Modèles et techniques en programmation parallèle hybride et multi-cœurs

Quelques notions d'optimisation séquentielle

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

12/09/2023

Deux “règles d’or” :

- Avant/après optimisation, avant/après parallélisation, il faut s’assurer que le code donne des résultats corrects.
- Avant de paralléliser un code et/ou avant d’optimiser le parallélisme, il faut optimiser la version séquentielle d’un code.

On parle en général de version Debug et Release d'un code :

- version Debug : avec les options de compilation (-g -O0 par exemple avec gcc/g++) qui désactivent les optimisations faites par le compilateur, qui ajoutent dans le code des points de repère pour exécuter un code instruction par instruction, testent la validité des indices de vecteurs, affichent la valeur de variables en cours d'exécution, etc.
- version Release : avec les options de compilation (-O2 ou -O3 par exemple avec gcc/g++) qui permettent au compilateur d'optimiser lui-même le code, et en enlevant tous les contrôles.
- En général, la version Release sera (beaucoup) plus rapide que la version Debug.

- Pendant l'écriture, les tests et la correction d'un code, on utilise en général une version Debug.
- A la fin, on fournit aux utilisateurs une version Release.
- Bien vérifier que la version Release donne les mêmes résultats que la version Debug (on a parfois des surprises).

Pour évaluer l'optimisation d'un code, on utilise des outils de mesure de performance. Il existe de nombreux moyens de mesurer le temps d'exécution de code ou de parties de code :

- Commande unix `time` : mesure globale (temps ressenti par l'utilisateur)
- Fonctions définies par le langage de programmation et utilisables depuis l'intérieur du code :
  - `second(...)` (fortran),
  - `gettimeofday(...)` (C/C++),
  - `timespec_get()` (C),
  - `std::clock()` (C++),
  - `time.time()` (python)
  - `tic/toc` (matlab),
  - ...

  Permet de mesurer le temps d'exécution d'un groupe d'instructions.
  Penser à vérifier dans la documentation quelle est la précision des mesures.

- Librairies qui fournissent des fonctions utilitaires, par exemple MPI, OpenMP, PAPI
  On ajoute dans le code des appels à ces fonctions.
  - MPI et OpenMP proposent des fonctions pour mesurer le temps de calcul : `MPI_Wtime()`, `omp_get_wtime()`.
  - PAPI

(`https://github.com/icl-utk-edu/papi/wiki/`)

est une librairie qui donne des informations très précises.

Permet de consulter des compteurs système très bas niveau (par exemple : nombre d'opérations, utilisation des caches, registres, etc.)

- Outils externes de “profilage”

  Ajoutent automatiquement des points de mesure dans le code (gprof), s’interposent entre le code et le système pour récupérer des informations (valgrind, perf, vtune (Intel), AMD μProf (AMD), etc.)

  Permet de connaître des informations intermédiaires : nombre d’appels et temps moyen d’exécution de fonctions par exemple (gprof, valgrind).
  Certains sont plus précis et descendent au niveau de l’instruction (perf, vtune)

En général, il faut utiliser les outils de profilage avec une version “Debug” (même s’ils peuvent fonctionner en version “Release”, ils donneront alors beaucoup moins d’informations).

Voir *Exemple0* pour un exemple d'utilisation de perf et gprof

Programmation séquentielle efficace

Pour obtenir un code efficace (en temps d'exécution), il faut:

- **utiliser les algorithmes les plus efficaces possible (pas couvert par ce cours)**
- **organiser le placement des données (améliorer la localité spatiale)**
- **organiser la séquence d'instructions (améliorer la localité temporelle)**
- **écrire les instructions pour qu'elles soient les plus rapides possibles (éviter de calculer plusieurs fois la même expression, éviter si possible les tests (instructions 'if'), etc.)**

Les compilateurs peuvent améliorer l'efficacité d'un code (par des options particulières) mais il vaut mieux le faire soi-même.

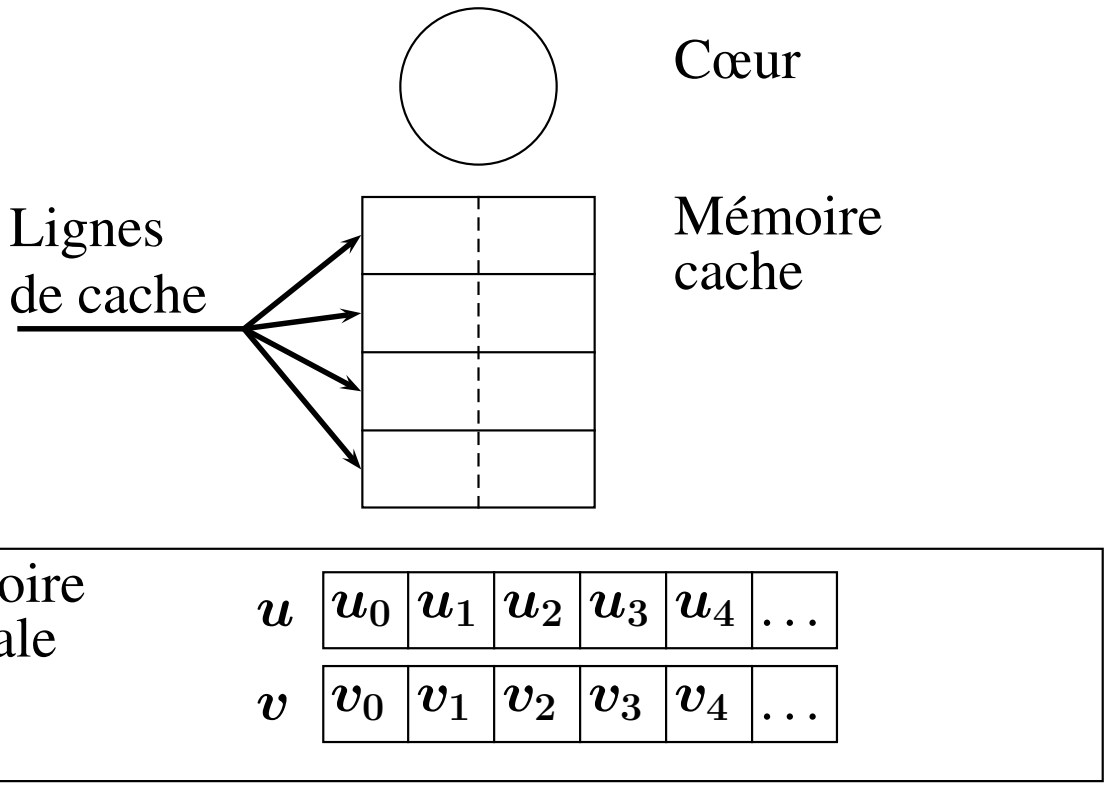
Une bonne utilisation de la mémoire est très importante pour l'optimisation de code.

Exemple : si `u` et `v` sont des vecteurs de taille `n > 4`, on veut calculer la boucle :

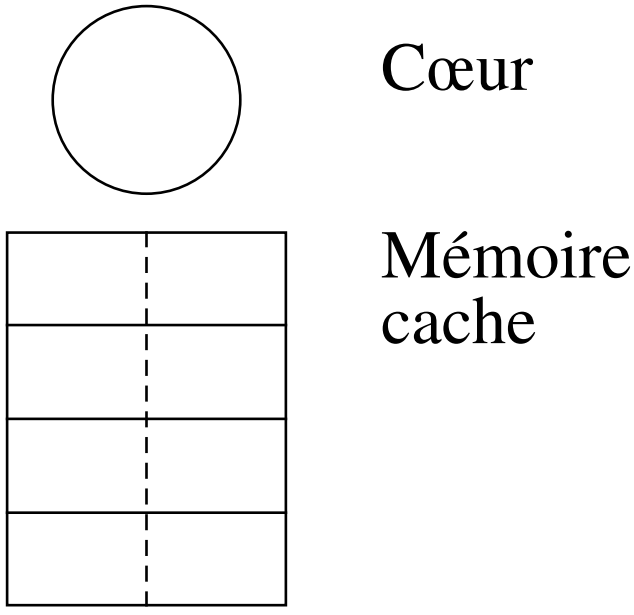
$$\begin{aligned} & v_0 = u_0 \\ & v_n = u_n \\ & for(i = 1; i < n - 1; i++) \\ & \quad v_i = (u_{i-1} + 2 * u_i + u_{i+1})/4; \end{aligned} \tag{1}$$

Supposons un système **idéalisé** par:

- un processeur (qui contient un seul cœur),
- un seul niveau de mémoire cache de taille 8 nombres réels (réparti en 4 lignes de cache de taille 2 nombres réels),
- la mémoire centrale



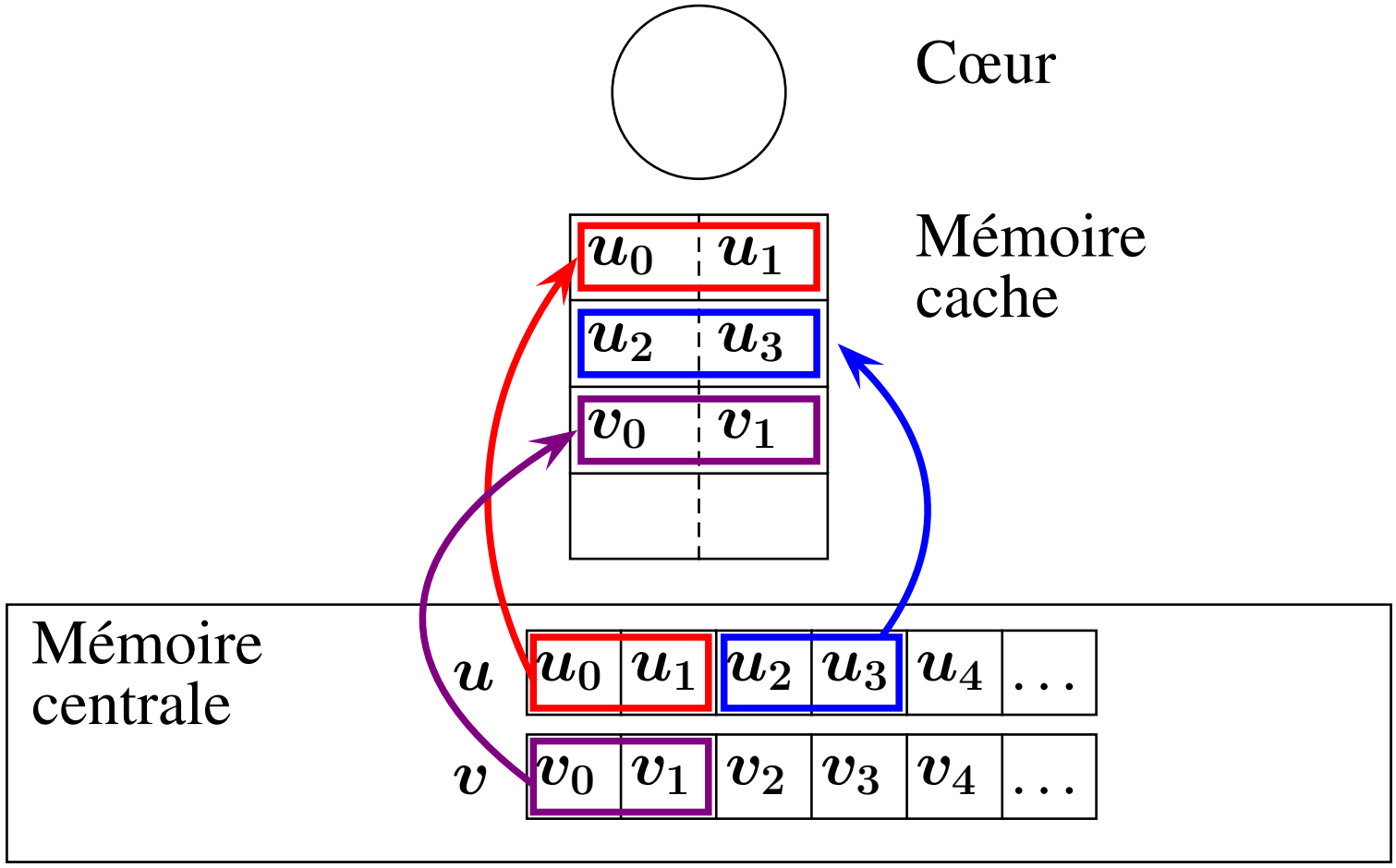
1. Avant d'exécuter $v_1 = (u_0 + 2 * u_1 + u_2)/4$:



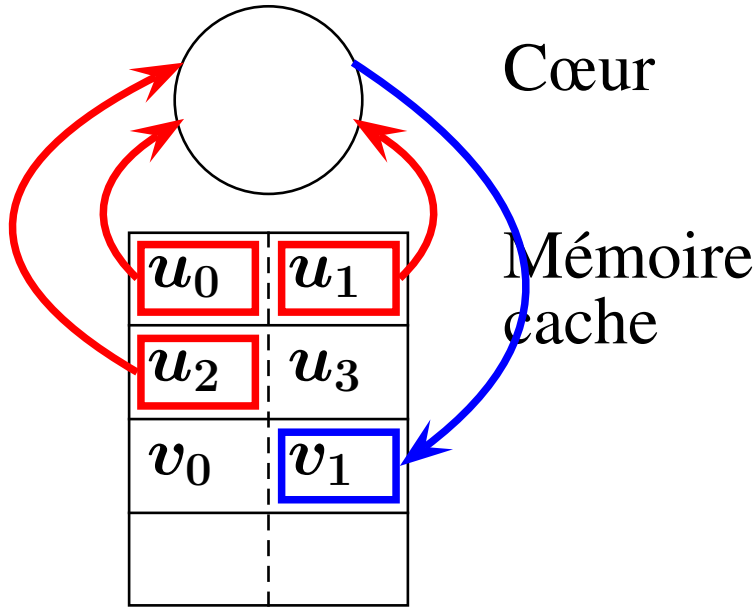
Mémoire centrale

u	u_0	u_1	u_2	u_3	u_4	$\ldots$
v	v_0	v_1	v_2	v_3	v_4	$\ldots$

2. Les blocs contenant u_0, u_1, u_2 et v_1 (3 blocs) sont copiés dans la mémoire cache :



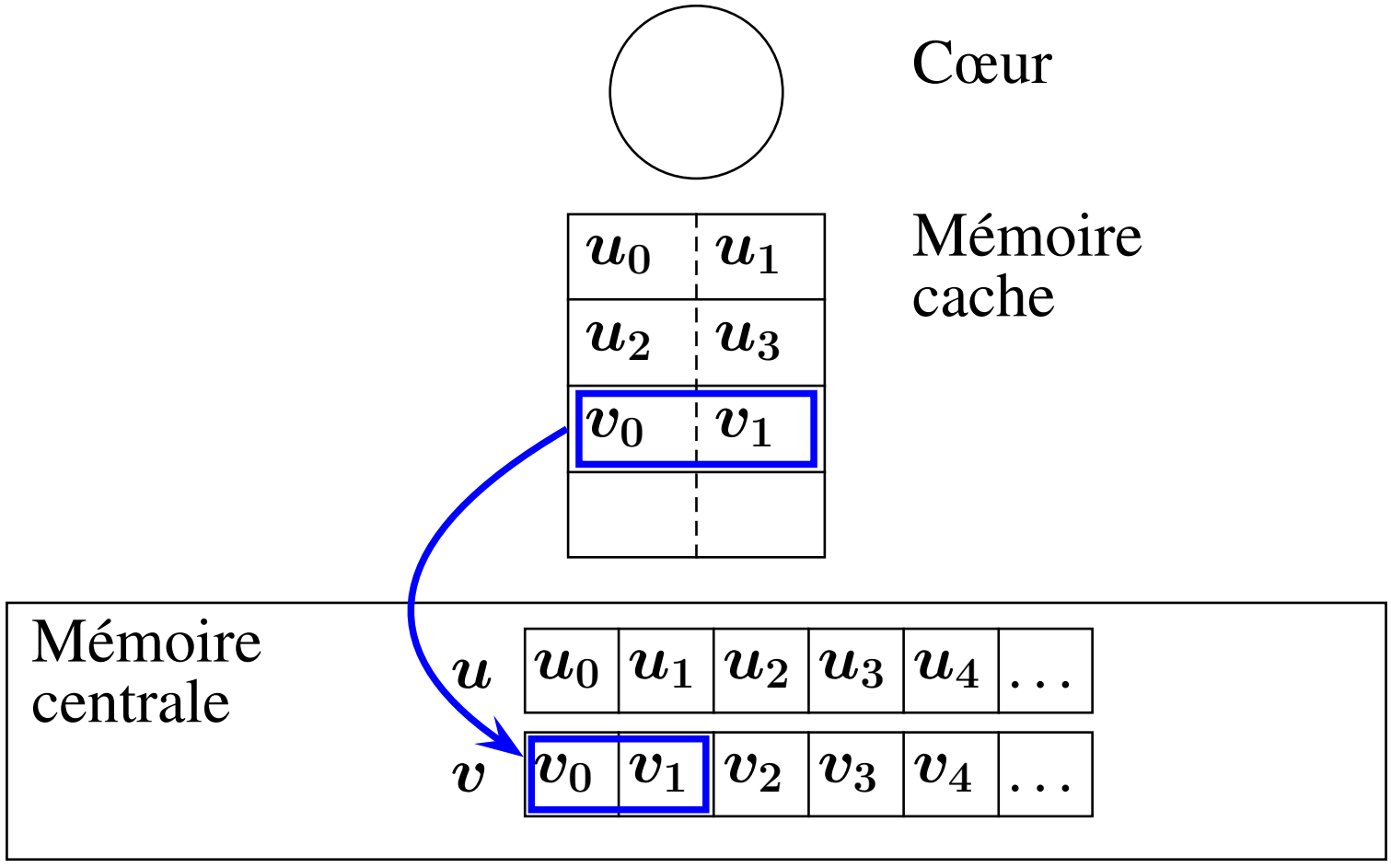
3. Le cœur utilise les copies de u_0, u_1, u_2 de la mémoire cache, calcule l'expression et place le résultat dans la mémoire cache :



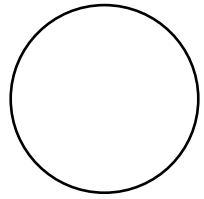
Mémoire centrale

u	u_0	u_1	u_2	u_3	u_4	...
v	v_0	v_1	v_2	v_3	v_4	...

4. Le bloc contenant le résultat est recopié dans la mémoire centrale :



5. Le calcul de l'instruction suivante $v_2 = (u_1 + 2 * u_2 + u_3)/4$ peut commencer:



Cœur

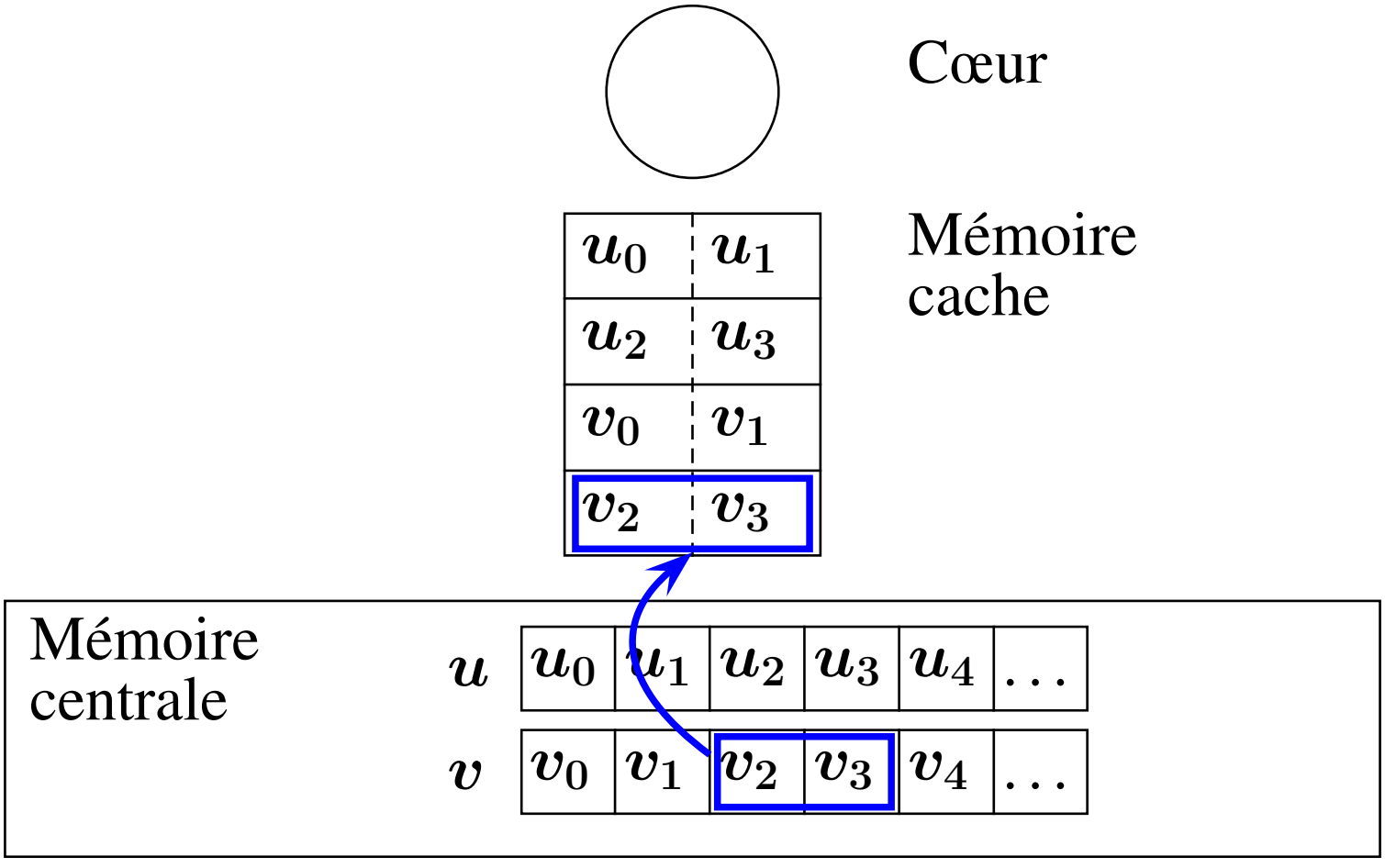
u_0	u_1
u_2	u_3
v_0	v_1

Mémoire cache

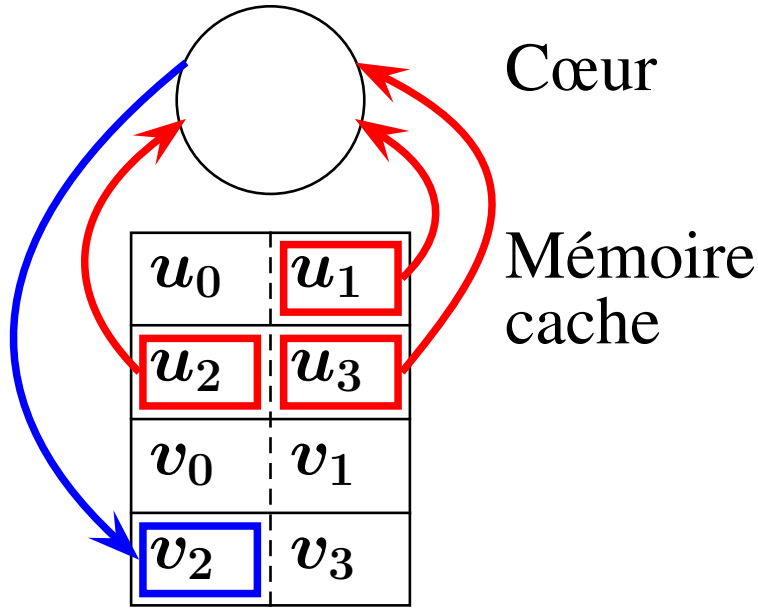
Mémoire centrale

u	u_0	u_1	u_2	u_3	u_4	$\ldots$
v	v_0	v_1	v_2	v_3	v_4	$\ldots$

6. Les composantes de u nécessaires sont déjà dans la mémoire cache, seul le bloc contenant la composante v_2 doit être copié dans la mémoire cache :



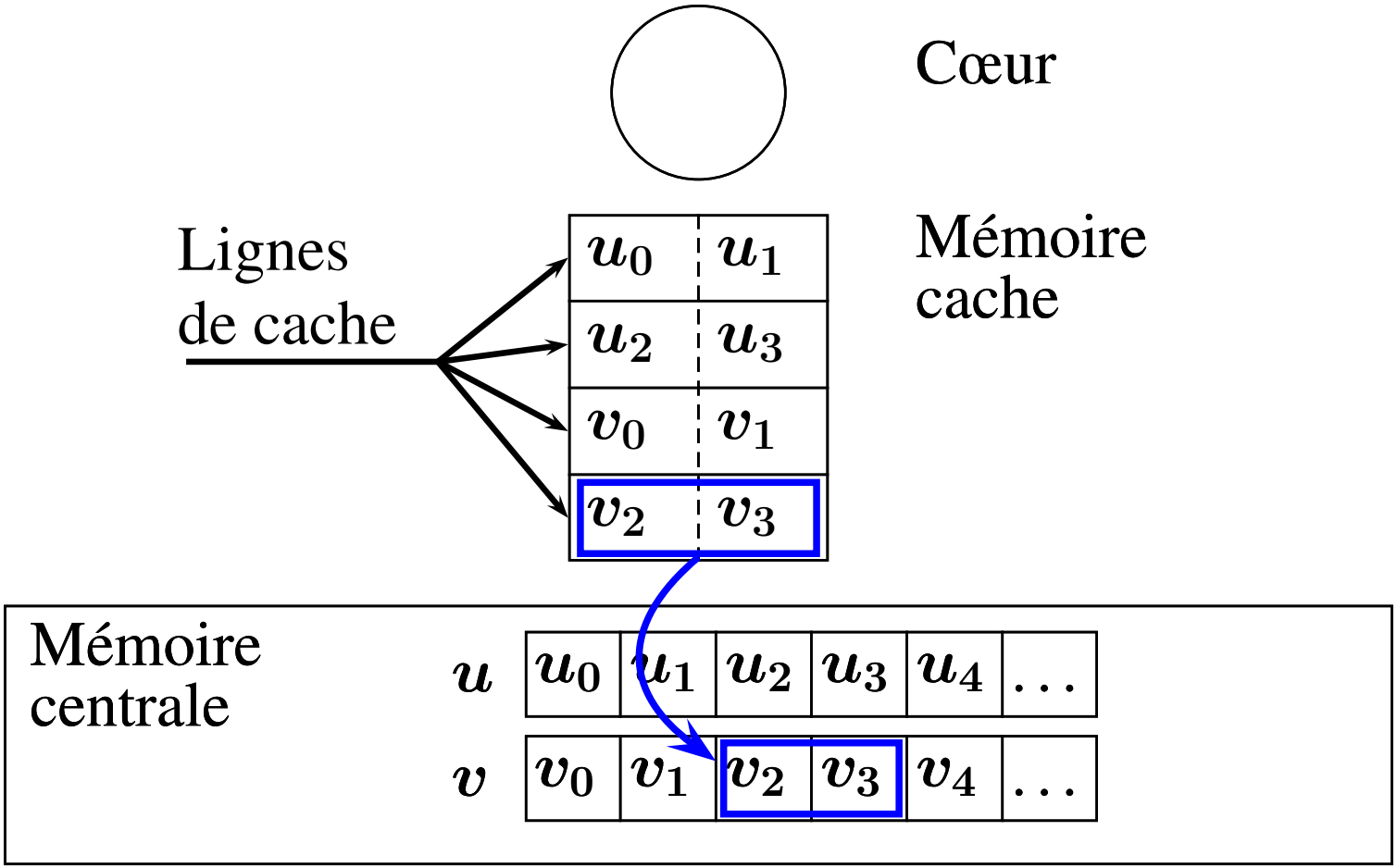
7. Le cœur utilise les copies de u_1, u_2, u_3 de la mémoire cache, calcule l'expression et place le résultat dans la mémoire cache :



Mémoire centrale

u	u_0	u_1	u_2	u_3	u_4	...
v	v_0	v_1	v_2	v_3	v_4	...

8. Le bloc contenant le résultat est recopié dans la mémoire centrale :



En résumé :

- La première instruction $v_1 = (u_0 + 2 * u_1 + u_2)/4$ utilise
  - 4 transferts (lents) mémoire centrale - mémoire cache
  - 4 transferts (rapides) mémoire cache - cœur
- La deuxième instruction $v_2 = (u_1 + 2 * u_2 + u_3)/4$ utilise
  - 2 transferts (lents) mémoire centrale - mémoire cache
  - 4 transferts (rapides) mémoire cache - cœur

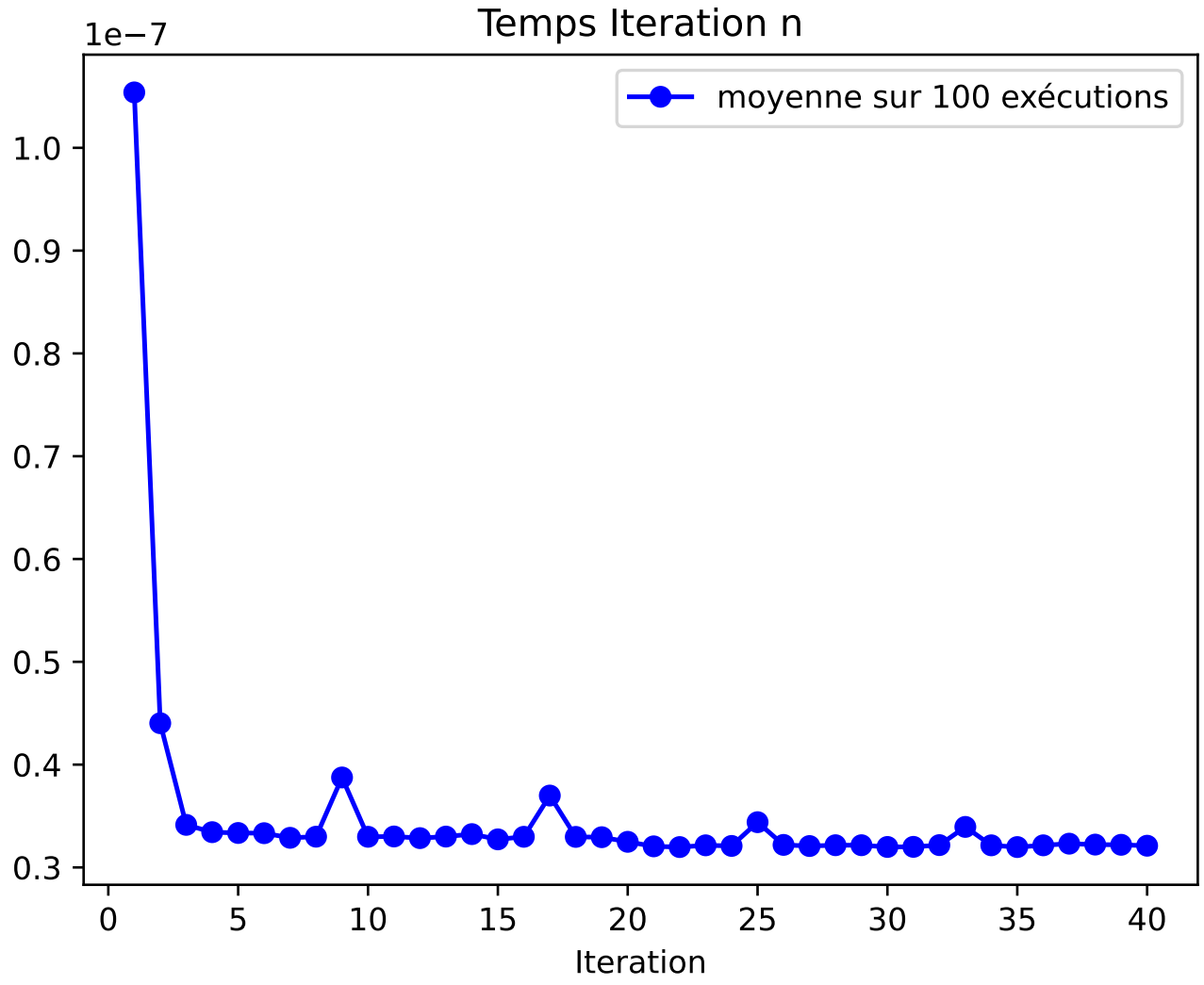
Attention : cet exemple est (très) simplifié : en général la mémoire cache est de taille plus grande que dans l'exemple et, de plus, il y a plusieurs types de mémoire cache dans un ordinateur

Dans les processeurs actuels, les lignes de cache ont une taille de quelques dizaines d'octets (entre 32 et 128 octets).

Examiner et tester l'exemple *Exemple1*
Lire le fichier Exemple1/README.txt pour des instructions de compilation et d'exécution.

- La première instruction prend plus de temps parce qu'elle remplit la mémoire cache.
- Sans mémoire cache, toutes les itérations prendraient (à peu près) le même temps que la première.
- Beaucoup d'instructions suivantes n'auront pas besoin de chercher toutes les valeurs dans la mémoire centrale.

La mémoire cache améliore beaucoup le temps de calcul de presque toutes les itérations.



La première itération prend 3 x plus de temps que les itérations suivantes.

On remarque aussi que les itérations d'indices $1 + k \times 8$ prennent un peu plus de temps que les autres:

> *C'est la conséquence de transferts supplémentaires mémoire cache - mémoire centrale (la taille d'une ligne de cache L3 est 64 octets* $= 8\times$ *la taille d'un double C++).*

Pour connaître la taille d'une ligne de cache, on peut taper la commande:

```
getconf -a | grep LINESIZE
```

On voit donc qu'il est important de bien utiliser la mémoire cache.

Pour obtenir le maximum de performances, il faudrait programmer en connaissant exactement l'organisation des mémoires cache.
Un code qui utilise l'information sur la mémoire cache de façon portable (indépendante de la machine), est (très) compliqué à écrire.

C'est pourquoi, on essaie d'appliquer les règles suivantes : optimiser les localités spatiale et temporelle.

Localité spatiale

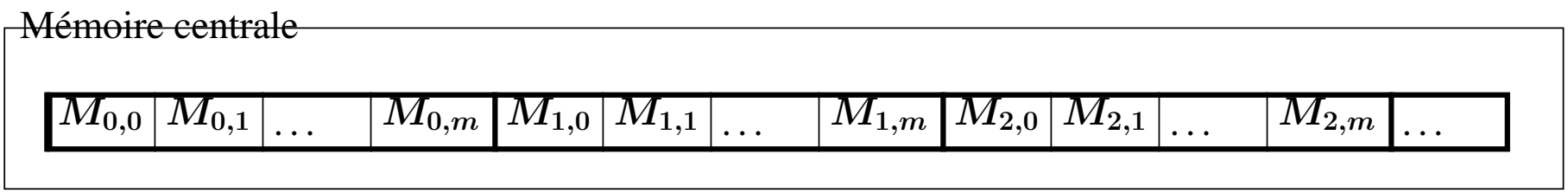
Règle: autant que possible, utiliser des zones mémoires proches les unes des autres dans une séquence d'instructions

Le transfert entre mémoire centrale et mémoire cache se fait par bloc de données.
Donc, si une donnée est à côté d'une donnée qui vient d'être utilisée (et donc transférée en mémoire cache), un nouveau transfert sera peut-être inutile.

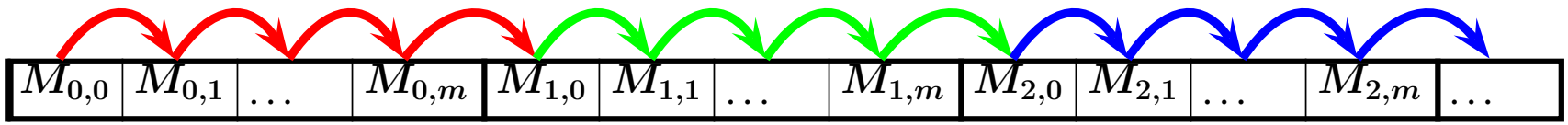
Voir l'Exemple2 : addition de 2 matrices.
Lire le fichier Exemple2/README.txt pour des instructions de compilation et d'exécution.

Le code compare 2 algorithmes de parcours des coefficients de matrice : lignes par lignes et colonnes par colonnes.

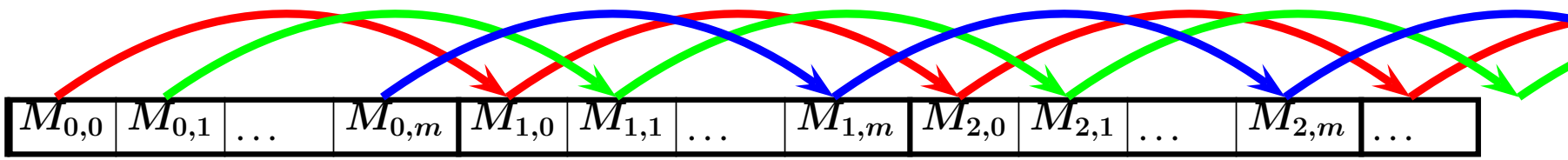
Dans cet exemple, une matrice $M(m, n)$ est rangée dans la mémoire centrale ligne par ligne:



Parcours ligne par ligne (flèches rouges, vertes puis bleues):



Parcours colonne par colonne (flèches rouges, vertes puis bleues):



Exécuter l'exemple 2 en tapant:

```
./ex_1_2.exe
```

qui additionne 2 matrices de taille 1024×1024 et comparer les temps de calcul.

Question.

Ré-éxécuter le code en ajoutant un paramètre entier :

```
./ex2.exe n
```

avec $n = 8, 16, 32, 64, 128, 256, 512, 1024, 2048$ *qui additionne des matrices de taille* $n \times n$.

Expliquer les résultats

Localité temporelle

Règle: autant que possible, les instructions qui utilisent une même zone mémoire, doivent s'exécuter de façon rapprochée dans le temps

> *La mémoire cache étant de petite taille, le gestionnaire mémoire, s'il a besoin de place, effacera dans la mémoire cache les données les plus anciennes.*
> *Si une donnée est utilisée par plusieurs instructions proches dans le temps, elle sera maintenue plus longtemps en mémoire cache (et donc nécessitera moins de transferts)*

Voir l'Exemple3
Lire le fichier Exemple3/README.txt pour des instructions de compilation et d'exécution.

Utilisation du // interne au processeur

En général, on laisse le compilateur essayer d'optimiser le code pour ce type d'amélioration, mais on peut essayer d'appliquer les règles suivantes (il peut y en avoir d'autres) :

- dérouler les boucles (si les instructions à l'intérieur de la boucle contiennent des opérations arithmétiques)
  utiliser au maximum la présence de plusieurs unités de calcul (additionneurs, multiplicateurs, etc) dans un cœur
- essayer d'éviter les tests
  simplifier le travail du processeur (enchaînement des instructions le plus déterministe possible).

Voir Exemple 4 : Déroulement de boucles (exemple inspiré de la librairie d'analyse numérique LAPACK)

Lire le fichier Exemple4/README.txt pour des instructions de compilation et d'exécution.

Dans calcul1, une itération de la boucle de longueur n
contient 1 addition et 1 multiplication.
Dans calcul2, une itération de la boucle de longueur $n/4$
contient 4 additions et 4 multiplications.

Donc l'ordinateur peut calculer simultanément certaines des opérations en fonction du nombre de circuits d'addition et/ou de multiplication disponibles.

A noter que calcul2 contient plus d'opérations sur les entiers que calcul1. Malgré cela, calcul2 est tout de même plus rapide que calcul1.