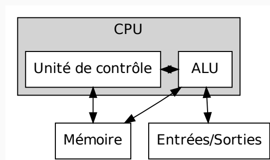
OPTIMISATION DES PERFORMANCES

Anthony Scemama, Nicolas Renon, Julien Bodart, Jérémie Gressier

April 22, 2018

Accès aux données



Il faut:

- alimenter l'unité de contrôle en continu avec des instructions
- alimenter les unités arithmétiques et logiques (ALU) en continu avec des données

Cette alimentation est le goulot d'étranglement (Neumann bottleneck)

Matrice des distances

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

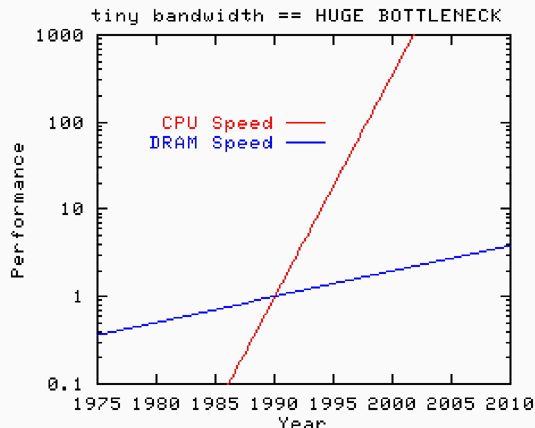
- `n`=100 (78 KiB) : 1.3 cycles / `dist[i][j]`
- `n`=4000 (125 MiB) : 3.0 cycles / `dist[i][j]`
- `n`=40000 (12.2 GiB) : 4.7 cycles / `dist[i][j]`

Matrice des distances

```
for (i=0 ; i<n ; i++)
  for (j=0 ; j<i ; j++)   // 2 fois moins de flops
    dist[i][j] = (X[0][i]-X[0][j])*(X[0][i]-X[0][j]) +
                 (X[1][i]-X[1][j])*(X[1][i]-X[1][j]) +
                 (X[2][i]-X[2][j])*(X[2][i]-X[2][j]) ;
for (i=0 ; i<n ; i++)
  for (j=i+1 ; j<n ; j++)
    dist[i][j] = dist[j][i];
```

- n=100 : 1.9 cycles : $\times 1.46$
- n=4000 : 18.9 cycles : $\times 6.3$
- n=40000 : 35.3 cycles : $\times 7.5$

LE "MUR" DE LA MÉMOIRE



- CPU : ×1.55 / an
- Mémoire : ×1.10 / an

Stream benchmark : http://www.cs.virginia.edu/stream

LE "MUR" DE LA MÉMOIRE

Latence Temps de transfert d'une seule donnée entre deux points

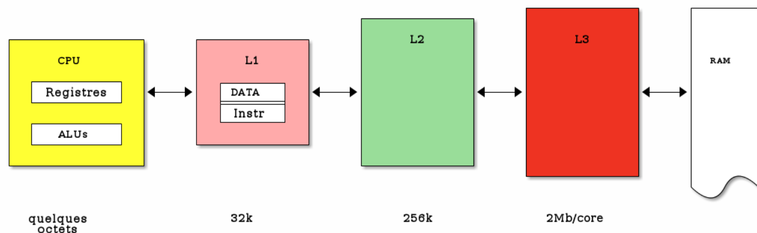
Bande passante Quantité de données qui passent par un point par unité de temps

Améliorations de la latence et de la bande passante sur 20 ans:

	Latence	Bande passante
Disque	8×	143×
RAM	4×	120×
Ethernet	16×	1000×
CPU	21×	2250×

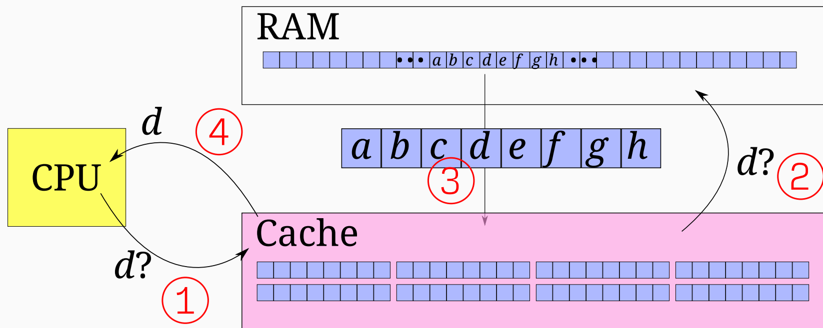
- Mémoires hiérarchiques (caches) : masquent les latences
- Les accès aléatoires sont de plus en plus coûteux

MÉMOIRES HIÉRARCHIQUES



- Une ALU fait un `LOAD` d
- Si d $\in$ L1, copie de d dans le registre
- Si d $\notin$ L1,
- Si d $\in$ L2, copie de d dans L1 et dans le registre
- etc

Quand un cache demande une donnée à un niveau supérieur, il transfère une ligne de cache : un bloc de taille fixe (typiquement 64 octets).



Localité

Spatiale : Si on demande e après avoir demandé d, e sera dans le cache

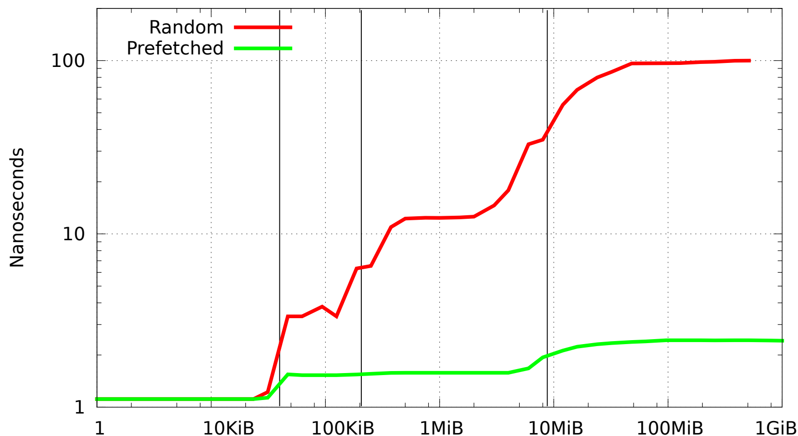
Temporelle : La ligne de cache remplacée par la nouvelle est celle qui est la plus ancienne (Least Recently Used, LRU)

Prefetching

Si un accès régulier aux données est détecté, les lignes de cache suivantes seront demandées en avance

LATENCE (NANOSECONDES)

Accès dans un tableau de taille croissante:



latence (nanosecondes)

Intel(R) Xeon(R) CPU E3-1271 v3 @ 3.60GHz

1 cycle = 0.28 ns, peak SP throughput = 0.0087 ns/flop

Integer	ADD	MUL	DIV	MOD	Bit
32 bit	0.28	0.84	6.60	7.07	0.28
64 bit	0.28	0.84	11.80	11.75	0.28
Floating Point	**ADD**	**MUL**	**DIV**		
32 bit	0.84	1.39	3.77		
64 bit	0.84	1.39	5.71		
Data read	**Random**	**Prefetched**			
L1	1.11	1.11			
L2	3.3	1.54			
L3	12.3	1.58			
RAM	100.	2.4			

`http://lmbench.sourceforge.net`

Latence (cycles)

Intel(R) Xeon(R) CPU E3-1271 v3 @ 3.60GHz

1 cycle = 0.28 ns, peak SP throughput = 32 flops/cycle

Integer	ADD	MUL	DIV	MOD	Bit
32 bit	1	3	23	25	1
64 bit	1	3	42	42	1
Floating Point	**ADD**	**MUL**	**DIV**		
32 bit	3	5	13		
64 bit	3	5	20		
Data read	**Random**	**Prefetched**			
L1	4	4			
L2	12	5.5			
L3	44	5.6			
RAM	357	8.6			

http://lmbench.sourceforge.net

BANDE PASSANTE

Intelő Xeonő Processor E5-2690 v3 (30M Cache, 2.60 GHz):

`a(i) = a(i) + b(i)*c(i)`

- 2 FMA vectoriels par cycle : `a = a + b*c`
  vecteurs de 4 flottants en DP (4×8 octets)
- 16 octets par flop
- Puissance crête
  2.6 GHz $\times$ ($2 \times 2 \times 4$ flops) $\times$ 12 coeurs = 499.2 GFlops/s (DP)
- Bande passante nécessaire :
  499.2 Gflops/s $\times$ 16 o/flops = 8 TiB/s
- Bande passante mémoire max
  4 canaux $\times$ 2133 MHz $\times$ 8 octets = 68.2 GiB/s

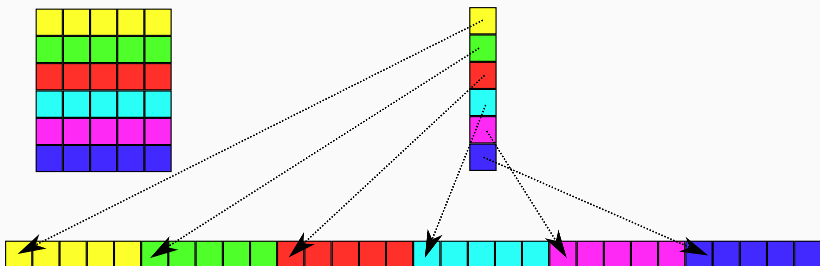
La bande passante mémoire est 117x trop faible !

```c
A = (double*)  malloc (n * sizeof(double));
```

- `malloc` alloue un bloc continu de mémoire
- `sizeof (double)` : nombre d'octets
- `(double*)` cast du type `void*` vers `double*` : permet le calcul d'adresse logique (A[i]) $\longrightarrow$ physique (0xa23b4)

Donc : les éléments d'un tableau 1D se suivent.

```
A    = (double**) malloc( n   * sizeof(double*) );
A[0] = (double* ) malloc( n*m * sizeof(double ) );
for (i=1 ; i<n ; i++)      // Incrémentation de l'adresse
  A[i] = A[i-1] + m;       // de m*sizeof(double) octets
}
```



ORGANISATION DES DONNÉES : TABLEAU 2D

```
A    = (double**) malloc( n   * sizeof(double*) );
A[0] = (double* ) malloc( n*m * sizeof(double ) );
for (i=1 ; i<n ; i++)       // Incrémentation de l'adresse
  A[i] = A[i-1] + m;        // de m*sizeof(double) octets
}
```

- Le 1er `malloc` alloue un bloc continu de mémoire
- Le 2ème `malloc` alloue un autre bloc continu de mémoire
- Le pointeur vers le 2ème bloc est stocké dans `A[0]`
- On affecte à `A[i]` l'adresse du début de chaque ligne

Donc : les éléments d'un tableau 2D se suivent en ligne.
On a aussi `&A[i][n] = &A[i+1][0]` .

Matrice des distances

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

- `n`=100 (78 KiB) : 1.3 cycles / `dist[i][j]`
- `n`=4000 (125 MiB) : 3.0 cycles / `dist[i][j]`
- `n`=40000 (12.2 GiB) : 4.7 cycles / `dist[i][j]`

- `X[0]`, `X[1]` et `X[2]` : lectures contigües
- `dist[i]` : écritures contigües

Limite : Bande passante

Matrice des distances

```
for (i=0 ; i<n ; i++)
  for (j=0 ; j<i ; j++)   // 2 fois moins de flops
    dist[i][j] = (X[0][i]-X[0][j])*(X[0][i]-X[0][j]) +
                 (X[1][i]-X[1][j])*(X[1][i]-X[1][j]) +
                 (X[2][i]-X[2][j])*(X[2][i]-X[2][j]) ;
for (i=0 ; i<n ; i++)
  for (j=i+1 ; j<n ; j++)
    dist[j][i] = dist[i][j];
```

- `n`=100 : 1.9 cycles : ×1.46
- `n`=4000 : 18.9 cycles : ×6.3
- `n`=40000 : 35.3 cycles : ×7.5

- `dist[j][i]` : Ecriture distante de `n` éléments, jamais dans le cache

Mauvais accès à la mémoire : limité par la latence.

```
typedef struct {
  double hx, hy;
  double c0, c1, c2;
  int rank;
  int size;
  int nx;
  int ny;
  int itmax;
  int n_bytes;
  char t;
} A ;
```

- Les données de chacun des champs du `struct` se suivent en mémoire
- Mettre `char t` en 1er décalerait toutes les adresses de 1 octet : mauvais!
- Il faut choisir un ordre tel que la taille des types décroît : accès correctement alignés

```c
typedef struct {
 double x,y,z;
} point;

dot = 0.;
for (i=0 ; i<n ; i++)
  dot += p[i].x * q[i].x + p[i].y * q[i].y
       + p[i].z * q[i].z;
```

On utilise toujours tous les éléments (`x,y,z`) du `struct` en même temps $\Longrightarrow$ Tableau de struct OK.

```c
typedef struct {
 char* Nom;
 int Z;
 int N_electrons;
 int N_protons;
 double masse;
 double x,y,z;
} atome;
```

```c
dot = 0.;
for (i=0 ; i<n ; i++)
  dot += p[i].x * q[i].x
       + p[i].y * q[i].y
       + p[i].z * q[i].z;
```

Chaque `x,y,z` est distant du précédent $\Longrightarrow$ Tableau de struct pas OK. Il faut faire plutôt un struct de tableaux.

```
typedef struct {
 char** Nom;
 int* Z;
 int* N_electrons;
 int* N_protons;
 double* masse;
 double* x,y,z;
} atomes;
```

```
dot = 0.;
for (i=0 ; i<n ; i++)
  dot += p.x[i] * q.x[i]
       + p.y[i] * q.y[i]
       + p.z[i] * q.z[i];
```

Les 3 flux `p.x`, `p.y` et `p.z` sont pre-fetchés en parallèle $\Longrightarrow$ augmentation de la bande passante.

Struct of array est (presque) toujours meilleur que array of struct .

Instructions

Quand on est sûr que la vitesse du programme n'est pas limitée par les accès à la RAM, on peut optimiser les instructions.

Intensité arithmétique: $\sigma = \mathrm{N(Flops)/N(octets)}$

Exemples

- Incrément de vecteur: `x[i] += y[i]` : $\sigma = \mathrm{N/(24N)} = 0.042$ flops/o
- Produit scalaire : $\mathrm{x} = \sum_{\mathrm{i}}^{\mathrm{N}} \mathrm{a_i} \times \mathrm{b_i}$: $\sigma = \mathrm{2N/(16N)} = 0.125$ flops/o
- Produit de matrices : $\mathrm{X_{ij}} = \sum_{\mathrm{k}} \mathrm{A_{ik}} \times \mathrm{B_{kj}}$: $\sigma = \mathrm{2N^3/(24N^2)} \propto \mathrm{N}$

Intel(R) Xeon(R) CPU E3-1271 v3 @ 3.60GHz

1 cycle = 0.28 ns

Integer	ADD	MUL	DIV	MOD	Bit
32 bit	1	3	23	25	1
64 bit	1	3	42	42	1
Floating Point	**ADD**	**MUL**	**DIV**		
32 bit	3	5	13		
64 bit	3	5	20		
Data read	**Random**	**Prefetched**			
L1	4	4			
L2	12	5.5			
L3	44	5.6			
RAM	357	8.6			

`http://lmbench.sourceforge.net`

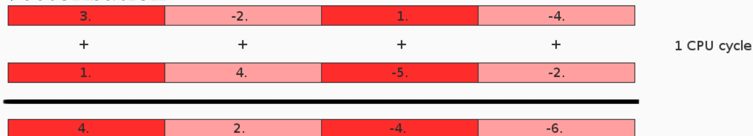
Instructions peu coûteuses

- ADD, MUL (int)
- ADD, MUL (float/double)
- AND, OR, XOR, ... (bool)

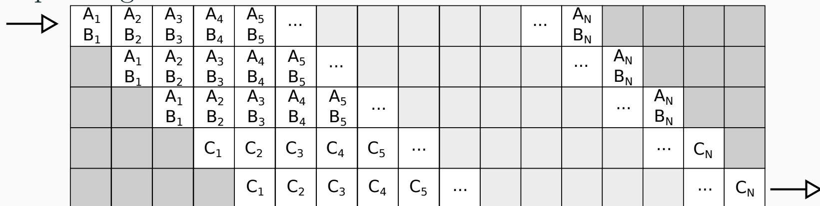
Instructions coûteuses

- DIV, MOD (int)
- DIV (double)
- SQRT (double)
- EXP, POW, LOG, SIN, COS, etc (float/double)

- Utilisent une seule unité (super-scalaire)
- Au moins 2 unités peuvent exécuter l'instruction
- Vectorisation



- Pipelining



Attention : Un `if` mal prédit vide le pipeline (bulles).

- Méthode de Newton:
  Division:

$$\mathrm{f(X) = (1/X) - D = 0, \quad X_{i+1} = X_i(2 - DX_i)}$$

  Sqrt:

$$\mathrm{f(X) = (1/X^2) - S = 0, \quad X_{i+1} = 1/2 \times X_i(3 - SX_i^2) \Longrightarrow 1/\sqrt{X}}$$

- Appels de bibliothèque

jeux d'instructions x86

MMX : MultiMedia eXtensions (1997). 57 instructions. Opérations vectorielles sur entiers (128 bits)

SSE : Streaming SIMD Extensions (1999). 70 instructions. Opérations vectorielles sur réels simple précision (128 bits)

SSE2 : (2001). 144 instructions. Opérations vectorielles sur entiers, réels en simple et double précision (128 bits). Remplace MMX.

SSE3 : (2004). 13 instructions. Opérations vectorielles horizontales (128 bits).

SSSE3 : (2006). 32 instructions. Shuffle, dot, ...

SSE4 : (2007). 54 instructions.

AVX : Advanced Vector Extensions (2011). Opérations vectorielles flottantes (256 bits). Instructions à 3 opérandes.

AVX2 : (2013) Opérations vectorielles entières + FMA.

AVX-512 : (2015). Opérations vectorielles (512 bits). FMA entier, prefetch, exp, div.

http://www.agner.org/optimize/instruction_tables.pdf

Compilation

Compilateur

Programme informatique qui transforme un code source écrit dans un langage de programmation (le langage source) en un autre langage informatique (le langage cible). (Wikipedia)

- Source : C, C++, Fortran, etc
- Cible : Langage Machine

```
double dot(int n, double x[n], double y[n]) {
  int i;
  double result;
  result = 0.;
  for (i=0 ; i<n ; i++)
    result += x[i]*y[i];
  return result;
}
```

```
$ gcc -g -c -O0 test.c
$ objdump -S -d test.o > test.asm
```

PROGRAMME COMPILÉ (-O0)

```
double dot(int n, double* x, double* y) {
 0:   55                      push   %rbp
 1:   48 89 e5                mov    %rsp,%rbp
 4:   89 7d ec                mov    %edi,-0x14(%rbp)      n
 7:   48 89 75 e0             mov    %rsi,-0x20(%rbp)      x
 b:   48 89 55 d8             mov    %rdx,-0x28(%rbp)      y
  result = 0.;
 f:   b8 00 00 00 00          mov    $0x0,%eax             copie 0 -> %eax (32 bits)
14:   48 89 45 f0             mov    %rax,-0x10(%rbp)      copie %rax -> result (64 bits)
  for (i=0 ; i<n ; i++)
18:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)       i=0
1f:   eb 38                   jmp    59 <dot+0x59>         saut en 59
result += x[i]*y[i];
21:   8b 45 fc                mov    -0x4(%rbp),%eax       copie i -> %eax
24:   48 98                   cltq                         conversion 64 bits
26:   48 c1 e0 03             shl    $0x3,%rax             %rax+8 -> %rax (adresse)
2a:   48 03 45 e0             add    -0x20(%rbp),%rax      &x[i-1] + 8
2e:   f2 0f 10 08             movsd  (%rax),%xmm1          load x[i] -> %xmm1
21:   8b 45 fc                mov    -0x4(%rbp),%eax       copie i -> %eax
35:   48 98                   cltq                         conversion 64 bits
37:   48 c1 e0 03             shl    $0x3,%rax             & -> &+8
3b:   48 03 45 d8             add    -0x28(%rbp),%rax      &y[i-1] + 8
3f:   f2 0f 10 00             movsd  (%rax),%xmm0          load y[i] -> %xmm0
43:   f2 0f 59 c1             mulsd  %xmm1,%xmm0           %xmm1 * %xmm0 -> %xmm0
47:   f2 0f 10 4d f0          movsd  -0x10(%rbp),%xmm1     load result -> %xmm1
4c:   f2 0f 58 c1             addsd  %xmm1,%xmm0           %xmm1 + %xmm0 -> %xmm0
50:   f2 0f 11 45 f0          movsd  %xmm0,-0x10(%rbp)     store %xmm0 -> result
  for (i=0 ; i<n ; i++)
55:   83 45 fc 01             addl   $0x1,-0x4(%rbp)       i + 1
59:   8b 45 fc                mov    -0x4(%rbp),%eax       copie i+1 -> i
5c:   3b 45 ec                cmp    -0x14(%rbp),%eax      comparaison (i, n)
5f:   7c c0                   jl     21 <dot+0x21>         saut a 21 si i<n
  return result;
61:   48 8b 45 f0             mov    -0x10(%rbp),%rax      copie result -> %rax
65:   48 89 45 d0             mov    %rax,-0x30(%rbp)      copie %rax -> return
69:   f2 0f 10 45 d0          movsd  -0x30(%rbp),%xmm0
6e:   5d                      pop    %rbp
```

PROGRAMME COMPILÉ (-O3 -MARCH=NATIVE)

```
for (i=0 ; i<n ; i++)
 0:   85 ff                 test   %edi,%edi                  n AND n -> %edi
result = 0.;
 2:   c5 f9 57 c0           vxorpd %xmm0,%xmm0,%xmm0          0. -> %xmm0 (64bits)
for (i=0 ; i<n ; i++)
 6:   7e 1e                 jle    26 <dot+0x26>              saut en 26 si < ou =
 8:   31 c0                 xor    %eax,%eax                  i=0 -> %eax (64bits)
 a:   66 0f 1f 44 00 00     nopw   0x0(%rax,%rax,1)
  result += x[i]*y[i];
10:   c5 fb 10 0c c6        vmovsd (%rsi,%rax,8),%xmm1        copie x[i] -> %xmm1
15:   c5 f3 59 0c c2        vmulsd (%rdx,%rax,8),%xmm1,%xmm1  y[i]*%xmm1 -> %xmm1
1a:   48 83 c0 01           add    $0x1,%rax                  i+1 ->
for (i=0 ; i<n ; i++)
1e:   39 c7                 cmp    %eax,%edi                  Comparaison i, n
  result += x[i]*y[i];
20:   c5 fb 58 c1           vaddsd %xmm1,%xmm0,%xmm0          %xmm1 + %xmm0 -> %xmm0
for (i=0 ; i<n ; i++)
24:   7f ea                 jg     10 <dot+0x10>              saut en 10 si >
26:   f3 c3                 repz retq
```

QUE FAIT -O3 ?

- Remplace des instructions coûteuses (DIV, exp) par des approximations
- Ne respecte pas exactement les parenthèse : (a+b)+c
- Inlining des fonctions
- Fusion/distribution des boucles
- Déroulage des boucles
- Inversion des boucles
- Vectorisation
- Prefetching software
- ...

QUE FAIT -O3 ?

Quelques options utiles:

- `-march=native` : Optimise pour l'architecture courante.
- `-ftree-vectorize` : Vectorisation
- `-funroll-loops` : Déroulage des boucles.
- `-fprefetch-loop-arrays` : Prefetch software.
- `-ffast-math` : Moins de précision sur certaines opérations.
- `-O3` : Optimisation agressive. Parfois moins efficace que `-O2`, il faut vérifier.

`man gcc`

Quelques astuces

- Optimiser seulement les points chauds (profiling avec gprof)
- Les boucles sont les zones chères
- Plus les boucles sont imbriquées, plus c'est cher

pour -1^{n}, ne pas faire `pow(-1.0,n)` :

- Si le bit de poids le plus faible est à 0, n est pair
- Si le bit de poids le plus faible est à 1, n est impair

```
double m1pn(int n) {
  static double memo[2] = { 1.0, -1.0 };
  return memo[ n & 1 ];
}
```

Lorsque le compilateur n'est pas sûr que deux tours de boucles sont indépendants, il ne peut pas faire certaines optimisations.

```
for (i=0 ; i<n ; i++)
 for (j=0 ; j<n ; j++)
  a[i][j] = a[i][j-1] + b;
```

Ici, il suffit d'inverser les boucles:

```
for (j=0 ; j<n ; j++)
 for (i=0 ; i<n ; i++)
  a[i][j] = a[i][j-1] + b;
```

Autre possibilité: ajouter une boucle

```
double b2[4] = { b, 2.*b, 3.*b, 4.*b };
for (i=0 ; i<n ; i++)
 for (j=0 ; j<n ; j+=4)
  for (k=0 ; k<4 ; k++)
    a[i][j+k] = a[i][j-1] + b[k];
```

FAUSSES DÉPENDANCES (ALIASING)

En C, les tableaux apparaissent comme des pointeurs. Parfois, le compilateur ne peut pas savoir que les tableaux ne se recouvrent pas:

```
double f( double* a, double* b ) {
for (i=0 ; i<n ; i++)
  a[i] = b[i] + c;
...
}
```

Peut-être que `&b[i+1] == &a[i]` ?

Il faut dire au compilateur que `a` et `b` ne se recouvrent pas:

```
double f( double *__restrict__ a, double *__restrict__ b ) {
for (i=0 ; i<n ; i++)
  a[i] = b[i] + c;
...
}
```

SORTIR LES IF DES BOUCLES

```
int i;
double a = 0.;
for (i=0 ; i<n ; i++) {
   if (i%2 == 1)
     a = a+1.0;
   else
     a = a+2.0;
}
```

```
int i;
double a = 0.;
for (i=0 ; i<n ; i+=2)
     a = a+1.0;
for (i=1 ; i<n ; i+=2)
     a = a+2.0;
```

SORTIR LES IF DES BOUCLES

```c
int i,j,k;
for (i=0 ; i<n ; i++)
 for (j=0 ; j<n ; j++)
  for (k=0 ; k<n ; k++)  {
   ...
   if ( fonction_test(i) ) {
     ...
   } else {
     ...
   }
   ...
  }
```

```c
int i,j,k;
for (i=0 ; i<n ; i++)
 if ( fonction_test(i) ) {
  for (j=0 ; j<n ; j++)
   for (k=0 ; k<n ; k++)  {
   ...
   }
 } else {
  for (j=0 ; j<n ; j++)
   for (k=0 ; k<n ; k++)  {
   ...
   }
 }
```

```
for (i=0 ; i<n ; i++)
 for (j=0 ; j<n ; j++)
  for (k=0 ; k<n ; k++)  {
    c[i][j] +=
      a[i][k]* b[k][j] / d[j];
  }
```

```
for (i=0 ; i<n ; i++)
 for (j=0 ; j<n ; j++) {
  tmp = 0.;
  for (k=0 ; k<n ; k++)
    tmp += a[i][k]* b[k][j]
  c[i][j] += tmp / d[j];
  }
// ----
double d_inv[n];
for (k=0 ; k<n ; k++)
  d_inv[k] = 1./d[k];
for (i=0 ; i<n ; i++)
 for (j=0 ; j<n ; j++) {
  tmp = 0.;
  for (k=0 ; k<n ; k++)
    tmp += a[i][k]* b[k][j]
  c[i][j] += tmp * d_inv[j];
  }
```

Avoir à la fois b et $b^\dagger$ permet de choisir la forme la plus adaptée pour les accès.

```
for (i=0 ; i<n ; i++)
 for (j=0 ; j<n ; j++)
  for (k=0 ; k<n ; k++)
    c[i][j] += a[i][k]* b[k][j];
```

```
for (i=0 ; i<n ; i++)
 for (j=0 ; j<n ; j++)
  for (k=0 ; k<n ; k++)
    c[i][j] +=
       a[i][k]* b_t[j][k];
```

INVERSER L'ORDRE DES BOUCLES

```
for (i=0 ; i<n ; i++)
 for (j=0 ; j<n ; j++)
  for (k=0 ; k<n ; k++)
    c[i][j] += a[i][k]* b[k][j];
```

```
for (i=0 ; i<n ; i++)
 for (k=0 ; k<n ; k++)
  for (j=0 ; j<n ; j++)
    c[i][j] += a[i][k]* b[k][j];
```

Le compilateur peur vectoriser automatiquement.

DÉROULER LES BOUCLES POUR RÉDUIRE LE NOMBRE DE STORE

```
for (i=0 ; i<n ; i++)
 for (j=0 ; j<n ; j++)
  for (k=0 ; k<n ; k++)
    c[i][j] += a[i][k]* b_t[j][k];
```

2 loads, 1 store

```
n4= n >> 2 << 2;  // Equivalent à
n4= (n / 4) * 4;
```

Décalage de 2 bits à droite, puis 2 bits à gauche: mets les 2 derniers bits à zero : donne le multiple de 4 immédiatement inférieur.

```
int n4= n >> 2 << 2;
for (i=0 ; i<n ; i++) {
 for (k=0 ; k<n4 ; k+=4)
  for (j=0 ; j<n ; j++)
    c[i][j] +=
       a[i][k  ]* b[k  ][j]
     + a[i][k+1]* b[k+1][j]
     + a[i][k+2]* b[k+2][j]
     + a[i][k+3]* b[k+3][j];
 for (k=n4 ; k<n ; k++)
  for (j=0 ; j<n ; j++)
    c[i][j] +=
       a[i][k]* b[k][j]; }
```

(8 loads, 1 store)/4

BLOCKING

```
#define B1 64
#define B2 64
#define B3 64
int n4= n >> 2 << 2;
for (ib=0 ; i<n ; i+=B1)
 for (jb=0 ; j<n ; j+=B2)
  for (kb=0 ; k<n ; k+=B3)
   for (i=ib ; i<n && i<ib+B1 ; i++)
    {
    for (k=kb ; k<n4 && k<kb+B3 ; k+=4)
     for (j=jb ; j<n && j<jb+B2 ; j++)
        c[i][j] +=
          a[i][k  ]* b[k  ][j] + a[i][k+1]* b[k+1][j]
        + a[i][k+2]* b[k+2][j] + a[i][k+3]* b[k+3][j];
    for (k=n4 ; k<n && k<kb+B3 ; k++)
     for (j=jb ; j<n && j<jb+B2 ; j++)
        c[i][j] += a[i][k]* b[k][j];
    }
```

```c
double fact(int n) {
  int i;
  double result = 1.;
  for (i=2 ; i<=n ; i++)
    result *= ((double) i);
  return result; }
}

double fact(int n) {
  int i; double i_d = 1.;
  double result = 1.;
  for (i=2 ; i<=n ; i++) {
    i_d += 1.;
    result *= i_d;
  }
  return result; }
}
```

```c
double fact(int n) {
  static double memo[20]; int i;
  static int memomax = -1; double result;
  if (n <= memomax) return memo[n];
  if (n<=20) {
    memo[0] = 1.;
    for (i=memomax+1 ; i<=n ; i++)
      memo[i] = memo[i-1] * ((double) i);
    memomax = n;
    return memo[n];
  } else {
    result = fact(20);
    for (i=21 ; i<=n ; i++)
        result *= ((double) i);
      return result; }
  }
```

- Éviter les conversions de types implicites :
  `x = x * 3` → `x = x * 3.0` si `x` est `double`
- Factoriser: `a*x + b*x + c*x` → `x*(a+b+c)`
- Remplacer une div par une multiplication par l'inverse :
  `x/4.` → `x*0.25`
- Utiliser la simple précision si possible
- Remplacer les petites puissances par des multiplications :
  `pow(x,2)` → `x*x`
- Éviter d'évaluer les fonctions chères si possible :
  `exp(x) < 1.e-20` quand $|x| < -46$
- Transformer des div entières par des décalages de bits :
  `i = i / 8` → `i »= 3` si `i`≥ 0
- Transformer des mod entiers par des opérations logiques :
  `i % 64` → `i & 63` si `i`≥ 0