Optimisations de code

Lionel Lacassagne

LIP6

Sorbonne University (UPMC) / LIP6 / ALSOC

https://www.lip6.fr/

lionel.lacassagne@lip6.fr

Plusieurs types d'optimisation

- Optimisations de bas niveau (LLT)
  - Réalisables par un compilateur optimisant,
  - ne change pas la sémantique du programme,
  - impact variable: faible (strength reduction) $\rightarrow$ élevé (vectorisation + CLP)
- Optimisations de haut niveau (HLT)
  - Non réalisables par un compilateur optimisant (change la sémantique du programme)
  - liées au domaine applicatif (algèbre linéaire, traitement du signal, traitement d'images)
  - liées (souvent) aux transformations mémoire (*memory layout*)
  - impact (très) important (combo de transformations très différentes)
  - optimisations scalaires (+)
  - optimisations de boucles (++)
  - optimisations de boucles pour la mémoire (+++)
- Objectifs: *ninja gap*
  - connaitre les LLT = se mettre à la place du compilateur, pour le guider
  - connaitre les HLT, pour les combiner avec les LLT, car parfois le compilateur n'est pas efficace
  - pipeline et/ou fusion d'opérateurs + transformation du *memory layout*
  - $\Rightarrow$ savoir "coder" pour les systèmes embarqués

Optimisation scalaire: *strength reduction*

Réduction de force

- Objectif
  - remplacement d'un opérateur complexe/lent par un opérateur simple/rapide
  - concerne principalement les opérateurs arithmétiques

```
for(int i=1; i<n; i++) {
  z=i*x;
}
```

```
z=0;
for(int i=1; i<n; i++) {
  z=z+x;
}
```

- Fonctionne aussi sans boucle
  - multiplication / division par une puissance de 2 (`y=2x`) → décalage (`y=x<<1`)
  - division (`y=x/5`)→ multiplication par l'inverse (`y=0.2*x`)
  - les processeurs récents possèdent des instructions réciproques (voir SIMD)
- Attention: perte de précision possible `y=x/3` → `y=0.33333*x`

Optimisation scalaire: *strength reduction*

Cas particuliers

- En lien avec le codage binaire
  - `r=x%8` $\rightarrow$ `r=x&7` (rappel: $(7)_{10}=(111)_2$)
  - masquage de bits quand le modulo est une puissance de 2

Constant folding & constant propagation

- *Constant folding*
  - simplification et évaluations des expressions arithmétiques constantes connues à la compilation
  - l'expression est souvent le résultat d'une expansion de macro

```
#define N 5
int a = 10+N;
int a=15;
```

- *Constant propagation*
  - propagation de l'opération de *constant folding* dans le code
  - l'expression est souvent le résultat d'une expansion de macro

```
x=3; y=x+4;
x=3; y=7;
```

Common Sub-expression Elimination (CSE)

- Factorisation des expressions arithmétiques
  - l'expression est souvent le résultat d'une expansion de macro

```
x=a+b+1;
y=2*(a+b);
```

```
t=a+b;
x=t+1;
y=2*t;
```

  - gain d'une addition

- Pour réfléchir : $(a+b)^4$
  - v_0 (3MUL+4ADD): $c = (a+b) \times (a+b) \times (a+b) \times (a+b)$
  - v_1 (12MUL+4ADD): $c = a_4 + 4 \times a_3 \times b + 6 \times a_2 \times b_2 + 4 \times a \times b_3 + b_4$
  - avec $a_2 = a \times a, \quad b_2 = b \times b, \quad a_3 = a_2 \times a, \quad b_3 = b_2 \times b, \quad a_4 = a_2 \times a_2, \quad b_4 = b_2 \times b_2$
  - v_2 (2MUL+1ADD): $s = a + b, \quad p = s \times s, \quad c = p \times p$
  - la vitesse n'est pas uniquement proportionnelle à la complexité, mais aussi aux dépendances de données et donc à la latence des opérations ...
  - ... et cette latence peut être masquée sur les processeurs *Out-of-Order* et/ou par du déroulage de boucle (voir après)

Dead code elimination & Expression simplification

- Dead Code elimination (code non atteignable)
  - après un return
  - dans des branches de test en fonction de la sémantique (voir *narrowing*)
- Expression simplification
  - résultats d'expansion de macros
  - éléments neutres arithmétique (`+0 *1 /1`), élément absorbant (`*0`)
  - simplification d'expressions arithmétiques (`x-x x/x`)

Narrowing

- Simplification / suppression de code
  - en fonction de la dynamique des variables,
  - et/ou des constantes connues à la ocmpilation

```
unsigned short s, x;
x=s>>20; // decalage trop grand
x=0;

if(s>0x10000) // toujours faux
```

Copy propagation & instruction combining

- Copy propagation
  - propagation/réutilisation de la première variable d'affectation (casse la dépendance de données)

```
y=x;
z=y;;
y=x;
z=x;
```

- instruction combining
  - regroupement d'instructions et remplacement par une instruction équivalente

```
i++;
i++;
i+=2;
```

Inlining

- Initialement en C++ et maintenant en C (`inline`)
  - en plus des options de compilations pour indiquer/forcer le compilateur à le faire
  - augmente les opportunités d'optimisation

```
int add(int x, int y) {return x+y;}
int sub(int x, int y) {return add(x,-y);} // C++ style
```

- En remplaçant:

```
int sub(int x, int y) {return x+(-y);} // reste un CHS
int sub(int x, int y) {return x-y);}
```

Optmisation de boucle: *scalarisation*

- Scalarisation: à combiner avec des techniques d'optimisation de boucles
  - remplacement d'une expression "tableau" par une expression "registres"
  - les accès mémoire sont mémorisés dans des variables (mises en registres ?)

```
for(int i=0;i<n;i++) {
  D[i]=A[i]+B[i];
  E[i]=A[i]*C[i];
}
for(int i=0;i<n;i++) {
  a=A[i]; b=B[i]; d=a+b; D[i]=d;
  c=C[i]; e=a*c; E[i]=e;
}
```

- Conseil:
  - séparation du code en 3 parties: lecture mémoire, calculs, écriture mémoire
  - très utile lorque le corps de boucle est complexe...

```
for(int i=0;i<n;i++) {
  a=A[i]; b=B[i]; c=C[i]; // Load
  d=a+b; e=a*c;           // Calc
  D[i]=d; E[i]=e;         // Store
}
```

Code hoisting

- Déplacement du code constant
  - le code constant à l'intérieur d'une boucle est sorti de la boucle
  - il est remonté d'autant de niveau(x) (de nids de boucles) que nécessaire (ici 1)

```
for(int i=0;i<n;i++) {
  A[i]=x+y;
}

t=x+y;
for(int i=0;i<n;i++) {
  A[i]=t;
}
```

Loop Invariant code motion #1

- Déplacement du code indépendant de la boucle
  - Avant: $2n^2$ MUL + n^2 ADD

```
for(int i=0;i<n;i++) {
  for(int j=0;j<n;j++) {
    X[i][j]=a*i+b*j;
  }
}
```

  - Après: $(n^2 + n)$ MUL + n^2 ADD

```
for(int i=0;i<n;i++) {
  u=a*i;
  for(int j=0;j<n;j++) {
    X[i][j]=u+b*j;
  }
}
```

Loop Invariant code motion #2

- Combinaison avec optimisations précédentes
  - suppression des multiplications associées à `j` (via *strength reduction*)
  - complexité: n MUL + $2n^2$ ADD

```
for(int i=0;i<n;i++) {
  u=a*i;
  v=0;
  for(int j=0;j<n;j++) {
    X[i][j]=u+v;
    v=v+b;
  }
}
```

  - suppression des multiplications associées à `i`
  - complexité: $2n^2 + n$ ADD, plus aucune multiplication ...

```
u=0;
for(int i=0;i<n;i++) {
  v=0;
  for(int j=0;j<n;j++) {
    X[i][j]=u+v;
    v=v+b;
  } // j
  u=u+a; // Attention:  a l'exterieur de la boucle j
}
```

Loop Fusion

- Initialement
  - diminution de l'impact du contrôleur de boucle
- Maintenant
  - améliorer la localité des données si ré-use d'un tableau (ici `A`)
  - ajouter de la scalarisation

```
for(int i=0;i<n;i++)
  D[i]=A[i]+B[i];
for(int i=0;i<n;i++)
  E[i]=A[i]*C[i];
```

```
for(int i=0;i<n;i++) {
  D[i]=A[i]+B[i];
  E[i]=A[i]*C[i];
}
```

- Attention
  - Les notations pointeurs ne sont pas plus rapides que les notations tableaux avec un compilateur moderne et optimisation
  - les notations pointeurs sont justes illisibles et *error prone*

```
for(int i=0;i<n;i++) {
  *D++ = *A++ + *B++;
  *E++ = *A++ * *C++;}
```

Loop Distribution

- L'inverse de loop-fusion
  - diminution la quantité de mémoire accédée par boucle $\Rightarrow$ les accès mémoire peuvent mieux tenir dans les caches
  - peut améliorer les opportunités de loop-blocking (toujours pour le cache)
  - peut diminuer le risque de code spilling (si plus de registres disponibles, sauvegardes des registres dans la pile, ie, écriture dans le cache L1)

Loop collapsing

- Représente le coté obscur de la Force
  - fusion de deux boucles imbriquées en une seule
  - diminue l'impact du contrôleur de boucle
  - améliore la prédiction de branchement puisqu'il n'y a plus de branchement
  - uniquement en dernier recours, après avoir épuisé les autres solutions
  - sachant que le compilateur sait le faire, et qu'il y a même une décoration OpenMP pour cela

```
int T[H][W];
for(i=0; i<H; i++) {
  for(j=0; j<W; j++) {
    T[i][j] = i*W+j;
  }
}
int *p = (int*) &T[0][0];
for(k=0; j<H*W; k++) {
  p[k] = k;
}
```

Loop normalisation

- En Info les indices commencent à 0, mais à 1 en Math ...assembleur
  - on veut $T[i] \leftarrow i$
  - error prone ...

```
int T[N+1];
for(i=1; i<=N; i++) {
  T[i]=i;
}
int for(i=0; i<N; i++) {
  T[i+1]=i+1;
}
```

Loop unrolling #1

- Le déroulage de boucle est l'une des optimisations les plus importantes
- Avant
  - diminue le temps passé dans le controleur de boucle
  - diminue le risque d'erreur de prédiction d'un branchement
  - augment les opportunités d'optimisation
  - ne pas oublier l'épilogue
- Maintenant
  - permet de cacher la latence des instructions
  - rôle particulier en Traitement du Signal et des Images (TSI)

```
for(i=0; i<n; i++) {
  f(i);
}
for(i=0; i<n; i+=2) {
  f(i+0);
  f(i+1);
}
// epilogue simple car deroulage d'ordre 2
if(n%2==1) f(n-1);
```

Loop unrolling #2

- Plusieurs façons d'écrire la même chose

```
for(i=0; i<N; i+=2) {
  f(i+0);
  f(i+1);
}
for(i=0; i<N/2; i++) {
  f(2*i+0);
  f(2*i+1);
}
```

- Problème: les 2 versions sont fausses ...
- Version 1:
  - espace d'itérations: $i \in [0, n-1] \Rightarrow (i+0) \in [0, n-1]$ mais $(i+1) \in [1, n]$
- Version 2:
  - espace d'itérations: $i \in [0, n/2-1] \Rightarrow (2i+0) \in [0, 2(n/2-1)]$ et $(2i+1) \in [1, 2(n/2-1)+1]$
  - car $2(n/2) \neq n$, cela dépend de la parité de n
  - $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$ si n est pair, $\frac{n-1}{2}$ si n est impair AN: $5/2 = (5-1)/2 = 2$

Loop unrolling #3

une méthode qui marche tout le temps

- Faire des "paquets" exacts d'itérations
  - calculer le reste de la division de n par l'ordre de déroulage: $r = n \mod k$
  - la boucle devient `for(i=0; i<n-r; i+=k)` pour un intervalle semi-ouvert
  - la boucle devient `for(i=0; i<=n-1-r; i+=k)` pour un intervalle fermé
- Gérer l'épilogue en énumérant les indices en fonctions de n

```
switch(r) {
  case 0: break;
  case 1: f(n-1); break;
  case 2: f(n-2); f(n-1); break;
}
```

- Problème: la taille de l'épilogue est quadratique $k(k-1)/2$
  - faire une boucle `for(i=n-r; i<n; i++) f(i);`
    on retrouve le terme `n-r` de l'intervalle semi-ouvert qui était alors exclu
  - faire un Duff's device: `https://en.wikipedia.org/wiki/Duff's_device`
    inventé par Tom Duff pour Lucasfilm, novembre 1983

Loop unrolling #3

Duff's device

- Duff's device pour l'épilogue
  - l'ordre du `switch-case` est inversé
  - il n'y a plus de `break` (le dernier a été ajouté par esthétisme)

```
i=n-r;
switch(r) {
  case 2: f(i); i++;
  case 1: f(i);
  case 0: break;
  }
```

- Le Duff's device complet (corps de boucle + épilogue)
  - avantage: code compact
  - inconvénient code illisible, debug et maintenance compliqués
    "the most dramatic use yet seen of fall through in C"
  - en SIMD il n'est pas toujours possible de faire un Duff's device, donc ...

```
i=0; r=n%3; m=(n+2)/3;
switch(r) {
  case 0: do { f(i++);
  case 2:      f(i++);
  case 1:      f(i++);
               } while(--m >0);
}
```

Loop unwinding

- Déroulage complet de la boucle
  - si les indices sont connus à la compilation
  - si le nombre d'itérations est petit

```
for(i=0; i<4; i++) {
  f(i);
}
f(0); f(1); f(2); f(3);
```

External loop unrolling #1

- Déroulage de la boucle externe
  - diminue l'impact sur les dépendances de données
  - notez la boucle supplémentaire dans l'épilogue ...

```
for(i=0; i<n; i++) {
  for(j=0; j<n; j++) {
    f(i,j);
  }
}
r=n%2;

for(i=0; i<n-r; i+=2) {
  for(j=0; j<n; j++) {
    f(i+0,j);
  }
  for(j=0; j<n; j++) {
    f(i+1,j);
  }
}
if(r) {
  for(j=0; j<n; j++) {
    f(n-1,j);
  }
}
```

External loop unrolling #2

- Déroulage de la boucle externe et fusion des boucles internes
  - les boucles internes ont par construction le même espace d'itérations

```
r=n%2;
for(i=0; i<n-r; i+=2) {
  for(j=0; j<n; j++) {
    f(i+0,j);
    f(i+1,j);
  }
}
if(r)
  for(j=0; j<n; j++) { f(n-1,j); }
```

Unroll & Jam #1

- Déroulage et mélange
  - diminue l'impact des dépendances de données
  - épilogue volontairement manquant

```
for(i=0; i<n; i++) {
  D[i]=A[i]+B[i]+C[i];
}
for(i=0; i<n-r; i+=2) { // Loop unrolling classique
  D[i+0]=A[i+0]+B[i+0]+C[i+0];  // i+0
  D[i+1]=A[i+1]+B[i+1]+C[i+1];} // i+1
// unroll & jam
for(i=0; i<n-r; i+=2) {
  D[i+0]=A[i+0]+B[i+0];  // i+0
  D[i+1]=A[i+1]+B[i+1];  // i+1

  D[i+0]=D[i+0]+C[i+0];  // i+0
  D[i+1]=D[i+1]+C[i+1];} // i+1
```

- Problème: augmentation du nombre d'accès mémoire
  - LU: 6 LOAD + 2 STORE
  - U&J: 8 LOAD + 4 STORE
  - on dénombre les accès mémoire écrit par l'utilisateur en C, pas ceux générés par le compilateur ...

Unroll & Jam #2

- Déroulage et mélange avec scalarisation

```
for(i=0; i<n-r; i+=2) {
  // Load
  a0=A[i+0]; a1=A[i+1];
  b0=B[i+0]; b1=B[i+1];
  c0=C[i+0]; c1=C[i+1];
  // Calc
  d0=a0+b0; d1=a1+b1;
  d0=d0+c0; d1=d1+c1;
  // Store
  D[i+0]=d0; D[i+1]=d1;
}
```

- Avantages
  - U&J: 6 LOAD + 2 STORE, comme LU
  - bonne lisibilité et extensible

Software pipelining #1

- Pipeline logiciel (à l'instar du pipeline du processeur)
  - très important pour les processeurs VLIW (DSP C6x),
  - important pour les processeurs classiques
  - diminue l'impact des dépendances de données
- Pipeline d'une expression ponctuelle `X[i]`

```
for(i=0; i<n; i++) {
  E[i]=(A[i]*B[i]+C[i])/D[i];
}
```

- Fonctionnement
  - découpage d'expressions complexes en expressions simples (comme U&J)
  - et entrelacement d'expressions issues de différentes itérations de boucles
  - plus l'expression est longue et plus la dépendance sera relâchée.
  - Notation: $x_k \Leftrightarrow$ `X[k]`

cycle	iter $i=0$	iter $i=1$	iter $i=2$	iter $i=3$
0	$e_0 = a_0 \times b_0$			
1	$e_0 = e_0 + c_0$	$e_1 = a_1 \times b_1$		
2	$e_0 = e_0/d_0$	$e_1 = e_1 + c_1$	$e_2 = a_2 \times b_2$	
3		$e_1 = e_1/d_1$	$e_2 = e_2 + c_2$	$e_3 = a_3 \times b_3$
4			$e_2 = e_2/d_2$	$e_3 = e_3 + c_3$
5				$e_3 = e_3/d_3$

Software pipelining #2

- Analyse du *scheduling*
  - recherche du premier cycle contenant toutes les instructions de la boucle initiale ($\times + /$)
  - avant: prologue et après: épilogue

cycle	iter $i = 0$	iter $i = 1$	iter $i = 2$
0	$e_0 = a_0 \times b_0$		
1	$e_0 = e_0 + c_0$	$e_1 = a_1 \times b_1$	
2	$e_0 = e_0 / d_0$	$e_1 = e_1 + c_1$	$e_2 = a_2 \times b_2$
3		$e_1 = e_1 / d_1$	$e_2 = e_2 + c_2$
4			$e_2 = e_2 / d_2$

```
i=0;   // la derniere valeur de la boucle
  E[i+0]=A[i+0]*B[i+0];
  E[i+0]=E[i+0]+C[i+0];E[i+1]=A[i+1]*B[i+1];
for(i=0; i<n-2; i++) {
  E[i+0]=E[i+0]/D[i+0];E[i+1]=E[i+1]+C[i+1];E[i+2]=A[i+2]*B[i+2];
}
i=n-3; // la derniere valeur de la boucle
                          E[i+1]=E[i+1]/D[i+1];E[i+2]=E[i+2]+C[i+2];
                                              E[i+2]=E[i+2]/D[i+2];
```

Software pipelining #3

- Remarques
  - le SP ressemble au U&J mais sans augmenter la taille du corps de boucle intéressant lorsque le corps de la boucle initiale est déjà important
  - le SP peut se combiner avec du LU ou du U&J
  - la scalarisation est complexe à réaliser (renommage de registre)
  - pour être efficace, disposer une d'architecture VLIW (non SSA)

Software pipelining #4

Pipeline d'un stencil

```
for(i=0; i<n; i++) {
  Y[i]=X[i]+X[i-1];
}
```

Loop splitting

- Découpage d'une boucle
  - pour des raisons particulières (multi-threading)
  - simplifie ou élimine une dépendance en coupant la boucle en deux parties
  - chaque boucle a le même corps de boucle, mais les espaces d'itérations sont différents (leur unions doit être identique à l'espace d'itérations initial)
  - exemple de cas particulier: *loop peeling*

Loop peeling

- Traitement particulier des premières ou dernières itérations de la boucle
  - pour la parallélisation / vectorisation de la boucle

```
for(i=1;i<=64;i++) {
  X[i]=X[1]+Y[i];
}

X[1]=X[1]+Y[1];
for(i=2;i<=64;i++) {
  X[i]=X[1]+Y[i];
}
```

- Et en Fortran 90 ?
  - notions de range et notation “tableau”,
  - simplifie le codage tout en augmentant la sémantique
  - ici chaque opération est intrinsèquement indépendante des autres

```
X(2:64)=X(1)+Y(2:64)
```

Loop unswitching #1

- Suppression des structures de test d'une boucle
  - permutation du test et de la boucle qui est templacée par 2 boucles sans test
  - extrêmement efficace pour le pipeline
  - et pour la vectorisation de code

```
for(i=0;i<n;i++) {
  if(c)
    X[i]=f(i);
  else
    X[i]=g(i);
}

if(c)
  for(i=0;i<n;i++) X[i]=f(i);
else
  for(i=0;i<n;i++) X[i]=g(i);
```

Loop unswitching #2

- *Unswitching* lorsque le test pour sur l'indice de boucle
  - il faut dés-entrelacer la boucle

```
for(i=0;i<n;i++) {
  if(i%2==0)
    X[i]=i/2; // cas pair
  else
    X[i]=i-1; // cas impair
}

for(i=0;i<n;i+=2) X[i]=i/2; // i<n est faux, vrai test plus comple
for(i=1;i<n;i+=2) X[i]=i-1; // i<n est faux, vrai test plus comple
```

- Remarques:
  - `i<n` sera géré après (problème orthogonal au sujet)
  - le test a disparu (car cas simple)
  - par contre il y a un problème, lequel ?
  - comment le résoudre ?

Loop unswitching #3

- *Unswitching* lorsque le test pour sur l'indice de boucle
  - déroulage de la boucle initial ou fusion des deux nouvelles boucles

```
for(i=0;i<n;i+=2) X[i]=i/2;
for(j=1;j<n;j+=2) X[j]=j-1;
```

- Fusion
  - faire un changement de variable: $i = j - 1$

```
for(i=0;i<n-0;i+=2) X[i  ]=i/2;
for(i=0;i<n-1;i+=2) X[i+1]=i;

r=n%2;
for(i=0;i<n-r;i+=2) {
  X[i+0]=i/2;
  X[i+1]=i;
}
if(r)                 // si n est impair
  X[n-1]=(n-1)/2; // n-1 est pair et divisible par 2
```

Forward store #1

- Initialement
  - pour éliminer l'accès à des variables globales
  - accessoirement, les variables globales sont à éviter ...

```
int s;
void f(int n) {
  s=0;
  for(int i=0; i<n; i++}
    s+=i;
  }
}
void f(int n) {
  int t;
  for(int i=0; i<n; i++}
    t+=i;
  }
  s=t;
}
```

Forward store #2

toujours scalariser

- Maintenant
  - pour éliminer des accès mémoires
  - exemple: vecteur de produits scalaires

```
for(i=0; i<n; i++) {
  C[i]=0;
  for(j=0; j<n; j++) {
    C[i] += A[i][j]*B[i][j];
  }
}

for(i=0; i<n; i++) {
  c=0;
  for(j=0; j<n; j++) {
    c += A[i][j]*B[i][j];
  }
  C[i]=c;
}
```

Scalar replacement

toujours scalariser

- Scalarisation des accès mémoire (multiples)
  - remplacer les accès mémoires à un tableau par une variable

```
for(int i=0;i<n;i++) {
  D[i]=A[i]+B[i];
  E[i]=A[i]*C[i];
}

for(int i=0;i<n;i++) {
  a=A[i];
  D[i]=a+B[i];
  E[i]=a*C[i];
}
```

Loop interchange

- Permutation de boucle
  - pour parcourir les tableaux dans le sens des caches (parcour horizontal)
  - le corps de boucle reste identique
  - les boucles doivent en général être dans le même ordre que les indices

```
for(j=0; j<n; j++) {
  for(i=0; i<n; i++) {
    C[i][j]=A[i][j]+B[i][j] // balayage vertical
  }
}

for(i=0; i<n; i++) {
  for(j=0; j<n; j++) {
    C[i][j]=A[i][j]+B[i][j] // balayage horizontal
  }
}
```

Loop tiling

- *Loop blocking*
  - et par extension, *cache blocking*
  - découpage de l'espace d'itération en bloc afin d'augmenter la localité et leur réutilisation
  - pour que les données "tiennent dans le cache"
  - = amélioration de la localité spatiale et/ou temporelle
- Comment
  - et par extension, *cache blocking*
  - autant de bloc que de niveau de cache
  - autant de bloc que de boucle, chaque découpage correspondant à un niveau de cache

Pourquoi faire tout cela

- Parce que tous les compilateurs ne le font pas tout le temps
  - surtout les compilateurs pour petits processeurs / micro-contrôleurs qui disposent de moins d'accélérateurs matériels (cache à 3 niveaux, prédicteur de branchement efficace, exécution spéculative, exécution dans le désordre)
- Pour que vous sachiez le faire
  - valeur ajoutées
- Parce qu'en SIMD, les compilateurs sont moins efficaces
  - idée: comprendre le fonctionnement du compilateur pour l'aider
  - en codant d'une manière efficace pour lui (tout en restant lisible)
  - afin que chacun fasse une partie du chemin : une ré-écriture de code peut permettre au compilateur d'activer un plus grand nombre d'optimisations ou combinaison d'optimisations