Optimisation d'expressions dans les applications scientifiques

Julien Zory

—

Équipe PIPS

Centre de Recherche en Informatique

École des Mines de Paris

Mai 1998

Introduction 2

➡ Contexte :

Applications de calcul scientifique

Evaluation répétée d'expressions (nids de boucles)

Architectures modernes (ILP - nouvelles instructions)

➡ Objectif :

Améliorer les performances

Optimiser les expressions

➡ Approche :

Utiliser les transformations algébriques

Expressions équivalentes au sens mathématique

MAIS temps d'évaluation non constant

➥ Etudier l'impact de ces transformations algébriques

➡ Difficultés

Explosion combinatoire du nombre d'expressions équivalentes

Tenir compte des capacités des processeurs récents

① Introduction

② Exemples

APPLU (SPEC FP95) - ONDE24 (IFP)

③ Optimisation d'expressions

Factorisation

Extraction des multiply-add

④ Implémentation et premiers résultats

⑤ Related Work

⑥ Conclusion et perspectives

Exemples - contexte 5

- Architecture

  processeurs de la gamme IBM RS6000 (SP2)

  power2 wide node 590 (66MHz)

  P2SC 595 (120 et 160 MHz)

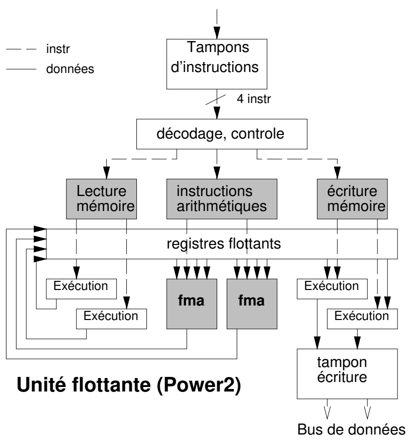
- Compilation

  compilateur natif IBM `XLFortran` (V3.2.5)

  options `-O3 -qarch=pwr2 -qhot`

  préprocesseur `KAP` pour IBM Fortran (V3.3)

Exemples - Le Power2 6



Unité flottante (Power2)

➡ 2 unités de calculs flottants (fpu)

➡ opérations pipelinées

  sauf division et racine carré

  2 opérations émises par cycle

➡ latence de 2 cycles pour

  l'addition

  la multiplication

➡ multiplication + addition en une seule instruction (fma)

➡ lecture/écriture d'un mot de 128 bits en mémoire (données contigues – quad load/store)

- Un extrait tiré des benchmarks **SPEC FP95**
- Code écrit en Fortran 77
- La même expression apparaît plusieurs fois
- La plupart des procédures pourraient être optimisées
- Evaluation naïve :

  $12 \times 5n^3$ additions et $30 \times 5n^3$ multiplications

  nx = ny = nz = 100 et réels double précision

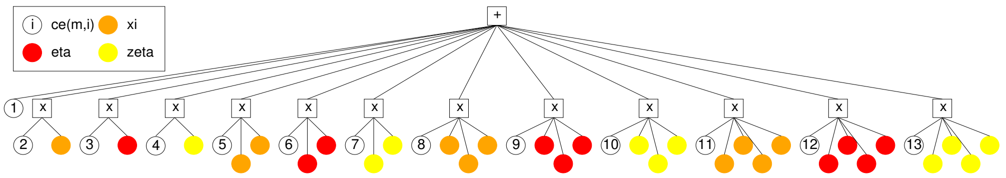
  temps d'exécution : 0.78 sec


```
[...]
do j = 2, ny - 1
 eta = ( dfloat(j-1) ) / ( ny - 1 )
 do i = 2, nx - 1
  xi = ( dfloat(i-1) ) / ( nx - 1 )
  do k = 1, nz
   zeta = ( dfloat(k-1) ) / ( nz - 1 )
   do m = 1, 5
    ue(m,k) =  ce(m,1)
            + ce(m,2) * xi
            + ce(m,3) * eta
            + ce(m,4) * zeta
            + ce(m,5) * xi * xi
            + ce(m,6) * eta * eta
            + ce(m,7) * zeta * zeta
            + ce(m,8) * xi * xi * xi
            + ce(m,9) * eta * eta * eta
            + ce(m,10) * zeta * zeta * zeta
            + ce(m,11) * xi * xi * xi * xi
            + ce(m,12) * eta * eta * eta * eta
            + ce(m,13) * zeta * zeta * zeta * zeta
   end do
[...]
```

Exemple 1 - APPLU (3/8)

Structure initiale

$12 \times 5n^3$ additions
$30 \times 5n^3$ multiplications

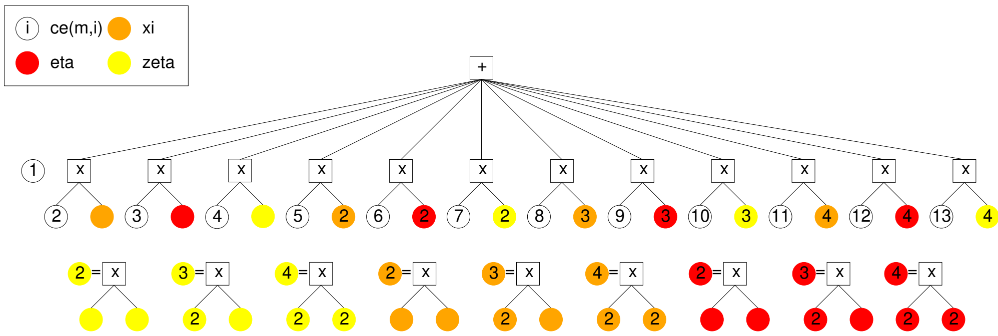


Exemple 1 - APPLU (4/8)

Invariant code motion

Non appliquée par le compilateur `xlf` (associatif gauche)

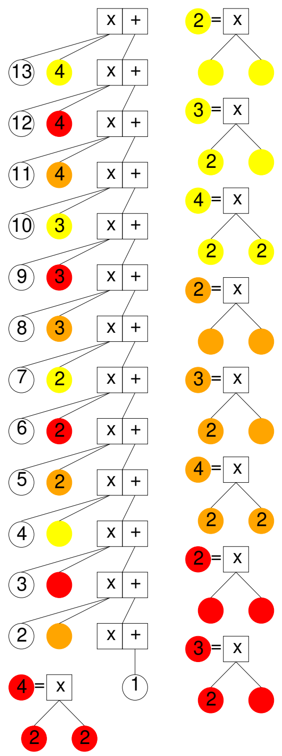
$12 \times 5n^3 + 3n^3 + 3n^2 + 3n$ multiplications
$12 \times 5n^3$ additions



Invariant code motion + extraction des multiply-add

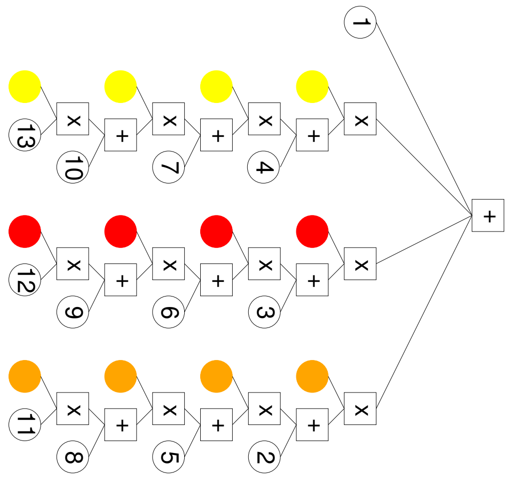
$3n^3 + 3n^2 + 3n$ multiplications

$12 \times 5n^3$ multiply-add



Factorisation

Moins d'opérations - Meilleur équilibrage



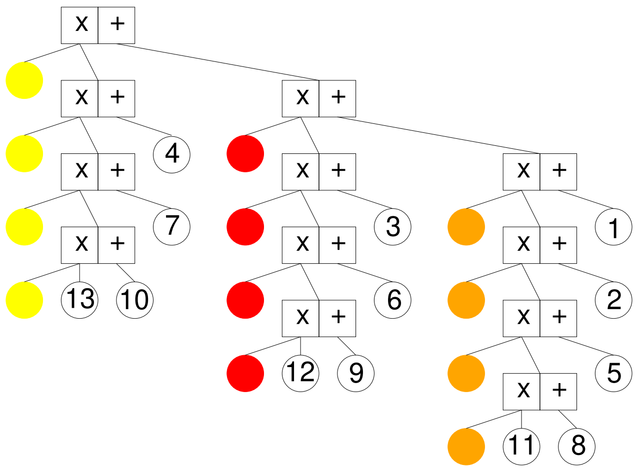
$12 \times 5n^3$ multiplications

$12 \times 5n^3$ additions

Exemple 1 - APPLU (7/8) 13

Factorisation + extraction des multiply-add

Encore moins d'opérations



$12 \times 5n^3$ multiply-add

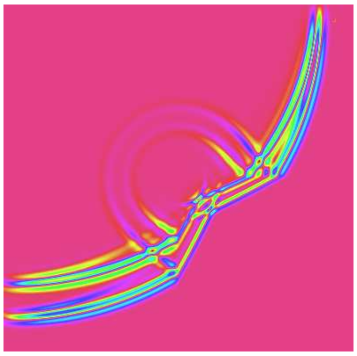
Exemple 1 - APPLU (8/8) 14

Résultats

sur processeur P2SC 595 cadencé à 160MHz

Version	Nombre d'opérations	Temps d'exécution (secondes)	Gain
Initiale	$210n^3$	0.78	-
Invariant code motion + fma	$135n^3$	0.49	37%
Factorisation	$120n^3$	0.35	55%
Factorisation + fma	$60n^3$	0.26	67%

Exemple 2 - ONDE24 (1/10)

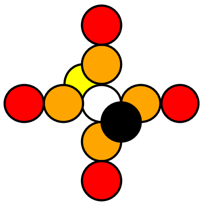


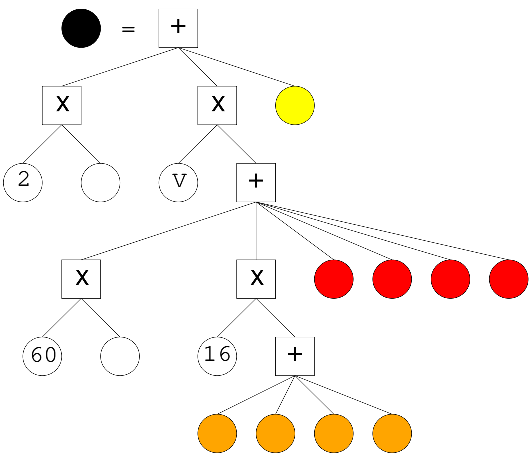
- ➡ Application de propagation d'ondes sismiques 2D
- ➡ 75% du temps passé dans une expression
- ➡ Code écrit en Fortran 77
- ➡ 10 additions et 4 multiplications
- ➡ au mieux 5 cycles pour 14 opérations

Processeur	MHz Rate	MFLOP/s Peak	max MFLOP/s	measured MFLOP/s
Power2 wide node 590	66	266	186	44
P2SC thin node 595	120	480	336	70
P2SC thin node 595	160	640	448	91

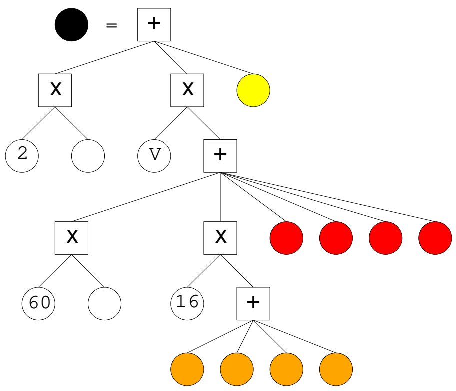
Exemple 2 - ONDE24 (2/10)


```
[...]
 DO J = 3,NP-2
   DO I = 3, NP-2
     U(I,J,KP) =
$      2 * U(I,J,KM) - U(I,J,KP)
$      - V(I,J) * ( 60*U(I,J,KM)
$      -16*( U(I+1,J,KM) + U(I-1,J,KM)
$      + U(I,J-1,KM) + U(I,J+1,KM))
$      + U(I+2,J,KM) + U(I-2,J,KM)
$      + U(I,J-2,KM) + U(I,J+2,KM))
   ENDDO
 ENDDO
[...]
```

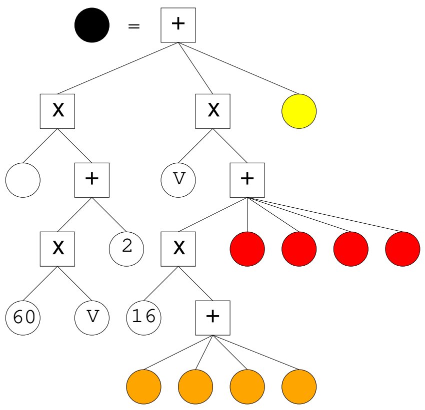




2 factorisations envisageables

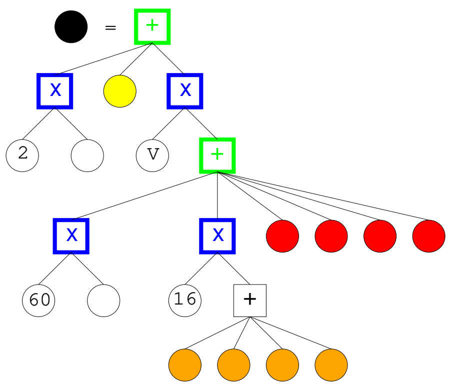


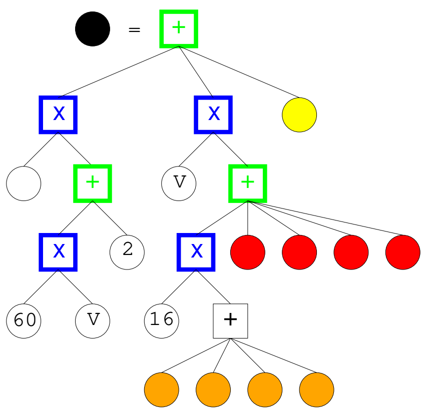
U(I,J,KM) apparaît 2 fois



V(I,J) apparaît 2 fois

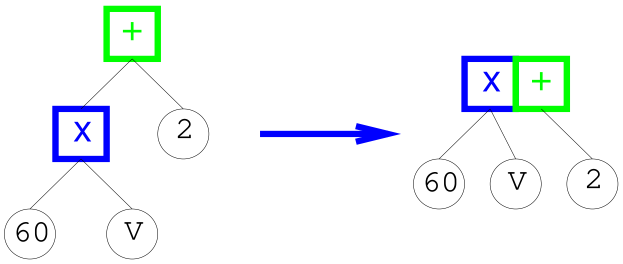
4 multiply-add



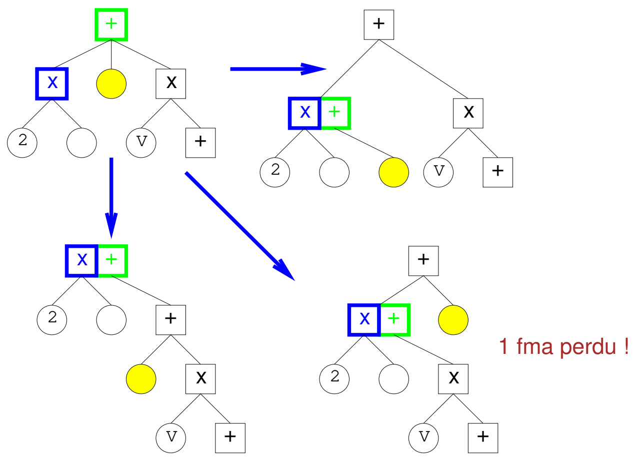


➡ Plusieurs alternatives à considérer

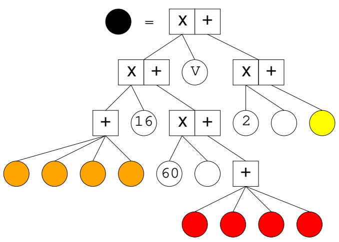
Un cas simple



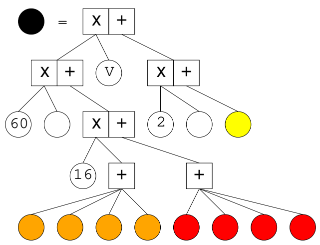
Un cas plus complexe



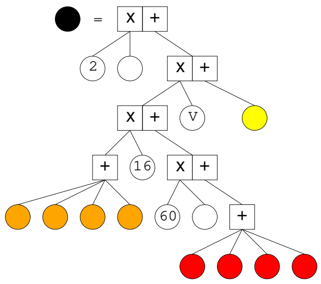
Exemple 2 - ONDE24 (7/10)



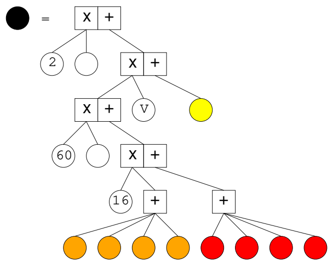
151 MFLOP/s



149 MFLOP/s

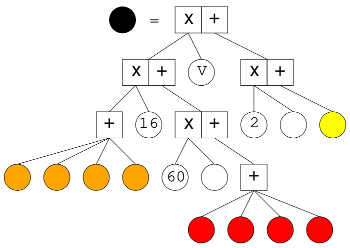


137 MFLOP/s

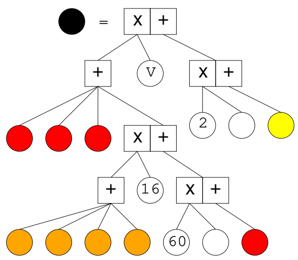


130 MFLOP/s

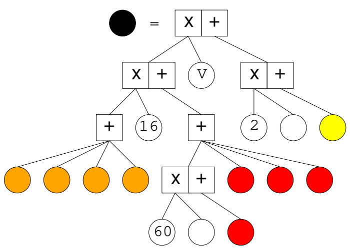
Exemple 2 - ONDE24 (8/10)



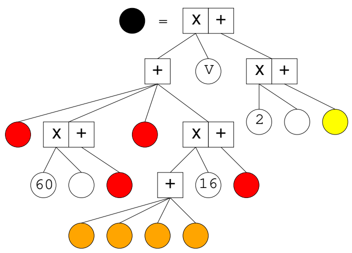
151 MFLOP/s



141 MFLOP/s



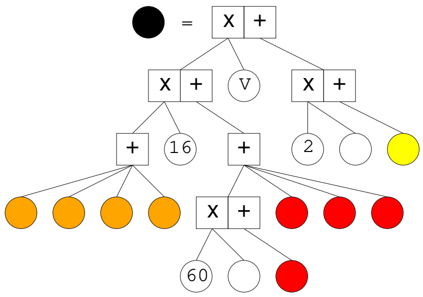
164 MFLOP/s



144 MFLOP/s

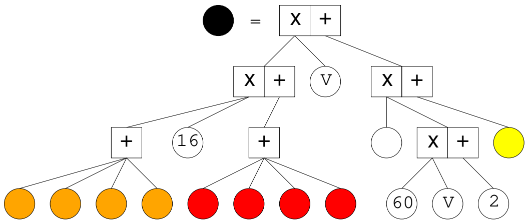
Exemple 2 - ONDE24 (9/10)

2 *bonnes* configurations



issue de la factorisation 1

➡ 164 MFLOP/s



issue de la factorisation 2

➡ 162 MFLOP/s

Exemple 2 - ONDE24 (10/10)

- ➡ De bons résultats :

  améliorations des performances jusqu'à 100%

- ➡ Un grand nombre de solutions finales
- ➡ Un impact non négligeable sur les performances (26%)!
- ➡ Comment choisir ?

Version	**P2SC 595 (160 MHz)** MFLOP/s	**Power2 590 (66 MHz)** MFLOP/s
Initiale	102	47
Meilleure solution	164 / 162	96
Moins bonne solution	126	62

Optimiser les expressions

- Le quoi :

  Parcourir l'espace des expressions
  équivalentes au sens algébrique

  Evaluer l'impact des transformations sur les performances

  Choisir une *bonne* combinaison de transformations

  Objectif : **améliorer les performances**

- Les contraintes

  Technique de compilation (beaucoup d'expressions)

  Combinatoire importante

  L'espace est **trop "vaste"** pour un parcours exhaustif

  Sans simuler ou exécuter les expressions pour choisir

- Le comment :

  Un problème d'optimisation

  Ne pas parcourir tout l'espace des solutions

  Se limiter à certaines transformations élémentaires

  Prendre des **décisions locales**

  Rater éventuellement de bonnes solutions

  Mettre en place des **critères de décisions** (approchés)

- ➡ Basées sur des propriétés algébriques

  Commutativité - Associativité

  Distributivité $x \times (y + z) = x \times y + x \times z$

  Elément neutre et absorbant $x + xy = x(1 + y)$

  Autres : Division $a/b = a \times b^{-1}$ - Carré $a^2 = a * a$

  Opérateurs logiques AND OR NOR ...

  Algèbre (max,+)

- ➡ A différents niveaux

  Dans la structure de donnée

  Au sein de réécritures

  Dans des algorithmes spécifiques

Comment comparer deux expressions ?

- ➡ De nombreux paramètres entrent en jeux

  nombre et latence des opérations

  chemin critique

  nombre d'unités de calculs - profondeur du pipeline

  pression de registres - caches ...

- ➡ **coût réel :** temps d'évaluation sur un processeur donné

  compliqué à mettre en oeuvre

  coûteux en temps

- ➡ **coût estimé :** valeurs théoriques

  sert de point de départ pour la *comparaison*

  peut être plus ou moins affiné

- poids total

  un *poids* w_i est associé à chaque opération (noeud i dans l'arbre représentant l'expression)

  valeur importante si une seule unité de calcul

  $W = \sum_{i=1}^{n} w_i$

- chemin critique (pondéré ou non)

  chaque noeud se trouve à une profondeur d_i

  valeur importante si *infinité* d'unités de calcul

  $C = \max_{i=1}^{n} d_i$

- gravité

  compromis entre W et C

  $G = \frac{\sum_{i=1}^{n} w_i \times d_i}{W}$

Deux phases distinctes

- phase de normalisation

  moteur de réécriture

  applique toutes les règles possibles

  déterminisme assuré par l'algorithme de pattern-matching ??

- 2 heuristiques (factorisation et extraction des multiply-add)

  prise de decision nécessaire

  explosion combinatoire du nombre de solutions

  processus itératif : décision locale

  critères de choix

- Objectifs

  supprimer toute mise en forme préalable
  (distribuer - forward substitution)

  appliquer des transformations algébriques simples

- Méthode

  techniques de **pattern-matching associatif-commutatif**

  basé sur STORM (Orléans - Stony Brook)

  règles de **réécriture** (avec gardes) (CAVEAT - CEA)

  représentation interne (associativité-commutativité)

  plusieurs bases de règles

  converge si pas de règles contradictoires

$\boxed{\text{Principes}}$

- appliquer une transformation

$$x \times y_1 + x \times y_2 + \ldots + x \times y_n + z \Rightarrow x \times (y_1 + y_2 \ldots + y_n) + z$$

- un gain de $(n-1)$ multiplications
- pour tout couple d'opérateurs distributifs - ex: $(\max,+)$
- prise en compte des opérateurs inverses

$$x \times y - x \times z = x \times (y - z)$$

➡Rechercher un pattern x et construire la nouvelle expression

Factorisation

pattern-matching inadapté

➡ manque d'expressivité

$x + x \times y_1 + x \times y_2 + \ldots + x \times y_n + z$

le x initial est *facultatif*

n est *inconnu* à priori

➡ prise de décision nécessaire entre TOUTES les factorisations

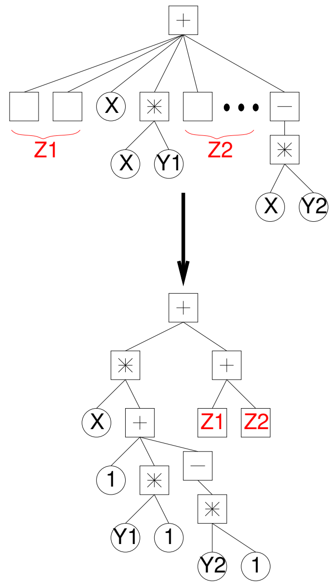
pour chaque *étape*

$i_1 \times i_2 \ldots \times i_p \times y_1 + \cdots + i_1 \times i_2 \ldots \times i_p \times y_n$

$(2^p - 1) \times n!$ matches

Factorisation - Algorithme

- Algorithme bottom-up
- Pour tous les noeuds dans l'arbre
- Recherche d'un pattern x
- Construction progressive des Y_i et Z



Function Factorization(n)
- *Input:* n is a node
- *Output:* a list of possible factorizations for this node

if (operator(n) is "+") **then**
  let *list of terms* = children of(n)
  let Z_1 = 0
  for each $term$ **in** *list of terms*
    let *list of successors* = successors of $term$ in *list of terms*
    for each X **in** list_of_candidates($term$)
      let Z_2 = 0
      let *list of Y* = (nil)
      for each $successor$ **in** *list of successors*
        if (X appears in list_of_candidates($successor$)) **then**
          add $successor$[X_found→"1"] to *list of Y*
        else
          Z_2 = Z_2 "+" $successor$
        end if
      end for each $successor$
      if (*list of Y* is not empty) **then**
        add $term$[X_tested→"1"] to *list of Y*
        (X,*list of Y*,Z_1 "+" Z_2) is a candidate for factorization
      end if
    end for each X
    Z_1 = Z_1 "+" $term$
  end for each $term$
end if

Function list_of_candidates(n)
- *Input:* n is a node
- *Output:* a list of candidates for the factorization

if (operator(n) is "-") **then** n = child of(n) **end if**
if (operator(n) is "*") **then return** children of(n)
else return singleton(n)
end if

Factorisation - Algorithme 37

➡ Complexité : $\mathcal{O}(n^2 m^2)$

pour un noeud de l'arbre

n termes avec chacun m candidats potentiels

si comparaisons en temps constant
(structure de donnée adéquate - hashage)

➡ Améliorations

conserver la liste des candidats (d'une étape à l'autre)

ne pas tester plusieurs fois le même x

implémentation paramétrique des opérateurs

Factorisation - Critères de choix 38

Comment choisir ?

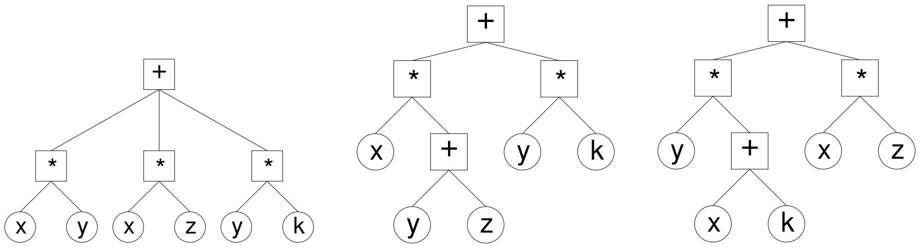
- Conflit possible entre plusieurs factorisations

  $a \times b + a \times x_1 + b \times x_2$

  $a \times (b + x_1) + b \times x_2$ ou $a \times x_1 + b \times (a + x_2)$

- nombre d'opérations **identique** dans les deux cas
- (si on suppose l'élimination des sous-expressions communes)
- *MAIS* **équilibrage** de l'arbre potentiellement **différent**

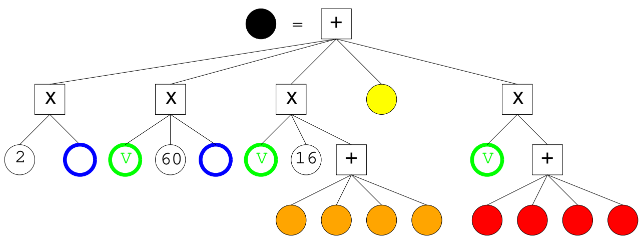
➡ utilisation de la gravité G comme critère de choix

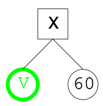
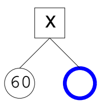
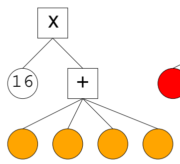
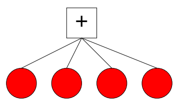


Version initiale — Factorisation 1 — Factorisation 2

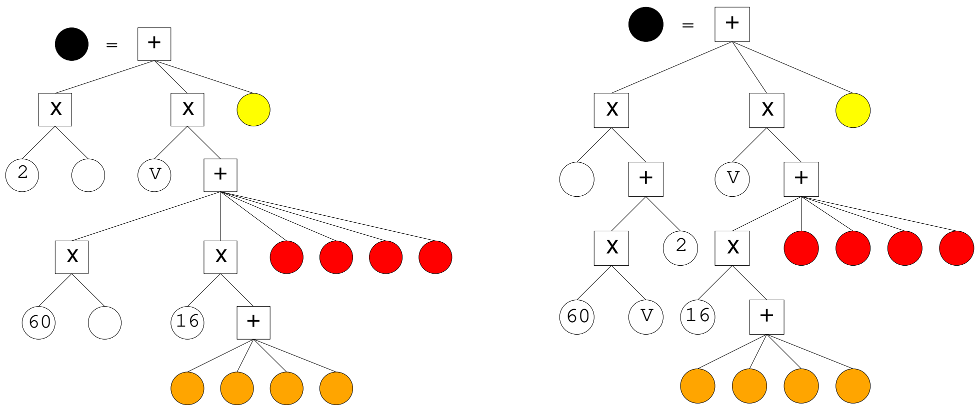
$$G_1 \leq G_2 \Leftrightarrow (W_y + W_z) \leq (W_x + W_k)$$

choisir la factorisation avec $\sum_{i=1}^{n} W_{Y\,i}$ minimal



X	Y1	Y2	Y3	W
	2			1
				4

2 factorisations



- transformation de N à N' telle que $d'_i = d_i + n$ et $w'_i = w_i$

$$G_{N'} \times W_{N'} = \sum_{i=1}^{n} w'_i \times d'_i = \sum_{i=1}^{n} (d_i + n) w_i = (G_N + n) W_N$$

- $G_1 = \frac{w_+ + 2\times(w_* + w_*) + 3\times w_+ + (G_x+2)W_x + (G_k+2)W_k + (G_y+3)W_y + (G_z+3)W_z}{2\times(w_+ + w_*) + W_x + W_y + W_z + W_k}$
- $G_2 = \frac{w_+ + 2\times(w_* + w_*) + 3\times w_+ + (G_y+2)W_y + (G_z+2)W_z + (G_x+3)W_x + (G_k+3)W_k}{2\times(w_+ + w_*) + W_x + W_y + W_z + W_k}$

$$G_1 - G_2 = (W_y + W_z) - (W_x + W_k)$$

Principe

- reconnaître une construction du type $x \times y + z$
- une opération de moins
- faire un choix judicieux
  - prendre en compte *associativité* et *commutativité*
  - extraire le *maximum* de multiply-add (parti pris)
  - examiner l'équilibrage de l'expression

Pattern-matching inadapté

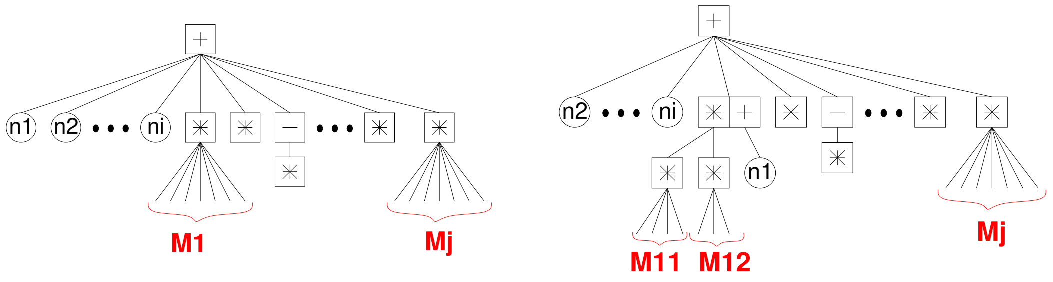
- un pattern relativement simple
- problème d'expressivité $+(x, \times(y))$
  - la multiplication est commutative $\Rightarrow$ au moins 2 opérandes
- un nombre de matches beaucoup trop grand
  - $i_1 + \ldots + i_m + j_1 \times j_2 \ldots \times j_n$
  - $(2^m - 2)(2^n - 2)$

Extraction des Multiply-add

- Favoriser le parallelisme d'instruction
- Combiner

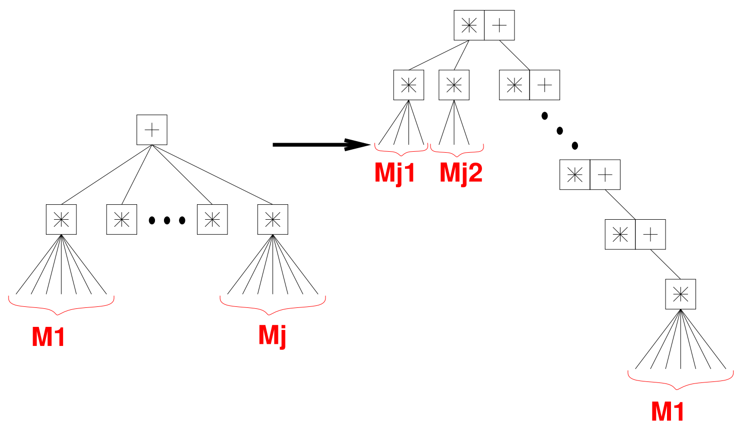
  la multiplication de *coût minimal*

  le terme **non multiplicatif** de *coût* minimal
- Permet de placer les termes ”légers” en bas



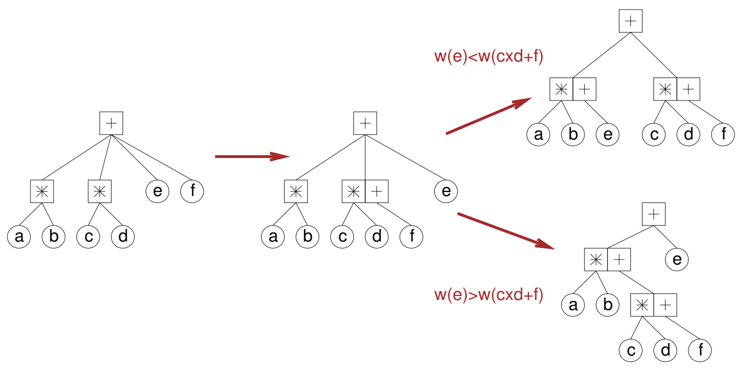
Extraction des Multiply-add

Cas particulier



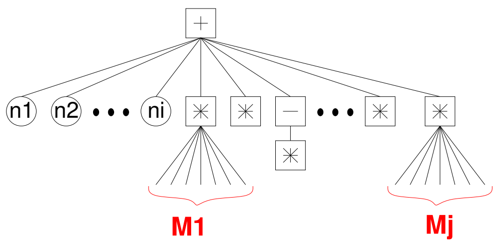
Exemple

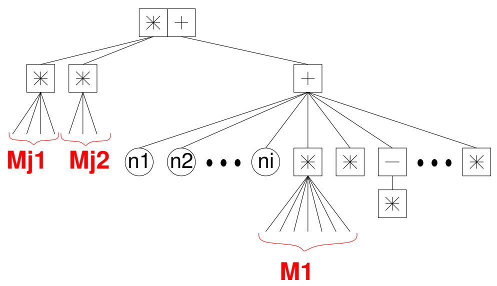
hypothèses : $W(a \times b) \geq W(c \times d)$ et $W(e) \geq W(f)$



MAIS !!!

Placer le plus haut possible dans l'arbre les sous-expressions les plus *lourdes*



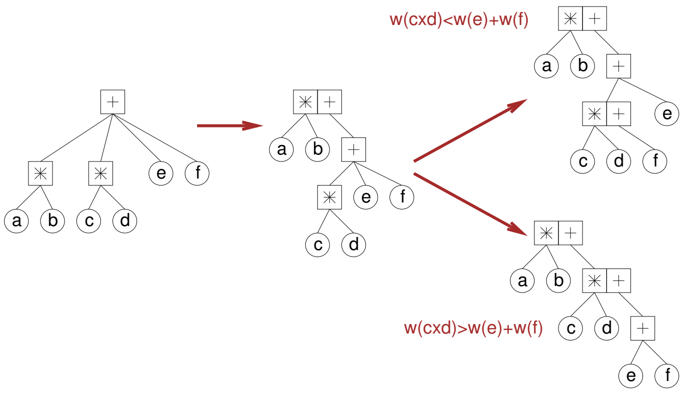


Séminaire A3/CRI
—Optimisation d'expressions—
Julien Zory — Centre de Recherche en Informatique — ÉCOLE DES MINES DE PARIS

Exemple (suite)

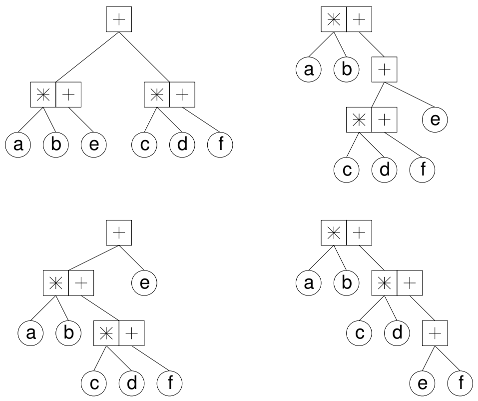
hypothèses (rappel) : $W(a \times b) \geq W(c \times d)$ et $W(e) \geq W(f)$

$$W(a \times b) \geq W(c \times d) + W(e) + W(f)$$



Exemple (fin)

hypothèses (rappel) : $W(a \times b) \geq W(c \times d)$ et $W(e) \geq W(f)$



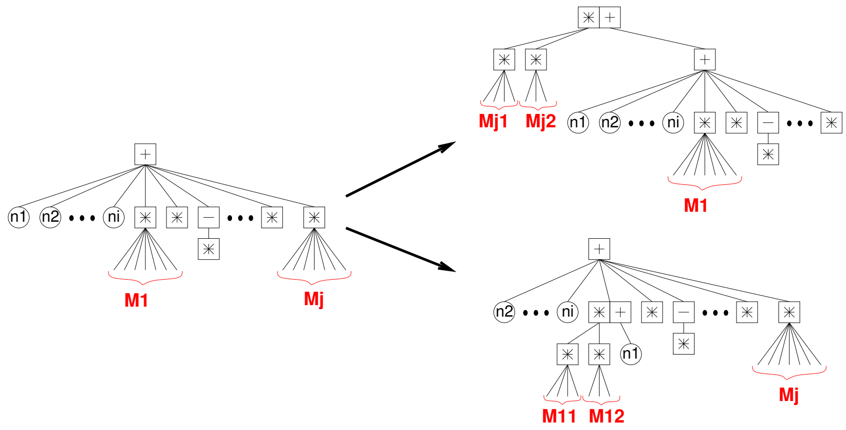
Function multiply-add extraction(n)
- *Input:* n is a node
- *Output:* a multiply-add extraction

if (operator(n) is "+") **then**
  let *list of Mult* = sorted_list_of_multiplications(n)
  let *list of Node* = sorted_list_of_remaining_nodes(n)
  let M_1 = First(*list of Mult*)
  let M_j = Last(*list of Mult*)
  if (M_j is very heavy) **then**
    (M_j1,M_j2) = heuristic_balance(M_j)
    (M_j1,M_j2,$n[M_j \rightarrow$ "0"]) is a fma construction
  else
    (M_{11},M_{12}) = heuristic_balance(M_1)
    let N_1 = First(*list of Node*)
    (M_{11},M_{12},N_1) is a fma construction
end if

Décisions

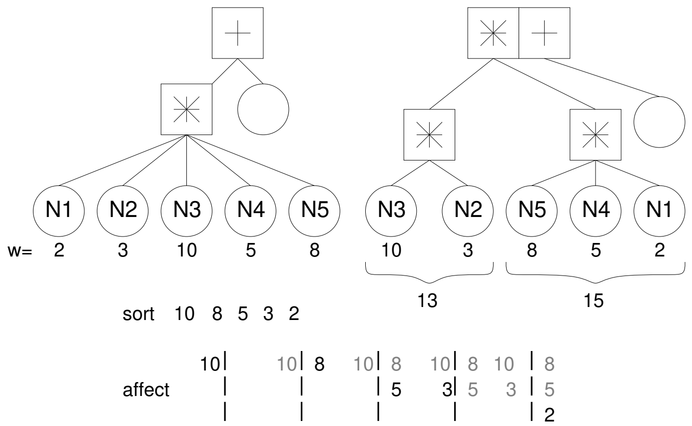
- on ne peut se contenter de toujours choisir
  - le plus lourd
  - le plus léger
- 2 cas de figures à considérer
- un seul objectif
  - placer les termes "lourds" le plus haut dans l'arbre

2 cas de figures



Construire deux sous-termes *équilibrés*

- Problème de partitionnement **NP-Complet**
- **Heuristique** classique
- Relativement rare en pratique



Un premier prototype

- Construit au dessus de

  STORM - Université d'Orléans et Stony Brook

  Simplificateur CAVEAT - CEA

- Ecrit en C++ (utilise la librairie STL)

- 38 000 lignes de code

  Storm 10 000

  Caveat 25 000

  Algorithmes 3 000

- interface avec PIPS prévue (parser fortran - informations)

- Optimisation **automatique** des expressions

  normalisation - factorisation - extraction des multiply-add

- Temps de calcul faible

  de l'ordre de la demi-seconde pour les exemples (SuperSPARC)

  4 secondes pour une expression comportant $97\times$ et $33+$

- Des résultats **très encourageants**

  on va jusqu'à diviser par 2 le temps d'exécution

  résultats parfois meilleurs que ceux obtenus manuellement

  APPLU : le meilleur est trouvé automatiquement

  ONDE24 : la solution à 162 MFLOP/s est choisie (factorisation 2)

Langages

- Transformations souvent interdites par les langages
  - à cause de l'instabilité numérique
  - en Fortran : associativité et commutativité autorisées si respect des parenthèses
  - en C : autorisé dans un premier temps puis interdit
- MAIS
  - diminution du nombre d'opérations (multiplications)
  - raisonnable si l'utilisateur n'a rien fait de spécial pour garantir les résultats
  - appliqué comme optimisation avec avertissement de l'utilisateur

Techniques de compilation

- De nombreuses techniques de compilation
  - ordonnancement des instructions
  - allocations de registres
  - software pipelining
- Les transformations algébriques
  - apportent plus de libertés
  - déroulage de boucle moins important
  - mais peuvent aussi entrer en conflit

Architecture

- ➡ plusieurs unités de calculs
- ➡ pipeline de plus en plus profond
- ➡ nouvelles instructions (fma)
- ➡ des processeurs de plus en plus nombreux

  IBM RS6K

  SGI MIPS R5000 et R10000

  HP PA-RISC

  Intel Merced (extensions multimédia MMX)

transformations algébriques

- ➡ utilisés dans d'autres domaines

  Optimisations de requêtes SQL

  tree-height reduction (VLSI DSP synthesis)

  compilation VLIW

En conclusion

augmenter les **performances** des applications

- beaucoup de temps passé à évaluer des expressions

  grosses expressions

  exécutées beaucoup de fois (nids de boucles)
- le programmeur peut difficilement optimiser

  nécessite des connaissances de bas niveau

  perte de lisibilité et de portabilité
- les compilateurs

  s'occupent des optimisations de plus bas niveau

  le code 3 adresses rend difficile de telles optimisations

En conclusion (suite)

➡ utiliser les **transformations algébriques** pour

  diminuer le **nombre d'opérations**

  profiter des **nouvelles instructions** des processeurs modernes

  fournir au compilateur plus de **parallélisme d'instruction**

  (en améliorant l'*équilibrage* de l'expression)

  permet de réduire le besoin de déroulage des boucles

➡ ne pas explorer tout l'espace des solutions

  pattern-matching ⇒ explosion combinatoire

  prise de décisions *locales* et *significatives*

  élaboration des critères de choix simples et efficaces

➡ Des résultats très encourageants et une première implémentation

Perspectives

- De nouvelles applications

  SPEC FP95

  propagation d'onde en 3 dimensions

  boucles de livermore

  Perfect Club

  code généré automatiquement (HPF - Rhône Poulenc)

- D'autres plate-formes

  processeurs R10000 - 21264 - HP PARISC - Merced ...

  compilateurs

- Un contexte plus général d'optimisations (assoc-comm)

  mieux équilibrer l'arbre

  invariant code motion

  élimination des sous-expressions communes

Informations

- Rapport de recherche disponible

  Using Algebraic Transformations to Optimize
  Expression Evaluation in Scientific Codes

  http://www.cri.ensmp.fr/~zory

- Contacts

  email : zory@cri.ensmp.fr

  Tel : 01.64.69.48.33