Formation en Calcul Scientifique - LEM2I

Performances et Optimisations

Violaine Louvet [1]

[1] Institut Camille jordan - CNRS

12-14/12/2011

Introduction

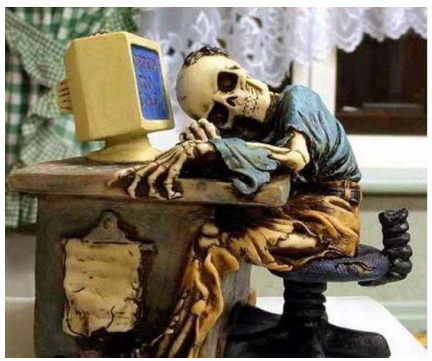
Optimisation ?

- Au sens lisibilité, portabilité, réutilisabilité
- Au sens améliorer les performances

Est-ce compatible ?

Optimiser pour :

- Exploiter au mieux l'architecture matérielle
- Réduire le temps de calcul
- Réduire l'empreinte mémoire
- Pouvoir faire tourner des calculs plus complexes, plus longs, plus gros

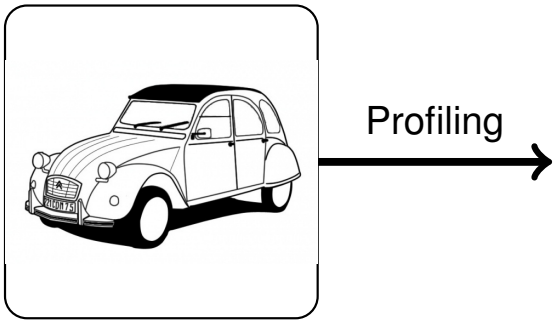


Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus

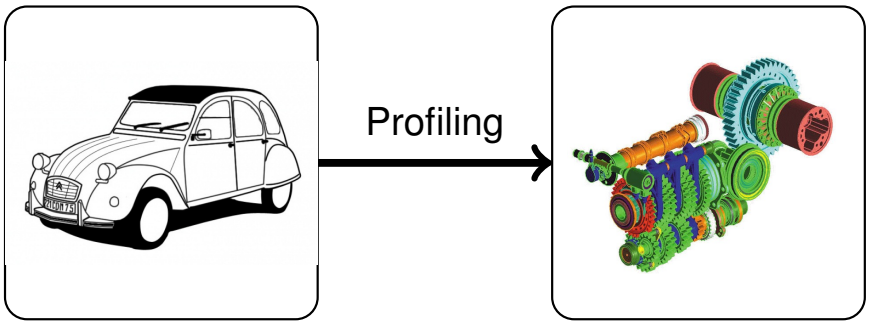
Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus
- Identifier les goulets d'étranglements, notamment en terme de temps de calcul
  - Les problèmes de mémoire entraînent la plupart du temps l'allongement du temps d'exécution



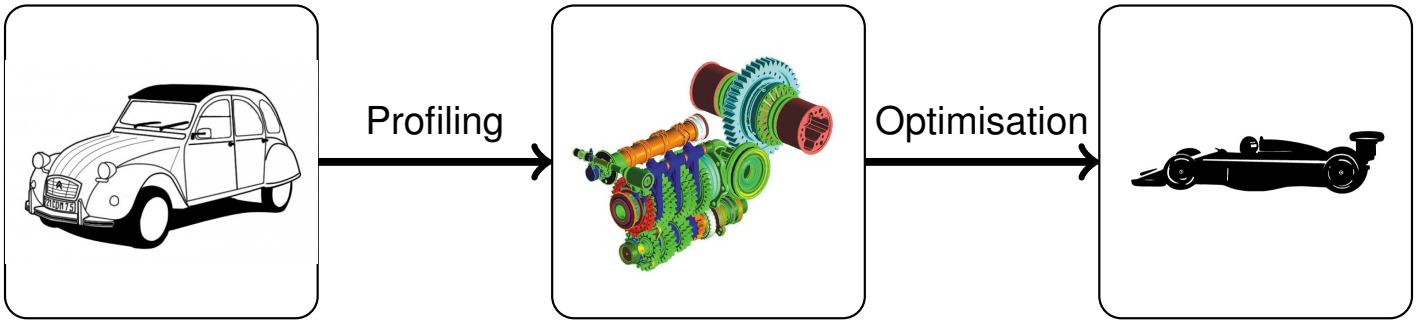
Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus
- Identifier les goulets d'étranglements, notamment en terme de temps de calcul
  - Les problèmes de mémoire entraînent la plupart du temps l'allongement du temps d'exécution
- Améliorer les parties les plus critiques



Stratégie

- Avoir un code qui fonctionne, et qui donne les résultats attendus
- Identifier les goulets d'étranglements, notamment en terme de temps de calcul
  - Les problèmes de mémoire entraînent la plupart du temps l'allongement du temps d'exécution
- Améliorer les parties les plus critiques
- Vérifier et valider le code au cours du processus d'optimisation



Sommaire

Exemple

```
program tp_opt
  real(kind=kind(1.d0)), allocatable    :: a(:,:),b(:,:)
  real(kind=kind(1.d0)), allocatable    :: c(:,:)   ! c = A*B
  real(kind=kind(1.d0)), allocatable    :: x(:)
  real(kind=kind(1.d0))                 :: ddot
  integer                               :: n  ! taille des matrices

  n = 1000
  ! initialisation des matrices
  allocate(a(n,n))
  call initA(n,a)
  allocate(b(n,n))
  call initB(n,b)
  ! Multiplication A*B
  allocate (c(n,n))
  call matmul(n,a,b,c)
  ! initialisation de x
  allocate(x(n))
  call initX(n,x)
  ! Produit matrice vecteur : x = C*x
  call matvec(n,c,x)
  ! Calcul de la norme
  call norm(n,x)
end program tp_opt
```

Sommaire

Sommaire

Mesure simple du temps

- Evaluer de façon non intrusive le temps d'exécution du programme : utilisation de la commande *time*.

```
user@ip :\$ time ./opt

 Norme :    6.99966213E+17

real 0m11.283s
user 0m11.261s
sys 0m0.008s
```

  - real : temps réel écoulé lors de l'exécution.
  - user : temps d'utilisation CPU en mode utilisateur.
  - sys : temps d'utilisation CPU en mode noyau (système).
- Dans le cas d'une exécution séquentielle : user $+$ sys $\sim$ real
- Les temps donnés par la commande *time* peuvent varier d'une exécution à l'autre, ils sont plutôt informatifs.

Mesure intrusive du temps

Mesure du temps

Evaluer plus finement de façon intrusive : utilisation de routines intrinsèques fortran.

Temps CPU : mesuré en secondes

```
real                              :: t1,t2
call CPU_TIME( t1 )
...
call CPU_TIME( t2 )
write(*,*) "Temps boucles : ",t2 - t1
```

Mesure intrusive du temps

Temps d'horloge : donne un temps *elapsed* en retournant un nombre de périodes d'horloge consommées

```
INTEGER :: nb_periodes_initial, nb_periodes_final
INTEGER :: nb_periodes_max, nb_periodes_sec, nb_periodes
REAL :: temps_elapsed
! Initialisations
CALL SYSTEM_CLOCK(COUNT_RATE=nb_periodes_sec, COUNT_MAX=nb_periodes_ma
CALL SYSTEM_CLOCK(COUNT=nb_periodes_initial)
...
CALL SYSTEM_CLOCK(COUNT=nb_periodes_final)
nb_periodes = nb_periodes_final - nb_periodes_initial
IF (nb_periodes_final < nb_periodes_initial) &
    nb_periodes = nb_periodes + nb_periodes_max
temps_elapsed   = REAL(nb_periodes) / nb_periodes_sec
write(*,*) "Temps elapsed : ",temps_elapsed
```

La valeur *count* est la pulsation courante de l'horloge interne qui est une valeur cyclique variant de 0 à *count_max*.
count_rate est la fréquence de l'horloge interne, *count_max* est la valeur maximale de l'horloge interne. Ce sont des constantes du système.

Travaux pratiques

Evaluer le temps de calcul du programme exemple :

- Sans intrusion dans le code avec la commande *time*
- En utilisant les routines intrinsèques fortran, et en évluant plus finement chacune des étapes du calcul

Sommaire

Profiling

Techniques précédentes limitées à des évaluations ponctuelles sur le temps d'exécution
Nécessité d'automatiser et de pouvoir avoir d'autres données d'analyse

Différents types d'approche

- Intrumentation statique : effectuée au plus tard à la compilation.
- Instrumentation dynamique : effectuée lors de l'exécution.

Objectifs

Outils de profiling

Outils indispensables pour optimiser de manière pertinente un code : ils permettent d'identifier les parties du code sur lesquelles il va falloir travailler :

- Temps passé dans chaque partie du programme.
- Nombre d'appels des fonctions.
- Interaction du programme avec l'environnement : accès mémoire, accès concurrents aux données ...

Vocabulaire

- Temps inclusif : temps total passé dans une fonction
- Temps exclusif : temps total passé dans une fonction duquel on soustrait les temps passés dans les fonctions appelées.

Principes

Différentes techniques

- Echantillonnage : l'exécution est échantillonnée régulièrement pour savoir quelles fonctions sont appelées. Pas d'intrusion mais résultat dépendant notamment de la fréquence d'échantillonnage.
- Instrumentation : Ajoute à chaque appel de fonction une fonction d'instrumentation qui va mesurer le temps d'appel, le nombre d'instructions exécutées ... grâce à des compteurs internes au processeur.
- Emulation : exécute le programme sur un processeur virtuel. Tout peut ainsi être mesuré de manière exacte mais cette méthode est très lente.

Sommaire

gprof

Instructions de profiling ajoutées à la compilation
Option de compilation (compilateurs GNU) : *-pg*

- Chaque fonction est modifiée à la compilation pour mettre à jour les structures de données stockant la fonction appelante et le nombre d'appels.
- Le temps d'exécution est évalué par échantillonnage.
- Pour des fonctions non instrumentées (appel de bibliothèques par exemple), seul l'information du temps passé dans la fonction est disponible.

En pratique

- Compilation : *gfortran -pg -g tp_opt.f90 -o opt*
- Exécution normale du programme : *./opt*
- Génération d'un fichier de données *gmon.out* dans le répertoire courant.
- Exécution de *gprof* : *gprof opt gmon.out*

Sorties de gprof

Profil plat

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 99.68     10.53    10.53        1    10.53    10.53  matmul_
  0.19     10.55     0.02   499499     0.00     0.00  poly_
...
```

Graphe d'appels

```
granularity: each sample hit covers 2 byte(s) for 0.09% of 10.58 seconds

index % time    self  children    called     name
                0.00   10.58       1/1           main [2]
[1]    100.0    0.00   10.58       1         MAIN__ [1]
               10.53    0.00       1/1           matmul_ [3]
                0.01    0.02       1/1           initb_ [4]
                0.01    0.00       1/1           inita_ [6]
                0.01    0.00       1/1           matvec_ [7]
                0.00    0.00       1/1           initx_ [8]
                0.00    0.00       1/1           norm_ [9]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00   10.58                 main [2]
                0.00   10.58       1/1           MAIN__ [1]
-----------------------------------------------
               10.53    0.00       1/1           MAIN__ [1]
[3]     99.5   10.53    0.00       1         matmul_ [3]
...
```

Description des sorties de gprof : profil plat

% time : pourcentage du temps d'exécution total passé à exécuter cette fonction.

cumulative seconds : temps total cumulé que le processeur a passé à exécuter cette fonction, ajouté au temps passé à exécuter les fonctions précédentes dans le tableau.

self seconds : nombre de secondes passées à exécuter cette seule fonction.

calls : nombre d'appels de la fonction.

self ms/calls : nombre de millisecondes passées dans la fonction par appel.

total ms/calls : nombre de millisecondes passées dans cette fontion et ses enfants.

name : nom de la fonction

Description des sorties de gprof : graphe d'appels

- Différentes parties séparées par des tirets.
- Chaque partie débute par la ligne primaire :
  - Elle comprend en début de ligne un nombre entre crochets.
  - Elle se termine par le nom de la fonction concernée.
  - Les lignes précédentes décrivent les fonctions appelantes.
  - Les lignes suivantes décrivent les fonctions appelées : fonctions enfants.
  - Les entrées sont classée par temps passé dans la fonction et ses enfants.

index : Index référençant la fonction.

% time : pourcentage du temps passé dans la fonction, incluant les enfants.

self : temps total passé dans la fonction.

Children : temps total passé dans les appels aux enfants.

called : nombre d'appels de la fonction. Si appels récursif, la sortie est de la forme *n+m*, *n* désigne le nombre d'appels non récursifs et *m* le nombre d'appels récursifs.

name : nom de la fonction avec son index.

TP : gprof

- Utiliser gprof sur le programme exemple.
- Sur quelle portion de code faire porter l'effort d'optimisation ?

Tau, Tuning and Analysis Utilities

Limites de *gprof*

- Contenu des traces assez limité.
- On veut pouvoir stocker dans ces traces des évènements définis par l'utilisateur.
- Profilage des applications parallèles.

TAU

- Ensemble d'outils d'analyse de performances de programmes.
- Permet de traiter beaucoup plus d'informations que *gprof* :
  - applications parallèles : informations par processus, par thread, par machine,
  - temps inclusif et exclusif pour chacune des fonctions,
  - accès aux compteurs internes,
  - information par classe, par instance, séparation des données selon les différentes instanciation des templates,
  - possibilité de profiler certaines parties arbitraires du code,
  - statistiques sur des évènements définis par l'utilisateur ...

Sommaire

Valgrind

Suite d'outils de profilage et de débogage mémoire qui permet de détecter des problèmes de gestion mémoire :

- Memcheck : un détecteur de fuites mémoires
- Cachegrind : un simulateur de caches
- Callgrind : un profileur

Fonctionnement

Le code est exécuté dans une machine virtuelle émulant un processeur équipé de nombreux outils d'analyse donnant des informations poussées sur le comportement d'un programme, qui ne pourraient être obtenues à l'aide d'un processeur matériel.

Quelques précisions

- Valgrind dégrade énormément les performances
- Ne surtout pas lancer une analyse Valgrind sur un programme complet long à exécuter

Débogage mémoire

En pratique

- Compilation du code :

```
\$ gfortran -g tp_opt.f90 -o opt
```

- On peut vérifier que l'exécution du code se déroule correctement :

```
\$ ./opt
```

- Exécution sous valgrind :

```
\$ valgrind --tool=memcheck ./opt
```

Débogage mémoire : analyse des sorties

```
...
==4264==
==4264== Invalid write of size 8
==4264==    at 0x4011F9: inita_ (tp_opt.f90:47)
==4264==    by 0x40141B: MAIN__ (tp_opt.f90:17)
==4264==    by 0x401A8D: main (tp_opt.f90:35)
==4264==  Address 0x653f240 is 0 bytes after a block of size 8,000,000 alloc'd
==4264==    at 0x4C28F9F: malloc (vg_replace_malloc.c:236)
==4264==    by 0x4013EE: MAIN__ (tp_opt.f90:16)
==4264==    by 0x401A8D: main (tp_opt.f90:35)
...
==4264==
==4264== Invalid read of size 8
==4264==    at 0x4009D2: norm_ (tp_opt.f90:139)
==4264==    by 0x4019A5: MAIN__ (tp_opt.f90:33)
==4264==    by 0x401A8D: main (tp_opt.f90:35)
==4264==  Address 0x59a3380 is 0 bytes after a block of size 8,000 alloc'd
==4264==    at 0x4C28F9F: malloc (vg_replace_malloc.c:236)
==4264==    by 0x401932: MAIN__ (tp_opt.f90:26)
==4264==    by 0x401A8D: main (tp_opt.f90:35)
==4264==
...
```

Erreurs mémoire

Les erreurs mémoire sont signalées en début de rapport. Sur l'exemple, 2 types d'erreurs :

- *Invalid write of size 8* : apparaît quand on tente d'écrire des données dans une zone incorrecte de la mémoire.
- *Invalid read of size 8* : apparaît quand on essaie de lire dans un bloc mémoire invalide.

Débogage mémoire : évaluer les fuites mémoire

```
...
==4264==
==4264== HEAP SUMMARY:
==4264==     in use at exit: 0 bytes in 0 blocks
==4264==   total heap usage: 25 allocs, 25 frees, 24,020,009 bytes allocated
==4264==
==4264== All heap blocks were freed -- no leaks are possible
```

Fuites mémoire

- Si le programme est complexe, on peut lancer *vagrind* avec l'option *–leak-check=full* pour avoir plus d'informations sur la provenance des erreurs.
- A noter : le compilateur se charge de nettoyer correctement la mémoire, mais si ce n'est pas le cas, notre programme provoque des fuites mémoire

TP : debugger les fautes de mémoire

- Utiliser *valgrind* et son outil *memcheck* pour corriger les problèmes mémoires du programme exemple

Utilisation de valgrind en mode profiler

```
\$ valgrind --tool=callgrind --dump-instr=yes ./prog
```

- *–dump-instr=yes* : le comptage d'évènement se fait au niveau des instructions. Facilite la comparaison avec le source.

- Génération d'un fichier *callgrind.out.numéro_pid*.
- Utilisation de l'outil KCacheGrind pour l'analyser.

Valgrind pour l'analyse de l'utilisation des caches

```
\$ valgrind --tool=cachegrind ./prog
```

- Il est possible de spécifier la configuration des caches I1/D1/L2.

```
==4305==
==4305== I    refs:      46,145,461,574
==4305== I1   misses:             1,679
==4305== LLi  misses:             1,645
==4305== I1   miss rate:           0.00%
==4305== LLi  miss rate:           0.00%
==4305==
==4305== D    refs:      19,055,666,894  (18,046,629,937 rd   + 1,009,036,957 wr)
==4305== D1   misses:     1,129,382,318  ( 1,128,130,135 rd   +     1,252,183 wr)
==4305== LLd  misses:         2,406,623  (     2,030,550 rd   +       376,073 wr)
==4305== D1   miss rate:            5.9% (           6.2%     +           0.1%
)
==4305== LLd  miss rate:            0.0% (           0.0%     +           0.0%
)
==4305==
==4305== LL refs:         1,129,383,997  ( 1,128,131,814 rd   +     1,252,183 wr)
==4305== LL misses:           2,408,268  (     2,032,195 rd   +       376,073 wr)
==4305== LL miss rate:              0.0% (           0.0%     +           0.0%
)
```

TP : utilisation de la suite Valgrind

- Utiliser *callgrind* pour identifier les parties du code à optimiser
- Utiliser *cachegrind* pour évaluer l'utilisation des caches

Kcachegrind

Utiliser l'outil graphique *KCachegrind* pour analyser les résultats

Sommaire

Résumé

Où en sommes nous ?

- Nous avons une estimation du temps CPU du programme
- Nous avons identifié les « points chauds » au niveau du temps d'exécution, c'est-à-dire les endroits du programme consommateurs en temps CPU
- Nous avons corrigé des erreurs liées à la mémoire
- Nous avons évalué l'usage des caches lors d'une exécution

Résumé

Où en sommes nous ?

- Nous avons une estimation du temps CPU du programme
- Nous avons identifié les « points chauds » au niveau du temps d'exécution, c'est-à-dire les endroits du programme consommateurs en temps CPU
- Nous avons corrigé des erreurs liées à la mémoire
- Nous avons évalué l'usage des caches lors d'une exécution

Où va-t-on ?

- Améliorer le temps CPU
- Améliorer l'usage des caches

$\longrightarrow$ optimiser

Niveaux d'optimisation

Différents niveaux d'optimisations :

- niveau algorithmique et numérique : il est souvent plus efficace de choisir un algorithme, une structure de données ou une méthode numériques plus adaptés au problème que de s'attaquer à de l'optimisation bas niveau.
- niveau intermédiaire : choix du langage, utilisation de bibliothèques existantes, parallélisation.
- bas niveau : travail sur les sources en lien avec le profiler, instructions sse, options du compilateur, ...

Sommaire

Optimisations haut niveau

- Méthode numérique : la méthode numérique choisit est-elle adaptée au problème, parallélisable, cohérente avec les structures de données ... ?
- Algorithme : évaluer la complexité de l'algorithme utilisé (nombre d'opérations), vérifier qu'il n'en existe pas de moins coûteux en temps, en mémoire, parallélisables ...
- Réutilisation : il y a de fortes chances que les opérations numériques nécessaires au code aient déjà été écrites : utiliser des bibliothèques existantes, souvent très optimisées et fiables.
- Accès aux données : évaluer la pertinence de la structure de données : l'accès est-il efficace, adapté à l'algorithme utilisé ?

Sommaire

Optimisations par le compilateur

Idéalement, le compilateur optimise automatiquement le code. En pratique, il n'a pas toujours assez d'informations (notamment la taille des données connue au run-time, ...).

Optimisations principales

- allocation optimale des registres, optimisation des accès mémoires
- élimination des redondances
- optimisation des boucles, en ne conservant à l'intérieur que ce qui est modifié ;
- optimisation du pipeline, utilisation du parallélisme d'instructions

Options de compilation

- Plus n est grand :
  - Plus l'optimisation est sophistiquée
  - Plus le temps de compilation est important
  - Plus la taille du code peut devenir grande
- Options de base :

  -O0 : aucune optimisation

  -O1 : optimisations visant à accélérer le code, en particulier quand il ne contient pas beaucoup de boucles.

  -O2 : *O1* + déroulage de boucles (considère notamment un aliasing strict). Augmente sensiblement le temps de compilation.

  -O3 : *O2* + transformation des boucles, des accès mémoire.

Attention

Certaines optimisations « agressives » peuvent modifier la précision des résultats (ordre des opérations par exemple).

TP : Optimisations par le compilateur

- Tester les différentes optimisations proposées par le compilateur sur le code exemple

Sommaire

Pourquoi est-ce critique ?

- Hiérarchie mémoire des processeurs actuels.
- En général, les boucles concentrent en peu de ligne un maximum de temps de calcul.
- Le compilateur fait ce qu'il peut mais il ne peut pas tout.
- Comportement régulier et répétitif : relativement simple à analyser statiquement.

Objectifs

- Améliorer l'accès aux données et donc leur localité spatiale.
- Aller plus vite.
- Réduire la surcharge dûe au contrôle des boucles.
- Investiguer les possibilités de vectorisation ou de parallélisation.

Le compilateur ne peut pas tout

- L'optimisation du compilateur est inhibée par :
  - Boucles sans compteur
  - Appel de sous-programmes dans une boucle
  - Condition de sortie non standard
  - Aliasing (pointeurs)
  - Tests à l'intérieur des boucles

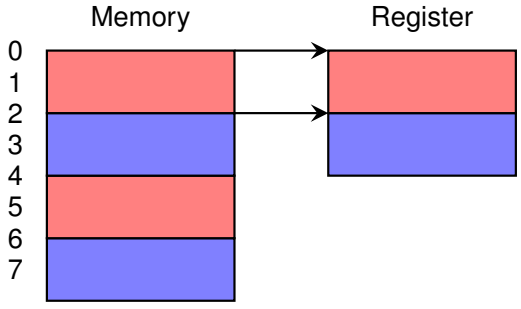
Aider le compilateur

- Il est donc important d'aider le compilateur
- En s'inspirant de ce qu'il cherche à faire :
  - Alignement de données
  - Transfomations de boucles
  - inlining
  - détection des invariants ...

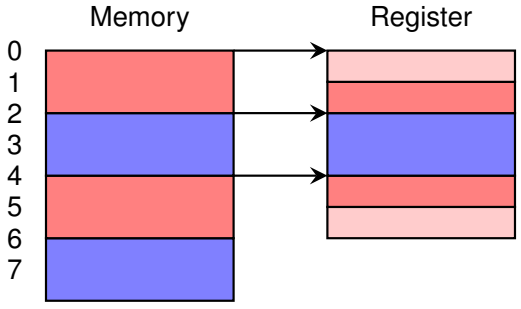
Alignement de données

- Positionner les instructions et données de manière efficace, quitte à perdre un peu d'espace mémoire
- L'accès à la mémoire se fait par morceaux de 2,4, 8, 16 ou 32 octets : granularité de l'accès mémoire
- Exemple : lecture de 4 octets avec une granularité mémoire de 2 octets

Lecture de 4 octets à l'adresse 0



Lecture de 4 octets à l'adresse 1



En pratique

- Quand on manipule des types de taille un multiple de 4 octets (32 bits) ou 8 octets (64 bits), ce qui est le cas la plupart du temps, pas trop de problèmes
- Le compilateur aligne correctement les données
- On peut forcer l'alignement par l'utilisation de directives type sse

Inlining et invariant des boucles

- Sortir les invariants des boucles :

Code original :

```
do i = 1,10
   a(i) = b(i) + c/d
end do
```

Code optimal :

```
tmp = c/d;
do i = 1,10
   a(i) = b(i) + tmp
end do
```

- Inliner les appels de petites procédures dans les boucles : permet d'éviter l'overhead lié à l'appel de la routine.

Code original :

```
do i = 1,10
   call funct(a(i))
end do
```

Code optimal :

```
do i = 1,10
   ... ! Corps de la routine funct
end do
```

Déroulage de boucles

- Loop unrolling : réduit le coût de la gestion de la boucle (nombre de comparaison total, branchements, incréments) et augmente le parallélisme d'instructions potentiellement exploitable par les processeurs.

Code original :

```
do i = 1,50
   a(i) = b(i)
end do
```

Code optimal :

```
do i = 1,50,2
   a(i) = b(i)
   a(i+1) = b(i+1)
end do
```

A noter

Le compilateur va probablement le faire tout seul en fonction des potentialités de l'architecture

Permutation de boucles

- Permutation des boucles imbriquées : alignement en mémoire des accès aux tableaux, réduction des défauts de cache.

Code original :

```
do i = 1,n
   do j = 1,m
      a(i,j) = 0.
   end do
end do
```

Code optimal :

```
do j = 1,m
   do i = 1,n
      a(i,j) = 0.
   end do
end do
```

- Attention, ce n'est pas toujours possible : dépendances des accès aux données dans les boucles

```
integer,parameter     :: m=???,n=???,ioff=???,joff=???

do i = 1,n
   do j = 1,m
      a(i,j) = funct(a(i+ioff,j+joff))
   end do
end do
```

Fusion de boucles

- Fusion de boucles : améliore la localité temporelle des données et réduit le nombre de branchements.

Code original :

```
do i = 1,n
   a(i) = ...
end do
do i = 1,n
   ... = ... a(i) ...
end do
```

Code optimal : l'accès en lecture de *a(i)* se fait encore après son accès en écriture. La dépendance des données est respectée.

```
do i = 1,n
   a(i) = ...
   ... = ... a(i) ...
end do
```

Distribution de boucles

- Distribution de boucles : améliore la vectorisation et/ou la parallélisation des boucles et optimise l'usage du cache pour chaque boucle.

Code original :

```
do i = 2,n-1
   a(i+1) = b(i-1) + c(i)
   b(i) = a(i)*k
   c(i) = b(i) - 1
end do
```

Code optimal :

```
do i = 2,n-1
   a(i+1) = b(i-1) + c(i)
   b(i) = a(i)*k
end do
do i = 2,n-1
   c(i) = b(i) - 1
end do
```

Boucles avec condition

- Eliminer les conditions des boucles : pas de test ni de branchement dans le corps de la boucle.

Code original :

```
do i = 1,n
   a(i) = a(i) + b(i)
   if (expression) d(i) = 0.
end do
```

Code optimal :

```
if (expression) then
   do i = 1,n
      a(i) = a(i) + b(i)
      d(i) = 0.
   end do
else
   do i = 1,n
      a(i) = a(i) + b(i)
   end do
end if
```

Partage de l'espace d'itération

- Partage de la boucle : permet souvent d'éliminer des itérations problématiques.

Code original :

```
do i = 1,n
   a(i) = b(i) + c(i)
   if (i > 10) then
      d(i) = a(i)+a(i-10)
   end if
end do
```

Code optimal :

```
do i = 1,10
   a(i) = b(i) + c(i)
end do
do i = 11,n
   a(i) = b(i) + c(i)
   d(i) = a(i)+a(i-10)
end do
```

Expansion des scalaires

- Expansion des scalaires : élimine de « fausses » dépendances.

Code original :

```
do i = 1,n
   t = a(i) + b(i)
   c(i) = t + 1
end do
```

Code optimal : la boucle peut maintenant être parallélisée.

```
allocate(t(n))
do i = 1,n
   t(i) = a(i) + b(i)
   c(i) = t(i) + 1
end do
deallocate(t)
```

Partitionnement de boucles

- Calcul par blocs : adapte la granularité (en gros la taille des blocs) à la hiérarchie mémoire.

Code original :

```
do j = 1,n
   do i = 1,n
      do k = 1,n
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      end do
   end do
end do
```

Code optimal : on choisit S (stride) pour qu'un bloc de a, de b et de c tiennent dans le cache.

```
do jj = 1,n,S
   do ii = 1,n,S
      do kk = 1,n,S
         do j = jj,min(n,jj+S-1)
            do i = ii,min(n,ii+S-1)
               do k = kk,min(n,kk+S-1)
                  c(i,j) = c(i,j) + a(i,k)*b(k,j)
               end do
            end do
         end do
      end do
   end do
end do
```

Divers

- Eviter les conversions de type inutiles :

Code original :

```
real a(n)
do i = 1,n
   a(i) = 2*a(i)
end do
```

Code optimal :

```
real a(n)
do i = 1,n
   a(i) = 2.0*a(i)
end do
```

- Eliminer les sous-expressions :

Code original : calcul du polynôme $y = a + bx + cx^2 + dx^3$, 7 multiplications et 3 additions

```
y = a + b*x + c*x**2 + d*x**3
```

Code optimal, 3 multiplications et 3 additions

```
y = a + (b + (c + d*x)*x)*x
```

TP

- A partir des informations obtenues via le profiling du code, retravailler les différents éléments du code afin d'améliorer le temps de calcul et l'accès mémoire
- Retester les différentes optimisations du compilateurs