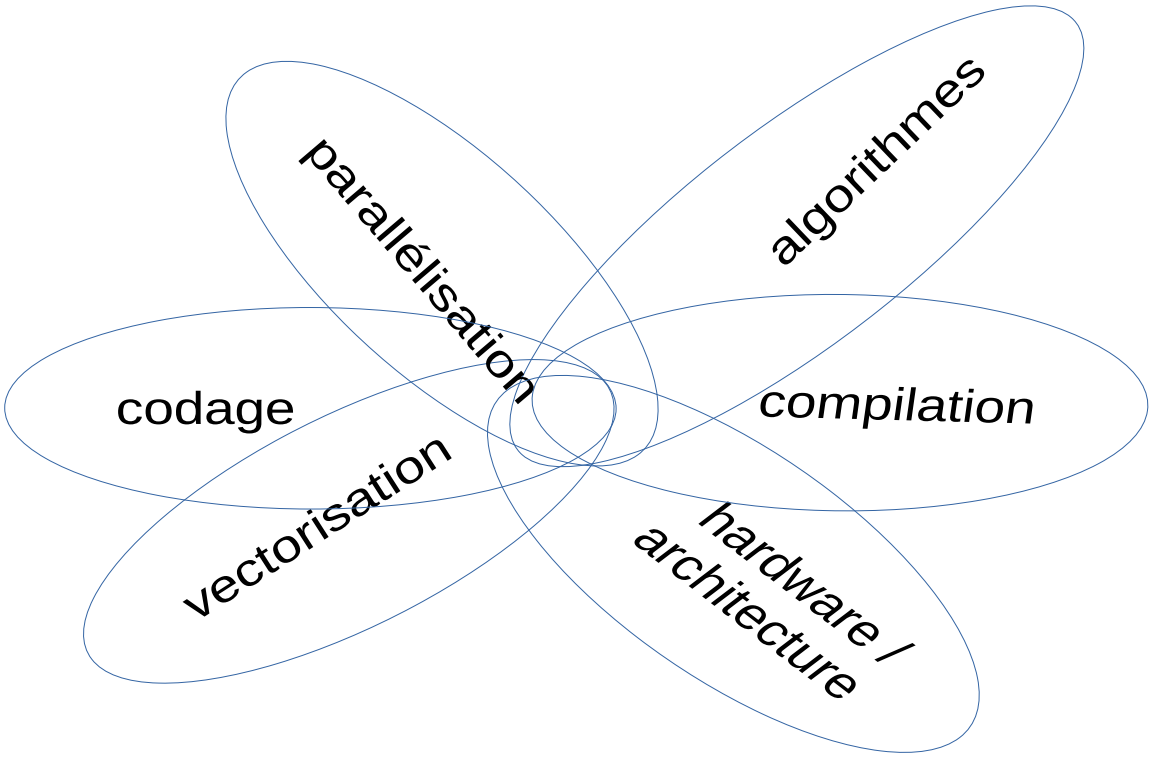
1. **Introduction au calcul haute-performance : optimisation et parallélisation**
   (→ Norbert KERN)
   - Quantifier la performance
   - Compiler avantageusement (premiers éléments)
   - Optimiser son code source
   - Jouer sur la hardware
   - Analyser (finement) les performance
   - Paralléliser (OpenMP, MPI)
   - Combiner C et Python

2. **Algorithmes : parallélisation d'un code de Dynamique Moléculaire classique**
   (→ Simona ISPAS)
   - Stratégies de parallélisation : décomposition atomique et spatiale
   - Implémentation pratique et vérification

3. **Mise en oeuvre : simulations ab initio**
   (→ Simona ISPAS)
   - Théorie de la fonctionnelle de la densité (DFT)
   - Dynamique moléculaire ab-initio
   - Implémentation pratique (notions de base)

INTRODUCTION AU CALCUL HAUTE-PERFORMANCE : OPTIMISATION ET PARALLÉLISATION



Cependant :

> « We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil** »
>
> Donald Knuth

Rappel : étapes de résolution un problème par approche numérique

i. Modélisation
ii. Conception des algorithmes
iii. Implémentation
iv. Compilation
v. Execution
vi. Exploitation/interprétation des données

(I) Modélisation :

c'est un métier … pas ici …
mais le choix peut impacter (Ex: se limiter aux puissances paires de la distance pour un potentiel d'interaction)

(II) Conception des algorithmes

Très important, surtout quand c'est mal fait … penser à ces deux 'échelles' :

- algorithmes à l'échelle globale (choix d'un algorithme d'intégration, agencement des étapes successives du code, etc), puis → *éviter de faire des calculs inutiles*
- implémentation à échelle 'locale' (programmer une opération, une boucle, une fonction) efficacement → *faire efficacement les calculs coûteux inévitables*

(III) Implémentation

Écrire le code source, organisé en bibliothèques, sous-fichiers etc; documentation; C (voir C++) ou Fortran

(IV) Compilation

Traduction du 'programme' dans un langage de programmation en 'programme' executable

Traduction du langage (plus ou moins) 'universel' au code spécifique pour l'architecture

```
Code (C, Fortran) → langage machine     → exécutable binaire
printf("%ld",x)   →    movl $8, %edx    → 10111011010100001010
                       int $0x80          01001000101110111010
                            ...           10010100010101010101
                                          11101000101111010101
                                                    ...
```

Rem:

- les données, mais aussi le code exécutable, sont binaires !
- but d'un code : transformer des données (binaires);
  ex: convertir cfg. initiale (00….110) en cfg. finale (01...101)

Ex : gcc truc.c && ./a.out

Rem : le '&&' enchaîne la deuxième commande, comme si elle était séparée par ';', mais uniquement si la première commande réussit

En réalité, en plusieurs étapes :

- *pré-compilation* : transformation du code source, selon spécifications.
  Ex (simple) :

```
g++ -DSILENT hello.C
```

  → le code est adapté en fonction de si le 'flag' **SILENT** est défini ou non

  Rem: d'autres exemples à venir
- puis *compilation*, dont la première partie consiste à produire le *code assembleur*

  Rem: on peut voir le code 'machine' (assembleur) : compiler avec l'option -S produit un fichier .s, qui contient ce code en clair

```
gcc -S hello.c && more hello.s
```

- fin de la compilation : fabrication d'un *binaire* exécutable (a.out par défaut)

```
gcc hello.c && ./a.out
```

Rem: plusieurs compilateurs (gnu, Intel, Portland, …); suivent les normes du langage mais certaines interprétations divergentes possibles
Ex: appplication des normes IEEE sur les nombres à virgule flottante), gestion du 'double zéro' des entiers, etc.

(V) Exécution

Chargement du binaire exécutable en mémoire, puis modification des données selon les algorithmes codés par application succesive des instructions binaires

Nécessite gestion des instructions (CPU) et des données (RAM)

(VI) Exploitation/interprétation des données

Peut être intégrée (logiciel 'monolithique), mais typiquement, en science :

- production des données (executable)
- sauvegarde sur disque
- analyse / interprétation 'post mortem'

Avantages/inconvenients : lenteur de l'eécriture; taille des données produites sur disque; facilité de scriptage; disponibilité des données pour analyse ultérieure modifiée; besoin du résultat des évaluations pour la suite de la simulation (ex: simulations à biais thermodynamique)

QUANTIFIER LA PERFORMANCE

Premier pas : quantifier, puis mesurer la 'performance'

Qqs indicateurs pertinents :

- temps d'exécution → utilisateur impatient
- flops (floating point operations per second) → cpu
- iops (input/output operations per second) → i/o
- paquets/octes par sec → réseau
- flops/Watt → consommation électrique
- flops/euro → budget
- scalabilité → calculs parallèles

Lequel utiliser ? Cela dépend ...
Quel indicateur pour les simulations atomistiques ? Cela dépend aussi …
La base, en pratique : mesurer le temps de calcul pour l'exécution d'un certain code est toujours instructif !

Mesurer le temps d'exécution (linux)

Rem: cela est possible aussi depuis un code même (bibliothèques en C, même python)

Ici: commande **`time`** :

Ex:

```
$ time sleep 5
real  0m5,002s
user  0m0,002s
sys   0m0,000s
```

'wall time' vs. 'user time'vs. 'cpu time'

- real = 'wall time' = chronomètre
- user = 'cpu time' = temps utilisé pour calculer
  (rem: cumulé si processeurs multiples)
- sys = 'system time' = appels de système
  (rem: cumulé sur sous-procesus comme forks etc.)

Rem: cpu<wall si processeur simple, mais pas nécessairement sur multi-processeur

À noter aussi aune subtilité : **`time`** est une commande de terminal, mais il existe aussi un executable **`/usr/bin/time`** (attention à la confusion involontaire !). Chacun a des avantages, mais le dernier peut être pratique ici car il permet de reconfigurer le format dans lequel le résultat est rendu.

Complexité algorithmique

L'**idée** est d'analyser comment le besoin de ressources dépend de la taille du 'système' (taille d'un tableau, chiffres retenus pour représenter un entier, longueur d'une chaîne de caractères, etc).

- on stipule une fonction f(N) décrivant comment les ressources nécessaire dépendent de la taille de l'entrée N
- ici, on considère le temps de calcul en fonction de la taille du système (mais des variantes caractérisent la mémoire nécessaire, etc.)
- on peut se baser soit sur scénario le plus défavorable, soit sur moyenne

ici donc :

taille du système N → temps de calcul f(N)

Pour une **opération simple** (comme par ex. l'addition, la multiplication ...)

- nécessite un certain temps de calcul
- ce temps varie d'un simple facteur d'une machine à l'autre
- la répétition de telles opérations multiplie simplement le temps total de calcul

La complexité caractérise donc directement l'algorithme utilisé

Exemples :

- l'addition est une opération de complexité O(N) par rapport à au nombre N de chiffres retenus : on somme sur N chiffres, de manière identique (somme de nombres entre 0 et 9, la retenue, et il faut gérer la nouvelle retenue)
- la multiplication est un algorithme de complexité $O(N^2)$ si on l'effectue comme c'est enseigné à l'école.

  Rem: il existe des algorithmes qui la ramènent, dans le meilleur des cas, à O(N*log(N))

Question/Exercice : quelle est la complexité algorithmique de la multiplication de deux matrices de dimension NxN ?

Pour des **opérations plus complexes**, malgré le fait que le temps d'exécution peut varier en fonction des circonstances (charge de calcul instantanée du processeur, taux d'occupation de la memoire vive, …), il reste pertinent de catractériser un algorithme par sa complexité.

Exemple : insertion successive de N eléments dans un tableau, initialement vide, en décalant tous les éléments suivants

- temps pour une affectation: T_1
- pire des cas : toutes les insertions au début du tableau
- nombre d'affectations du n-ième élément à insérer : n (càd n-1 décalages, 1 nouvelle affectation)
- temps total: $T_N = \sum_{n=0}^{N-1} n = \frac{(N-1)(N-2)}{2} = \frac{1}{2} N^2 + ... \simeq N^2$
- donc complexité $O(N^2)$

Rem: lorsque l'on insère systématiquement en dernière position, la complexité est $O(N)$.

Question/Exercice : et si l'insertion se fait à une position aléatoire de la liste ?

Travaux pratiques : généralités et fonctionnement

Ces consignes resteront *a priori* identiques pour toutes les séances de TP.

Les travaux pratiques se déroulent dans les salles informatiques, sous Linux.

Attention, le choix de Linux est délibéré, l'accès que l'utilisation du terminal de commande donne à l'exécution du code permet de faciliter l'accès aux informations nécessaires.
Attention aussi : vous pouvez bien entendu effectuer les mêmes manipulations sur d'autres systèmes Linux, mais les résultats ne seront pas le mêmes. D'une part, une performance différente affectera les résultats quantitatifs mesurant la performance. D'autre part, et plus subtilement, les résultats peuvent aussi différer *qualitativement* ! Cela vient du fait que le compilateur peut différer, son comportement par défaut peut différer et, à la base de tout, l'architecture diffère nécessairement. Vous aurez été avertis !

Évaluation :

Le critère d'evaluation reposera davantage sur la démarche que sur vos résultats : peut-on comprendre votre démarche ? Pourquoi avez-vous concu vos essais de telle manière ? Les paramètres, ont-ils été choisis de la bonne manière pour éviter des soucis ? Vos résultats, que signifient-ils ? Peut-on y croire ou pas ? Qu'est-ce qu'en apprend ? Etc.

A déposer :

- un document de synthèse **complet** : il doit documenter votre démarche et vos conclusions, permettre de comprendre et de reproduire vos résultats
  - en pdf (par exemple produit à partir de LibreOffice)
  - code et scripts éventuels (intégrées dans le texte)
  - puis surtout : idée de la démarche, graphes produits, démarche d'exploitation des données, conclusion, appréciations critiques, etc.
- votre code, et d'éventuels scripts, en code 'source'

Délais :

Il vous est demandé de compléter votre compte-rendu au fur et à mesure, en ajoutant la suite avant chaque nouvelle séance, mais il restera amendable. La totalité du document sera évalué à la fin de ce bloc du cours.

Travaux pratiques : Mesurer la performance

But : se familiariser avec les grandes lignes, et très probablement quelques subtilités, de la manière de quantifier la performance d'une opération simple dans un code.

Exercice : Coût des opérations élémentaires

On se propose d'de quantifier la performance pour des opérations simples, et en particulier de comparer la multiplication à la division, et si vous avez le temps à l'addition.
Implémentez un code effectuant N multiplications de deux nombres réels. Puis, en vous servant de la commande **`time`**, déterminez le temps de calcul nécessaire pour effectuer une multiplication. Faites de même pour une division et concluez. Complétez par l'addition si vous en avez le temps. Vous pouvez aussi contraster avec le résultat pour la multiplication de deux entiers : est-ce pareil ?

Quelques questions qui pourraient vous être utiles :

- combien de multiplications choisissez-vous d'effectuer? Dans l'idéal, et en pratique ? Pourquoi ?
- en répétant la mesure du temps d'exécution N fois, avec le même nombre N, que constatez-vous ?
- coment gérez-vous le problèm qu'il y a un 'overhead' (surcout) dans le temps de calcul qui est associé au lancement de votre programme ?

Une suggestion pratique :
Il est possible de recevoir un argument dans un program C/C++, ceci est illustré par le code fourni **`commandlinearguments.C`** . Après compilation, vous pouvez l'appeler comme ceci

```
> g++ -o cmdlargs.out commandlinearguments.C
> ./cmdlargs.out  testtesttest
```

et l'argument fourni ('testtesttest') est réceptionné par le code C comme chaîne de caractère.

Rem: En Fortran, le même type de mécanisme existe (https://gcc.gnu.org/onlinedocs/gfortran/GETARG.html et https://gcc.gnu.org/onlinedocs/gfortran/GET_005fCOMMAND_005fARGUMENT.html)

COMPILER AVANTAGEUSEMENT (PREMIERS ÉLÉMENTS)

On peut souvent obtenir des gains très importants
seulement en pilotant le compilateur correctement !

Illustration de flags de compilation

En fait, ce sont autant des flags de précompilation, qui modifient la manière dont le code executable est produit à partir d'un seul et même fichier source.

Exemple :

```
/* programme compiltest.c */
include <math.h>
#include <stdio.h>


main() {
  double x=0;
  for (long int i; i<1000000000; i++) {
    x += sinh(x);
  }
  printf("%lf" ,x);
}
```

En compilant cela par

```
g++ sometest.C -lm && time a.out
```

on obtient (à titre d'exemple) un temps 'user' de 5,366s, alors qu'en compilant ainsi

```
g++ -O2 sometest.C -lm && time a.out
```

cela devient 0.001s.

- gain d'un facteur de l'ordre de 5000 en performance (pour ce code, sur un système en particulier)
- typiquement les gains sont plus modestes mais pertinents
- c'est gratuit (!!!)
- sauf : temps de compilation risque d'augmenter

Illustration du principe

- pre-compilateur modifie le code source.

Ex : le code `double y = x/2.0` peut être remplacé par double `y=x*0.5` , opération donc une opération moins coûteuse !
Ex : le code **const** `x=42; y=x**2;` peut être remplacé par `x=42; y=1764;` et on peut ainsi économiser une multipication

- en réalité il s'agit d'un ensemble d'interventions : le comportement du compilateur est piloté par l'option de compilation -O

- le niveau de l'optimisation est choisi par le nombre spécifié :
  - -O0 : aucune optimisation
  - -O1 : un premier niveau d'optimisation, typiquement faciles à réaliser
  - -O2 : optimisation avec interventions plus lourdes
  - -O3 : optimisation très lourde
  - d'autres flags permettent d'aller au-delà … (à suivre)

- la différence des niveaux se joue sur le type d'opérations que l'on tente d'améliorer, le nombre de lignes que l'on regarde 'en avance' pour identifier des optimisations à faire, etc

Attention !

Augmenter le niveau d'optimisation requis
- va typiquement (mais pas toujours) optimiser la performance
- mais augmentera le temps de compilation
- et aussi la taille du binaire exécutable
- et potentiellement la mémoire vive nécessaire à l'exécution du programmer

Attention, on parle aussi de '**agressivité**' de l'optimisation
- le code source sera modifié de manière de plus en plus importante
- en produisant, *en principe,* un exécutable équivalent
- **mais** : les optimisations très aggressives peuvent entraîner des problèmes et subtilités, et occasionnellement cela conduit à des résultats numériques modifiés !!!
  Règle de base :
  - jusqu'à `-O2` l'optimisation est considérée 'safe'
  - pour `-O3` les problèmes sont rares mais peuvent exister pour des cas particuliers
  - le niveau -`Ofast` passe par-dessus certains standards stricts, en s'exposant à l'introduction d'erreurs par la compilation. **ATTENTION !!!** Ce niveau, et l'optimisation par d'autres flags 'sur mesure' (voir plus loin), sont susceptibles de poser problème …

Rem : voir la documentation https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

OPTIMISER SON CODE SOURCE

D'abord quelques rappels sue les variables, le typage et la gestion de la mémoire

Typage et types de variable (C) :

Compilation pour éfficacité

- compilation exploite le typage:
  → le type de la variable est *figé* et donc connu au moment de compiler
- types de variables alphanumériques
  - entiers : `int` et variantes : `short int`, `long int`, `unsigned int` ...
  - à virgule flottante : `float` (4 octets), `double` (8 octets)
  - et caractères: char (en réalité un `unsigned int`)
- Attention: valeurs minimales/maximales dépendent de l'architecture
  → voir limits.h

Coût des opérations simples :

Typiquement:

addition, soustraction ≪multiplication ≪division ≪appel de fonctions (sqrt etc)

Attention: cela dépend de l'architecture.

Ex : arithmétique d'entiers sur architecture Intel 64

- add/sub: 1 cycle CPU
- imul (integer multiply): 10 cycles
- idiv (integer division): 66–80 cycles(!)

→ https://www.quora.com/Is-there-any-performance-difference-between-the-basic-operations-sum-subtraction-multiplication-division-when-the-computer-performs-them

Surcoût d'un appel de fonctions

- surcoût inévitable (assembler : transmettre argument, chercher code de la fonction, etc)
- typiquement petit
- donc sans importance pour la plupart des fonctions : fonctions appelées rarement, fonctions lentes (i/o), etc
- potentiellement significatif sur les très petites fonctions

Mais, attention :

→ souvent : performance/optimisation vs. lisibilité/structuration/modularité du code

→ optimisations doivent cibler le code critique pour la performance (cf. Knuth !)

Ex : MD avec calcul des forces, images périodiques, etc

Donc, en pratique :

Pour les parties du code (uniquement !) qui sont critiques pour la performances

- remplacer une division par une multiplication, là où cela est possibles
  Ex: `s=x*Li` , plutôt que `s=x/L,` (avec `Li=1./L` calculé une fois pour toutes)
- éviter les appels de fonctions (si possible)
  Ex: `x2=x*x`, à la place de x`2 = pow(x,2)`
- utiliser des *variables temporaires* si besoin
  Ex: `r3=r*r*r; r6=r3*r3;` à la place de `r6=r*r*r*r*r*r;`

Mais, ***les calculs les plus rapides sont ceux qu'on évite de faire !***
→ d'autres économies peuvent être faites en amont

- conception de l'algorithme
  → complexité algorithmique
- choix du modèle (!)
  → le plus simple possible
  → numériquement simple
  Ex: Lennard-Jones : pour calculer les forces et l'énergie, on n'a **jamais** besoin de calculer la distance `r` entre particules !
  → baser tout sur r^2
  → éviter ainsi d'appeler la fonction `sqrt` (souvent ... !!!)

Ex : potentiel de Lennard-Jones

code 'naif':

```
double u(double sig,double eps, double r) {
  return 4*eps*(pow(sig,12)/pow(r,12) - pow(sig,6)/pow(r,6) );
}
```

améliorations:

- pour commencer, appliquer la puissance au ratio, plutôt que prendre le ratio des puissances
- mieux : éviter les facteurs **`sig, eps`**
- éviter **`l'appel de pow()`**
- **`éviter de calculer la racine`**
- **`utiliser des variables temporaires pour calculer les puissances en plusieures étapes`**
- **`etc ...`**

Fonctions 'inline' : encore une astuce de compilation

But : déclarer des fonctions en évitant le surcoût lié à l'appel

Pour cela, l'idée est simple : il suffirait de remplacer, dans le code source, chaque appel de fonction par le code complet

Ex : puissance de six

```
double psix(double x) {
   double x2 = x*x; /*temporary variable*/
   return x2*x2*x2; }
sigma6 = psix(sigma);
alpha6 = psix(alpha);
```

pourrait être remplacé avantageusement (en ce qui concerne la performance) par

```
double tmp;
tmp=sigma*sigma;   sigma6=tmp*tmp*tmp;
tmp=alpha*alpha;   alpha6=tmp*tmp*tmp;
```

Inconvenient : multiplicité du code, mauvaise lisbilité, …

On peut obtenir le même comporteent par le mot clé **`inline`** !

```
inline double psix(double x) {
   double tmp x2 = x*x;
   return x2*x2*x2; }
sigma6 = psix(sigma);
alpha6 = psix(alpha);
```

À savoir :

- cela est utile *seulement* pour de *très* petites fonctions
- en reálité ils s'agit d'une *suggestion* pour le compilateur : c'est son algorithme qui décide si oui ou non la fonction est repliquée ou préservée en tant que telle

  Rem: dans l'exemple, la présence d'une déclaration de variable pèse déjà lourdement en contre ...
- par conséquent, le comportement précis dépend du compilateur
- et les flags d'optimisation renforcent (ou déjouent) la recommendation faite par le mot clé **`inline`**

Rem : il existe, en C/C++ un mécanisme plus rudimentaire appelé un **`macro`**, qui atteint le même but, mais de manière **certaine** :

```
#define sq(x) ( x*x )
x=sq(1989)
y=sq(1984)
```

Ici, aucune fonction n'est définie, mais le pré-proceseur va réellement substituer **`x*x`** à cahque fois qu'il rencontre le code **`sq(x)`**. C'est très basique, mais ca marche bien (attention à qqs subtilités cependant, par exemple par rapport à la précédence de l'opérateur '*' une fois la substitution faite … tester !

Rem: il s'agit d'un exemple de directives de pré-processeur, qui propose d'autres mécanismes, e.g. pour une compilation conditionnelle selon si un certain flag est activé ou non **`(#ifdef)`**.

Une autre astuce de pré-processeur: tests par compilation conditionnelle

Une autre situation qui arrive souvent : faire exécuter, ou non, une certaine partie du code

Mécanisme : **compilation conditionnelle**, gérée au niveau de la pré-compilation

Ex : afficher, mais seulement pour tester, une valeur

```
for (int i=0; i<10; i++) {
      int i2 = i*i;
      #ifdef TEST
      printf("i**2 :  %d",i2);
      #endif
}
```

Le résultat à l'exécution dépend alors de la compilation, selon le paramètre précisé (ou non) :

```
> gcc moncode.c
> gcc -DTEST moncode.c
```

Ce mécanisme peut par exemple servir pour généraliser un code 2d à 3d, si cela est demandé (mais attention à tout maintenir de manière cohérente …).

Quelques variantes :

- l'opposé (négation) :

```
#IFNDEF … #ENDIF
```

- traitement selon le cas (if .. else) :

```
#IFDEF … #ELSE … #ENDIF
```

- transmission d'une valeur, par ex 'LEVEL' (comme 'TEST' ci-dessus, par -DLEVEL)

```
#if  LEVEL ==  1 … # endif
```

- etc

Se renseigner sur des directives de **préprocesseur**

Assertions

Un raccourci souvent bienvenu, justement pour sécuriser du code, reprend cette idée en forme d'une bibliothèque permettant de gérer des '**assertions**' ('affirmations'). Par exemple, le code

```
#include <stdio.h>
#include <assert.h>

void count(int N) {
    assert( N>0 );
    … code of function …;
}
```

permet de sécuriser la fonction : si jamais une valeur négative de N est passée, cela provoque l'arrêt du programme (avec un message d'erreur renvoyant à cette ligne de code).

En revanche, si cela est souvent justifié pour tester, cela ne peut pas être souhaitable en mode 'production', par exemple si cette fonction est appellée très fréquemment (ex: calcul des conditions périodiques aux bords). On peut alors d'en débarasser simplement en précisant

```
gcc -DNDEBUG moncode.c
```

et toutes les assertions seront simplement éliminées par le pré-processeur !

C’est assez simple, n’affecte donc pas la performance lors de l’utilisation du code, et permet de détecter un certain nombre de soucis avant d’avoir très mal à la tête ...

Travaux pratiques : Compilation et optimisation du code source

But : se familiariser avec l'idée de pouvoir influer sur la performance par des flags de compilation; comprendre l'impact de qqs flags simples sur la performance

ATTENTION : dans ce qui suit, il est parfois important de comparer à ce que fait le compilateur 'bêtement'. Or, il s'avère que le comportement de base peut déjà être configuré pour réaliser certaines optimisations (et ce comportement peut différer d'un système à l'autre, y compris pour un même compilateur …). Il est donc important, lorsque l'on cherche une référence pour juger l'impact d'une optimisation précise, de compiler le code de référence en précisant l'option `-O0` , afin d'empêcher explicitement toute optimisation.

Exercice : coût d'opérations simples (sur les réels)

Reprendre votre analyse des opérations simples (addition, multiplication, division, éventuellement sqrt(…)), en testant les divers niveaux de compilation `-On`, en vous focalisant sur les nombres réels (`float` ou `double`). Conclusion ? Et si vous activez l'option `-ffast-math` à la compilation ?

Question extra : en vous appuyant sur la documentation, pouvez-vous indiquer, voire mettre en évidence, un exemple ou le résultat numérique change suite à cette optimisation ?

Exercice : Interaction Lennard-Jones

On se donne les positions (x1,y1) et (x2,y2) de deux particules. L'interaction se fait selon le potentiel Lennard-Jones, en deux dimensions, tronqué à un rayon R_c et décalé pour assurer la continuité du potentiel. On a besoin d'une fonction.

```
const double Rc=2.5  /* global constant */

void interactions( double x1, double y1, double x2, double y2, \
                   double &u12, double &f12x, double &f12y ) {
  ...
  }
```

qui calcule l'énergie d'interaction u12 ainsi que la force f12 de l particule no. 1 sur la particule no. 2 et modifie les valeurs définies par référence en fonction. Optimisez-la !

Cette question est donc un concours : quelle est la meilleure performance que vous pouvez obtenir sur les machines de la salle TP ?

Exercice (si vous avez le temps) : surcoût lié à l'appel de fonction / optimisation

Mettez en place une stratégie pour déterminer le surcoût lié à l'appel d'une fonction. A quel temps estimez-vous ce surcoût pour le système en salle de TP ? Est-elle significative ?

Rem : pour être à l'abris de chercher des erreurs inexistants, *il vaudra mieux empêcher le compilateur d'optimiser par défaut, en considérant que vous avez probablement voulu déclarer votre fonction comme* ***`inline`*** *: le flag de compilation -****`fnoinline`*** *(tout collé) vous permet d'empêcher cela.*

Rem : *attention à ne pas généraliser – si vous appelez une fonction en C++ qui passe comme argument un objet, et vous passez cela par valeur, le surcoût peut être extrême, dû à l'opération de copie etc … En Python, le surcoût pour appeler une fonction aura aussi tendance à être plus important.*

JOUER SUR LA HARDWARE

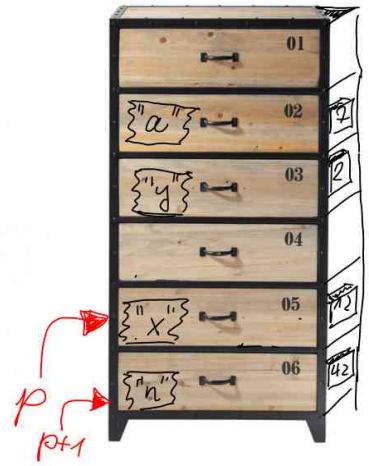
Un petit rappel sur l'organisation de la mémoire, et la manière d'y accéder (en C), s'impose, avant d'aborder comment cela peut être exploité :

Rappels : Variables, mémoire et 'pointeurs' (en C)

- adresse de mémoire:
  → cf. étiquettage de l'apothécaire

- pointeur :
  → une adresse de mémoire

```
double x = 12;
    // le nombre 12 dans un tiroir (ici le #05),/
       étiquetté 'x'
double *p = &x;
    // numéro du tiroir (ici #05)
```



Rem :

- la variable (pointeur) **p** contient donc une adresse en mémoire, celle où **x** est stocké
- en regardant la déclaration de **p**, la seule chose que l'on sait de son contenu est que c'est un double

- déréférencement d'un pointeur :
  → remonter de l'adresse au contenu

```
printf("%lf",*p);
      // *p : contenu du tiroir de cette équette: 12
```

Rem : à distiguer de l'affichage de l'adresse en mémoire

```
printf("%ld",p);
      // p : adresse de la mémoire référencée par p
      // (codé en int)
```

- arithmétique des pointeurs
  puisque les pointeurs ne sont que des adresses, et donc des nombres entiers (ou prèsque), on peut donner un sens à l'addition, par ex.

```
double *p = &x; p+=1
```

mais que cela veut-il dire ??? Simplement l'incrément à la case de mémoire suivante

→ **p** équivaut maintenant au tiroir #06, son contenu est **42**

→ c'est équivalent à un pointeur sur la variable **n**

→ Attention ! Dans cet exemple de tiroirs, les types ne correspondent pas cependant – on voit bien comment cela va poser problème si le type de **n** est différent du type de **p** (ce sera réglé pour les tableaux, voir dessous)

Rem: (ou plutôt: Avertissement) : le C se veut proche de la machine, et autorise des choses dangereuses – à vous d'assumer ...

Tableaux (1d), pointeurs et arithmétique de pointeurs

Avec cette idée, on a de quoi faire des tableaux : des tiroirs qui se suivent ...

- tableau (statique) :

  → ensemble de valeurs d'un même type

```
double x[100];
```

- accès par l'indice ('étiquette')

```
x[42] = 0.07;   // élément numéro 43 (!)
```

- stockage en mémoire contigue (en principe)

  → les entrées d'un tableau se suivent (cf tiroirs successifs)

- c'est très proche des pointeurs :

```
double *px = &x[42];  *px=3.1415;  /*ou près que*/
```

  Rem: ce pointeur est simplement l'adresse d'un élément du tableau.

  Rem: la variable 'tableau' même (ici **x**) est simplement l'adresse du premier élément du tableeau !

- donc idéal pour l'allocation dynamique

```
double *x = malloc(100 * sizeof(double));  // C
double *x = new double[100]; /* C++ */
x[42] += 0.01;
```

- quitte alors à aller jusqu'au bout : **arithmétique des pointeurs** :

  pointeur = adresse en mémoire, donc un entier (ou près que, non-signé etc)

  → arithmétique de pointeurs selon leur type !

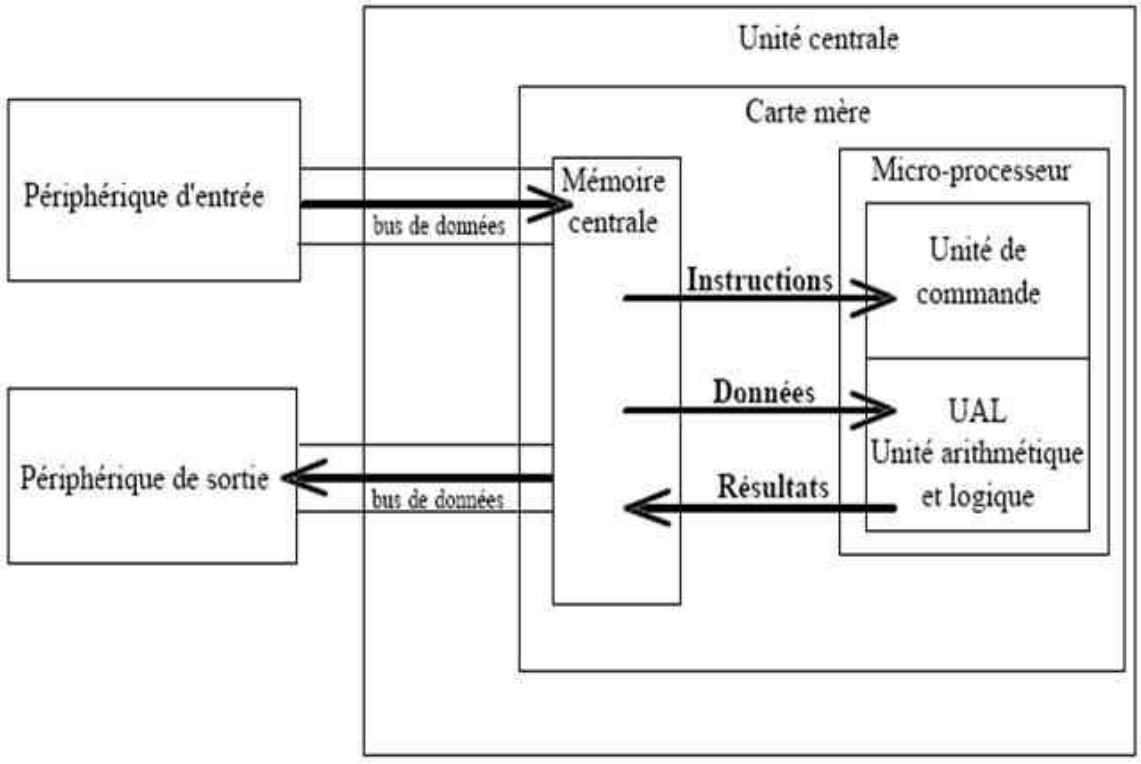
```
int *p = new int[100];
int *ip = p[40]; /* pointe sur l'élément no.41 */
ip++; /*avancer à la case d'après */
```

```
printf("%d",*ip); /* contenu de l'élément p[41] ! */
```

Rem : chaînes de caractères sont des tableaux de char: **`char[]`** où **`char*`**

Rappels : Accès à la mémoire - calculs simples

- les données sont stockées dans la mémoire vive : **RAM**
- les calculs se font dans la **CPU**
- la CPU dispose d'une (très petite) mémoire : **régistres**



[source : https://slideplayer.fr/slide/1310698/]

Une opération

```
C = A + 2*B;  /* par exemple*/
```

implique donc

- de lire les valeurs A et B de la mémoire (RAM → régistre CPU)
- d'effectuer la somme pondérée (CPU)
- d'écrire le résultat dans la mémoire de C (régistre CPU → RAM)

Rem: cet échange doit accompagner la lecture des instructions (code binaire), qui doivent aussi lues progressivement

Cela suggère donc plusieurs pistes pour accélérer les choses :

- *éviter* des attentes au niveau du processeur
  → '**pipelining**' : hardware; compilateur
  Illustration (simplifiée, en réalité pour instructions machine) :

```
a+=2; b/=2; c=sqrt(b)   /* successivement ...*/
a+=2; b/=2; x=sqrt(c)   * … ou pas, car pas de dépendance */
```

  → traiter plusieurs instructions (machine) en même temps permet de bien charger le processeur et de gagner du temps – si les instructions suivantes le permettent …

- *anticiper* la mise à disposition des données dans les régistres, en sorte de les avoir disponibles pour les opérations à suivre, en sorte de pouvoir effectuer plusieurs opérations simultanément

- *réduire* le nombre d'accès à la RAM en étant conservateur:
  garder les données disponibles au cas d'une nouvelle utilisation
  → exploiter la **mémoire cache** : par le compilateur, aidé par les flags de compilation (*i.e. vous !*)

- réduire les accès à la RAM en étant prévoyant :
  → charger des données autour de celles nécessaires, au cas où il y en aurait besoin : par le compilateur, aidé par les flags de compilation (*vous !)* et le code (v*ous !!*)

On voit plus clairement les enjeux pour un exemple légèrement plus complexe comme le suivant

Ex : calculer une combinaison linéaire de deux vecteurs

```
for (int i=0; i<dim; i++) {
      z[i] = a*x[i] + b*y[i] ;
}
```

Cela nécessite donc, *à chaque pas*, d'effectuer les opérations indiquées ci-dessus:
- lecture de x[i], y[i] dans la RAM
- exécution de l'opération numérique par la CPU
- écriture de z[i] dans la RAM

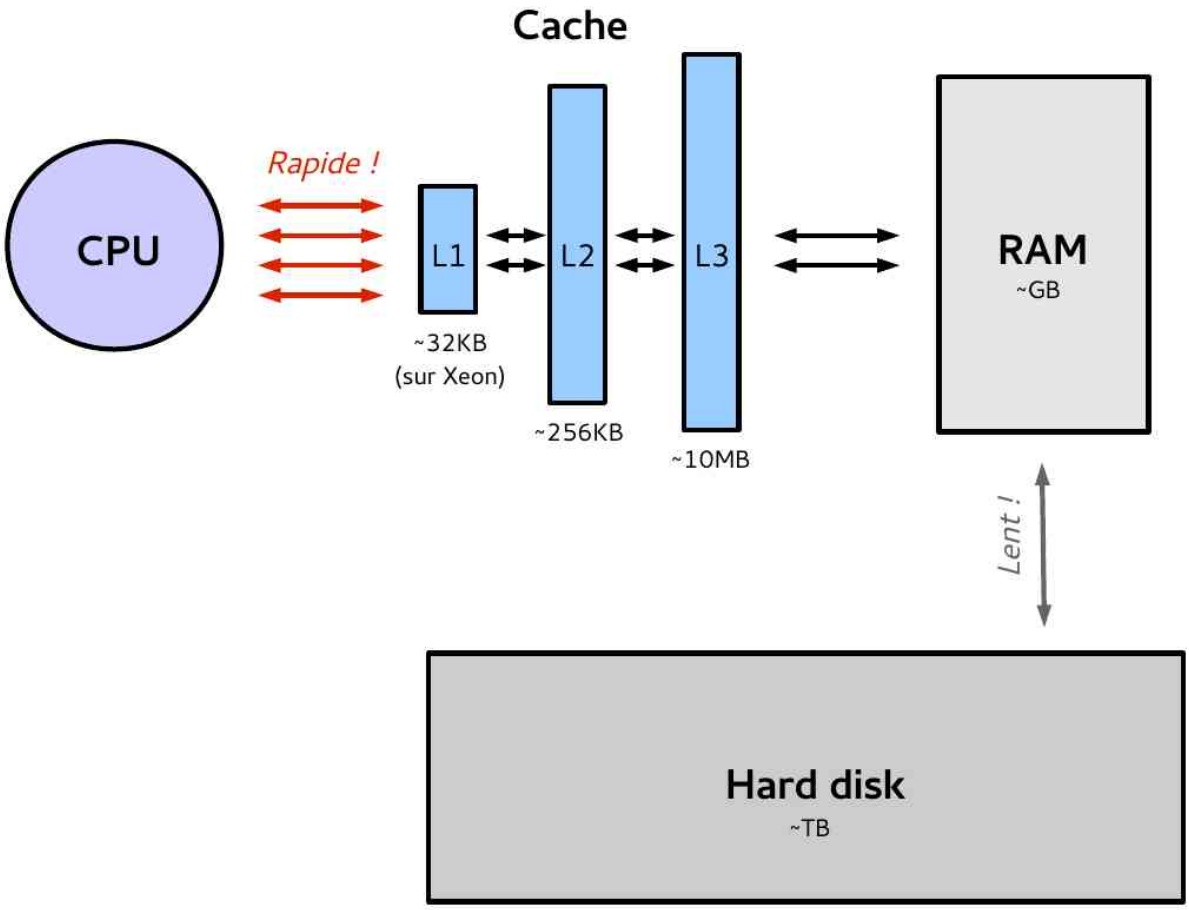
Rem: les valeurs de a, b peuvent être conservées dans un régistre de la mémoire pour toute la boucle

Rappels : Notion d'une mémoire cache et architecture résultante

Idée : garder l'information requise souvent à la portée des mains.

--> Illustration par l'étagère de D. Coslovich

→ Étagère 'tampon'
→ en mémoire, le tampon s'appelle 'cache'
→ elle existe à plusieurs endroits au niveau de l'architecture d'un ordinateur :



[source : D. Coslovich]

Ex : notion de *cache* entre cpu / mémoire RAM; entre RAM / disque dur; entre ordinateur / internet, ordinateur / clé usb, etc.

Accès à la mémoire par 'cache' - capacité, latence, bande passante

- ***capacité*** : taille de la mémoire, i.e. la quantité d'information qu'elle peut contenir
- ***latence*** : temps nécessaire pour transférer une donnée.
- ***bande passante*** : débit de transfert de données (bytes/sec)

Typiquement, on rechercherait les trois critères, mais ils sont contradictoires (et chers …)

→ agencement par plusieurs niveaux de cache :

	Latence (cycles CPU)	Bande passante	taille
CPU			
↕		~ 100 Gb/s	
L1	4 - 5		~100 kO
↕			
L2	dizaine		26kO – 2 MO
↕			
L3 (LLC)	centaine		1-8 MO
↕		~ 10 Gb/s	
RAM			

Rem : c'est un peu plus compliqué que cela …

- la cache L3 n'existe pas dans tous les PCs
- la gestion des niveaux de cache peut être organisée de manière *inclusive* (en cascade) ou de manière *exclusive* (par accès direct à chaque niveau) : la deuxième approche est plus simple à gérer mais moins performante
- l'attribution de la mémoire cache à miroiter des blocs peut se faire de manière plus ou moins complexe (*directe, N-fold, totale*) avec des avantages et des inconvénients
- la bande passante est aussi limitée par le 'bus', càd la composante électronique qui assure la transmission des données entre les emplacements de mémoire
- pour des processeurs multi-coeur, les caches L2 et L3 *peuvent* être partagées (par 2 processeurs, tous les processeurs, …) *ou non* ...
- etc etc etc

Un lien pour creuser : http://patrick.hede.free.fr/wolf/CNAM/lamemoirecache.htm

Rem: une difficulté dans la gestion de la mémoire cache est qui'il faut garantir la **cohérence du cache,** i.e. s'assurer que tous les changements qui se font dans le cache, soient faits de manière identique, que ce soit depuis la RAM vers les régistres de la cpu (LOAD) ou en sens inverse (STORE). Sinon : contradictions, pertes de données, …

Information sur l'architecture, cpu, cache etc

On peut obtenir des informations sur son système par

```
> more /proc/cpuinfo
```

ou, de manière encore plus détaillée, de l'information sur la cpu et les caches, dans un répertoire dédié

```
> cd /sys/devices/system/cpu
```

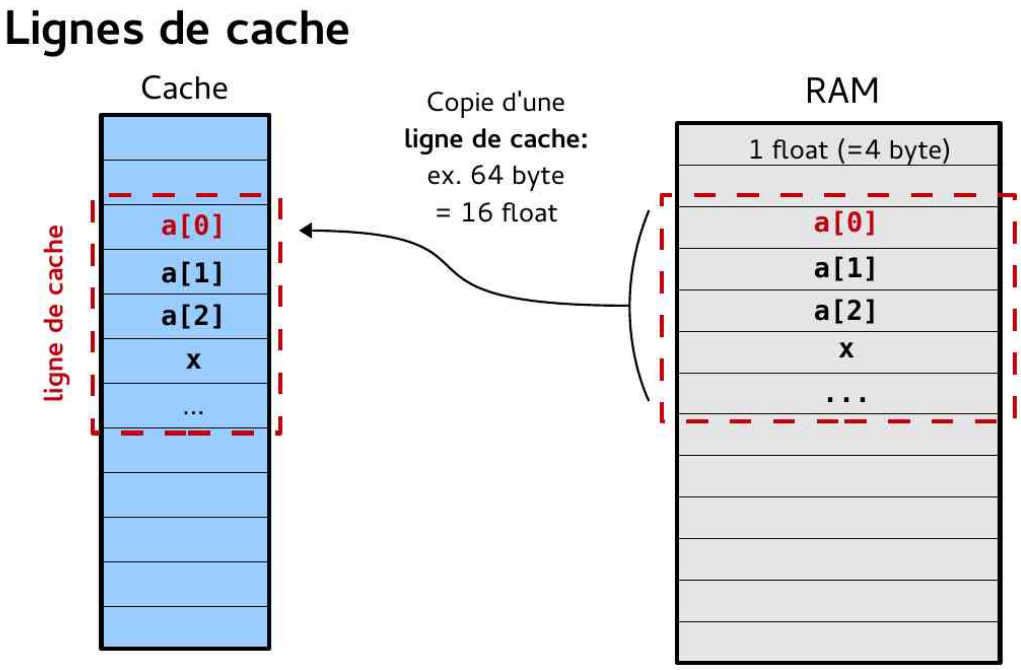
dans lequel l'information est organisé en fichiers/répertoire.
Par ex :

```
> grep . cpu0/cache/index*/size
> grep . cpu0/cache/index*/type
```

vous renseignera sur la taille et les types de mémoire cache de votre système etc.

Fonctionnement du cache

Une ***'ligne de cache'*** est attribuée, càd une zone contigue de la mémoire est sélectionnée pour être miroitée dans le cache.



Pour connaître la taille de la ligne de cache (en bytes) de votre machine

```
grep . /sys/devices/system/cpu/cpu0/cache/i*/coherency_line_size
```

[source : D. Coslovich]

Rem :

- plusieurs démarches possibles (mapping direct, associatif, etc)
- c'est l'attribution de la ligne de cache qui décidera de l'efficacité de la mémoire tampon !

Performance :

- la démarche consiste donc à ce qu'une recherche de données, avant d'être adressée à la RAM (lente), soit d'abord adressée à la mémoire cache (rapide): si elle est disponible, on a gagné du temps … (sinon, on en a perdu !)
- la réussite réside donc dans la réussite des recherches dans le cache (ratio hit/miss)
- on peut savoir ce taux de réussite par la commande **`perf`** (installer linux-tools-common sous ubuntu) ou par l'outil **`valgrind`**

```
> perf stat -e cycles,LLC-load-misses,LLC-loads, \
          L1-dcache-load-misses,L1-dcache-loads <executable>
```

Accès à la mémoire : tableaux + boucles; loop-interchange

Tableau 1d :

- zone ***contigue*** de mémoire pour stocker N variables du même type
- les éléments correspondent à des adresses de mémoire régulières

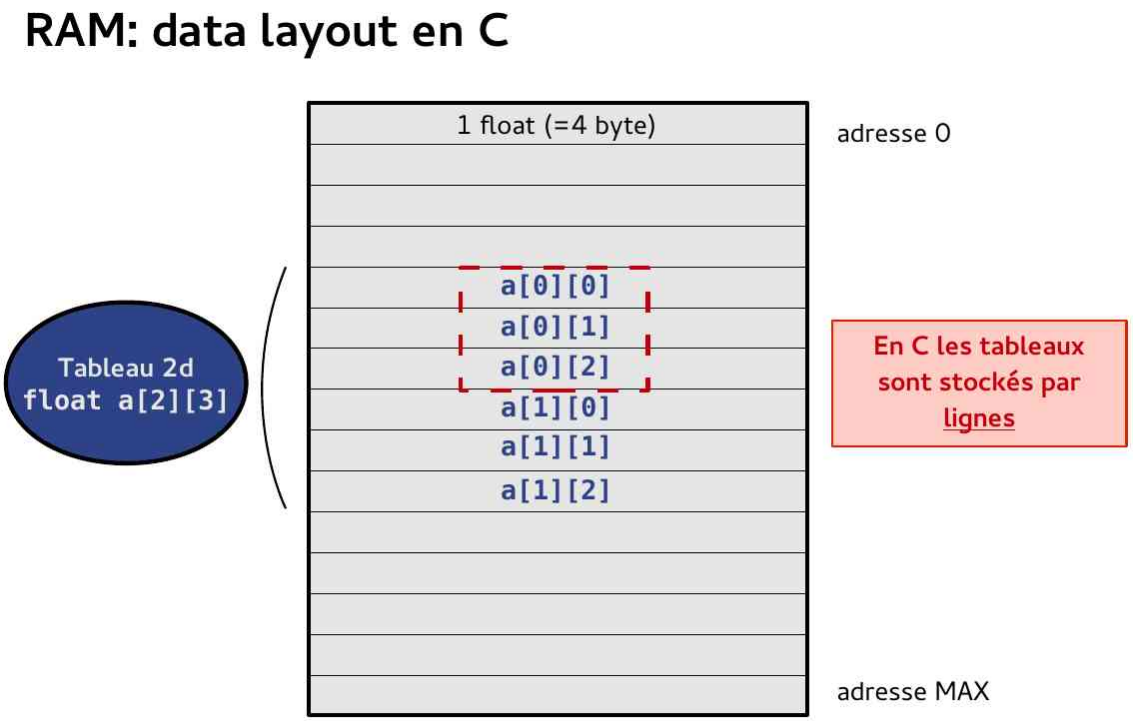
  Ex: incrément d'un octet pour un tableau de float – dans ce cas, les adresses sont des entiers simplement !
- parcourir le tableau est simple et efficace (cf. arithmétique des pointeurs en C !)

Pour un tableau 2d (ou plus !)

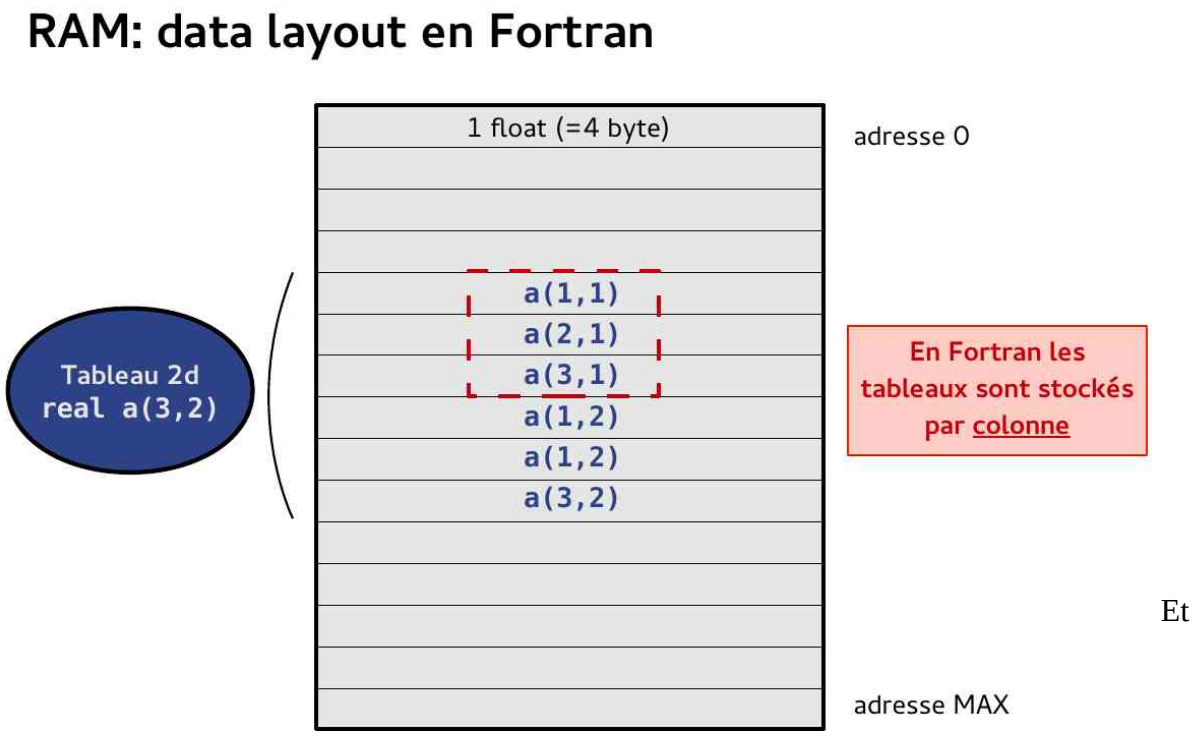
- toujours une zone contigue de mémoire
- l'adressage simple n'est plus possible (deux indices ...)
- en réalité, une conversion doit être faite entre un entier de [0..(NxM)-1] en mémoire en une paire d'indices [0..N-1),(0..M-1)] à utiliser dans le code
- on peut raisonner en 'tableau de tableaux'
- il faut donc décomposer, soit en 'lignes' soit en 'colonnes'

Le choix dépend : C vs. Fortran

En C : organisation des tableaux **par lignes**



En Fortran : organisation des tableaux **par <u>colonnes</u>**



Et pourquoi est-ce important ?

→ ***mémoire cache*** !

- lorsqu'un élément est sollicité dans la RAM, les éléments successifs seront aussi transférés au cache
- donc disponibles plus rapidement
- idéal pour l'itération, qui accède aux éléments dans l'ordre
- mais : seulement si on les utilise dans l'ordre adapté

```
/* bon (en C) : ligne par ligne*/
for  (i=0; i<ndim; i++){ /*ligne i*/
    for (j=0; j<ndim; j++) { /*colonne j*/
       b[i][j] = a[i][j];
    }
}
```

```
/* mauvais (en C) : colonne par colonne*/
for (j=0; j<ndim; j++) { /*colonne j*/
    for  (i=0; i<ndim; i++){ /*ligne i*/
       b[i][j] = a[i][j];
    }
}
```

À retenir donc: en C, c'est le ***dernier*** indice qui doit varier rapidement !

Rem : en Fortran, c'est donc le contraire …

Travaux pratiques : Jouer sur la hardware : le cache

But : illustrer quelques aspects de l'impact de la bonne utilisation de la hardware sur la performance

Exercice : loop interchange

Code fourni : ***loop-interchange-bad.c***

Recopiez ce code en ***loop-interchange-good.c***, puis adaptez-le pour optimiser la manière dont les tableaux sont parcourus.

- mesurez le temps d'exécution pour les codes, en considerant les deux variantes proposées pour les paramètres (nombre d'itérations et dimension des tableaux). Interprétez !
- confrontez votre analyse aux résultats obtenus avec le flag de compilation **-O3**. Explications ? Interprétations ? Interrogations ?
- finalement, utilisez le programme **perf** pour obtenir le pourcentage de *cache misses* au niveau de la cache L1 (vos machiens ne possèdent pas de cache L3). Aide : la commande **perf** que vous cherchez pourrait être

```
perf stat -e L1-dcache-load-misses,L1-dcache-loads ./a.out
```

Exercice : loop fusion

Codes fournis : ***loop-fusion-bad.c***

Recopiez ce code en ***loop-fusion-good.c***, dans lequel vous fusionnerez les boucles pour éviter de les re-traverser inutilement.
Refaire une analyse similaire à celle de l'exercice précédent et synthétiser votre compréhension de l'impact du cache.