

Optimisation de Code Séquentiel

Fabien.COELHO }
Ronan.KERYELL } @cri.ensmp.fr

Centre de Recherche en Informatique de
l'École des Mines de Paris

Master Recherche 2^{ème} année Informatique de Rennes 1 — ENSTBr
ISIA — ENSMP
octobre 2006–février 2007
Version 1.10

Stratégie

2

- Cerner *besoins* de performance
- Investissement matériel ou humain (optimisation) ? Gain de 1% de 1GF=10MF...
- Étude théorique de vitesse maximale (*complexité* + vitesse processeur & mémoire)
- Trouver goulets d'étranglement (*gprof*). *Boucles internes*
- Optimiser (algorithmes + code) les bouts de code spécifiques en *langage source*
- Étudier documentations internes de l'ordinateur (Internet...)
- Éventuellement changer les objectifs (algorithme plus cru...)



But

1

- Besoins de performance de plus en plus grand
- Baisse des coûts
- Utilisation ordinateur parallèle ?
 - ▶ Ordinateur // \equiv de plus en plus collection d'ordinateurs séquentiels
 - ▶ Si optimisation séquentielle : gain sur ordinateurs séquentiels *et* parallèles
 - ▶ Décharge ordinateurs parallèles pour de plus grands travaux
- Optimisations algorithmiques non traitées ici



Stratégie

3

- Regarder langage machine généré et modifier langage source
- Recoder en langage machine certains bouts



- + Optimisations algorithmiques
- Code plus spécialisé
- Code moins lisible (macros `cpp`, `#ifdef`,...)
- Difficile à maintenir
- Heurte certaines règles de programmation spécifiques dans l'industrie (arithmétique de pointeurs interdite, etc.)
- + Plus portable que du matériel spécifique

Compromis...



DÉPARTEMENT INFORMATIQUE — ENSI BRETAGNE

—Introduction—



Programmes étalons

6

- MFLOPS = Millions d'Opérations FLottantes Par Secondes
 \rightsquigarrow GFLOPS, TFLOPS



Les MIPS et MFLOPS sont à utiliser dans un contexte donné

Meaningless Information Per Se ☹



DÉPARTEMENT INFORMATIQUE — ENSI BRETAGNE

—Introduction—



Estimation d'une puissance de machine *a priori*

- Programmes réels (compilateur `gcc`, `TEX`, `Spice`, *votre code*, jeux) interpellant les utilisateurs
- Échantillon de programmes réels : <http://www.spec.org> CPU2000 CINT et CFPU
- Noyaux extraits de programmes (Linpack, Livermore)
- Jouets (crible d'Ératosthènes, puzzle, quicksort)
- Benchmarks synthétiques (Whetstone (flottant), Dhrystone (entier)) : créés pour représenter des comportements de programmes réels.
- MIPS = Millions d'Instructions Par Secondes \rightsquigarrow GIPS



DÉPARTEMENT INFORMATIQUE — ENSI BRETAGNE

—Introduction—



Programmes étalons

7

Problèmes :

- Importance du compilateur (version, niveaux d'optimisations, triche)
- Importance du système (version, charge)
- Comportements différents sur les tests selon les machines

Performances très différentes de la performance crête
 (garantie de non dépassement !) ☹

Perfect Benchmark : en général 1 % de la performance crête ☹



DÉPARTEMENT INFORMATIQUE — ENSI BRETAGNE

—Introduction—



- Parallélisme intraprocresseur (superscalaire), pipeline
- Hiérarchie mémoire
- Outils
- Compilation
- Algorithmique & structures de données (cours d'informatique...)
- Transformations de boucles
- Codage de l'information



Architecture des processeurs

10

- ▶ Superscalaire (planification dynamique) : processeurs modernes
- ▶ VLIW (planification statique) : Itanium
- Tailles et débit des mémoires et caches ↗



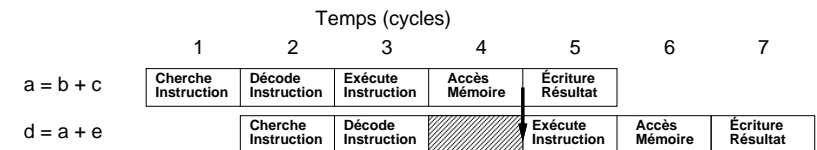
- Évolution technologies d'intégration (loi de MOORE, +60 %/an)
 - ▶ Taille transistor ↘
 - ▶ Fréquence d'horloge ↗
- Pipeline (travail à la chaîne) ↗
- Reduced Instruction Set Computer : ne réaliser que instructions courantes
 - ▶ Instructions $r1 = r2 \text{ op } r3$
 - ▶ Mouvement entre mémoire et registre
 Simple donc rapide
- Unités fonctionnelles ↗



Pipeline

11


- Augmentation du débit par travail à la chaîne
 - ▶ Intel Pentium : 5 étages
 - ▶ PentiumPro : 12 étages
 - ▶ Pentium 4 : ≈ 20 étages
- Éviter les vraies dépendances qui gèlent le pipeline ⚠



Méthode :

- ▶ Démarrer instructions au plus tôt



- ▶ Utiliser résultats le plus tard
-  Problème : nécessite de garder les données \rightsquigarrow « pression » sur registres du processeur



Opérations CISC

14

- Complex Instruction Set Computer : usine à gaz pour tout faire en matériel \rightsquigarrow veau
- Disparition processeurs CISC sauf... *x86*
 - ▶ Mais Intel PentiumPro : opérations compliquées transformées en micro-opérations RISC envoyée dans 5 pipelines en parallèle :
 - 2 d'exécution (U & V)
 - 1 de chargement
 - 1 de calcul de l'adresse de stockage
 - 1 de stockage

Assure compatibilité binaire sur jeu d'instructions style 1970



Temps d'exécution typiques d'opérations (RISC) :

- Entières simples (+, logique) : 1 cycle
- Entières compliquées (\times) : quelques cycles
- Division entière : > 32 cycles
- Flottantes simples (+, \times) : 3 cycles
- \div flottante : 30 cycles (approximation successive méthode NEWTON)



Opérations CISC

15

- ▶ Temps :
 - Style RISC (add, cmp, logique, mov registres, jCC, jmp, ...) : 1 cycle
 - Flottant fadd 3, fmul 5
 - imul 4
 - fdiv 32 bits : 18
 - fsqrt 80 bits : 69
 - Compliquées (*esi)++ : inc dword ptr [esi] traduite en 4 micro-opérations
- \rightsquigarrow Éviter les instructions complexes !



Problème des tests dans les programmes

- Temps de branchement ou de non branchement : changer déroulement du travail à la chaîne \rightsquigarrow mettre toutes les instructions à la poubelle ☹
- \exists mécanisme de prédiction de branchement dans processeurs (mémoire du passé,...) pour deviner très tôt le flot d'exécution
- Écrire le code pour minimiser les temps de branchement :
 - ▶ Supprimer en déroulant les boucles
 - ▶ Remplir les emplacements des branchements retardés le cas échéant
 - ▶ Ne pas brusquer le mécanisme de prédiction : éviter tests avec résultat aléatoire



Parallélisme intraprocresseur

18

- Arranger le code pour alimenter les unités
- Rajouter des `nop` (des riens...) peuvent aider ! Pentium...

```

.repeat                                .repeat
add esi,4      ; 1; Modify esi      add esi,4      ; 1; Modify esi
mov eax,ebx    ; 0;                mov eax,ebx    ; 0;
shl ebx,1      ; 1;                shl ebx,1      ; 1;
mov [esi],eax  ; 1; AGI stall hits  nop            ; 0; NOP !!!
inc ebx        ; 1;                mov [esi],eax  ; 1; no AGI stall
dec ecx        ; 0;                inc ebx        ; 0;
                                dec ecx        ; 1;
.until ZERO?   ; 1; Loop          .until ZERO?   ; 0; Loop
;;; 5 cycles                                ;;; 4 cycles

```



- Plusieurs unités fonctionnelles en parallèle
- Exécution en parallèle si possible
 - ▶ Dans l'ordre (Pentium instruction + instruction suivante) statique
 - ▶ Dans le désordre (PentiumPro)
- Planification : heuristique + tests d'occupation et de dépendance
 - ▶ RISC : plus simple à prédire
 - ▶ x86 : usine à gaz... quasi-impossible
 - ▶ VLIW : tout est statique, simple
- Difficile de prédire le temps d'exécution (dépend du contexte en plus...)



Instructions SIMD

19

Instruction « multimédia » style Intel Pentium MMX :

- Concatène 8 valeurs 8 bits (ou 4×16 bits ou 2×32 bits) dans 1 mot de 64 bits (registre flottant)
- Instructions de tassement/décompactage
- Additions/soustractions saturées ou pas
- Multiplications (3 cycles), logiques & décalage
- Exécutables souvent par paire :

`paddb mm0, mm1; 1 cycle`

`paddb mm2, mm3; 0 cycle, recouvre la précédente`

- Passage du mode MMX au flottant : ≈ 50 cycles...
- Amélioration Pentium 4 SSE2 128 bits : 4 op float ou 2 op double/cycle $\times 2$



Pour l'instant, pas utilisées par les compilateur... Dites-le en assembleur ou bibliothèques !



Hiérarchie mémoire

22

↔ Instauration d'une hiérarchie basée sur le compromis
taille × vitesse :

1. Registres du PE
2. Tampons d'instructions
3. Mémoire cache primaire, secondaire, tertiaire,...
4. Mémoire principale
5. Mémoire étendue (CRAY)
6. Mémoire d'autres processeurs
7. Disque magnétiques
8. Disque d'autres processeurs
9. Disques optiques



Problème : difficile de faire des mémoires rapides et grandes ☹

Heureusement ∃ localité dans les programmes :

Temporelle : une donnée accédée risque de l'être encore prochainement

Spatiale : une donnée accédée risque d'être suivie par un accès voisin

Encore plus vrai dans du code structuré : boucles, fonctions, objets, fichiers,...



Hiérarchie mémoire

23

10. Bandes (migration) étend les disques, mais très long

11. Bandes (sauvegarde)

Optimisation :

- Exploiter au mieux la hiérarchie
- Tasser dans des registres
- Rester dans le cache
- Mémoire virtuelle :
 - Rester dans les pages de mémoire accessibles (TLB)
 - Rester dans les pages en mémoire physique
- Utiliser la connaissance des mécanismes sous-jacents (précharge)



- Souvent contradictoire avec exploitation parallélisme intra-processeur



Registres

26

Mécanisme actuel : IBM Power : 6 registres en plus des 32 visibles, PentiumPro 40 registres physiques !



- Très rapide...
- Très rares ! Compromis...
- Attention aux compilateurs C avec champs de bits...
- RISC : 32 registres entiers + 32 flottants
- Intel : 8 registres entiers + 8 registres flottants
- Renommage de registres si dépendance lecture-écriture

$a = b;$

$b = c + d;$

$e = f(b);$

Exécuté en interne (table de renommage)

$a = b; b' = c + d;$

$e = f(b');$

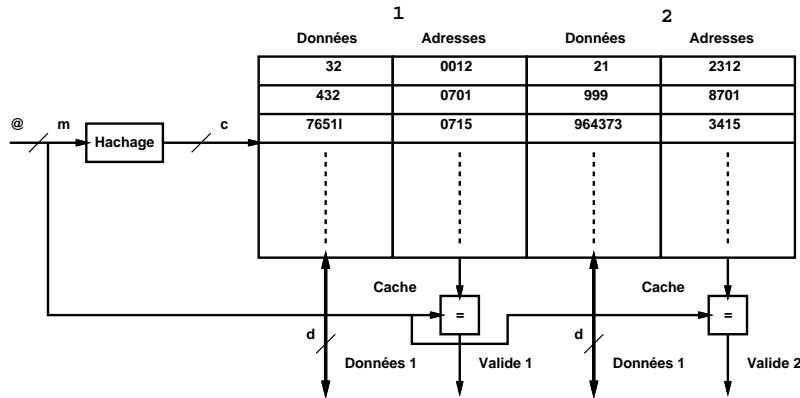


Cache

27

- Mémoire rapide et... rare
- *Essaye* de garder copie des données les plus souvent utilisées
- Organisé en lignes de caches (Pentium : 8Ko en 512 lignes de 16 octets, 128Ko L1 et 256Ko L2 AMD Athlon XP)
- n (= associativité) lignes regroupées en ensemble (Pentium : 2)





- Une valeur d'adresse (F, e, ℓ) peut être stocké dans n lignes différentes de l'ensemble e de l'élément de ligne ℓ
- Heuristique du choix de la ligne dans un ensemble (LRU :



Mémoire virtuelle

30

- Extension mémoire physique sur des disques durs
- Évite fragmentation par plusieurs processus
- Cache de traduction d'adresses virtuelles \rightarrow adresses réelles (TLB : *translation look-aside buffer*)
- Aide du système d'exploitation : algorithme de choix style LRU
 - ▶ Choix des TLB (logiciel ou matériel)
 - ▶ Choix des pages à bouger entre mémoire physique et disque (lent!) : système d'exploitation
- Essayer de respecter la localité en fonction de l'algorithme de choix : accès séquentiels, prévisibles
- Assez semblable au mécanisme de cache mais plus gros grain



least recently used)

- Plusieurs niveaux de caches possibles (secondaire dans le PentiumPro)...

Mauvais sur Pentium :

```
for(i = 0; i < N; i++) {
    ... = c[i] + c[i + 0x1004] + c[i + 0x2000]
```

S'arranger pour faire des accès sans conflits :

- Espacer les tableaux en mémoire



Mémoire virtuelle

31

- Si taux de *swap* \nearrow , performances \searrow
- Technique logicielle *out of core* : remplacer le *swap* par des échanges explicites



Modification de fonctionnement possible

- Vissage de processus en mémoire (`plock`)
- Vissage de pages en mémoire (`mlock`)
- Modification de l'algorithme de remplacement de page (`vadvise`) pour des programmes avec ramasse-miette (vision à long terme nuisible pour Lisp ou Java)



Exemple de C : pointeurs

34

- Compliquent (empêchent ?) analyses automatiques
- Donc moins d'optimisations

```
p = &a;
if (b > 0)
    p = &b;
*p = 5;
```

Lequel de a ou b vaudra 5 ?



- Langage de programmation
- Compilateurs et options
- Outils de mesures



Exemple de C : aliasing

35

Deux références (pointeurs) sur une même zone

```
void init(float * array, float * val, int size)
{
    float * end = array + size;
    while (array < end) *array++ = *val + 1.0;
}
```

- Stockage de l'adresse de `val` en registre, et recharge à chaque tour car éventuellement dans `array` !
- Parade ? `const` ? registre explicite ?
- Pas toujours possible
- Rajouter un test et écrire 2 codes différents :

```
void init(float * array, float * val, int size)
{
```



```
float * end = array + size;
float v = *val + 1.0;
if (val >= array && val < end) {
    // Cas à problème :
    while (array <= val) *array++ = v;
    v += 1.0;
    while (array < end) *array++ = v;
}
else {
    // Pas de problème :
    while (array < end) *array++ = v;
}
}
```



```
ad = db + cd; \emph{// double add}
a = (float) ad;
```

Passe son temps dans les conversions !

- Parade
 - Options de compilation
 - Données double
- C++ gère ça avec le polymorphisme



- Si beaucoup de données, utilisation possible de float (simple précision, 4 octets), au lieu de double (double précision, 8 octets) par exemple : gain de mémoire et débit mémoire
- Promotions en double lors des appels

```
float c, r;
extern double cos(double); // Bibliothèque mathématique
c = cos(r);
// transformé en fait en :
c = (float) cos((double) r); // 2 conversions
```



SUN C par défaut tout traduit en double

```
// float a, b, c; a = b + c; \emph{// traduit en fait en :}
float a, b, c; double ad, bd, cd;
bd = (double) b; cd = (double) c;
```



Stockage contigu sur les colonnes

```
double A[I][J];
for (i=0; i<100; i++)
    for (j=0; j<100; j++)
        A[i][j] = 1.0;
```

Linéarisation :

$A[i][j] \equiv *(&A[0][0] + I*i + j)$



- Plus propre, plus modulaire, plus beau...
- Et plus cher ! problèmes de C, et d'autres !
- Polymorphisme (surcharge des opérateurs,...) pour éviter des conversions de type
- Utilisation de `inline` pour éviter des appels à des fonctions
- Fonctions virtuelles décidées à l'exécution
 - ▶ Surcoût à chaque appel
 - ▶ À n'utiliser que si nécessaire
- Pas de propriétés algébriques (distributivité et Hörner)
 $a*x*x+b*x+c \quad (a*x+b)*x+c$
- Concept de *template* : excellent car compilé (optimisé) pour chaque cas



Exemple de C++ : temporaires à gogo

42

Surcharge des opérateurs et expressions : bien !
 ... mais insertion implicite de temporaires

```
void dosomething(Matrix & A, Matrix & B, Matrix & C, Matrix & D)
{
    A = B * C + D ; // En fait t1 = B * C; t2 = t1 + D; A = t2;
}
// Comparé à :
void dosomething(Matrix & A, Matrix & B, Matrix & C, Matrix & D)
{
    A = B;
    A *= C;
    A += D; // aucun temporaire...
}
```

Mais pas encore de fusion de boucles



- STL : ensemble de *templates* standards



Exemple de Fortran 77

43

- Langage du calcul scientifique
- Simple et suranné (peu structuré, concepts de 1960)
- Conçu pour être optimisable
 - ▶ Aliasing interdit
 - ▶ Modifications algébriques des expressions : légal
 - ▶ Allocation statique (pas de récursion)
 - ▶ Passages par référence
- Pas de structures de données \rightsquigarrow horreurs manuelles les simulant à coup de tableaux d'indices \rightsquigarrow \ performances
 ☹




Mais qui programme en vrai Fortran 77 ???



En général les programmes réel sont faux par rapport à la norme... ☹



- Rétroingénierie du pauvre pour échanger des dimensions :
#define A(a,b) B(b,a)
-  si mélange de C et de Fortran par exemple



Stockage contigu sur les lignes

```
real A(100,100)

do j=1, 100
  do i=1, 100 ! Plus rapide pour le cache
    A(i,j) = 1.0
  enddo
enddo

! À comparer à :
do i=1, 100
  do j=1, 100
    A(i,j) = 1.0
  enddo
enddo
```

- Linéarisation :

$$A(i,j) \equiv A(100*j + i)$$



∃ type de données complex

```
complex a(n), b(n)
real rho, c(n)

do i=1, n
  ! 2 * réelles, 2 / réelles
  a(i) = rho * b(i) / c(i)
enddo

!
do i=1, n
  ! 2 * réelles, 1 / réelles
  a(i) = (rho / c(i)) * b(i)
enddo
```



- Nouvelle mouture de Fortran
- Structuré, records, récursion : maintenance plus aisée
- Moindres performances...
 - Expressions de tableaux \rightsquigarrow surcoût (débit mémoire)

```

real, dimension(100,100):: A, B, C, D
! Potentiellement 3 nids de boucles :
A = f(...)
B = A+A
C = B*B - A + B
! À comparer à (localité cache) :
do j=1, 100
  do i=1, 100
    A(i,j) = f(...)
    B(i,j) = A(i,j)+A(i,j)
    C(i,j) = B(i,j)*B(i,j) - A(i,j) + B(i,j)
  enddo
enddo

```



Motivations pour le choix d'un langage

50

- Compétence des utilisateurs et... culture (locale et relationnelle)!
- Objectifs souvent contradictoires
 - performances (assembleur, C, F77,...)
 - vitesse de développement (shell, tcl/tk, perl, Python)
 - maintenance (Ada, Fortran 95,...)
 - puissance du langage (Java, C++, OCaml,...)
 - portabilité (C, F77, Java/JVM, C#/CLR,...)



- Allocation dynamique (allocate), pointeurs : analyse automatique compliquée
- Passage souvent par copie/restoration (style C++ sans &)


```
call initialize(A(1 :100 :3,1 :100 :2))
```

 \rightsquigarrow perte de temps



Compilation

51

- choix du compilateur (cc acc gcc lcc, f77 f90 f95 g77)
- Choix des options (IBM : typiquement 150 options!!!)
- Choix de l'arithmétique (IEEE ou machine...)



- Option -g
 - ▶ Parfois incompatibles avec les optimisations (CRAY)
 - ▶ Compatibles mais pas à pas difficile (gcc)
- Vérifications statiques : gcc -ansi -Wall ...
- Outils externes de vérification : lint
- Options -u pour Fortran (pas de déclarations implicites !)

- SUN : f77 -fast
- IBM : xlf -O2 -qhot -qarch=pwr2
Higher Order Transformations, pas toujours mieux
- CRAY : f90 -O 3 -O unroll 2
- GNU : g77 -ansi -Wall -Wimplicit -O2

Code assembleur : souvent -S (génère .s au lieu d'un .o)

| | |
|--|---|
| For details on optimization, see the Fortran Programming Guide chapters Performance Profiling, and Performance time, and may also degrade performance when mance and Optimization. | Program's optimization algorithms take more compilation time, and may also degrade performance when applied to too large a fraction of the source program. |
| -O Optimize at the level most likely to give close to the maximum performance for many realistic applications (currently -O3). | Optimization at this level is more likely to improve performance if it is done with profile feedback. See -xprofile=p. |
| -O1 Do only the minimum amount of optimization (peephole). | -fast Optimize for speed of execution using a selection of options. |
| -O2 Do basic local and global optimization. This usually gives minimum code size. -O3 is preferred over -O2 unless -O3 results in excessive compilation time, running out of swap space, or excessively large code size. | Select the combination of options that optimizes for speed of execution without excessive compilation time. This option provides close to the maximum performance for many realistic applications. |
| -O3 Adds global optimizations at the function level. Usually generates larger executable files. | -fast sets the following options: |
| -O4 Adds automatic inlining of functions in the same file. -g suppresses automatic inlining. In general, -O4 results in larger code. | o The -xtarget= native hardware target. If the program is intended to run on a different target than the compilation machine, follow the -fast with the appropriate -xtarget= option. For example: f95 -fast -xtarget=ultra ... |
| -O5 Attempt aggressive optimizations Suitable only for that small fraction of a program that uses the largest fraction of compute time. | o The -O5 optimization level. |

| | |
|--|---|
| o The -libmli option to inline certain math library routines. | o The -xvector=yes option to enable use of the vectorized math library. |
| o The -fsimple=2 option to optimize floating-point operations. | Note that this option is a particular selection of other options that is subject to change from one release of the compiler to another, and between compilers. For details on the options set by -fast, see the Fortran User's Guide. |
| o The -dalign option to allow generation of faster double word load/store instructions. | |
| o The -xlibmopt option to link the optimized math library. | Do not use this option with programs that depend on IEEE standard exception handling; you can get different numerical results, premature program termination, or unexpected SIGFPE signals. |
| o The -depend option to better optimize DO loops. | |
| o The -fns option for possibly faster handling of underflow. | For separate compile and link steps: if you compile with -fast, then be sure to link with -fast. |
| o The -ftrap=common option to set trapping of floating-point exceptions (this is the default for f95). | o The -fsimple=[n] option to select floating-point optimization preferences for f95. Allow the optimizer to make simplifying assumptions concerning floating-point arithmetic. If n is present, it must be 0, 1, or 2. The defaults are: |
| o The -f option to align double and quad data in memory. | |
| o The -pad=common option to improve use of cache. | |

With no -fsimple, f95 uses -fsimple=0
 With only -fsimple, f95 uses -fsimple=1

-fsimple=0
 Permit no simplifying assumptions. Preserve strict IEEE 754 conformance.

-fsimple=1
 Allow conservative simplifications. The resulting code does not strictly conform to IEEE 754, but numeric results of most programs are unchanged.

With -fsimple=1, the optimizer can assume the following:

IEEE 754 default rounding/trapping modes do not change after process initialization.

Computations producing no visible result other than potential floating point exceptions may be deleted.

Computations with Infinity or NaNs as operands need not propagate NaNs to their results; e.g. `rfast` sets -fsimple=2. `x*0` may be replaced by 0.

-ftrap=t
 Computations do not depend on sign of zero.

With -fsimple=1, the optimizer is not allowed to optimize completely without regard to roundoff or exceptions. In particular, a floating-point computation cannot be replaced by one that produces different results with rounding modes held constant at run time.

-fsimple=2
 Permit aggressive floating-point optimizations that may cause many programs to produce different numeric results due to changes in rounding.

For example, -fsimple=2 will permit the optimizer to attempt replacing computations of `x/y` with `x*z`, where `z=1/y` is computed once and saved in a temporary, thereby eliminating costly divide operations.

Even with -fsimple=2, the optimizer still is not permitted to introduce a floating-point exception in a program that otherwise produces none.

-ftrap=t
 Set floating-point trapping mode

—Compilation—



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE



This option sets the IEEE floating-point trapping mode. All turns on all the trapping modes, and will cause trapping of spurious and expected exceptions. Use common instead.

t is a comma-separated list that consists of one or more of the following:

%all, %none, common, [no%]invalid, [no%]overflow, [no%]underflow, [no%]division, [no%]inexact.

The f95 default is -ftrap=common. (Note that the default with f77 is -ftrap=%none.)

-inline=r1
 Request inlining of the specified user-written routines. For example, to optimize by inlining the specified user-written routines named in the list r1. The list is a comma-separated list of functions and subroutines.

Example: Set all traps, except inexact.

-ftrap=%all,no%inexact

The meanings are the same as for the `ieee_flags` function, except that:

If you prefix the name of a routine on the list with `no%`, inlining of that routine is inhibited.

—Compilation—



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE



For example, to enable automatic inlining while doing inlining of a specific routine (`gflub`), use `-O3 -inline=%auto,no%gflub`.

Only routines in the file being compiled are considered. The optimizer decides which of these routines are appropriate for inlining.

A routine is not inlined if any of the following conditions apply, with no warnings:

- o Optimization is less than -O3
- o The routine cannot be found.
- o Inlining it is not profitable or safe.
- o The source is not in the file being compiled.

-xcrossfile.

-autopar
 Enable automatic loop parallelization

Parallelization features require a Sun WorkShop license.

For example, to enable automatic inlining while doing inlining of a specific routine (`gflub`), use `-O3 -inline=%auto,no%gflub`.

Only routines in the file being compiled are considered. The optimizer decides which of these routines are appropriate for inlining.

A routine is not inlined if any of the following conditions apply, with no warnings:

- o Optimization is less than -O3
- o The routine cannot be found.
- o Inlining it is not profitable or safe.
- o The source is not in the file being compiled.

-xcrossfile.

-autopar
 Enable automatic loop parallelization

Parallelization features require a Sun WorkShop license.

To improve performance, also specify the `-stackvar` option when using any of the parallelization options, including -autopar.

Also, -autopar is inappropriate on a single-processor system, and will degrade performance.

For more information, see the Parallelization chapter in the Fortran Programmer's Guide.

Number of Threads: To run a parallelized program in a multithreaded environment, you must set the `PARALLEL` or `OMP_NUM_THREADS` environment variables prior to execution. This tells the runtime system the maximum number of threads the program can create. The default is 1. In general, set `PARALLEL` or `OMP_NUM_THREADS` to the available number of processors on the target platform.

—Compilation—



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE



If -autopar is specified but -explicitpar is not, %none: Ignore all parallelization directives. explicit parallelization directives are ignored.

Sun-style parallelization directives start with `C$PAR` or `!$PAR`. Cray-style parallelization directives start with `CMIC$` or `!MIC$`. Either style can appear in uppercase or lowercase.

If you use -autopar and compile and link in separate steps, then you must link with f95 -autopar as well.

OpenMP directives start with `C$OMP`, `!$OMP`, `*$OMP`, in fixed format, or `!$OMP` in free format.

You can combine OpenMP directives with either Cray or Sun style directives in the same compilation unit. But both Sun and Cray style directives cannot both be active in the same compilation unit. For example:

WorkShop license and generates a list of messages on -mp=sun,openmp and -mp=cray,openmp are permitted, but NOT -mp=sun,cray

-mp=[%none|sun|cray|openmp]
 Specify the style for parallelization directives

The default is sun.

sun: Accept Sun-style MP directives.
 cray: Accept Cray-style MP directives.
 openmp: Accept OpenMP directives

You must also specify -explicitpar to have these directives enable parallelization. Also, -stackvar should be specified with parallelization. For example:

-explicitpar -stackvar -mp=openmp
 -openmp

—Compilation—



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE



| | |
|--|---|
| Enable explicit parallelization with Fortran 95 OpenMP Static local and not initialized, or directives. | <ul style="list-style-type: none"> o In common blocks |
| This option is a macro for the combination of options either one, the arrays or character strings can not be equivalenced. | |
| -mp=openmp -explicitpar -stackvar -D_OPENMP | |
| The Fortran 95 OpenMP directives are described in Fortran User's Guide. | <p>If <code>=p</code> is present, it must be one of the following (no spaces):</p> <ul style="list-style-type: none"> local: Pad local variables common: Pad variables in common blocks local,common: Both local and common padding is done common,local: Both local and common padding is done |
| To run a parallelized program in a multithreaded environment, you must set the PARALLEL or OMP_NUM_THREADS environment variables prior to execution. This tells the runtime system the maximum number of threads the program can create. The default is 1. Without the <code>-pad[=p]</code> option, no padding. In general, set PARALLEL or OMP_NUM_THREADS to the available number of processors on the target platform. | <p>Defaults:</p> <p>Without the <code>-pad[=p]</code> option, no padding.</p> <p>The following are equivalent:</p> <ul style="list-style-type: none"> <code>-mp -pad any.f</code> <code>f95 -pad=local,common any.f</code> <code>f95 -pad=common,local any.f</code> |
| Fortran parallelization features require a Sun Workshop HPC license. | |
| -pad[=p] | |
| Insert padding for efficient use of cache. | Restrictions on <code>-pad=common</code> : |
| This option inserts padding between arrays or character strings if they are: | If <code>-pad=common</code> is specified for a file that references a common block, it must be specified for all |

—Compilation—



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE



| | |
|--|---|
| files that reference that common block. | Parallelize loops chosen automatically by the compiler and explicitly specified by user supplied directives. |
| o With <code>-pad=common</code> specified, declarations of block variables in different program units must be the same except for the names of the variables. | Optimization level is automatically raised to -O3 if it must lower. |
| o Padding is dependent on the setting of <code>-xcache</code> . All files must be compiled with the same <code>-xcache</code> settings when <code>-pad=common</code> is used. | To improve performance, also specify the <code>-stackvar</code> option when using any of the parallelization options, including <code>-autopar</code> . |
| o EQUIVALENCE declarations involving common block variables will cause warning messages that padding has been inhibited by EQUIVALENCE when compiled with <code>-pad=common</code> . | Avoid <code>-parallel</code> if you do your own thread management. See the discussion of <code>-mt</code> in the discussion of <code>-parallelization</code> options like <code>-parallel</code> are intended to produce executable programs to be run on multiprocessor systems. On a single-processor system, parallelization generally degrades performance. |
| -parallel | |
| Enable a combination of automatic and explicit parallelization features. | If you compile and link in separate steps, if <code>-parallel</code> appears on the compile command it must also appear on the link command. |
| This option is a macro for a combination of options: | |
| -autopar -explicitpar -depend | See also the discussion of <code>-autopar</code> . |
| Parallelization features require a Sun Workshop HPC license. | For more information, see the chapter on parallelization in the Fortran Programming Guide. |

—Compilation—



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE



| | |
|---|---|
| Number of Threads: To run a parallelized program in a multithreaded environment, you must set the PARALLEL or OMP_NUM_THREADS environment variables prior to execution. This tells the runtime system the maximum number of threads the program can create. The default is 1. In general, set PARALLEL or OMP_NUM_THREADS to the available number of processors on the target platform. | <p>Analyze loops for reduction in automatic parallelization. To enable parallelization of reduction loops, specify both <code>-reduction</code> and <code>-autopar</code>.</p> <p>A loop that transforms the elements of an array into a single scalar value is called a reduction operation. For example, summing the elements of a vector is a typical reduction operation. Although these operations violate the criteria for parallelizability, the compiler can recognize them and parallelize them as special cases when <code>-reduction</code> is specified. See the Fortran Programming Guide chapter Parallelization for more information on reduction operations recognized by f95.</p> <p>If you specify <code>-reduction</code> without <code>-autopar</code>, the compiler issues a warning.</p> <p>This option also produces counting code in the manner of <code>-p</code>, but invokes a runtime recording mechanism that keeps more extensive statistics and produces a gmon file at normal termination. You can then generate an execution profile using <code>gprof(1)</code>.</p> <p>Example: <code>demo% f95 -autopar -reduction any.f</code></p> |
| -pg | |
| Compile for profiling with <code>gprof</code> . | |
| Prepare the object files for profiling with <code>gprof(1)</code> . This option makes profiles by procedure, showing the number of calls to each procedure and the percent of time used by each procedure. | |
| This option also produces counting code in the manner of <code>-p</code> , but invokes a runtime recording mechanism that keeps more extensive statistics and produces a gmon file at normal termination. You can then generate an execution profile using <code>gprof(1)</code> . | |
| For separate compile and link steps, if you compile with <code>-pg</code> , then link with <code>-pg</code> . | There is always potential for roundoff error with reduction. |
| -reduction | |
| Parallelize reduction operations in loops | If you have a reduction loop to be parallelized, then use <code>-reduction</code> (with <code>-autopar</code> , of course). Do not use |

—Compilation—



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE



| | |
|--|--|
| an explicit pragma, because the explicit pragma prevents reduction for that loop, resulting in wrong answers. | Parallelization options require a Sun Workshop HPC license. |
| -S | |
| Compile and only generate assembly code. | As the compiler detects each explicitly parallelized loop that has dependencies, it issues a warning message but the loop is parallelized. |
| Compile the named programs and leave the assembly language output on corresponding files suffixed <code>.asm</code> (a <code>.o</code> file is created). | Use with <code>-explicitpar</code> and a <code>C\$MIC DOALL</code> parallel pragma. |
| Example: <code>demo% f95 -explicitpar -vpara any.f</code> | |
| -unroll=n | |
| Enable unrolling of DO loops n times where possible. | |
| n is a positive integer. | |
| n = 1, inhibits all loop unrolling | |
| n > 1, this option suggests to the optimizer that unroll loops n times. | Enable the compiler to generate prefetch instructions on those architectures that support prefetch, such as UltraSPARC II (<code>-xarch= v8plus, v9plusa v8plusb, v9, v9a, v9b</code>) |
| If any loops are actually unrolled, then the execution time is larger. | Example values for val are: |
| -vpara | |
| Show verbose parallelization messages | auto enable automatic generation of prefetch instructions |
| | no/auto disable automatic generation |

—Compilation—



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE



| | | | |
|--|---|--|--|
| explicit | enable explicit prefetch | PREFETCH_DIRECTIVES | \$PRAGMA SPARC_PREFETCH_WRITE_MANY (address) |
| no%explicit | disable explicit prefetch directives | -xvector[={yes no}] | |
| yes | equivalent to -xprefetch=auto,explicit | -xvector=yes | permits the compiler to transform math library calls within D0 loops into single calls to the equivalent vector math routines when such transformations are possible. This could result in a performance improvement for loops with large loop counts. |
| no | equivalent to -xprefetch=no%auto, no%explicit | | |
| The first default, when -xprefetch is not specified is: | | | |
| -xprefetch=no | | The default if not specified is -xvector=no. Specifying -xvector is equivalent to -xvector=yes. | |
| The default when just -xprefetch is specified is: | | | |
| -xprefetch=yes | | This option also triggers -depend if -depend is not already specified prior to -xvector on the command line. However, -nodepend following -xvector on the command line will cancel the effect of -xvector. | |
| With -xprefetch=auto, the compiler is free to insert prefetch instructions into the code it generates. This may result in a performance improvement on UltraSPARC II processors. | | | |
| With -xprefetch=explicit, the compiler will recognize the directives: | | | |
| \$PRAGMA SPARC_PREFETCH_READ_ONCE (address) | | | |
| \$PRAGMA SPARC_PREFETCH_READ_MANY (address) | | | |
| \$PRAGMA SPARC_PREFETCH_WRITE_ONCE (address) | | | |
| \$PRAGMA SPARC_PREFETCH_WRITE_MANY (address) | | | |



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE



—Compilation—

Exemples d'outils

55

- Debugging : ddd (affiche les structures) gdb (x) xgdb dbx xdbx. Interface générique en Emacs
- Profiling : prof gprof tprof
permet de sélectionner les zones à optimiser
- Mesures : time rs2hpm sp2flops, fonctions
gettimeofday(),...
- Maintenance :
 - ▶ rcs cvs sccs (garde historique),
 - ▶ Mettre en place des tests de non-régression. Difficile si instabilités numériques



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE

—Outils—



| | |
|--|--|
| Compile for loop profiling by looptool. | C\$PRAGMA C(suba, subz) |
| | C\$PRAGMA SUN UNROLL 2 |
| Prepare object files for the loop profiler looptool. | C\$PRAGMA WEAK FUNK |
| part of the Sun Workshop. This option requires a Workshop license. | C\$PRAGMA SUN OPT=4 |
| | C\$PRAGMA PIPELOOP=5 |
| | C\$PRAGMA SPARC_PREFETCH_READ_ONCE (name) |
| If you compile and link in separate steps, and you use the -Zlp option, you must use the -Zlp option to link the object files with -Zlp, then be sure to link with -Zlp. | C\$PRAGMA SPARC_PREFETCH_READ_MANY (name) |
| | C\$PRAGMA SPARC_PREFETCH_WRITE_ONCE (name) |
| | C\$PRAGMA SPARC_PREFETCH_WRITE_MANY (name) |
| If you compile one subprogram with -Zlp, you need not compile all subprograms of that program with -Zlp. However, you get loop information only for the files compiled with -Zlp, and no indication that the program includes other files. | General Directives: f95 recognizes Sun-style parallel compiler directives starting with C\$PAR, !SPAR, or *SPAR, and many uppercase or lowercase is allowed. Examples: |
| DIRECTIVES | C\$PAR DOALL |
| General Directives: f95 allows general compiler directives starting with C\$PRAGMA, !SPRAGMA, or *SPRAGMA and many uppercase or lowercase is allowed. Examples: | C\$PAR DOSERIAL |
| | C\$PAR DOSEMP |
| | C\$PAR DOCOMMON |
| | Cray-style parallelization directives are also recognized. The directive sentinel is CMIC\$ or !MIC\$ |



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE




—Compilation—

GPROF

56

Call Graph Profile

- Instrumentation des codes C, F77 : option `-pg`
- Exécution du programme : génère un `gmon.out`
- Post analyse : `gprof executable gmon.out`
-  Surcoût à l'exécution important (facteur 2). Modifie l'exécution et le code :
 - ▶ Registres
 - ▶ Pipeline
 - ▶ Cache

Mais donne néanmoins les grandes lignes



DÉPARTEMENT INFORMATIQUE — ENST BRETAGNE

—Outils—



- Évaluation *statistique* du temps d'exécution pour chaque routine par interruptions régulières
- Comptage des appels pour chaque routine
 - ▶ Ses appelants
 - ▶ Ses appelés
- Exemple de surprises dans l'application PIPS :
 - ▶ 25% du temps passé dans `malloc` et `free`. Attention, certains `malloc` peuvent traverser toute la mémoire pour trouver un bloc vide... Meilleur `malloc` ou version personnelle ?
 - ▶ 20% du temps à calculer des fonctions de hashage



Exemple

59

```
> executable
> gprof -F main -E mcount executable gmon.out > resultat
> cat resultat
Each sample counts as 0.01 seconds.
% cumulative self      self      total
time  seconds seconds  calls ms/call ms/call name
28.29    1.83    1.83
4.03    2.09    0.26 108078  0.00    0.00 _doprnt
3.72    2.33    0.24 136580  0.00    0.00 malloc
3.57    2.56    0.23  64222  0.00    0.01 gen_trav_leaf
3.57    2.79    0.23 424090  0.00    0.00 strcmp
3.57    3.02    0.23  87643  0.00    0.00 .rem
3.41    3.24    0.22 125213  0.00    0.00 insert
3.41    3.46    0.22 125213  0.00    0.01 free
2.95    3.65    0.19  35981  0.01    0.02 gen_trav_obj
2.79    3.83    0.18   154    1.17   1.18 Matrix_Alloc
2.33    3.98    0.15 123523  0.00    0.00 delete[...]
[...]
```



- Fonctions classées par % du temps
 - ▶ Temps direct dans la fonction
 - ▶ Temps dans ses appelés
 - ▶ Nombre d'appels
- Appelants au dessus
 - ▶ Nombre d'appel à la fonction
 - ▶ Impact sur le temps direct et indirect
- Appelés au dessous
 - ▶ Par des appels de cet appelé
 - ▶ Temps direct dans l'appelé
 - ▶ Temps indirect (appelés des appelés)




Exemple

60

| index | % time | self | children | called | name |
|-------|--------|------|----------|---------------|----------------------|
| | | | | | <spontaneous> |
| [1] | 91.6 | 0.00 | 4.23 | | start [1] |
| | | 0.00 | 4.22 | 1/1 | main [2] |
| | | 0.01 | 0.00 | 1/1 | exit [404] |
| | | 0.00 | 0.00 | 1/2 | on_exit [1497] |
| [...] | | | | | |
| | | 0.01 | 0.00 | 5098/136580 | strdup [252] |
| | | 0.01 | 0.01 | 8284/136580 | contrainte_new [210] |
| | | 0.06 | 0.03 | 31961/136580 | alloc [132] |
| | | 0.15 | 0.07 | 84662/136580 | vect_new [74] |
| [51] | 7.7 | 0.24 | 0.11 | 136580 | malloc [51] |
| | | 0.03 | 0.02 | 112/112 | morecore [173] |
| | | 0.04 | 0.01 | 32319/123523 | delete [87] |
| | | 0.02 | 0.00 | 104261/104261 | demote [270] |



- Tous les processeurs modernes ont des compteurs pour faire des statistiques
- Souvent globaux à toute la machine et donc à tous les processus... 
- Exemple du SP2
 - ▶ rs2hpm (*RIOS 2 Hardware Performance Monitor*)
 - ▶ Compteurs sur le processeurs
 - ▶ Opérations flottantes, cache,... totales (pas par processus...)
 - ▶ Mesure distribuée et statistique : sp2flops



- Hiérarchie mémoire
 - ▶ Registres
 - ▶ Cache
 - ▶ Mémoire principale
 - ▶ Mémoire virtuelle
- Importance des boucles
 - ▶ Transformations pour améliorer la localité
 - ▶ Utilisation des registres




```
> rs2hpm executable
POWER2 Performance Statistics with sampled overall measurements
RS2HPM Version 2.0 - 960605      Author: Jussi.Maki@csc.fi
This program shows all processes CPU usage during the execution
CPU seconds      : 49.8762    CP executing   : 3326743932
Elapsed seconds  : 51.3962
FPU0 results/sec : 13.70M    F.P. in Math0 : 683472924
FPU1 results/sec : 1.46M     F.P. in Math1 : 73032915
F.P. add ops/sec : 8.93M     F.P. add      : 445445847
F.P. mul ops/sec : 0.00M     F.P. mul      : 222
F.P. ma ops/sec  : 5.81M     F.P. ma       : 290021229
MFLOPS ratio     : 14.75M    F.P. math ops : 735467298
Fixed instr/sec E0 : 20.99M   Fixed instr E0 : 1046907048
Fixed instr/sec E1 : 15.46M   Fixed instr E1 : 771189993
ICU instr/sec     : 4.64M     ICU instr.    : 231392898
Integer MIPS      : 41.09     Total instr.   : 2049489939
I Cache reloads/sec : 6.34k
D Cache reloads/sec : 1231.76k
D Cache storebacks/sec : 217.87k
D Cache misses/sec : 1217.57k
Total TLB misses/sec : 846.49k
```



- Temps majoritairement passé là !
- Nomreuses transformations de boucles possibles
 - ▶ Inversion
 - ▶ Déroulage (*loop unrolling*)
 - ▶ Blockage (*tiling*)
 - ▶ Software pipelining
 - ▶ ...
- attention à la légalité ! Respecter les dépendances (causalité), voire la non-associativité du flottant



- Motivation : localité

- Selon langage 

- Inverser i et j

```
do i = 1, n
  do j = 1, n
    A(i,j) = B(i,j) + C(i,j) ! Mauvais
  enddo
enddo
```

! Devient légalement :

```
do j = 1, n
  do i = 1, n
    A(i,j) = B(i,j) + C(i,j) ! Bon pour cache et mémoire
  enddo
enddo
```



Software pipelining

67

- Écartement des dépendances...
- Mais aussi plus de registres nécessaires...

```
do i = 1, n
  load[i]
  oper[i]
  store[i]
enddo
! devient : prélude
load[1]
load[2]
oper[1]
do i = 1, n - 2 ! Le cœur
  load[i + 2]
  oper[i + 1]
  store[i]
enddo
oper[n] ! Postlude
store[n - 1]
```



- Meilleure utilisation des registres et unités de calcul en //
- Diminution du surcoût de contrôle

```
do i = 1, n
  a(i) = c*b(i) + d
enddo
! unroll 4
do i = 1, n, 4
  a(i)   = c*b(i)   + d
  a(i+1) = c*b(i+1) + d
  a(i+2) = c*b(i+2) + d
  a(i+3) = c*b(i+3) + d
enddo
do i = 1 + 4*(n/4), n
  a(i) = c*b(i) + d
enddo
```



Software pipelining

68

store[n]



- Blocs d'opérations
- Souvent ajustés au cache

```
do j = 1, n
  do i = 1, n
    a(i,j) = ...
  enddo
enddo
! devient :
do j = 1, n, b1
  do i = 1, n, b2
    do j1 = j, min(j + b1, n)
      do i1 = i, min(i + b2, n)
        a(i + i1, j + j1) = ...
      enddo
    enddo
  enddo
enddo
```



Invariant code motion

71

- Pas la peine de calculer des choses inutilement
- Extraction des invariants de boucles

```
do i = 1, n
  a(i) = rho/norm*a(i)
enddo
! devient :
f = rho/norm
do i = 1, n
  a(i) = f*a(i)
enddo
```



Souvent déjà réalisé dans les bibliothèques spécialisées



Restructurations de code

72

- Borné par l'imagination ☺

```
do i = 1, n
  if (x < y) then
    a(i) = b(i)
  else
    a(i) = -b(i)
  endif
enddo
! devient :
if (x < y) then
  do i = 1, n
    a(i) = b(i)
  enddo
else
  do i = 1, n
    a(i) = -b(i)
  enddo
endif
```



- Propagation de constantes

```
const double pi = 3.14.....;
[...]
c = 2*pi*r;
```

- Évaluation à la compilation (évaluation partielle)

```
c = 2*pi*r;
```

```
//
c = 6.28....*r;
```



- Ordonnancement des instructions

- ▶ Selon l'architecture
- ▶ Complexe vu les contraintes nombreuses
 - Nombre de registre
 - Unités fonctionnelles disponibles
 - Contraintes de dépendances
 - Devient flou sur les processeurs modernes (x86)
 - ...



- Détection des sous expressions communes

```
do i=1, n, 2
  a(i) = b(i-1)+ b(i)+b(i+1)
  a(i+1) = b(i)+b(i+1)+b(i+2)
enddo
! devient :
do i=1, n, 2
  c = b(i)+b(i+1)
  a(i) = b(i-1)+c
  a(i+1) = c+b(i+2)
enddo
```

Couplable à du déroulage ou pipeline ici



- Utiliser un codage efficace
- Exploiter le maximum de bits du processeur
- Pré-mâcher les données si possible
- Utiliser le binaire : $\div (32 = 2^5) \equiv >> 5$,
 $\times (256 = 2^8) \equiv << 8, \dots$
- Compromis efficacité
(compactage/décompactage)-densité



- Tableau de structures ou structure de tableaux ?
 - ▶ Tableau de structures : localité mémoire au niveau de la structure
 - ▶ Structure de tableaux : localité mémoire au niveau d'un tableau
- Revoir le stockage : données accédées par un arbre \rightsquigarrow éclatement de la structure pour avoir un arbre compact avec des pointeurs vers une zone de données
- Problème si mesures de performances mènent à revoir toutes les structures de données... ☹



Applications codage binaire : multispin-coding

79

Exemple : bibliothèques de cassage de codes cryptographiques (détection mots de passe faibles avec John the Ripper), traitement d'image, traitement du signal, codage, optimisation de programmes...



- Exploitation du parallélisme en bits
- Ranger plusieurs petites données par mot machine
- 4 opérations sur 64 bits/cycle \equiv 256 opérations sur 1 bit/cycle !
- Opérations binaire style \wedge , $\&$, $|$, \sim sans problème
- Jeux d'instructions :
 - 1 Alpha 21164 à 600 MHz \equiv 76,8 GIPS 1 bit, 9,6 GIPS 8 bits
 - 1 Pentium 4 SSE3 à 4 GHz : 2 opérations 128 bits/cycle \equiv 1 TIPS (10^{12} opérations par secondes) 1 bit
- Idée : plutôt que de résoudre 1 problème à la fois, éclate problème en binaire pour calculer 256 tranches de problèmes binaires à la fois



Application utilisant des additions 9 et 6 bits

80

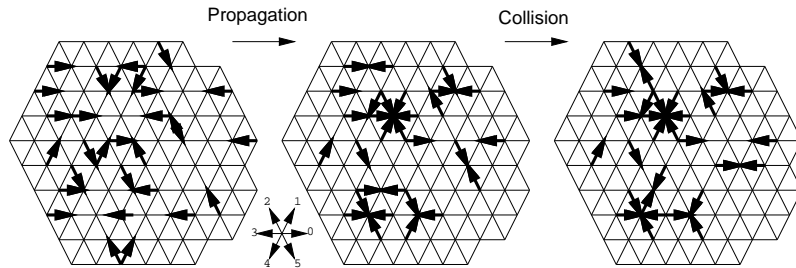
- Compactage dans 32 bits `a_xxs_yys` :

| quality | q_a | 0 | q_x | 0 | q_y |
|---------|-----|---|-----|---|-----|
| 8 | 6 | 1 | 8 | 1 | 8 |

- `q_a` sur 6 bits, `q_x` et `q_y` sur 8 bits
- Opérations sur `q_x` et `q_y` sur 9 bits \rightsquigarrow stockage sur 9 bits aussi (évite l'extraction)
- Garde des 0 délimiteurs absorbant les retenues (*masque*)
- Besoin de tester $(q_x, q_y) \in [-128, 127]^2$:
 - ▶ Changement repère (biais +128) $\rightsquigarrow (q'_x, q'_y) \in [0, 255]^2$
 - ▶ Test de `a_xxs_yys_&_((1<<8)_+_1(1<<18))_=_0` : 1 instruction !



- Sites contenant des particules se déplaçant quantiquement
- Interactions entre particules sur chaque site



- Tableau de sites contenant 1 bit de présence d'1 particule allant dans 1 direction
- Symétrie triangulaire



```

14 double_ad_=(a&d&~(b_|_c_|_e_|_f));
   double_be_=(b&e&~(a_|_c_|_d_|_f));
16 double_cf_=(c&f&~(a_|_b_|_d_|_e));
   ____/*_The_exchange_of_particles_:_*/
18 change_ad_=_triple_|_double_ad_|_(r&double_be)|_(nr&double_cf);
   change_be_=_triple_|_double_be_|_(r&double_cf)|_(nr&double_ad);
20 change_cf_=_triple_|_double_cf_|_(r&double_ad)|_(nr&double_be);
   ____/*_Where_there_is_blowing_,_collisions_are_no_longer_valuable_:
22 bl_=_blow[N_DIR];
   s_&=_~bl;
24 ns_&=_~bl;
   ____/*_Effects_the_exchange_where_it_has_to_do_according_the_solid
26 lattice[RIGHT]_=_(((a^change_ad)&ns)|_(d&s))
   ____|_bl&blow[RIGHT];
28 lattice[TOP_RIGHT]_=_(((b^change_be)&ns)|_(e&s))

```



- Compactage de 32 ou 64 sites/int par direction

```

1 a_=_lattice[RIGHT];
2 b_=_lattice[TOP_RIGHT];
   c_=_lattice[TOP_LEFT];_//_Particules_qui_montent_à_gauche
4 d_=_lattice[LEFT];_//_Particules_qui_vont_à_gauche
   e_=_lattice[BOTTOM_LEFT];
6 f_=_lattice[BOTTOM_RIGHT];
   s_=_solid;_//_Une_condition_limite
8 ns_=_~s;
   r_=_lattice[RANDOM];_//_Un_peu_d'aléa
10 nr_=_~r;
   ____/*_A_triplet_:_*/
12 triple_=(a^b)&(b^c)&(c^d)&(d^e)&(e^f);
   ____/*_Doubles_:_*/

```



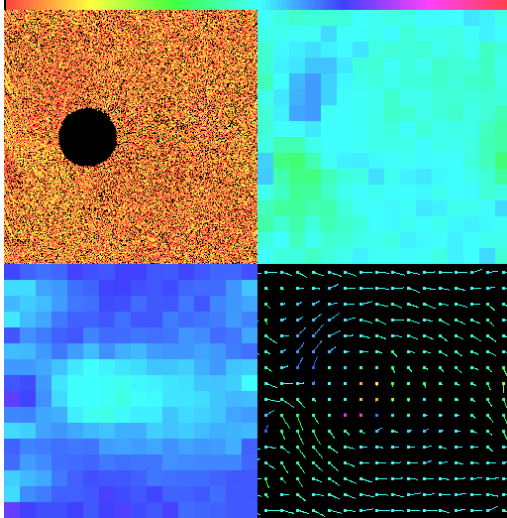
```

   ____|_bl&blow[TOP_RIGHT];
30 lattice[TOP_LEFT]_=_(((c^change_cf)&ns)|_(f&s))
   ____|_bl&blow[TOP_LEFT];
32 lattice[LEFT]_=_(((d^change_ad)&ns)|_(a&s))
   ____|_bl&blow[LEFT];
34 lattice[BOTTOM_LEFT]_=_(((e^change_be)&ns)|_(b&s))
   ____|_bl&blow[BOTTOM_LEFT];
36 lattice[BOTTOM_RIGHT]_=_(((f^change_cf)&ns)|_(c&s))
   ____|_bl&blow[BOTTOM_RIGHT];

```



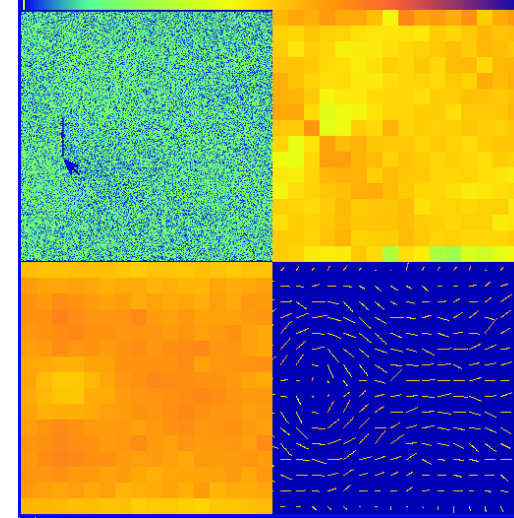
- Cylindre



- Passage en SSE2 : 2×128 sites traités par cycles
- Utilisation possible de cartes graphiques
- Autre méthode par table de collision mais problème débit mémoire/cache (cf scatter/gather des processeurs vectoriels)



- Instabilités de Von KARMAN



- Besoin de représenter des valeurs plus « continues » et plus de dynamique que types entiers
- Sous-ensemble de $\mathbb{D}(\subset \mathbb{R})$
- Représentation souvent au format IEEE 754-1985

$$f = (-1)^S \times \mathcal{M} \times 2^E$$

- ▶ S : bit de signe
- ▶ \mathcal{M} : mantisse (entier positif)
- ▶ E : exposant
- Plusieurs tailles de flottants
 - ▶ Simple précision (**float**)
 - ▶ Double précision (**double**)



► Précision étendue (**long_double**)

• Codage en mémoire

| Taille en bit | | | | |
|---------------|-----------|-------|-----------|-----------|
| Format | Totale | Signe | Exposant | Mantisse |
| Simple | 32 | 1 | 8 | 24 |
| Double | 64 | 1 | 11 | 53 |
| Étendu | ≥ 80 | 1 | ≥ 15 | ≥ 64 |

► Exposant codé en biaisé : $\mathcal{E}_{\text{réel}} = \mathcal{E}_{\text{stocké}} - \mathcal{E}_{\text{biais}}$

► Tri lexicographique sur bits compatible avec tri flottant !
Même si on ne gère pas le flottant on sait trier ☺



► NaN (*Not a Number*) pour $\frac{0}{0}$ ou $\sqrt{-1}$

Seul cas où $x \neq x$ lorsque x vaut NaN. Existe en signé et en version déclenchant exception (SNaN)

~~~ Peuvent simplifier programmation et calcul si bien géré (éviter tests cas particuliers...)

- $\exists$  Nombreux choix d'arrondi (plus proche, +, -, vers 0,...)
- <http://grouper.ieee.org/groups/754> En cours de révision  
[http://en.wikipedia.org/wiki/IEEE\\_754r](http://en.wikipedia.org/wiki/IEEE_754r) si vous voulez participer ☺

⚠ Plein de subtilités ⚠ !



| Format | Minimum en dénormalisé      | Minimum en normalisé        | Maximum fini               | $2^{-N}$ (grain)           | Chiffres significatifs |
|--------|-----------------------------|-----------------------------|----------------------------|----------------------------|------------------------|
| Simple | $1,4 \cdot 10^{-45}$        | $1,2 \cdot 10^{-38}$        | $3,4 \cdot 10^{38}$        | $5,96 \cdot 10^{-8}$       | 6–9                    |
| Double | $4,9 \cdot 10^{-324}$       | $2,2 \cdot 10^{-308}$       | $1,8 \cdot 10^{308}$       | $1,11 \cdot 10^{-16}$      | 15–17                  |
| Étendu | $\leq 3,6 \cdot 10^{-4951}$ | $\leq 3,4 \cdot 10^{-4932}$ | $\geq 1,2 \cdot 10^{4932}$ | $\leq 5,42 \cdot 10^{-20}$ | $\geq 18-21$           |

Nombreux paramètres définis dans `<float.h>`

- Possibilité de déclencher exceptions (division par 0, débordement,...) (fonction exécutée sur événement)
- Rajout de quantités symboliques (déclarées dans `<math.h>`)
  - $+0$  et  $-0$ . Néanmoins  $+0 = -0$  est vrai
  - $+\infty$  et  $-\infty$  (par exemple  $\frac{1}{+0}$  et  $\frac{1}{-0}$ ) (HUGE\_VAL...)



« What Every Computer Scientist Should Know About Floating-Point Arithmetic », David GOLDBERG, Computing Surveys, mars 1991, ACM

- Nombres flottants  $\equiv$  pale imitation de  $\mathbb{R}$  et même de  $\mathbb{D}$  ☹
- Nombreuses approximations
- ⚠ ⚠ ⚠ Propriétés algébriques de  $\mathbb{R}$  non vérifiées : non associatif


$$(1 \oplus 10^{40}) \ominus 10^{40} = 0$$

$$1 \oplus (10^{40} \ominus 10^{40}) = 1$$

- Changement des résultats possibles selon optimisations...

☹



- Notion d'équivalence séquentielle de programme entre différentes versions
  - ▶ Forte : le programme obtenu donne le même résultat
  - ▶ Faible : le programme obtenu donne le même résultat modulo les problèmes numériques précédents
- Choisir programmation prenant en compte ces caractéristiques
  - ▶ T<sub>E</sub>X écrit en virgule fixe 16+16 bits pour portabilité multi-plateforme ☺
  - ▶ Compromis entre performances & précision
-  Compilateurs devraient en tenir compte (pas optimisations sauvages)



## Algorithme de sommation de flottants

95

- Solution triviale

```

1 double_x[N];
2 double_s=0;
  for (int i=0; i<N; i++)
4   s+=x[i];
  
```

$$s = \sum_{i=0}^{N-1} x_i(1 + \delta_i)$$

avec  $\delta_i < (n - i)\epsilon$



- Exemples
  - ▶  $(x - y)(x + y)$  plus précis (voire plus rapide) que  $x^2 - y^2$
  - ▶ Algorithme somme de flottants



## Algorithme de sommation de flottants

96

- Version KAHAN

```

1 double_x[N];
2 double_s=x[0];
   double_c=0; // Erreur d'arrondi
4 for (int i=1; i<N; i++){
   double_y=x[i]-c; // Compense erreur précédente
6   double_t=s+y; // Nouvelle somme
   c=(t-s)-y; // Estime l'erreur d'arrondi
8   s=t;
  }
  
```

$$s = \sum_{i=0}^{N-1} x_i(1 + \delta_i) + \mathcal{O}(N\epsilon^2 \sum_{i=0}^{N-1} |x_i|) \quad \text{avec} \quad \delta_i \leq 2\epsilon$$

Optimisations incontrôlées du programme fait des ravages



ici... et revient à algorithme trivial ! ☹



## Pourquoi des nombres flottants dénormalisés ?

99

- Soustraction de 2 nombres normalisés, par exemple en simple précision

$$a = 2,05 \cdot 10^{-37}$$

$$b = 2,03 \cdot 10^{-37}$$

$$a - b = 2 \cdot 10^{-39}$$

$$a \ominus b = 0$$

$$a \neq b$$

Seule solution car  $\mathcal{M}$  ne peut pas commencer par 1... ☹

- Idée : rajouter mode dénormalisé pour très petits nombres où  $\mathcal{M}$  peut ne pas commencer par un 1
- Permet *underflow* (dépassement de capacité par le bas)



- Possible de représenter des nombres de plusieurs manières

$$\mathcal{M}' = 2^{-a} \mathcal{M}$$

$$\mathcal{E}' = \mathcal{E} + a$$

- Problème des codages redondants : comparaisons difficiles ☹
- Idée 1 : normaliser ! Exemple : choisir le  $\mathcal{M}$  le plus grand
- Idée 2
  - ▶  $\forall \mathcal{M} \neq 0$  : commence toujours par 1 en binaire
  - ▶  $\rightsquigarrow$  Ne pas stocker ce 1 évident...



$\rightsquigarrow$  Flottant normalisé : gagne 1 bit de précision ☺



## Pourquoi des nombres flottants dénormalisés ?

100

progressif

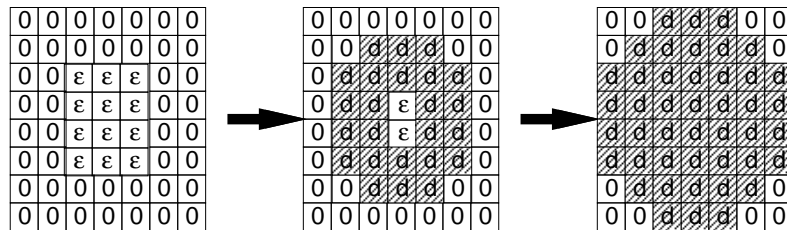
-  Si flottant dénormalisé non géré directement en matériel : génère exception et calculs terminés par... système d'exploitation  $\rightsquigarrow$  performances  $\searrow \searrow$  ☹
-  Parfois autorisation exception = : suppression pipeline (DEC Alpha) ☹



- Exception IEEE-754 générée lors de la dénormalisation
- Typiquement un programme de différences finie avec un domaine avec de petites valeurs  $\epsilon$  entouré de 0 :

$$x_{i,j}^n = \frac{x_{i-1,j}^v + x_{i+1,j}^v + x_{i,j-1}^v + x_{i,j+1}^v}{4}$$

~> Propagation d'ondes de dénormalisation



- Besoin de calculer  $f(i)$ ,  $f$  long et petit nombre de
- Réaliser un cache logiciel :

$$\forall i, F[i] = f(i)$$

- Utiliser partout  $F[i]$  dans le programme au lieu de  $f[i]$
- Éventuellement construction à la volée de  $F[i]$
- Cas numérique : possibilité d'interpoler des  $F[i]$  si pas trop regardant sur les valeurs



- À pleurer sur machine parallèle si ordonnancement statique : tous les processeurs attendent le plus lent ! ☹
- Rajout d'un biais pour ne plus être au voisinage de 0. Mais perte de dynamique... Compromis

$$y_{i,j}^n = x_{i,j}^n + b \quad (1)$$

$$y_{i,j}^n = \frac{y_{i-1,j}^v + y_{i+1,j}^v + y_{i,j-1}^v + y_{i,j+1}^v}{4} \quad (2)$$



Exemple (simplifié) : compter des mots dans un texte

```
for (i = 0; i < taille - 1; i++)
    if (c[i] != ' ' && c[i + 1] == ' ')
        nombre_mots++;
```

- Code tabulé :

```
for (i = 0; i < taille - 1; i++)
    nombre_mots += table[(c[i] << 8) + c[i + 1]];
Code à optimiser classiquement
```

- Construction d'une table pour remplacer le test :

```
for (c1 = 0; c1 <= 127; c1++)
    for (c2 = 0; c2 <= 127; c2++)
        table[(c1 << 8) + c2] = (c1 != ' ' && c2 == ' ') ? 1 : 0;
```



- Importance des optimisations dans le coût d'un produit final
- Estimer la complexité et le temps minimal
- Optimiser le strict nécessaire (pas la peine de faire du pipeline logiciel ou autre si le compilateur le fait déjà)
- Employer des algorithmes & structures de données adaptés
- Ne pas réinventer la roue
  - Utilisation de bibliothèques spécialisées
  - Bibliographie sur les algorithmes classiques
- Importance des boucles
- Connaître sa machine, ses outils & son compilateur !



- Demander de l'aide à des spécialistes



## Liste des transparents

- 0 Le trez jolly titre gracieux
- 0 Optimisation de Code Séquentiel
- 1 But
- 0 Introduction
- 2 Stratégie
- 4 Portabilité ?
- 5 Programmes étalons
- 8 Plan
- 9 Architecture des processeurs
- 8 Architecture des Processeurs
- 11 Pipeline
- 13 Opérations
- 14 Opérations CISC
- 16 Branchements
- 17 Parallélisme intraprocasseur
- 19 Instructions SIMD
- 21 Hiérarchie mémoire
- 25 Registres
- 27 Cache
- 26 Cache
- 30 Mémoire virtuelle

## 29 Mémoire virtuelle

- 32 Mémoire virtuelle Unix
- 33 Choix et outils pour l'optimisation

## 32 Langages

- 34 Exemple de C : pointeurs
- 35 Exemple de C : aliasing
- 37 Exemple de C : promotions
- 39 Exemple de C : stockage des tableaux
- 40 Exemple de C++
- 42 Exemple de C++ : temporaires à gogo
- 43 Exemple de Fortran 77
- 45 Exemple de Fortran : stockage des tableaux
- 47 Exemple de Fortran 77 : complex
- 48 Exemple de Fortran 95
- 50 Motivations pour le choix d'un langage
- 51 Compilation

## 50 Compilation

- 52 Debugage
- 53 Options d'optimisations, exemples
- 54 Options optimisation Sun Forte 6 F95
- 55 Exemples d'outils

## 54 Outils

- 56 GPROF
- 57 Résultats
- 58 Interprétation des résultats

- 59 Exemple
- 61 Mesures matérielle des performances
- 62 Exemple
- 63 Programmation pour l'optimisation
- 62 Programmation
- 64 Importance des boucles
- 65 Inversion de boucles
- 66 Déroulage de boucle
- 67 Software pipelining
- 69 Blocage (*tiling*)
- 71 Invariant code motion
- 72 Restructurations de code
- 73 Optimisations Scalaires classiques
- 76 Codage de l'information

## 75 Codage de l'information

- 77 Structures de données
- 78 Applications codage binaire : multispin-coding
- 80 Application utilisant des additions 9 et 6 bits
- 81 Gaz sur réseau
- 88 Nombres flottants
- 92 Nombres flottants  $\neq$  réels !
- 95 Algorithme de sommation de flottants
- 98 Nombres flottants normalisés
- 99 Pourquoi des nombres flottants dénormalisés ?
- 101 Dénormalisation flottante
- 103 Cachage/tabulation de valeurs/calculs
- 104 Cachage de calculs : comptage des mots
- 105 Conclusion