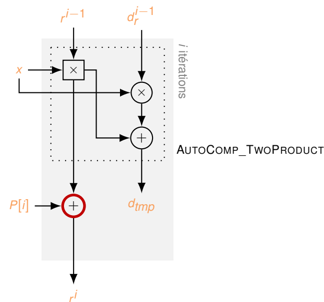
Compteurs (d'opérations flottantes)		
nombre total	latence par itération	latence totale
$(2+18)n$	9	?

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r, tmp, d_tmp, d_r, c, rh, rl, xh, xl, d_2p ;
  int i ;

  r = P[n] ;
  d_r = 0.0 ;
  for(i = n-1; i >= 0; i--) {
    tmp = r * x ;
    c = r * 134217729 ;
    rh = c - (c - r) ;
    rl = r - rh ;
    c = x * 134217729 ;
    xh = c - (c - x) ;
    xl = x - xh ;
    d_2p = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl) ;
    d_tmp = d_2p + d_r * x ;
    r = tmp + P[i] ;
  }

  return r ;
}
```



Transformation de la somme `r = tmp + P[i]`

$$[r, d_r] = \text{AUTOCOMP_TWOSUM}(tmp, d_{tmp}, P[i], \varnothing)$$

coût : 7 (latence de 6) opérations flottantes

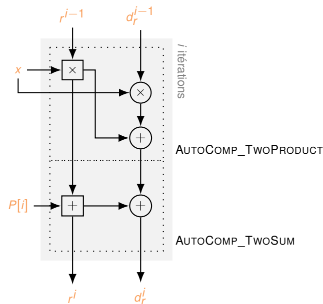
Compteurs (d'opérations flottantes)		
nombre total	latence par itération	latence totale
$20n$	9	?

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r, tmp, d_tmp, d_r, c, rh, rl, xh, xl, d_2p, u, d_2s ;
  int i ;

  r = P[n] ;
  d_r = 0.0 ;
  for(i = n-1; i >= 0; i--) {
    tmp = r * x ;
    c = r * 134217729 ;
    rh = c - (c - r) ;
    rl = r - rh ;
    c = x * 134217729 ;
    xh = c - (c - x) ;
    xl = x - xh ;
    d_2p = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl) ;
    d_tmp = d_2p + d_r * x ;
    r = tmp + P[i] ;
    u = r - tmp ;
    d_2s = (tmp - (r - u)) + (P[i] - u) ;
    d_r = d_2s + d_tmp ;
  }

  return r ;
}
```



Transformation de la somme `r = tmp + P[i]`

$$[r, d_r] = \text{AUTOCOMP_TWOSUM}(tmp, d_{tmp}, P[i], \varnothing)$$

coût : 7 (latence de 6) opérations flottantes

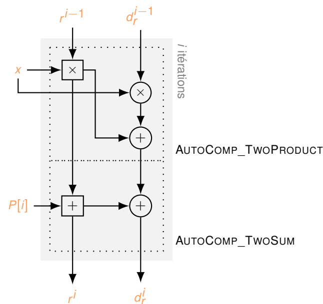
Compteurs (d'opérations flottantes)		
nombre total	latence par itération	latence totale
$(20+6)n$	10	?

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r, tmp, d_tmp, d_r, c, rh, rl, xh, xl, d_2p, u, d_2s ;
  int i ;

  r = P[n] ;
  d_r = 0.0 ;
  for(i = n-1; i >= 0; i--) {
    tmp = r * x ;
    c = r * 134217729 ;
    rh = c - (c - r) ;
    rl = r - rh ;
    c = x * 134217729 ;
    xh = c - (c - x) ;
    xl = x - xh ;
    d_2p = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl) ;
    d_tmp = d_2p + d_r * x ;
    r = tmp + P[i] ;
    u = r - tmp ;
    d_2s = (tmp - (r - u)) + (P[i] - u) ;
    d_r = d_2s + d_tmp ;
  }

  return r ;
}
```



Fermeture de la séquence d'opérations

Remarque : chevauchement des itérations sur 8 cycles d'une itération à la suivante, la latence pour les n itérations est donc de $2n+8$

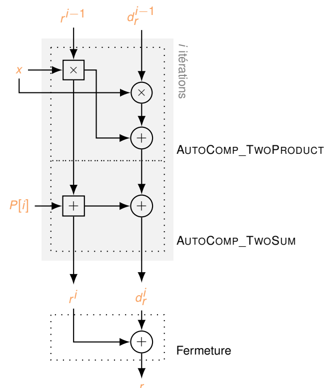
Compteurs (d'opérations flottantes)		
nombre total	latence par itération	latence totale
$26n$	10	$2n+8$

Transformation automatique du programme d'évaluation polynomiale HORNER

```
double
Horner(double *P, unsigned int n, double x) {
  double r, tmp, d_tmp, d_r, c, rh, rl, xh, xl, d_2p, u, d_2s ;
  int i ;

  r = P[n] ;
  d_r = 0.0 ;
  for(i = n-1; i >= 0; i--) {
    tmp = r * x ;
    c = r * 134217729 ;
    rh = c - (c - r) ;
    rl = r - rh ;
    c = x * 134217729 ;
    xh = c - (c - x) ;
    xl = x - xh ;
    d_2p = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl) ;
    d_tmp = d_2p + d_r * x ;
    r = tmp + P[i] ;
    u = r - tmp ;
    d_2s = (tmp - (r - u)) + (P[i] - u) ;
    d_r = d_2s + d_tmp ;
  }

  return r + d_r ;
}
```



Fermeture de la séquence d'opérations

$$r = r + d_r$$

coût : 1 opération flottante

Compteurs (d'opérations flottantes)		
nombre total	latence par itération	latence totale
$26n+1$	10	$2n+9$

auto-comp vs *double-double* et compensation 1/2

Comparaison des codes sources

```
double r, tmp, d_tmp, d_r, c, rh, rl, xh, xl, d_2p, u,
      d_2s ;
int i ;

r = P[n] ;
d_r = 0.0 ;

for(i = n-1; i >= 0; i--) {
  tmp = r * x ;
  c = r * 134217729 ;
  rh = c - (c - r) ;
  rl = r - rh ;
  c = x * 134217729 ;
  xh = c - (c - x) ;
  xl = x - xh ;
  d_2p = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl
      ) ;
  d_tmp = d_2p + d_r * x ;
  r = tmp + P[i] ;
  u = r - tmp ;
  d_2s = (tmp - (r - u)) + (P[i] - u) ;
  d_r = d_2s + d_tmp ;
 }

return r + d_r ;
```

ACHORNER

```
double r_h, r_l, t_h, t_l, x_hi, x_lo, hi, lo, t ;
int i ;

t = x * 134217729 ;
x_hi = t - (t - x) ;
x_lo = x - x_hi ;
r_h = P[n] ;
r_l = 0.0 ;

for(i = n-1; i >= 0; i--) {
  t = r_h * 134217729 ;
  hi = t - (t - r_h) ;
  lo = (r_h - hi) ;
  t_h = r_h * x ;
  t_l = (((hi * x_hi - t_h) + hi * x_lo) + lo * x_hi) +
         lo * x_lo ;
  t_l += r_l * x ;
  r_h = t_h + t_l ;
  r_l = (t_h - r_h) + t_l ;
  t_h = r_h + P[i] ;
  t = t_h - r_h ;
  t_l = ((r_h - (t_h - t)) + (P[i] - t)) ;
  t_l += r_l ;
  r_h = t_h + t_l ;
  r_l = (t_h - r_h) + t_l ;
 }

return r_h ;
```

DDHORNER

	ACHORNER	DDHORNER	COMPHORNER
opérations flottantes	$26n+1$	$28n+4$	
latence	$2n+9$	$17n+2$	

auto-comp vs *double-double* et compensation 1/2

Comparaison des codes sources

```
double r, tmp, d_tmp, d_r, c, rh, rl, xh, xl, d_2p, u,
       d_2s ;
int i ;

r = P[n] ;
d_r = 0.0 ;

for(i = n-1; i >= 0; i--) {
  tmp = r * x ;
  c = r * 134217729 ;
  rh = c - (c - r) ;
  rl = r - rh ;
  c = x * 134217729 ;
  xh = c - (c - x) ;
  xl = x - xh ;
  d_2p = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl
       ) ;
  d_tmp = d_2p + d_r * x ;
  r = tmp + P[i] ;
  u = r - tmp ;
  d_2s = (tmp - (r - u)) + (P[i] - u) ;
  d_r = d_2s + d_tmp ;
 }

return r + d_r ;
```

ACHORNER

```
double p, r, c, pi, sig, x_hi, x_lo, hi, lo, t ;
int i ;

t = x * 134217729 ;
x_hi = t - (t - x) ;
x_lo = x - x_hi ;
r = P[n] ;
c = 0.0 ;

for(i = n-1; i >= 0; i--) {
  p = r * x ;
  t = r * 134217729 ;
  hi = t - (t - r) ;
  lo = r - hi ;
  pi = (((hi * x_hi - p) + hi * x_lo) + lo * x_hi) + lo
       * x_lo ;
  r = p + P[i] ;
  t = r - p ;
  sig = (p - (r - t)) + (P[i] - t) ;
  c = c * x + (pi+sig) ;
 }

return r + c ;
```

COMPHORNER

	ACHORNER	DDHORNER	COMPHORNER
opérations flottantes	$26n+1$	$28n+4$	$22n+5$
latence	$2n+9$	$17n+2$	$2n+9$

auto-comp vs *double-double* et compensation 1/2

Comparaison des codes sources

```
double r, tmp, d_tmp, d_r, c, rh, rl, xh, xl, d_2p, u,
       d_2s ;
int i ;

r = P[n] ;
d_r = 0.0 ;

for(i = n-1; i >= 0; i--) {
  tmp = r * x ;
  c = r * 134217729 ;
  rh = c - (c - r) ;
  rl = r - rh ;
  c = x * 134217729 ;
  xh = c - (c - x) ;
  xl = x - xh ;
  d_2p = rl * xl - (((x - rh * xh) - rl * xh) - rh * xl
        ) ;
  d_tmp = d_2p + d_r * x ;
  r = tmp + P[i] ;
  u = r - tmp ;
  d_2s = (tmp - (r - u)) + (P[i] - u) ;
  d_r = d_2s + d_tmp ;
 }

return r + d_r ;
```

ACHORNER

```
double p, r, c, pi, sig, x_hi, x_lo, hi, lo, t ;
int i ;

t = x * 134217729 ;
x_hi = t - (t - x) ;
x_lo = x - x_hi ;
r = P[n] ;
c = 0.0 ;

for(i = n-1; i >= 0; i--) {
  p = r * x ;
  t = r * 134217729 ;
  hi = t - (t - r) ;
  lo = r - hi ;
  pi = (((hi * x_hi - p) + hi * x_lo) + lo * x_hi) + lo
       * x_lo ;
  r = p + P[i] ;
  t = r - p ;
  sig = (p - (r - t)) + (P[i] - t) ;
  c = c * x + (pi+sig) ;
 }

return r + c ;
```

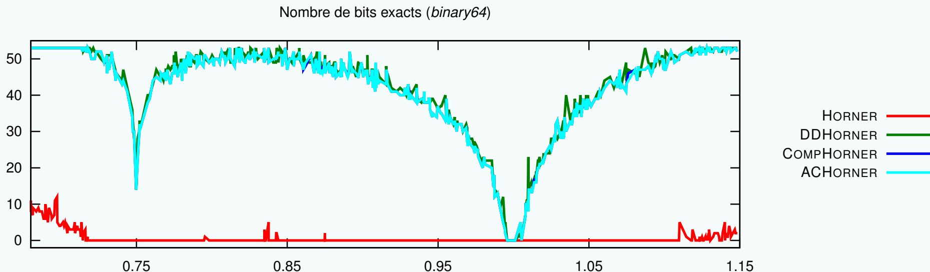
COMPHORNER

	ACHORNER	DDHORNER	COMPHORNER
opérations flottantes	$\mathbf{22n+5}$	$28n+4$	$22n+5$
latence	$2n+9$	$17n+2$	$2n+9$

auto-comp vs *double-double* et compensation 2/2

Comparaison avec les résultats de la littérature

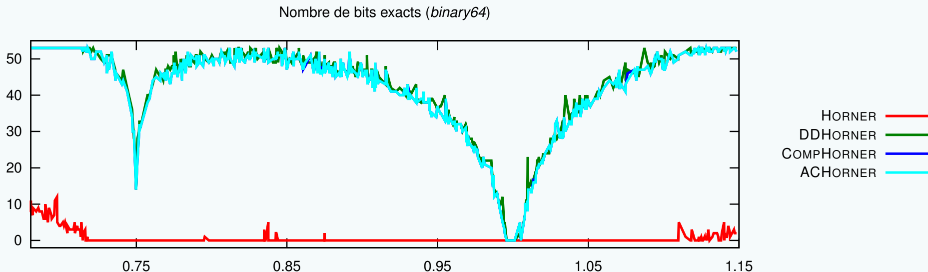
Mesures de précision



auto-comp vs *double-double* et compensation 2/2

Comparaison avec les résultats de la littérature

Mesures de précision

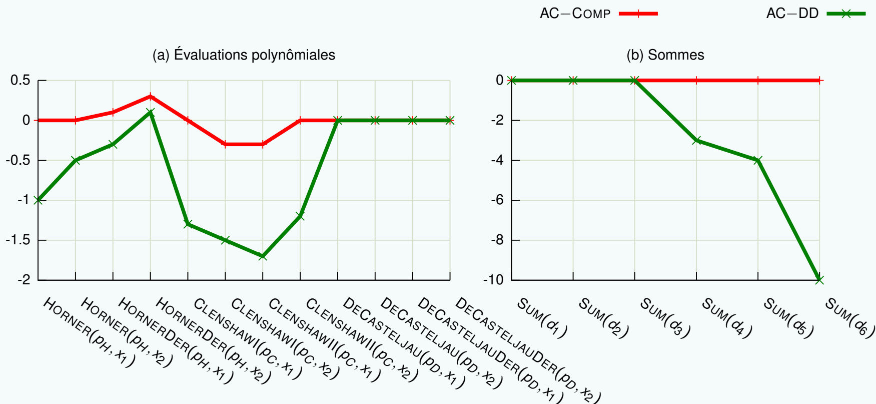


Mesures de performances (PAPI : moyenne sur 10^6 mesures, Ubuntu 12.04, Core i5 2.53GHz, Gcc -O2)

	PAPI : **Instructions**	**Cycles**	**IPC**	PERPI : **Instructions**	**Cycles**	**IPC**
HORNER	42	57	0,73	76	37	2,05
COMPHORNER	532	277	1,99	566	62	9,12
DDHORNER	658	920	0,72	676	325	2,08
ACHORNER	553	303	1,82	581	77	7,54
AC/COMP	1,04	1,09	0,95	1,02	1,24	0,82
AC/DD	0,84	0,33	2,55	0,85	0,23	3,62

Résultats expérimentaux : la précision

Différence du nombre de bits corrects de AC vs. COMP et AC vs. DD



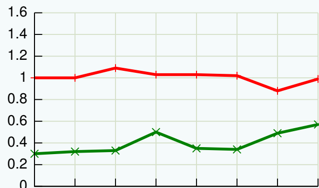
p_H	polynôme $p_H(x) = (x - 0,75)^5 (x - 1,0)^{11}$
p_C	polynôme $p_C(x) = (x - 0,75)^7 (x - 1,0)^{10}$
p_D	polynôme $p_D(x) = (x - 0,75)^7 (x - 1,0)$
x_1	256 valeurs tirées uniformément dans $\{0,85 : 0,95\}$
x_2	256 valeurs tirées uniformément dans $\{1,05 : 1,15\}$

d_1	32 jeux de 10^4 valeurs de conditionnement 10^8
d_2	32 jeux de 10^5 valeurs de conditionnement 10^8
d_3	32 jeux de 10^6 valeurs de conditionnement 10^8
d_4	32 jeux de 10^4 valeurs de conditionnement 10^{16}
d_5	32 jeux de 10^5 valeurs de conditionnement 10^{16}
d_6	32 jeux de 10^6 valeurs de conditionnement 10^{16}

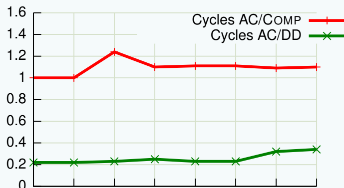
Résultats expérimentaux : les performances

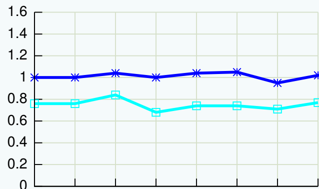
Ratio (cycles et instructions) entre les algorithmes AC/Comp et AC/DD

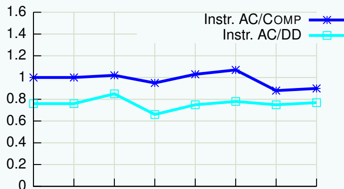
(a) PAPI



(b) PERPI







Plan de l'exposé

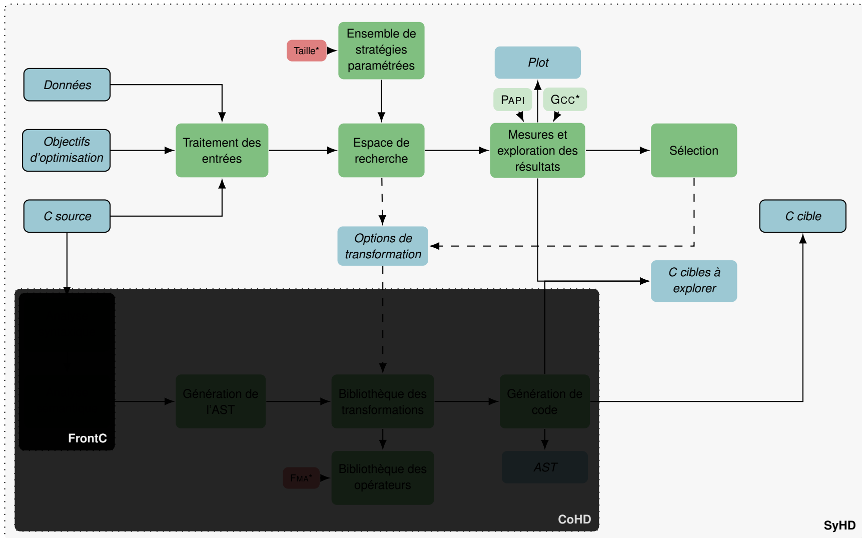
1. Notions d'arithmétique flottante et d'amélioration de la précision
   - Arithmétique flottante IEEE 754
   - Transformations sans erreur
   - Arithmétique double-double et algorithmes compensés

2. Transformation automatique de programme
   - Mise en place : l'outil CoHD
   - Méthodologie : correction de la précision
   - Les transformations de la somme et du produit
   - Résultats expérimentaux avec comparaisons à l'existant

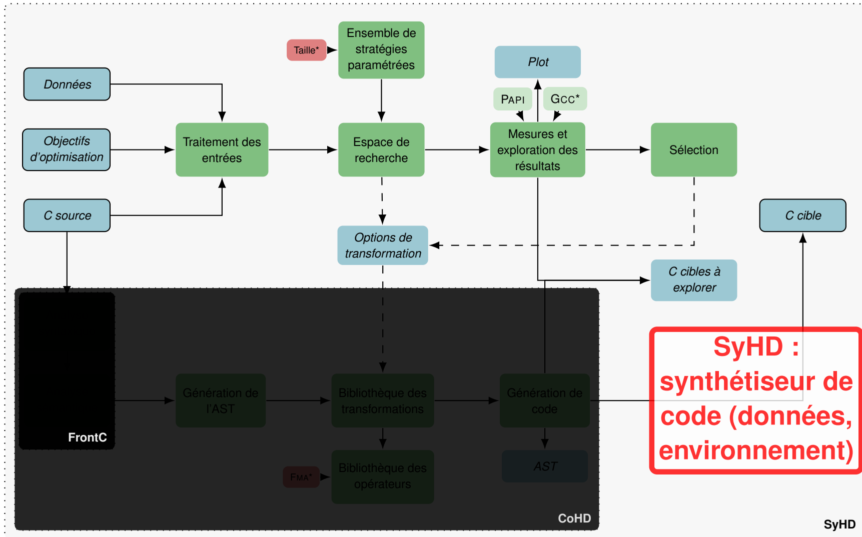
3. Synthèse de code pour des compromis entre précision et temps d'exécution
   - Mise en place : l'outil SyHD
   - Des stratégies de transformation de programme avec compromis
   - Ensemble de programme et critères de sélection
   - Résultats expérimentaux

4. Conclusions et perspectives

Un outil pour assurer la synthèse : **SyHD**

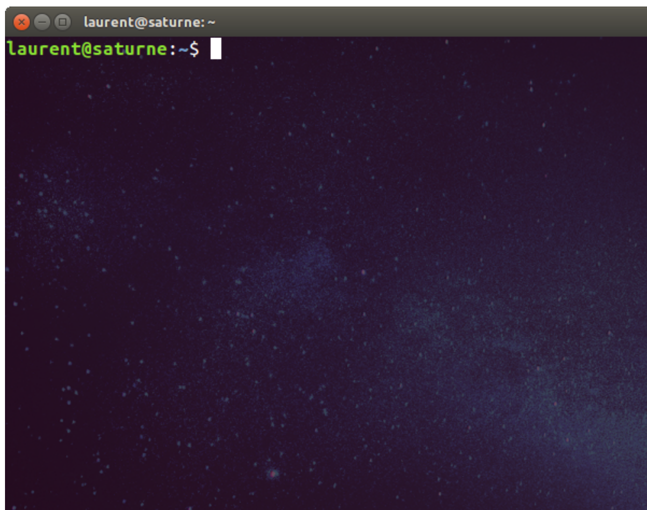


Un outil pour assurer la synthèse : **SyHD**



Modules | Outils externes | Options (*) | Entrées/sorties

Démonstration de la **synthèse de code**



Des stratégies de transformation de programme

Représentation d'une trace : exécution séquentielle

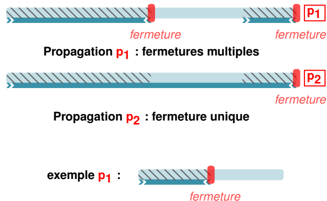
Des stratégies de transformation de programme

Représentation d'une trace : exécution séquentielle

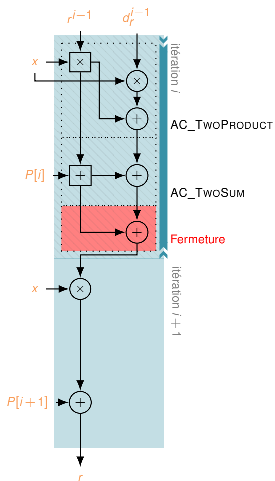
Propagation p_1 : fermetures multiples

Propagation p_2 : fermeture unique

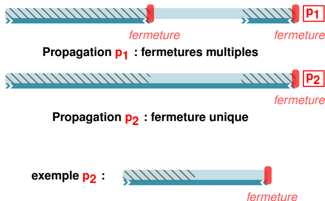
Des stratégies de transformation de programme



Propagation $\mathbf{p_1}$

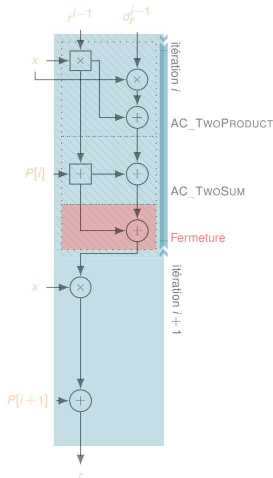


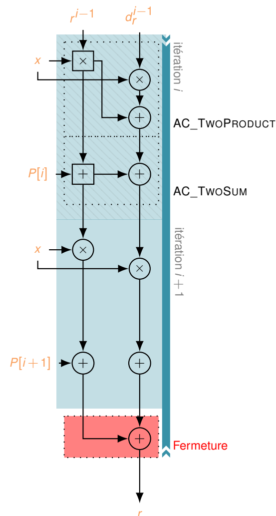
Des stratégies de transformation de programme



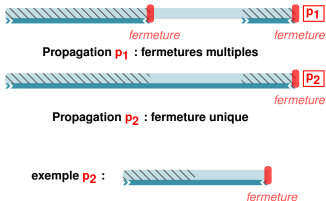
function AUTOPROP_TWOSUM($(a,\delta_a),(b,\delta_b)$)
  $s \leftarrow RN(a+b)$
  $\delta_s \leftarrow RN(\delta_a+\delta_b)$
  return (s,δ_s)
end function

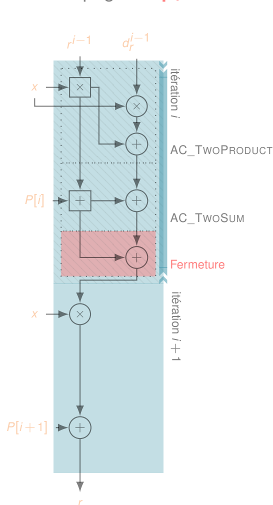
function AUTOPROP_TWOPRODUCT($(a,\delta_a),(b,\delta_b)$)
  $s \leftarrow RN(a\times b)$
  $\delta_s \leftarrow RN(a\times\delta_b + b\times\delta_a)$
  return (s,δ_s)
end function

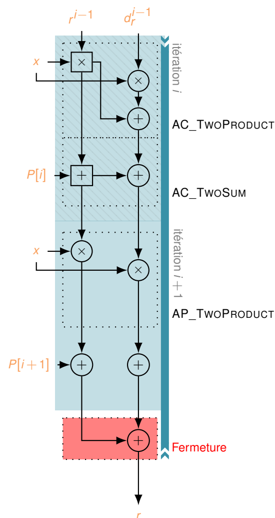




Des stratégies de transformation de programme



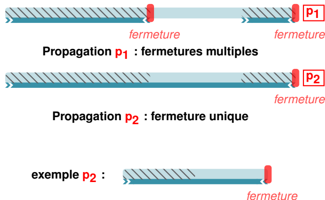


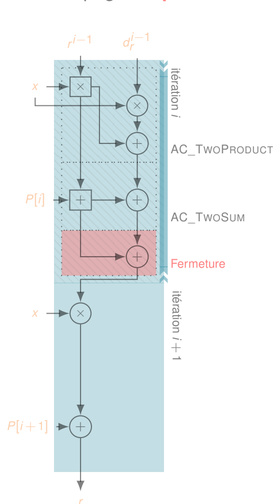


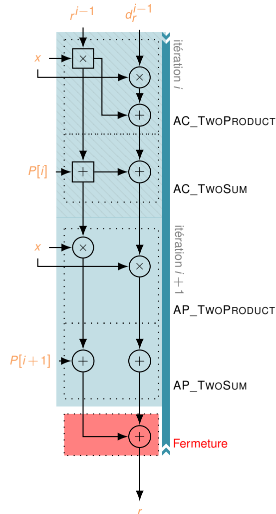
function AutoProp_TwoSum($(a,\delta_a),(b,\delta_b)$)
  $s \leftarrow RN(a+b)$
  $\delta_s \leftarrow RN(\delta_a+\delta_b)$
  return (s,δ_s)
end function

function AutoProp_TwoProduct($(a,\delta_a),(b,\delta_b)$)
  $s \leftarrow RN(a\times b)$
  $\delta_s \leftarrow RN(a\times\delta_b + b\times\delta_a)$
  return (s,δ_s)
end function

Des stratégies de transformation de programme





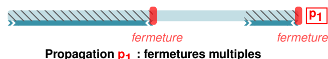


function AUTOPROP_TWOSUM$((a,\delta_a),(b,\delta_b))$
  $s \leftarrow RN(a+b)$
  $\delta_s \leftarrow RN(\delta_a+\delta_b)$
  return (s,δ_s)
end function

function AUTOPROP_TWOPRODUCT$((a,\delta_a),(b,\delta_b))$
  $s \leftarrow RN(a\times b)$
  $\delta_s \leftarrow RN(a\times\delta_b+b\times\delta_a)$
  return (s,δ_s)
end function

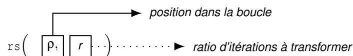
Des stratégies de transformation de programme

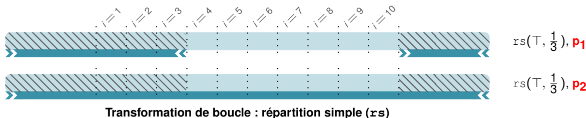
Représentation graphique d'une trace



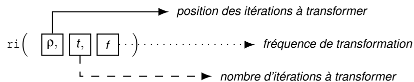
Propagation p_1 : fermetures multiples

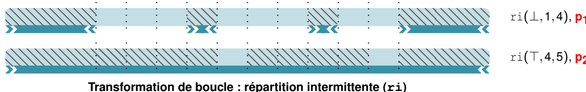
Propagation p_2 : fermeture unique





Transformation de boucle : répartition simple (**rs**)





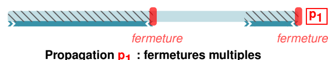
Transformation de boucle : répartition intermittente (**ri**)

Les stratégies

- $\mathtt{rs}(\rho, r)$
- $\mathtt{ri}(\rho, t, f)$

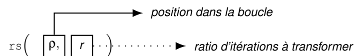
Des stratégies de transformation de programme

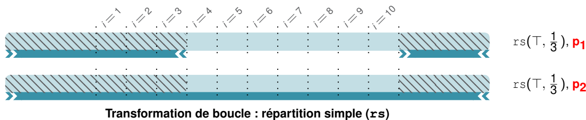
Représentation graphique d'une trace



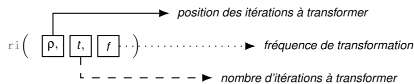
Propagation p_1 : fermetures multiples

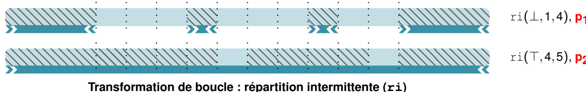
Propagation p_2 : fermeture unique





Transformation de boucle : répartition simple (rs)

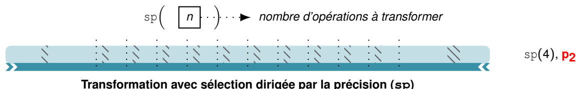




Transformation de boucle : répartition intermittente (ri)

Les stratégies

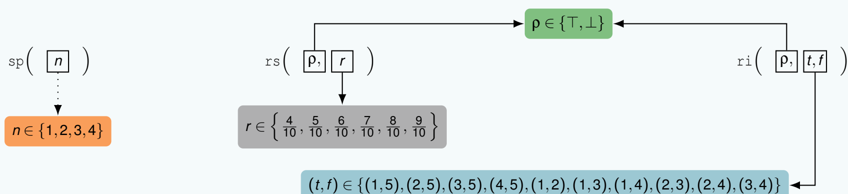
- $\mathtt{rs}(\rho, r)$
- $\mathtt{ri}(\rho, t, f)$
- $\mathtt{sp}(n)$



Transformation avec sélection dirigée par la précision (sp)

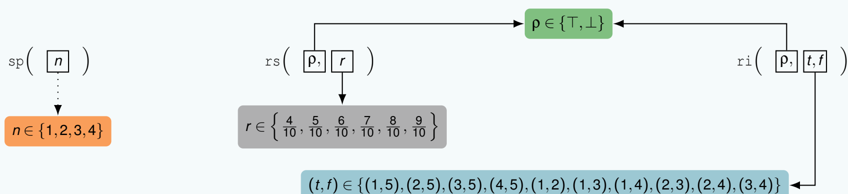
Un espace de recherche E

Pour un ensemble représentatif des transformations possibles



Un espace de recherche *E*

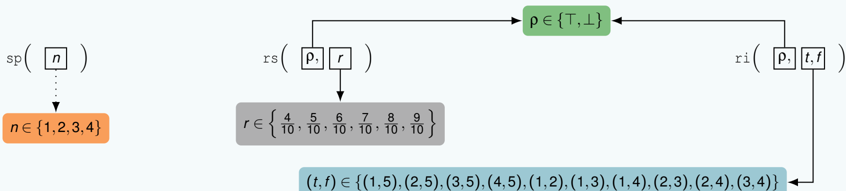
Pour un ensemble représentatif des transformations possibles



- L'utilisation supplémentaire des propagations p_1 et p_2 complète l'ensemble

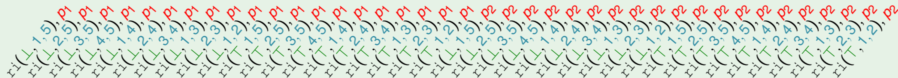
Un espace de recherche E

Pour un ensemble représentatif des transformations possibles



- L'utilisation supplémentaire des propagations p_1 et p_2 complète l'ensemble

Exemple : ensemble de stratégies générées pour la synthèse du programme HORNER selon les paramètres $\rho, r, t, f, n, p_1, p_2$



Des critères de sélection R^0, R^1, R^2

Objectifs

Un maximum de précision

$$R^0 = \min_{s \in \{\max_{s \in E} r_{bits\ exacts}(s)\}} r_{cycles}(s)$$

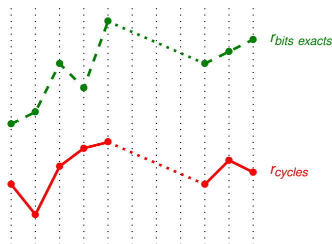
- r_{cycles} ratio de cycles entre AC et DD
- $r_{bits\ exacts}$ ratio de bits exacts entre AC et DD

Un bon compromis temps-précision

$$R^1 = \min_{s \in E} \left(\alpha \cdot r_{bits\ faux}(s) + \beta \cdot r_{cycles}(s)\right)$$

- où $\alpha + \beta = 1$ et $r_{bits\ faux} = 1 - r_{bits\ exacts}$

$$R^2 = \max_{s \in E} \left| r_{bits\ exacts}(s) - r_{cycles}(s) \right|$$



Des critères de sélection R^0, R^1, R^2

Objectifs

Un maximum de précision

$$R^0 = \min_{s \in \{\max_{s \in E} r_{bits\ exacts}(s)\}} r_{cycles}(s)$$

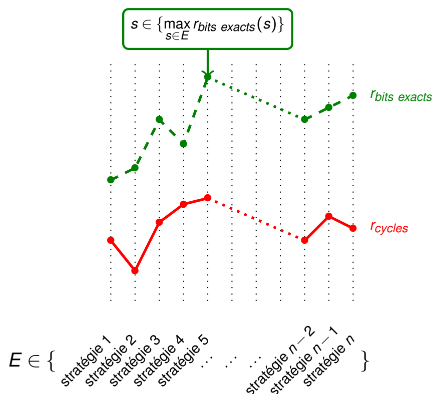
- r_{cycles} ratio de cycles entre AC et DD
- $r_{bits\ exacts}$ ratio de bits exacts entre AC et DD

Un bon compromis temps-précision

$$R^1 = \min_{s \in E} \left(\alpha \cdot r_{bits\ faux}(s) + \beta \cdot r_{cycles}(s)\right)$$

- où $\alpha + \beta = 1$ et $r_{bits\ faux} = 1 - r_{bits\ exacts}$

$$R^2 = \max_{s \in E} \left| r_{bits\ exacts}(s) - r_{cycles}(s) \right|$$



Des critères de sélection R^0, R^1, R^2

Objectifs

Un maximum de précision

$$R^0 = \min_{s \in \{\max_{s \in E} r_{bits\ exacts}(s)\}} r_{cycles}(s)$$

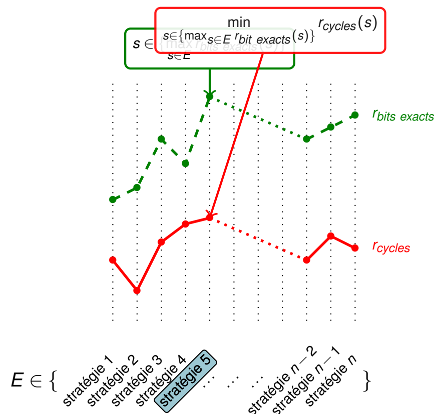
- r_{cycles} ratio de cycles entre AC et DD
- $r_{bits\ exacts}$ ratio de bits exacts entre AC et DD

Un bon compromis temps-précision

$$R^1 = \min_{s \in E} \left(\alpha \cdot r_{bits\ faux}(s) + \beta \cdot r_{cycles}(s)\right)$$

- où $\alpha + \beta = 1$ et $r_{bits\ faux} = 1 - r_{bits\ exacts}$

$$R^2 = \max_{s \in E} \left| r_{bits\ exacts}(s) - r_{cycles}(s) \right|$$



Des critères de sélection R^0, R^1, R^2

Objectifs

Un maximum de précision

$$R^0 = \min_{s \in \{\max_{s \in E} r_{bits\ exacts}(s)\}} r_{cycles}(s)$$

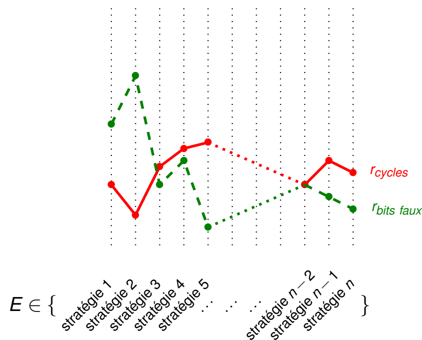
- r_{cycles} ratio de cycles entre AC et DD
- $r_{bits\ exacts}$ ratio de bits exacts entre AC et DD

Un bon compromis temps-précision

$$R^1 = \min_{s \in E} \left(\alpha \cdot r_{bits\ faux}(s) + \beta \cdot r_{cycles}(s)\right)$$

- où $\alpha + \beta = 1$ et $r_{bits\ faux} = 1 - r_{bits\ exacts}$

$$R^2 = \max_{s \in E} \left| r_{bits\ exacts}(s) - r_{cycles}(s) \right|$$



Des critères de sélection R^0, R^1, R^2

Objectifs

Un maximum de précision

$$R^0 = \min_{s \in \{\max_{s \in E} r_{bits\ exacts}(s)\}} r_{cycles}(s)$$

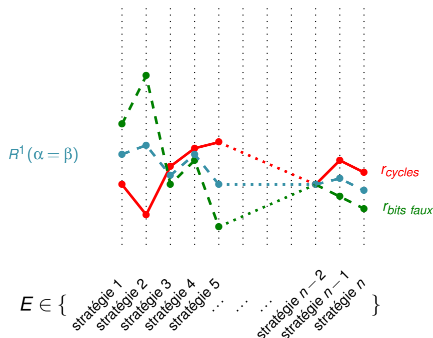
- r_{cycles} ratio de cycles entre AC et DD
- $r_{bits\ exacts}$ ratio de bits exacts entre AC et DD

Un bon compromis temps-précision

$$R^1 = \min_{s \in E} \left(\alpha \cdot r_{bits\ faux}(s) + \beta \cdot r_{cycles}(s)\right)$$

- où $\alpha + \beta = 1$ et $r_{bits\ faux} = 1 - r_{bits\ exacts}$

$$R^2 = \max_{s \in E} \left| r_{bits\ exacts}(s) - r_{cycles}(s) \right|$$



Des critères de sélection R^0, R^1, R^2

Objectifs

Un maximum de précision

$$R^0 = \min_{s \in \{\max_{s \in E} r_{bits\ exacts}(s)\}} r_{cycles}(s)$$

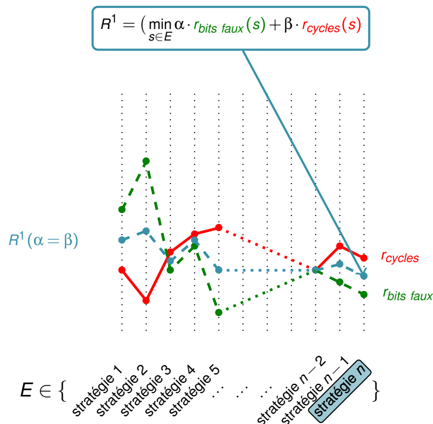
- r_{cycles} ratio de cycles entre AC et DD
- $r_{bits\ exacts}$ ratio de bits exacts entre AC et DD

Un bon compromis temps-précision

$$R^1 = \min_{s \in E} \left(\alpha \cdot r_{bits\ faux}(s) + \beta \cdot r_{cycles}(s)\right)$$

- où $\alpha + \beta = 1$ et $r_{bits\ faux} = 1 - r_{bits\ exacts}$

$$R^2 = \max_{s \in E} \left| r_{bits\ exacts}(s) - r_{cycles}(s) \right|$$



Des critères de sélection R^0, R^1, R^2

Objectifs

Un maximum de précision

$$R^0 = \min_{s \in \{\max_{s \in E} r_{bits\ exacts}(s)\}} r_{cycles}(s)$$

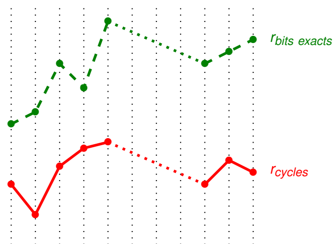
- r_{cycles} ratio de cycles entre AC et DD
- $r_{bits\ exacts}$ ratio de bits exacts entre AC et DD

Un bon compromis temps-précision

$$R^1 = \min_{s \in E} \left(\alpha \cdot r_{bits\ faux}(s) + \beta \cdot r_{cycles}(s)\right)$$

- où $\alpha + \beta = 1$ et $r_{bits\ faux} = 1 - r_{bits\ exacts}$

$$R^2 = \max_{s \in E} \left| r_{bits\ exacts}(s) - r_{cycles}(s) \right|$$



Des critères de sélection R^0, R^1, R^2

Objectifs

Un maximum de précision

$$R^0 = \min_{s \in \{\max_{s \in E} r_{bits\ exacts}(s)\}} r_{cycles}(s)$$

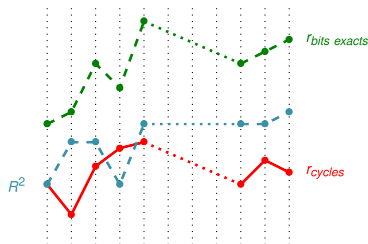
- r_{cycles} ratio de cycles entre AC et DD
- $r_{bits\ exacts}$ ratio de bits exacts entre AC et DD

Un bon compromis temps-précision

$$R^1 = \min_{s \in E} \left(\alpha \cdot r_{bits\ faux}(s) + \beta \cdot r_{cycles}(s)\right)$$

- où $\alpha + \beta = 1$ et $r_{bits\ faux} = 1 - r_{bits\ exacts}$

$$R^2 = \max_{s \in E} \mid r_{bits\ exacts}(s) - r_{cycles}(s) \mid$$



Des critères de sélection R^0, R^1, R^2

Objectifs

Un maximum de précision

$$R^0 = \min_{s \in \{\max_{s \in E} r_{bits\ exacts}(s)\}} r_{cycles}(s)$$

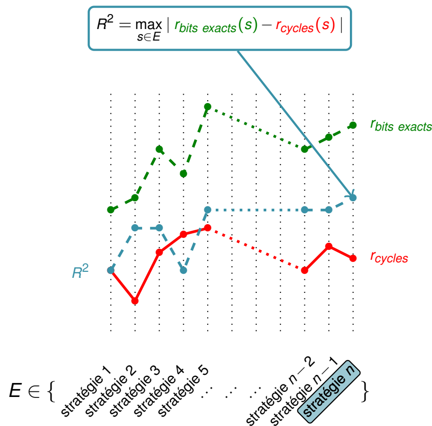
- r_{cycles} ratio de cycles entre AC et DD
- $r_{bits\ exacts}$ ratio de bits exacts entre AC et DD

Un bon compromis temps-précision

$$R^1 = \min_{s \in E} \left(\alpha \cdot r_{bits\ faux}(s) + \beta \cdot r_{cycles}(s)\right)$$

- où $\alpha + \beta = 1$ et $r_{bits\ faux} = 1 - r_{bits\ exacts}$

$$R^2 = \max_{s \in E} \mid r_{bits\ exacts}(s) - r_{cycles}(s) \mid$$



Retour sur la démonstration : le programme HORNER

```
int main(int argc, char** argv)
{
  int n=16;
  double x=atof(argv[1]);

  double P[]={
    2.373046875000000e-01, -4.192382812500000e+00,
    3.467285156250000e+01, -1.781982421875000e+02,
    6.370019531250000e+02, -1.679423828125000e+03,
    3.378095703125000e+03, -5.288271484375000e+03,
    6.511538085937500e+03, -6.327524414062500e+03,
    4.836465820312500e+03, -2.877295898437500e+03,
    1.306113281250000e+03, -4.373437500000000e+02,
    1.018750000000000e+02, -1.475000000000000e+01,
    1.000000000000000e+00};

  double r, res;
  int i;



  r = P[n];


  for(i=n-1; i>=0; i--) {
    r = r * x + P[i];
  }

  res = r;



  printf("%.16e %.16e\n",x,res) ;
  return(0);
}
```

Conditions

Polynôme $p_H(x) = (x - 0,75)^5 (x - 1,0)^{11}$

Données 256 $x \in \{0,35 : 0,45\}$ (dist. uniforme)

temps $r_{cycles} = 0,05$

précision $r_{bits\ exacts} = 0,64$

Environnement Linux 3.2.0.51, Core i5 2.53GHz, gcc 4.6.3, -O2, papi 5.1.0.2

Retour sur la démonstration : le programme HORNER

```c
int main(int argc, char** argv)
{
  int n=16;
  double x=atof(argv[1]);

  double P[]={
    2.373046875000000e-01, -4.192382812500000e+00,
    3.467285156250000e+01, -1.781982421875000e+02,
    6.370019531250000e+02, -1.679423828125000e+03,
    3.378095703125000e+03, -5.288271484375000e+03,
    6.511538085937500e+03, -6.327524414062500e+03,
    4.836465820312500e+03, -2.877295898437500e+03,
    1.306113281250000e+03, -4.373437500000000e+02,
    1.018750000000000e+02, -1.475000000000000e+01,
    1.000000000000000e+00};

  double r, res;
  int i;

#pragma COHD time block
{
  r = P[n];

#pragma COHD for decrease
  for(i=n-1; i>=0; i--) {
    r = r * x + P[i];
  }

  res = r;
}

#pragma COHD accuracy res
  printf("%.16e %.16e\n",x,res) ;
  return(0);
}
```

Conditions

Polynôme $p_H(x) = (x-0,75)^5(x-1,0)^{11}$

Données 256 $x \in \{0,35 : 0,45\}$ (dist. uniforme)

temps $r_{cycles} = 0,05$

précision $r_{bits\ exacts} = 0,64$

Environnement Linux 3.2.0.51, Core i5 2.53GHz, gcc 4.6.3, -O2, papi 5.1.0.2

Préparation du code

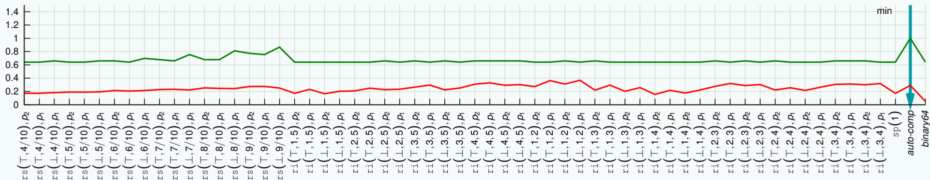
Ajout de directives de pré-compilation

- `#pragma COHD time block`
- `#pragma COHD accuracy res`

Résultats sur le programme HORNER

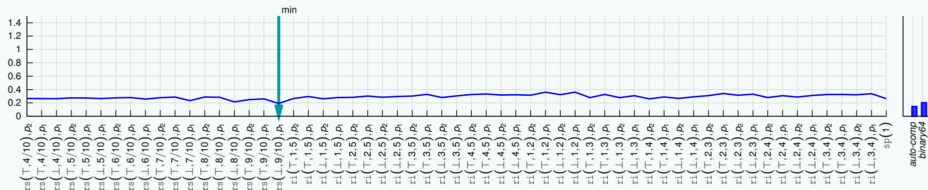
$R^0 \longrightarrow$ *auto-comp*

r_{cycles}, $r_{bits\ exacts}$

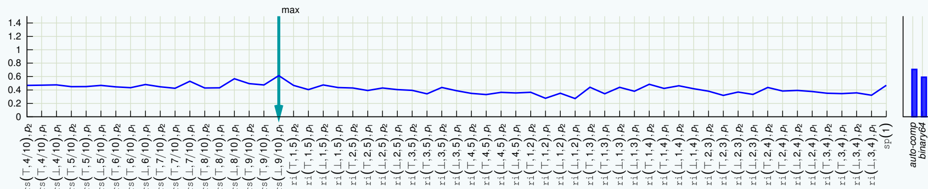


Résultats sur le programme HORNER

$R^1 \rightarrow \mathtt{rs}(\perp, 9/10), p_1$

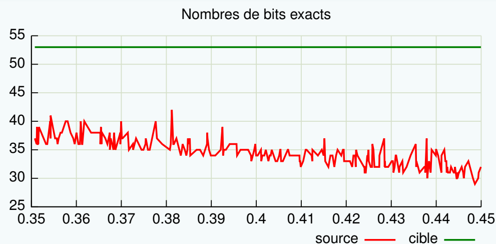


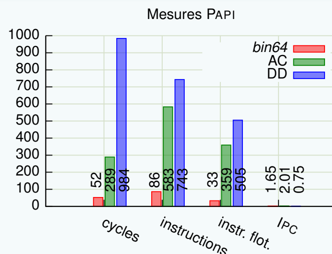
$R^2 \rightarrow \mathtt{rs}(\perp, 9/10), p_1$



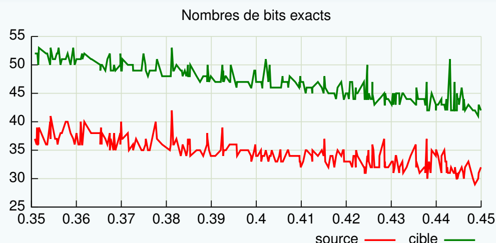
Résultats sur le programme HORNER

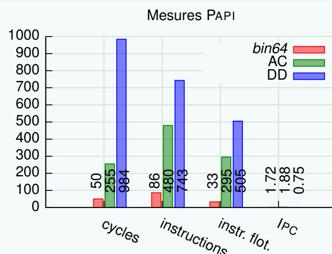
R^0





R^1 / R^2





Résultats expérimentaux : synthèse

Algorithmes et données	Ratio de sélection R^0	✓/✗	R^1/R^2	✓/✗
SUM(d_1)	*auto-comp*	✓	$\mathtt{ri}(\top, 10^3, 5\cdot 10^3), p_2$	✗•
SUM(d_2)	*auto-comp*	✓	$\mathtt{ri}(\top, 10^4, 5\cdot 10^4), p_2$	✗•
HORNER(p_H, d_4)	*auto-comp*	✓	$\mathtt{rs}(\top, 9/10), p_1$	✓
HORNER(p_H, d_5)	*auto-comp*	✓	$\mathtt{rs}(\top, 9/10), p_2$	✓
HORNERDER(p_H, d_4)	*auto-comp*	✓	$\mathtt{sp}(1)$	✗•†
HORNERDER(p_H, d_5)	*auto-comp*	✓	$\mathtt{rs}(\top, 9/10), p_2$	✓
CLENSHAWI(p_C, d_4)	*auto-comp*	✓	$\mathtt{rs}(\bot, 8/10), p_1$	✓
CLENSHAWI(p_C, d_6)	*auto-comp*	✓	$\mathtt{rs}(\top, 8/10), p_2$	✓
CLENSHAWII(p_C, d_4)	*auto-comp*	✓	$\mathtt{rs}(\bot, 8/10), p_1$	✓
CLENSHAWII(p_C, d_6)	*auto-comp*	✓	$\mathtt{rs}(\top, 8/10), p_2$	✓
DECASTELJAU(p_D, d_8)	*auto-comp*	✓	$\mathtt{rs}(\bot, 9/10), p_1$	✗◦
DECASTELJAU(p_D, d_9)	*auto-comp*	✓	$\mathtt{rs}(\bot, 9/10), p_1$	✗◦
DECASTELJAUDER(p_D, d_8)	*auto-comp*	✓	$\mathtt{rs}(\bot, 9/10), p_1$	✗◦
DECASTELJAUDER(p_D, d_9)	*auto-comp*	✓	$\mathtt{rs}(\bot, 9/10), p_1$	✗◦

• Échec dû au manque de précision
◦ Échec dû au manque de performances
CLENSHAW : la stratégie $\mathtt{rs}(\top, 9/10)$ n'est pas retenue ici car trop proche du programme en AC
† Il existe une meilleure solution qui répond avec succès : la stratégie $\mathtt{rs}(\bot, 7/10), p_1$

Plan de l'exposé

1. Notions d'arithmétique flottante et d'amélioration de la précision
- Arithmétique flottante IEEE 754
- Transformations sans erreur
- Arithmétique double-double et algorithmes compensés

2. Transformation automatique de programme
- Mise en place : l'outil CoHD
- Méthodologie : correction de la précision
- Les transformations de la somme et du produit
- Résultats expérimentaux avec comparaisons à l'existant

3. Synthèse de code pour des compromis entre précision et temps d'exécution
- Mise en place : l'outil SyHD
- Des stratégies de transformation de programme avec compromis
- Ensemble de programme et critères de sélection
- Résultats expérimentaux

4. Conclusions et perspectives

Contributions

Un nouveau moyen d'améliorer la précision

- Via des transformations source-à-source :

complètes résultats équivalents à ceux de la bibliographie : *algorithmes compensés*

- doublement de la précision
- cas du raffinement itératif pour la résolution de système linéaires

partielles stratégies de transformation pour des compromis performance–précision

- génération fine de code selon des attentes d'utilisateurs

Contributions

Un nouveau moyen d'améliorer la précision

- Via des transformations source-à-source :

complètes résultats équivalents à ceux de la bibliographie : *algorithmes compensés*

- doublement de la précision
- cas du raffinement itératif pour la résolution de système linéaires

partielles stratégies de transformation pour des compromis performance–précision

- génération fine de code selon des attentes d'utilisateurs

Un processus d'optimisation automatique

- Assuré par une implémentation logicielle de la synthèse de code
  - **CoHD** transformation de code C
  - **SyHD** synthèse de code : objectifs d'optimisation, données, environnements

Contributions

Un nouveau moyen d'améliorer la précision

- Via des transformations source-à-source :

complètes résultats équivalents à ceux de la bibliographie : *algorithmes compensés*

- doublement de la précision
- cas du raffinement itératif pour la résolution de système linéaires

partielles stratégies de transformation pour des compromis performance–précision

- génération fine de code selon des attentes d'utilisateurs

Un processus d'optimisation automatique

- Assuré par une implémentation logicielle de la synthèse de code
  - **CoHD** transformation de code C
  - **SyHD** synthèse de code : objectifs d'optimisation, données, environnements

Publications

- SCAN12 : *Automatic Code Transformation to Optimize Accuracy and Speed in Floating-Point Arithmetic* [LMT12]
- SCAN10 : *Trade-off Between Accuracy and Time for Automatically Generated Summation Algorithms* [LMT10a]
- PASCO10 : *Accuracy Versus Time : A Case Study with Summation Algorithms* [LMT10b]

Perspectives

Compléter les contributions existantes

- Vers le support :
  - d'un ensemble plus large d'opérations : $\div$, $\sqrt{}$
  - complet du C (*boucle* `while`, *récursivité*), ou autre langage
- Vers de nouvelles stratégies de transformation de programme : *compléter* sp
- Vers l'intégration de nouveaux critères d'optimisation : *taille du code, consommation*
- Vers plus d'automatisme : *gestion dynamique des stratégies de transformation*
- Vers une amélioration automatique de la précision non bornée : SumK, HornerK
- Vers une application partielle des expansions
- Ouverture vers d'autres domaines : *temps réel, arithmétique fixe*

Perspectives

Compléter les contributions existantes

- Vers le support :
  - d'un ensemble plus large d'opérations : $\div, \sqrt{}$
  - complet du C (*boucle* `while`, *récursivité*), ou autre langage
- Vers de nouvelles stratégies de transformation de programme : *compléter* sp
- Vers l'intégration de nouveaux critères d'optimisation : *taille du code, consommation*
- Vers plus d'automatisme : *gestion dynamique des stratégies de transformation*
- Vers une amélioration automatique de la précision non bornée : SUMK, HORNERK
- Vers une application partielle des expansions
- Ouverture vers d'autres domaines : *temps réel, arithmétique fixe*

Valorisations

- Publication des résultats des derniers chapitres de la thèse
- Mise en application : post-doc NUMALIS

M E R C I

[Dek71] T.J. Dekker
A Floating-Point Technique for Extending the Available Precision, 1971

[GLL09] S. Graillat, P. Langlois, N. Louvet
Algorithms for Accurate, Validated and Fast Polynomial Evaluation, 2009

[HBL01] Y. Hida, X.S. Li, D.H. Bailey.
Algorithms for Quad-Double Precision Floating Point Arithmetic, 2001

[IEEE754] IEEE Standard for Floating-Point Arithmetic
Microprocessor Standards Committee of the IEEE Computer Society, 3 Park Avenue, New York, NY 10016-5997, USA, 2008

[Iou13] A. Ioualalen
Transformation de programmes synchrones pour l'optimisation de la précision numérique, 2013

[JBL11] H. Jiang, R. Barrio, H. Li, X. Liao, L. Cheng, F. Su
Accurate Evaluation of a Polynomial in Chebyshev Form, 2011

[JGH13] H. Jiang, S. Graillat, C. Hu, S. Li, X. Liao, L. Chang, F. Su
Accurate Evaluation of the k-th Derivative of a Polynomial and its Application, 2013

[JLCS10] H. Jiang, S. Li, L. Cheng, F. Su
Accurate Evaluation of a Polynomial and its Derivative in Bernstein Form, 2010

[Lou07] N. Louvet.
Algorithmes compensés en arithmétique flottante : Précision, validation, performances, 2007

[LL07] P. Langlois, N. Louvet.
More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms, 2007

[LMT10a] P. Langlois, M. Martel, L. Thévenoux
Trade-off Between Accuracy and Time for Automatically Generated Summation Algorithms

[LMT10b] P. Langlois, M. Martel, L. Thévenoux
Accuracy Versus Time : A Case Study with Summation Algorithms

[LMT12] P. Langlois, M. Martel, L. Thévenoux
Automatic Code Transformation to Optimize Accuracy and Speed in Floating-Point Arithmetic, 2012

[MBdD10] J.M. Muller, N. Brisebarre, F. de Dinechin, C.P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, S. Torres.
Handbook of Floating-Point Arithmetic, 2010

[MPFR] L. Fousse, G. Hanrot, V. Levèfre, P. Pélissier, P. Zimmermann
MPFR : A Multiple-Precision Binary Floating-Point Library with Correct Rounding

[ROO05] S.M. Rump, T. Ogita, S.Oishi.
Accurate Sum and Dot Product, 2005

[She97] J.R. Shewchuk
Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates, 1997