Optimisation de code

Luca Cirrottola (INRIA)

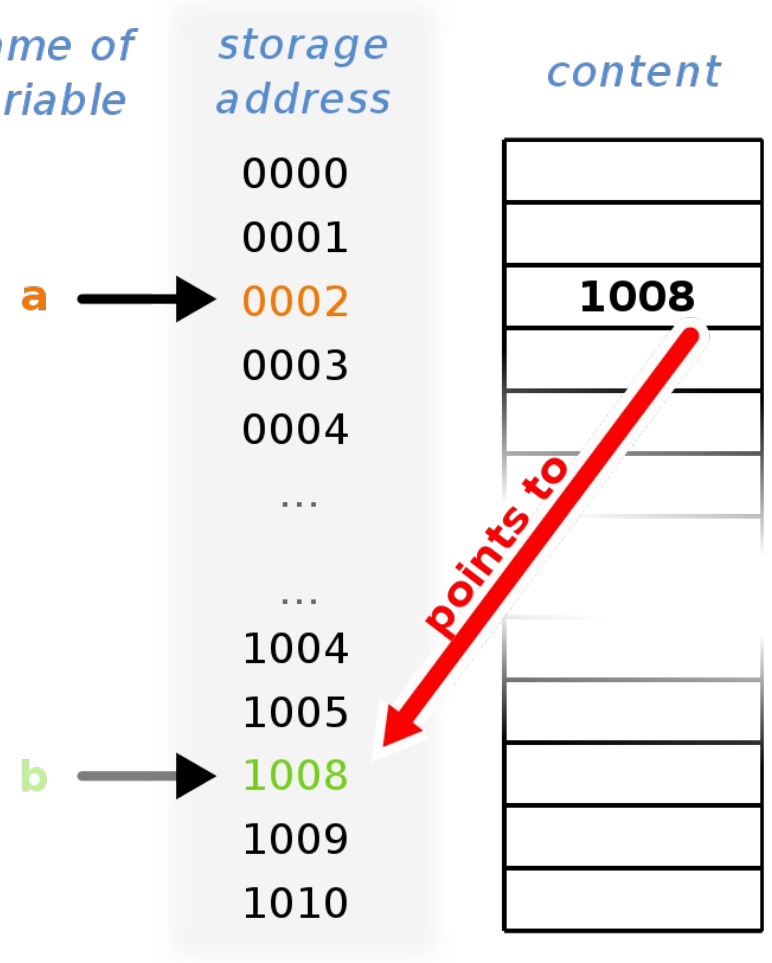
luca.cirrottola@inria.fr

Bordeaux INP ENSEIRB-MATMECA, Université de Bordeaux

Automne 2024

Avant de continuer….

Pointeurs



1. // C
2. int *a;
3. int b;
4.
5. a = &b;
6. // so: *a == b

https://en.wikipedia.org/wiki/File:Pointers.svg#/media/File:Pointers.svg

Vecteurs dynamiques

```
1. // C
2. int n = 1000;
3. int *A;
4.
5. A = (int*) malloc( n * sizeof(int) );
6. // access elements as A[0], A[1], … A[n-1]
```

Matrices (double vecteur) - 1

```
01.  // C
02.  int n = 1000;
03.  int **A;
04.  // Allocate one pointer per row
05.  A = (int**) malloc( n * sizeof(int*) );
06.  // Allocate each row pointer
07.  for ( int i = 0; i < n; i++ ) { // row
08.    A[i] = (int*) malloc( n * sizeof(int) );
09.  }
10.  // WARNING: no guaranties of memory contiguity from
     one row to another
```

Matrices (double vecteur) - 2

```
01. // C
02. int n = 1000;
03. int **A;
04. // Allocate one pointer per row
05. A = (int**) malloc( n * sizeof(int*) );
06. // Allocate the first pointer for the full matrix content
07. A[0] = (int*) malloc( n * n * sizeof(int) );
08. // Set the other pointers to point to the following rows
09. for ( int i = 1; i < n; i++ ) { // row
10.   A[i] = A[i-1] + n;
11. }
12. for ( int i = 1; i < n; i++ ) { // row
13.   for ( int j = 0; j < n; j++ ) { // column
14.     A[ i ][ j ] = 0;
15.   }
16. }
```

Matrices (vecteur singulier)

```
01. // C
02. int n = 1000;
03. int *A;
04. A = (int*) malloc( n * n * sizeof(int) );
05. for ( int i = 0; i < n; i++ ) { // row
06.   for ( int j = 0; j < n; j++ ) { // column
07.     A[ i*n+j ] = 0;
08.   }
09. }
```

- Optimisation de code
- Outils d’analyse des performances

Optimisation de code

Optimisation de code

- *Pourquoi ?*
  - Pour accélerer notre programme
  - Pour mieux utiliser les ressources de calcul (qui sont limitées!).
- *Comment ?*
  1. ***Test*** (“dans quelles conditions le programme est inefficient?”)
  2. ***Profilage*** (“dans quels algorithmes on passe le plus de temps/on consomme le plus de mémoire?”)
  3. ***Optimisation*** (“on modifie les algorithmes les plus critiques pour les performances”)

Test → Profilage → Optimisation

Optimisation de code

- Par expérience:

  ”We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.” *(D. Knuth, “Structured Programming with go to Statements”)*

- L’optimisation demande du temps (au développeur).
- Ce temps devrait être dédié aux parties les plus critiques du code.

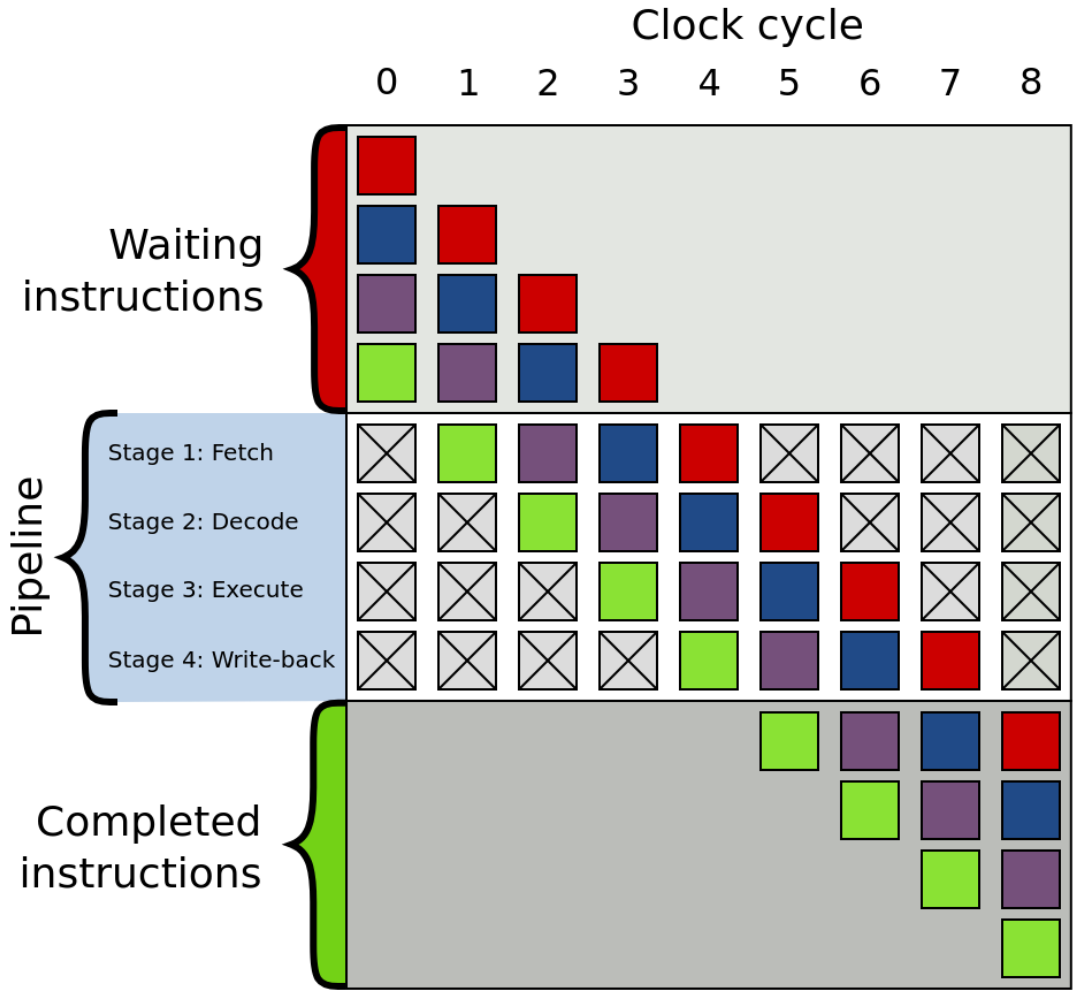
Test → Profilage → Optimisation

Optimisation de code

- Par qui?
  - Le *processeur* essaie déjà d'optimiser l'exécution du programme.
  - Le *compilateur* fait déjà des optimisations de code.
  - Le *développeur* arrive après.

Au moment de l’exécution

- *Instruction pipeline*:



https://commons.wikimedia.org/wiki/File:Pipeline,_4_stage.svg

Au moment de l'exécution

- Example:

```
for ( int i = 0; i < N; i++ ) {
  some_function( &condition, i );
  if ( condition == true ) {
    do_something();
  }
}
```

  - *Pipelining*
  - *Speculative execution*
    - *Branch prediction*
    - …
  - …

Optimisation par le compilateur

- Examples:
  - *inlining*
  - *loop transformations*
    - *loop unrolling*
    - *loop interchange*
    - *vectorization*
    - …
  - …

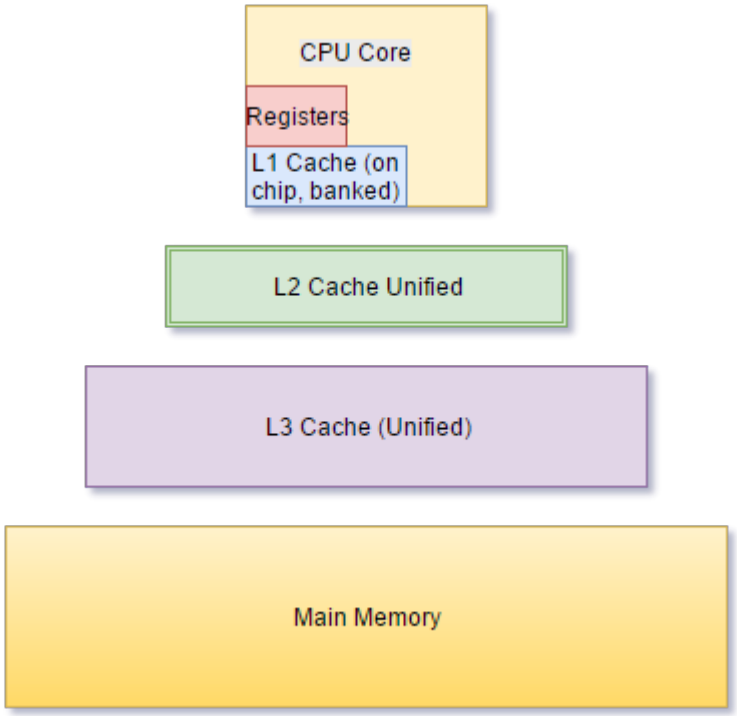
Optimisation par le développeur

- Examples:
  - *loop transformations*
    - *loop fusion*
    - *loop fission*
    - *loop interchange*
    - *loop nest optimization (tiling/blocking)*
  - …

Mémoire physique

- Beaucoup de ces optimisations concernent les *accès mémoire*.
- En effet la vitesse des processeurs a grandi beaucoup plus vite que celle des mémoires.
  - Le processeur risque de passer plus du temps à attendre des données qu'à faire des calculs.
  - => "Rapprocher" la mémoire au processeur pour des accès plus rapides?

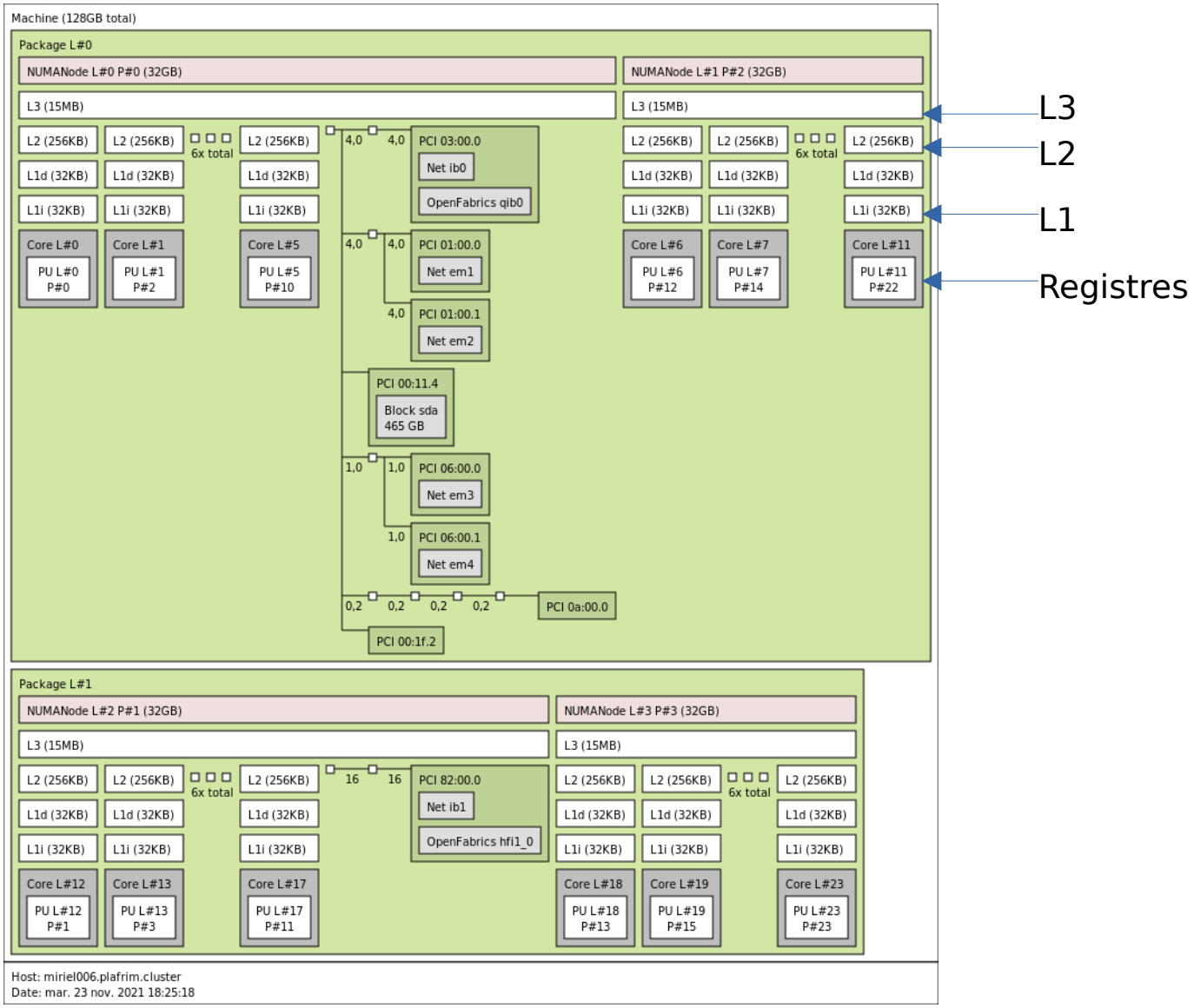
Mémoire physique



- Hiérarchie de mémoires *de plus en plus "proches" du processeur* (donc plus rapides).
- Le niveau inférieur (plus petit) contient normalement une partie des donnés du niveau supérieur.

https://commons.wikimedia.org/wiki/File:Cache_Hierarchy_Updated.png

Mémoire physique

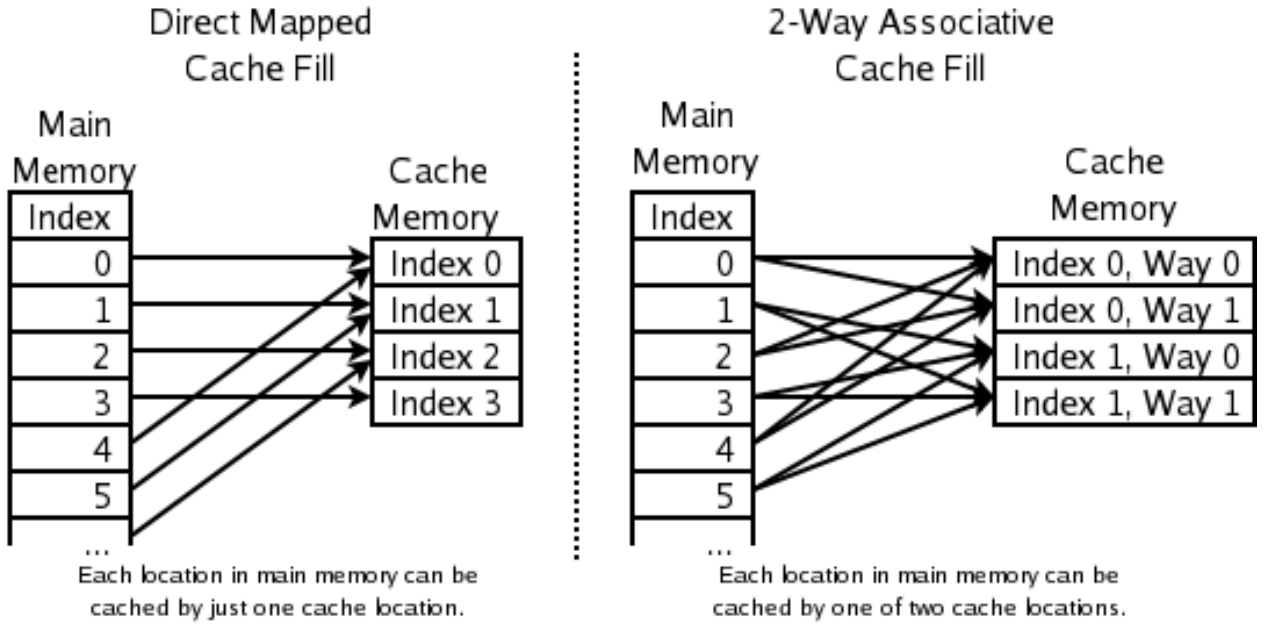


Mémoire physique

Storage Area	Register	L1 Cache	L2 Cache	RAM	Swap
Cycles to Access	≤ 1	≈ 3	≈ 14	≈ 240	$\approx 10^7$
Town	Talence	Pessac	Cestas	Toulouse	Mars

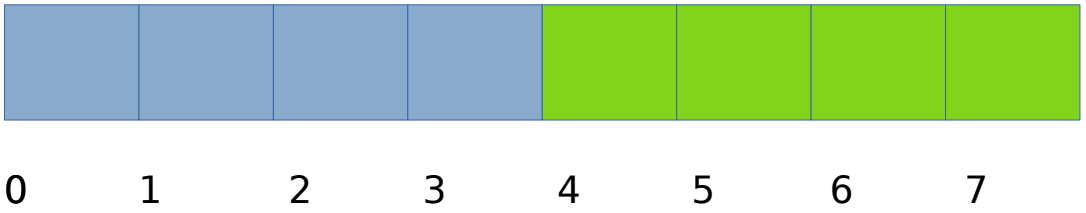
Correspondance

- Cache *chargée par* ***lignes***.
- Chaque ligne du niveau supérieur peut aller dans une (ou plus) ligne de mémoire cache.



https://en.wikipedia.org/wiki/CPU_cache#/media/File:Cache,associative-fill-both.png

Localité

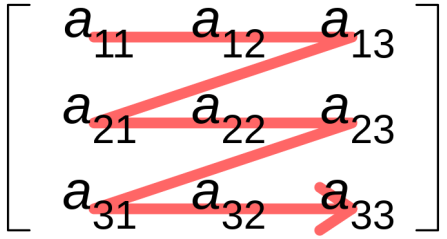


- **Localité spatiale**: utiliser des données contigues en mémoire.
- **Localité temporelle**: re-utiliser une même donnée le plus possible avant de passer à la suivante.
- ***Cache hit***: la donnée recherchée est déjà dans la ligne de cache utilisée.
- ***Cache miss***: la donnée recherchée n'est pas dans la ligne de cache utilisée.

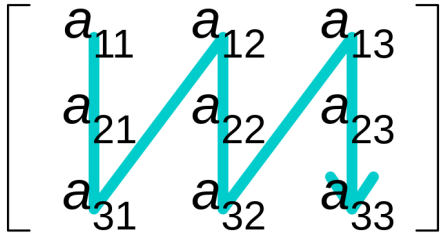
Localité

- **Example: $A_{ij} = B_{ij} + C_{ij}$**
- **Faut-il boucler d'abord sur l'indice i (ligne) ou j (colonne)?**
  - Fortran: stockage des matrices par colonnes.
  - C: stockage des matrices par lignes.
- **Fortran: j-i.**
- **C: i-j.**

Row-major order



Column-major order



https://commons.wikimedia.org/wiki/File:Row_and_column_major_order.svg

Outils d’analyse des performances

Outils

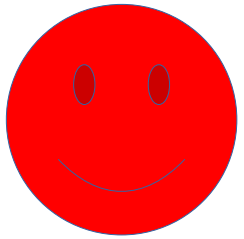
- **En bref**
  - Les ressources de calculs sont limitées
  - Des parties de code insoupçonnées peuvent ralentir tout le programme / consommer beaucoup de ressources
- **Démarche**
  - Trouver les zones critiques (*profilage*)
  - Améliorer l'implémentation ou changer d'algorithme (*optimization*)

Outils

- **Rappel:**
  - Effectuez l’optimisation:
    - Sur une version complète et testée du code
    - Sur une version optimisée par le compilateur (-O2, -O3)
    - Avec des paramètres / des entrées du programme réalistes

Outils

- **Basique...**
  - printf(“%i”,time(NULL)) ;
  - `time ./program`



- ***Profilers*:**
  - *Call stack sampling*
  - Instrumentation des appels à fonctions
  - Simulation hardware
  - Compteurs hardware

Outils

- **Exemples:**
  - gprof (compilation avec -pg)
    - *Flat profile, call graph*..
    - Demande une recompilation
  - Valgrind
    - memcheck, massif, cachegrind, callgrind
    - Assez lent ...
  - Performance Application Programming Interface (PAPI)
    - Compteurs hardware

Outils

- **Exemples:**
  - gprof (instrumentation *statique*)
    - Par échantillonage
  - Valgrind (instrumentation *dynamique*)
    - Memcheck – heap profiler
    - Callgrind – graphe des appels à fonction
    - Cachegrind – interaction avec mémoire cache
  - PAPI
    - Compteurs hardware

Usage

Usage

- **[Rappel:] Les performances sont beaucoup liées à la localité de la mémoire...**

Storage Area	Register	L1 Cache	L2 Cache	RAM	Swap
Cycles to Access	≤ 1	≈ 3	≈ 14	≈ 240	$\approx 10^7$
Town	Talence	Pessac	Cestas	Toulouse	Mars