T1 Architecture des processeurs

Optimisations logicielles

Alain Mérigot

M2 SETI 2022-23

Introduction

L'architecture permet de connaitre les moyens d'améliorer les programmes.
Les principaux facteurs à considérer sont :

- les délais de branchement
- la hiérarchie mémoire et la bonne utilisation du cache
- les dépendances de données

Préambule

Make it right before to make it fast

The three rules of optimization:

Rule #1: Don't

Rule #2: Don't ... yet

Rule #3: Profile before optimizing

- N'optimiser qu'un programme déjà fonctionnel
- dans ses parties critiques (utiliser un *profiler* pour connaître les fonctions les plus coûteuses)
- en se rappelant qu'on peut plus gagner avec des optimisations de haut niveau (structures de données et algorithmes adaptés) que de bas niveau

L'optimisation a souvent tendance à rendre le code plus obscur et plus difficile à maintenir et doit donc être faite à bon escient.

Réduction des pénalités de branchement

Dans les processeurs actuels, prédiction du comportement des branchements.

En cas d'erreur de prédiction, pénalité de $\approx$ 20 cycles.

Dans beaucoup de situations (notamment les boucles et surtout les boucles `for`), la prédiction est bonne

Mais difficulté en cas de branchement non prédictible.

Concerne les branchements sous leurs différentes formes : `if()`, `switch()`, `(?:)`, `&&`, ||.

Réduction des pénalités de branchement (cont.)

Remplacer des branchements par une table de correspondance (*look-up table*)

```
void f(float a[N]; int b[N]){
  for(int i =0; i<N; i++)
    a[i] += (b[i]>=0 ? +1.5f : -2.2f) ;
     // un branchement non prévisible
}
```

```
void f(float a[N]; int b[N]){
  static float corr[2]={1.5f, -2.2f};
  int isneg;
  for(int i =0; i<N; i++){
    isneg=(b[i]<0);//0 si b>=0, 1 sinon
    a[i] += corr[isneg]
  }
}
```

Particulièrement efficace pour des branchements multiples.

```
switch(b&0x07){
  case 0: a+= 4; break ;
  case 3: a+= 5; break ;
  case 4: a+= 6; break ;
  case 6: a+= 2; break ;
  default:a+=7;
}
// realisé sous forme de
//  plusieurs branchements
//  imbriqués
```

```
switch(b&0x07){
  case 0: a+= 4; break ;
  case 1:
  case 2: a+= 7; break;
  case 3: a+= 5; break ;
  case 4: a+= 6; break ;
  case 5: a+= 7; break;
  case 6: a+= 2; break ;
  case 7: a+= 7;
}
// Meilleur. Tous les cas du
// switch sont couverts. Se
// fait avec un brcht indexé
```

```
static int look[8]={4,7,7,5,
                    6,7,2,7};
a += look[(b&0x07)];
// Encore meilleur.
// Aucune rupture de séquence
// 10x plus rapide sur la
// plupart des architectures
```

Réduction des pénalités de branchement (cont.)

Beaucoup de tests simples peuvent être supprimés

```
#define max(a,b) ((a)>(b) ? (a) : (b))
```

```
static int max(int a, int b) {
  b = a-b;
  return  a - (b & (b>>31));
}
```

```
#define abs(a) ((a)<0 ? -(a) : (a))
```

```
#define abs(a) ((a)-(((a)+(a))&(a)>>31)))
```

Ces « astuces » peuvent être intéressantes,
mais cela dépend beaucoup du processeur, du compilateur, du caractère prédictible des données, etc.
Mesurer les temps préalablement.

```
c=(cond ? a : b) ;
```

```
int mask=(cond==0)-1; //0 si !cond,-1 sinon
c=(mask&a)|(~mask&b);
```

Intéressant sauf si le processeur possède des mouvements conditionnels.

Réduction des pénalités de branchement (cont.)

Calcul du nombre d'éléments négatifs d'un tableau

```
unsigned nb_neg_v1(int t[]){
  unsigned res=0;
  for(int i=0; i<N; i++)
    if (t[i]<0) res++;
  return res;
}
```

```
unsigned nb_neg_v2(int t[]){
  unsigned res=0;
  for(int i=0; i<N; i++)
    res += (t[i]<0);
  return res;
}
```

```
unsigned nb_neg_v3(int t[]){
  unsigned res=0;
  for(int i=0; i<N; i++)
    res -= (t[i]>>31);
  return res;
}
```

Cycles par itération sur un tableau aléatoire avec différents pourcentages de nombres négatifs et pour différents niveaux d'optimisation (intel core i7-6000 (*skylake*)).

	nb_neg_v1			nb_neg_v2			nb_neg_v3		
	50%	25%	0	50%	25%	0	50%	25%	0
`gcc -O0`	22.6	13.3	8.16	6.92	6.87	7.05	6.83	6.97	7.01
`gcc -O1`	14.6	10.1	1.49	1.44	1.42	1.45	1.46	1.41	1.45
`gcc -O3`	11.1	8.19	3.48	0.52	0.54	0.52	0.53	0.53	0.52

L'optimisation `-O3` déroule les boucles et utilise la vectorisation simd

Pour la version `v1` (avec `if`), notez l'effet des mauvaises prédictions (avec 50% de négatifs aléatoirement répartis) et l'absence de vectorisation simd en -O3.
`v2` at `v3` sont quasiment identiques. Préférer `v2` qui est plus lisible.

Prise en compte des caches

Cache : trois types de défauts (les trois “C”) :

Obligatoires (*compulsory*) Lors de la première utilisation d’une information

Capacité (*capacity*) Taille des données trop importante par rapport à la taille du cache pour bien exploiter la localité temporelle

Conflits (*conflicts*) Données allants dans la même ligne de cache

Amélioration : les trois règles en “R”

Réarranger Réorganiser pour améliorer la localité spatiale

Réduire Moins d’information = moins de défaut de caches

Réutiliser Améliorer la localité temporelle

Prise en compte des caches (cont.)

Réorganiser le code :

- Garder proches des fonctions utilisées simultanément
  - lors de la déclaration dans le source
  - par l'ordre des fichiers dans le *makefile*
  - en utilisant `__attribute__(section([nom])` et édition de lien (*linker-scripts* de `ld`)
- Se méfier de l'expansion de code des fonctions `inline`, des macros, des déroulages, etc. Peut parfois ralentir l'exécution au lieu de l'accélérer.
- Etudier le code généré, regarder les emplacements en mémoire des informations (*link-map*)

Disposition des données

Alignement : une donnée de taille 2^k est à une adresse multiple de 2^k.
Permet d'assurer qu'une donnée est toujours stockée dans une même rangée mémoire ou ligne de cache.
Il est généralement préférable de générer du code aligné[1]
Mais crée certaines contraintes et peut conduire à une perte mémoire.

Un `struct` aligné

- est toujours aligné sur la taille de son plus grand champ,
- a une taille multiple de ce champ,
- et chaque champ est aligné.

```
struct {
  char c;  // + 7o de remplissage
  double d;// 8 octets
  short s; // +6 pour que taille (struct)
           // soit multiple de 8
} s[100] ;
// s fait 24*100 octets
```

```
struct {
  double d;
  short s;
  char c;  // +5 pour taille X8
} s[100] ;
// s fait 16*100 octets
// utilisation du cache plus efficace
```

[1] même si le processeur peut exécuter du code non aligné (comme le pentium)

Disposition des données (cont.)

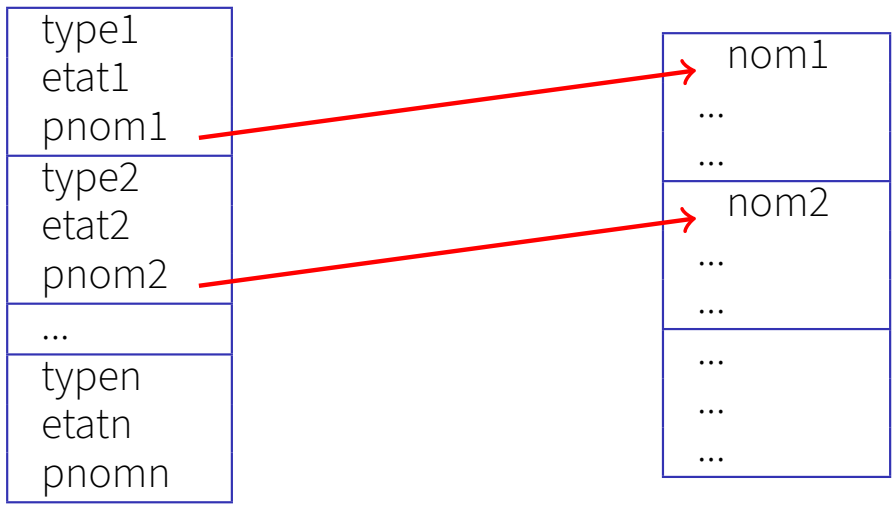
Garder proches des données utilisées simultanément.

```
struct {
  char type;
  char nom[100];
  char etat;
} s[200];
// Il est probable que type et état
// seront utilisés le plus couramment
// et nom[] sera rarement de 100 octets
```

```
struct {
  char type;
  char etat;
  char nom[100];
} s[200];
// Cette disposition est donc
// nettement préférable
```

Si une partie de `struct` est peu utilisée, on peut avoir intérêt à la stocker indépendamment.

```
struct {
  char type;
  char etat;
  char * pnom;
} s[200];
```



Disposition des données (cont.)

Tableaux de `struct`

Quand on manipule simultanément des tableaux, on peut avoir intérêt à définir un `struct` (*array of struct*)

```
int a[N], b[N];
for(int i=0; i<N; i++){
  a[i] = f(b[i]);
}
```

L'accès à `b[]` fera souvent des défauts de page.

```
struct {
  int a;
  int b;
} ab[N];
for(int i=0; i<N; i++){
  ab[i].a = f(ab[i].b);
}
```

L'accès aux données est totalement linéaire.

Disposition des données (cont.)

Traiter les tableaux/matrices par ligne

```
//multiplication de matrices ijk
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum +=  a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Chaque itération de la boucle interne fait un défaut de cache sur **b**

```
// multiplication de matrices ikj
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

c et **b** sont traités par ligne. Il n'y a plus que les défauts de cache obligatoires.

Fission et fusion de boucles

```
for(int i=0; i<N; i++)
  A[i]=b*B[i];
for(int i=0; i<N; i++)
  C[i]=c*B[i];
```

fusion

fission

```
for(int i=0; i<N; i++) {
  A[i]=b*B[i];
  C[i]=c*B[i];
}
```

La fission évite les défauts dus à des conflits sur des caches à faible associativité (par exemple entre `A[i]` et `C[i]`).

La fusion évite les défauts obligatoires dus au rechargement des données communes à plusieurs boucles (`B[i]` dans notre cas).

En règle générale, il est préférable de fusionner les boucles, surtout si le programme est limité par le débit processeur mémoire (*memory-bound*).

Mais dans certain rares cas (faible associativité des caches), il peut être préférable de faire une fission.

prefetching

Il peut être intéressant de faire une préacquisition dans le cache (*prefetch*) ou un préchargement dans une variable (*preload*) des données.
Doit être fait suffisamment à l'avance,

... mais pas trop pour être sur que la ligne reste dans le cache.

```
float t[N];
for(i=0; i<N; i+=4){
  __builtin_prefetch(&(t[i+4]));
  traiter(t[i]);
  traiter(t[i+1]);
  traiter(t[i+2]);
  traiter(t[i+3]);
}
```

```
float t[N], a,b,c,d,e;
a=t[0];
for(i=0; i<N; i+=4){
  b=t[i+1]; c=t[i+2];
  d=t[i+3]; e=t[i+4];
  traiter(a);     traiter(b);
  traiter(c);     traiter(d);
  a=e;
}
```

Sur des tableaux, les processeurs ont un *prefetch* très efficace et le gain est souvent faible.
Mais il peut être important sur des structures plus irrégulières.

```
void parcours_arbre_binaire (noeud *n) {
  __builtin_prefetch(n->gauche);
  __builtin_prefetch(n->droite);
  traiter(n);
  // condition de fin. Sur une feuille ...
```

```
  // ... les enfants pointent sur le noeud n
  if((n->gauche==n)&&(n->droite==n)) return;
  parcours_arbre_binaire(n->gauche);
  parcours_arbre_binaire(n->droite);
}
```

Allocation mémoire

Eviter les `malloc`s/`new`s. L'algorithme prend du temps et les données provoqueront des défauts de cache.
Alternatives :

- Allouer statiquement une donnée par une variable globale réutilisée autant que possible (par exemple pour acquérir une suite d'images).
- Si l'objet est temporaire plutôt l'allouer dans la pile par `alloca()`. La localité sera bien meilleure et l'allocation beaucoup plus rapide.
- Si une allocation dynamique est nécessaire, faire sa propre allocation à partir d'un réservoir (*pool*) que l'on gère. On peut ainsi définir une méthode `new()` dans la classe, par exemple.

Aliasing mémoire

Il y a *aliasing* mémoire quand plusieurs pointeurs peuvent modifier une même zone mémoire.
C'est un des principaux problèmes d'optimisation que l'on rencontre en pratique.

```
void sommerangee1(double *a, double *b) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      a[i*n + j] += b[i] ;
  }
}
```

`b[]` peut être dans la même zone mémoire que `a[]`. Par exemple avec un appel `sommerangee1(a,a+10);`

L'écriture dans `a[i*n+j]` peut donc modifier `b[i]`. Pour traiter ce cas, le compilateur doit *recharger* `b[i]` à *chaque* itération.
L'*aliasing* créée un grande inefficacité dans le code et interdit la plupart des optimisations que peut faire un compilateur, notamment le déroulage de boucles et la vectorisation SIMD.
Il donc l'identifier et le supprimer autant que possible.

Aliasing mémoire (cont.)

Souvent les compilateurs arrivent à éviter les problèmes d'*aliasing* en dupliquant le code généré :

```
void une_fonction_avec_aliasing(int a[], int b[]){
  vérifier si les zones mémoires de a[] et b[] intersectent
  si oui
    exécuter un code rechargeant les zones avec des alias
  si non
    code optimal sans alias
  fin
}
```

Mais dans un certain nombre de cas, il n'est pas possible de déterminer cette intersection sur un CPU.
Sur des GPU, il n'est pas possible de modifier le code en fonction des adresses et cette technique ne peut pas s'appliquer.
De plus, la taille du code est augmentée.

Aliasing mémoire (cont.)

Deux solutions :

- Supprimer l'aliasing en utilisant une variable locale pour la donnée problématique
- Déclarer les pointeurs comme `__restrict__`. Un pointeur/tableau ainsi déclaré indique au compilateur que la zone mémoire ne sera modifiée que par ce pointeur (C99).

```
void sommerangee2(double *a, double *b) {
  double tmp;
  for (int i = 0; i < n; i++) {
    tmp= b[i];
    for (int j = 0; j < n; j++)
      a[i*n + j] += tmp;
  }
}
```

Plus aucun problème, même si `a[]` et `b[]` sont partiellement dans la même zone.

```
void sommerangee3(double *__restrict__ a,
                  double *__restrict__ b)
{
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      a[i*n + j] += b[i];
  }
}
```

`__restrict__` indique au compilateur que les zones memoire de `a[]` et `b[]` sont disjointes. `b[i]` ne peut pas être modifié et aucun rechargement n'est nécessaire.

Aliasing mémoire (cont.)

L'aliasing est notamment très problématique dans les classes C++.

```
class tampon {
  private :
    int *buffer, nombre;
  public :
  void effacer() {
    for(int i=0; i<nombre; i++)
      buffer[i]=0;
    }
    ...
  }
  ...
```

Rien ne garantit que `buffer[i]` ne va pas écrire dans la zone mémoire où est situé `nombre`[2] et le modifier. Pour traiter ce cas, le compilateur doit recharger `nombre` depuis la mémoire à chaque itération.

```
class tampon {
  private :
    int *buffer, nombre;
  public :
  void effacer() {
    int n;
    for(int i=0, n=nombre; i<n; i++)
      buffer[i]=0;
    }
    ...
  }
  ...
```

La borne d'itération ne peut plus être modifiée. Code beaucoup plus efficace.

[2] Il va sans dire que c'est une pratique de programmation à proscrire **absolument**.

Accélération du code par le compilateur

Les compilateurs modernes effectuent un grand nombre d'optimisations (que le programmeur n'a plus à faire) :
Mise en ligne des fonctions (*function inlining*)
Peut se faire manuellement avec des macros, mais il est généralement préférable d'utiliser des fonctions.

```
#define carre(a) ((a)*(a)) // les () sont indispensables pour pouvoir faire carre(x+1)
y=carre(f(x)) ; // f inutilement evaluée deux fois. Utiliser plutot des fonctions inline.
```

Les compilateurs peuvent « *inlin*-er » des fonctions très efficacement.
Est relativement systématique pour les "petites" fonctions, surtout si elles sont déclarées `static`.
L'utilisation de `template`s permet d'avoir une certaine généricité.

```
template <typename T>
static void echange(T & a, T & b) { // peut être déclarée aussi static inline void ...
  T c;
  c=a;
  a=b;
  b=c;
}
...
float f1,f2;
echange (f1,f2);
```

Accélération du code par le compilateur (cont.)

Propagation des constantes

```
const float trois=3.0f;
#define demi 0.5f
...
float a;
a+= trois*demi;
```

```
const float trois=3.0f;
#define demi 0.5f
...
float a;
a+=1.5f;
```

Suppression des branches mortes

Supprime les tests inutiles (après propagation des constantes).

```
#define un 1
...
if (un > 0) x++;
else x=0;
```

```
#define un 1
...
x++;
```

Accélération du code par le compilateur (cont.)

Fusion des branches identiques

```
double x, y, z;
bool bb;
if (bb) {
  y = sin(x);
  z = y + 1.0;
} else {
  y = cos(x);
  z = y + 1.0;
}
```

```
double x, y, z;
bool bb;
if (bb) {
  y = sin(x);
} else {
  y = cos(x);
}
z = y + 1.0;
```

Suppression de sauts

```
1 int f(int a, bool b) {
2   if (b)
3     a = a * 2;
4   else
5     a = a * a;
6   return a + 1;
7 }
```

```
1 int f(int a, bool b) {
2   if (b) {
3     a = a * 2;
4     return a + 1;
5   } else {
6     a = a * a;
7     return a + 1;
8   }
9 }
```

Il faut un saut vers la ligne 6 après la ligne 3.

Accélération du code par le compilateur (cont.)

Déplacement de code invariant en dehors des boucles

```
float x[N], y;
for (int i = 0; i < N; i++) {
  a[i] = sqrt(y) + 1.0f;
}
```

```
float x[N], y, tmp;
tmp = sqrt(b) + 1.0f;
for (int i = 0; i < N; i++) {
  a[i] = tmp;
}
```

Variables d'induction (calculs linéaires dans une boucle)

```
int a[100];
for (int i = 0; i < 100; i++) {
  a[i] = i * 9 + 5;
}
```

```
int a[100], b;
for (int i = 0, b=5; i < 100; i++, b+=9) {
  a[i] = b;
}
```

Fait systématiquement pour supprimer des multiplications dans l'indexation des tableaux.

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

```
int a[N*N];
for (int i=0; i<N; i++) {
  for (int j=0, t=i; j<N; j++, t+=N) {
    a[t] ++ ;
  }
}
```

Déroulage de boucle

```
void tableau_carre(float t[N]) {
  for(int i=0; i<N; i++)
    t[i] *= t[i];
  return a;
}
```

```
void tableau_carre(float t[N]) {
  for(int i=0; i<N; i+=4)
    t[i] *= t[i];
    t[i+1] *= t[i+1];
    t[i+1] *= t[i+2];
    t[i+1] *= t[i+3];
  return a;
}
```

Le déroulage de boucle permet de :

- réorganiser le code pour limiter les aléas de données
- réduire le code associé au contrôle
- paralléliser (par exemple avec les extensions SIMD SSE, AVX, Neon, etc).

La plupart des compilateurs modernes (*gcc*, *clang*, *icc*, etc), effectuent un déroulage de boucle efficace en optimisation `-O3`.
Il est donc généralement inutile de l'effectuer "à la main", mais...

Déroulage de boucle (cont.)

... dans certains cas, cela peut être intéressant :
1/ Utiliser des accumulateurs multiples pour réduire les dépendances.

```
void somme_table(float t[N]) {
  float a=0.0f;
  for(int i=0; i<N; i++)
    a += f[i];
  return a;
}
```

Il y a une dépendance sur `a` entre les différentes itérations qui va ralentir l'exécution.

```
void somme_table(float t[N]) {
  float a1=0.0f, a2=0.0f;
  for(int i=0; i<N; i+=2){
    a1 += f[i];
    a2 += f[i+1];
  }
  return a1+a2;
}
```

Dépendance plus faible sur `a1` et `a2`.
Temps d'exécution ↘

2/ Simplifier le code interne aux itérations

```
void plus_moins(float t[N], v[N]) {
  for(int i=0; i<N; i++)
    if((i&1)==0) // itérations paires
      v[i] = t[i]+t[i+1];
    else         // itérations impaires
      v[i] = t[i-1]-t[i];
}
```

Le `if` va créer des aléas de contrôle.

```
void plus_moins(float t[N], v[N]) {
  for(int i=0; i<N; i+=2) {
    v[i]   = t[i]+t[i+1];
    v[i+1] = t[i]-t[i+1];
  }
}
```

Le déroulage permet de supprimer le `if`.

Pipeline logiciel

Le *pipeline logiciel* permet de masquer les attentes liées aux accès mémoire (comme un *preload*) et/ou celles liées à la latence des opérateurs dans une boucle.
Chaque itération va effectuer une partie du traitement pour des indices différents.

```
void somme2(float a[N], b[N]){
  float u, v, w;
  v=b[0];
  for (int i=0; i<N-1; i++) {
    //calcul de a[i]=b[i]+b[i+1]
    w=b[i+1];
    u=v+w;
    a[i]=u;
    v=w;          //=b[i+1]
  }
}
```

L'introduction des « registres » `v` et `w` évite de recharger `b[i]`.

```
void somme2(float a[N], b[N]){
  // avec pipeline logiciel
  // prologue
  float u, v, w;
  v=b[0];
  w=b[1];
  u=v+w ;                 // u=calcul de a[0]
  v=w;                    // v=b[1]
  w=b[2];
  for (int i=0; i<N-4; i++) {
    a[i]=u;               // ranger a[i]
    u=v+w;                // calculer a[i+1]
    v=w;
// copier b[i+2] dans v
    w=b[i+3];
// et charger b[i+3] dans w
  }
  //épilogue
  a[N-3]=u;
  u=v+w;
  a[N-2]=u;
}
```

Pipeline logiciel (cont.)

Le pipeline logiciel est fréquemment couplé à un déroulage de boucles et à une *rotation de registres*

```
void somme2(float a[N], b[N]){
  // prologue
  float u, v, w;
  v=b[0];
  w=b[1];
  u=v+w ;           // u=calcul de a[0]
  v=w;              // v=b[1]
  w=b[2];           // w=b[2]
  for (int i=0; i<N-4; i+=2) { // déroulage X2
       // v->b[i] w->b[i+1]
    a[i]=u;         // ranger a[i]
    u=v+w;
// calculer a[i+1]=b[i+1]+b[i+2]
    v=b[i+3];       // charger b[i+3] dans v
      //v->b[i+1] w->b[i]
    a[i+1]=u        // ranger a[i+1]
    u=w+v;
// calculer a[i+2]=b[i+2]+b[i+3]
    w=b[i+4];       // charger b[i+4] dans w
  }
  //épilogue
  a[N-3]=u;
  u=v+w;
  a[N-2]=u;
}
```

Programme précédent avec déroulage de boucle d'ordre 2. Pour éviter des transferts entre registres, les roles de **v** et **w** sont inversés pour les itérations paires et impaires.

Le pipeline logiciel est une méthode d'optimisation très efficace sur CPU ou GPU, mais potentiellement complexe pour des pipelines profonds.

Réordonnancement de code

Les compilateurs vont réordonner le code pour augmenter le parallélisme et limiter les dépendances.
Mais dans certains cas, il est nécessaire de les aider.

```
float a, b, c, d, x;
x = a + b + c + d;
```

Le standard C précise que ceci **doit** être calculé comme `((a+b)+c)+d`. Chaîne de dépendance pénalisante.

```
float a, b, c, d, y;
y = (a + b) + (c + d);
```

En réorganisant le code ainsi, `(a+b)` et `(c+d)` peuvent être évalués sans dépendances en parallèle et le calcul est plus rapide.

Les compilateurs ne prennent pas l'initiative de ce réarrangement[3] car, à cause des arrondis, les opérations flottantes ne sont pas associatives et les résultats des deux calculs peuvent (dans certains cas) être légèrement différents.

Pour des entiers, aucun problème d'arrondi et ce réarrangement est systématiquement fait pour augmenter le parallélisme.

[3] Sur `gcc` ou `clang`, `-funsafe-math-optimizations` change ce comportement.

Eviter les opérations coûteuses

Latence de quelques instructions (pentium *skylake)* :

Addition entière	1	Addition flottante	3
Multiplication entière	3–4	Multiplication flottante	5
Division entière	22–30	Division flottante	15–21
Conversion entier–flottant ou flottant–flottant			4–6

Eviter les divisions

```
a=b/1.67;
```

```
a=b*(1.0/1.67);
```

```
if(x > y / z) {
```

```
if(x * z > y) {
```

```
a=x1/y1 + x2/y2 ;//2 divisions
```

```
a=(x1*y2+x2*y1)/(y1*y2);//1 division
```

Les conversions `int-float` ou `float-double` ont un coût très important.
Faire attention aux conversions automatiques, aux déclarations, aux constantes...

```
float b;
b += 1.7; // 1.7 est double
   // b=(float)((double)b+1.7);
   // latence : 3+5+5=13 cy
```

```
float b;
b += 1.7f; // 1.7f est float
   // pas de conversion
   // latence 3 cy
```

Bonnes pratiques de programmation

Un code lisible et facile à maintenir est aussi généralement plus facilement optimisable par un compilateur.

Utiliser le plus possible des prototypes de fonctions.

Utiliser des `const` dans les déclarations de variables constantes et les prototypes de fonctions. Cela aide le compilateur et améliore la lisibilité du programme.

Préférer les tableaux aux pointeurs (très difficiles à vectoriser, sujets aux alias mémoire, etc).

Préférer les `enums` aux `#define`

etc.

Optimisation d'une application

L'analyse des performances d'une application nécessite de prendre en compte de nombreux facteurs concernant :

- l'algorithme (localité des calculs, transferts de données, dépendances, opérations effectuées, etc)
- l'architecture (caches, unités de traitement, parallélisme, etc)

Un paramètre important est l'**intensité de calcul** qui est le rapport entre

- le nombre d'opérations effectuées (en millards d'opérations par seconde (GFLOPS))
- le débit entre processeur(s) et mémoire(s) (en millards d'octets transférés par seconde (Gb/s))

Optimisation d'une application (cont.)

Une faible *intensité de calcul* indique que la facteur limitant est le transfert avec la mémoire « ***memory bound*** »
Une forte *intensité de calcul* indique que la limite vient d'une puissance de calcul insuffisante « ***compute bound*** »

L'idéal est d'avoir une *intensité de calcul* autour de 1 qui indique un algorithme et une architecture équilibrés.

En embarqué, les applications sont souvent *memory bound*, mais en calcul haute performance, les deux types de limites sont courantes.

Connaitre le type de limitation permet de déterminer les optimisations à effectuer :

- *memory bound* : améliorer la localité, réduire les défauts de cache, faire du *prefetching*, etc
- *compute bound* : optimiser l'utilisation du (ou des) processeurs (branchements, parallélisme d'instruction, etc), paralléliser le traitement, etc

Conclusion

L'optimisation de code peut permettre de gagner des facteurs importants (pouvant aller jusqu'à un ordre de grandeur).

Mais ne pas oublier :

- on n'optimise qu'un code fonctionnel
- après avoir détecté les zones coûteuses (*hot spots*) par profilage
- et avoir considéré des améliorations de haut niveau (algorithme, stucture de données, etc).