Modèles et techniques en programmation parallèle hybride et multi-cœurs

Quelques notions d'optimisation séquentielle

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

10/08/2020

Deux “règles d’or” :

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

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++),
  - `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, par exemple MPI, OpenMP, PAPI
  On ajoute dans le code des appels à des fonctions de la librairie.
  - MPI et OpenMP proposent des fonctions pour mesurer le temps de calcul : `MPI_Wtime()`, `omp_get_wtime()`.
  - PAPI

(`https://icl.cs.utk.edu/projects/papi/wiki/Main_Page`)

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), 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)

Les outils de mesure perturbent les temps de calcul et, en général, il faut les utiliser avec une version “debug”. Ils donnent seulement une indication sur les endroits du code les plus intéressants à optimiser. De toute façon, à la fin, il faut mesurer les temps calculs sur la version “release” (on a parfois des surprises) .

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, "faciliter le travail du compilateur")**

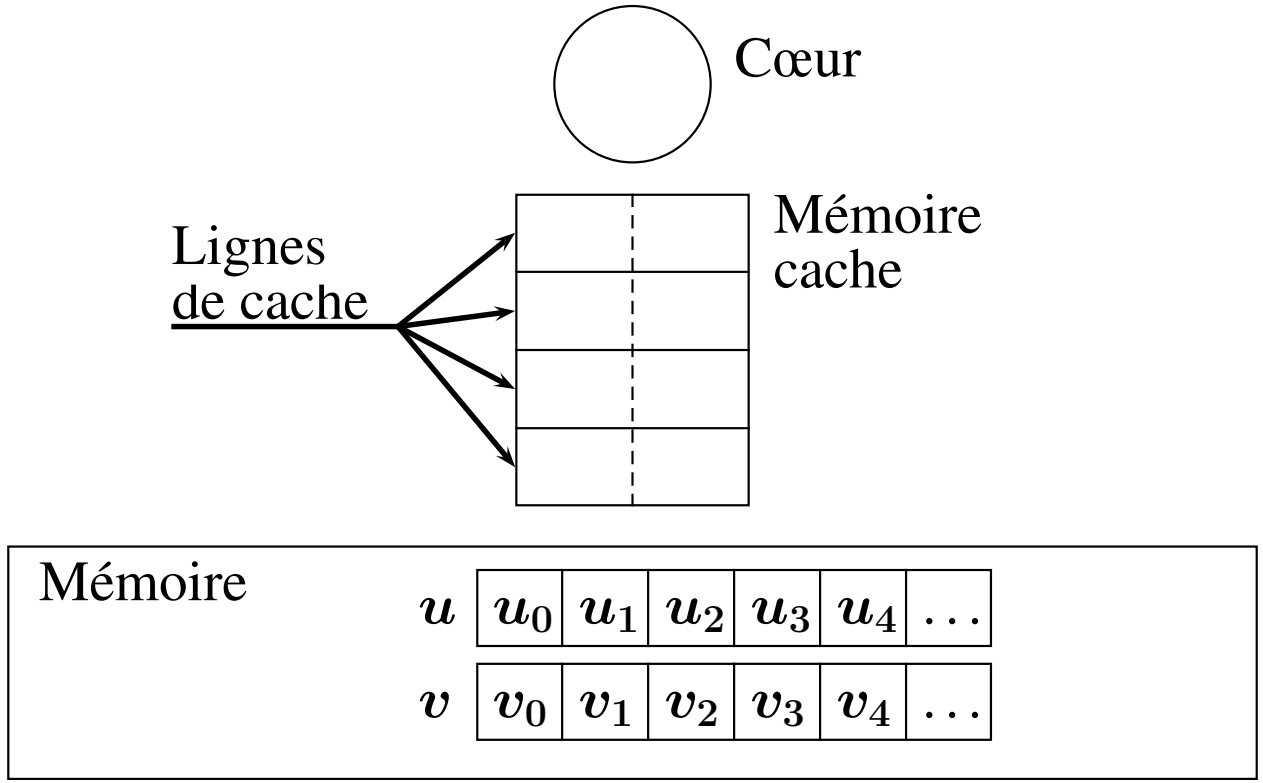
Une bonne utilisation de la mémoire est très importante pour l'optimisation de code.

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

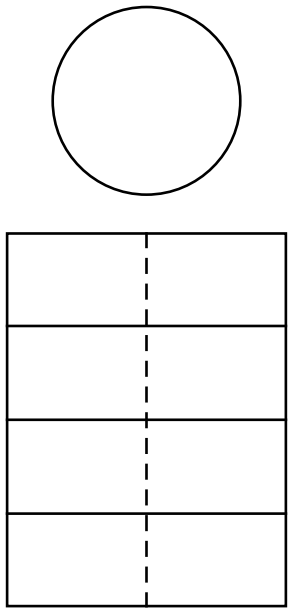
$$\begin{aligned} &for(i = 1; i < n - 1; i++) \\ &\quad v_i = (u_{i-1} + 2 * u_i + u_{i+1})/4; \end{aligned} \tag{1}$$

Supposons un système idéalisé par:

- un processeur,
- une mémoire cache de taille 8 (4 lignes cache de taille 2),
- la mémoire centrale

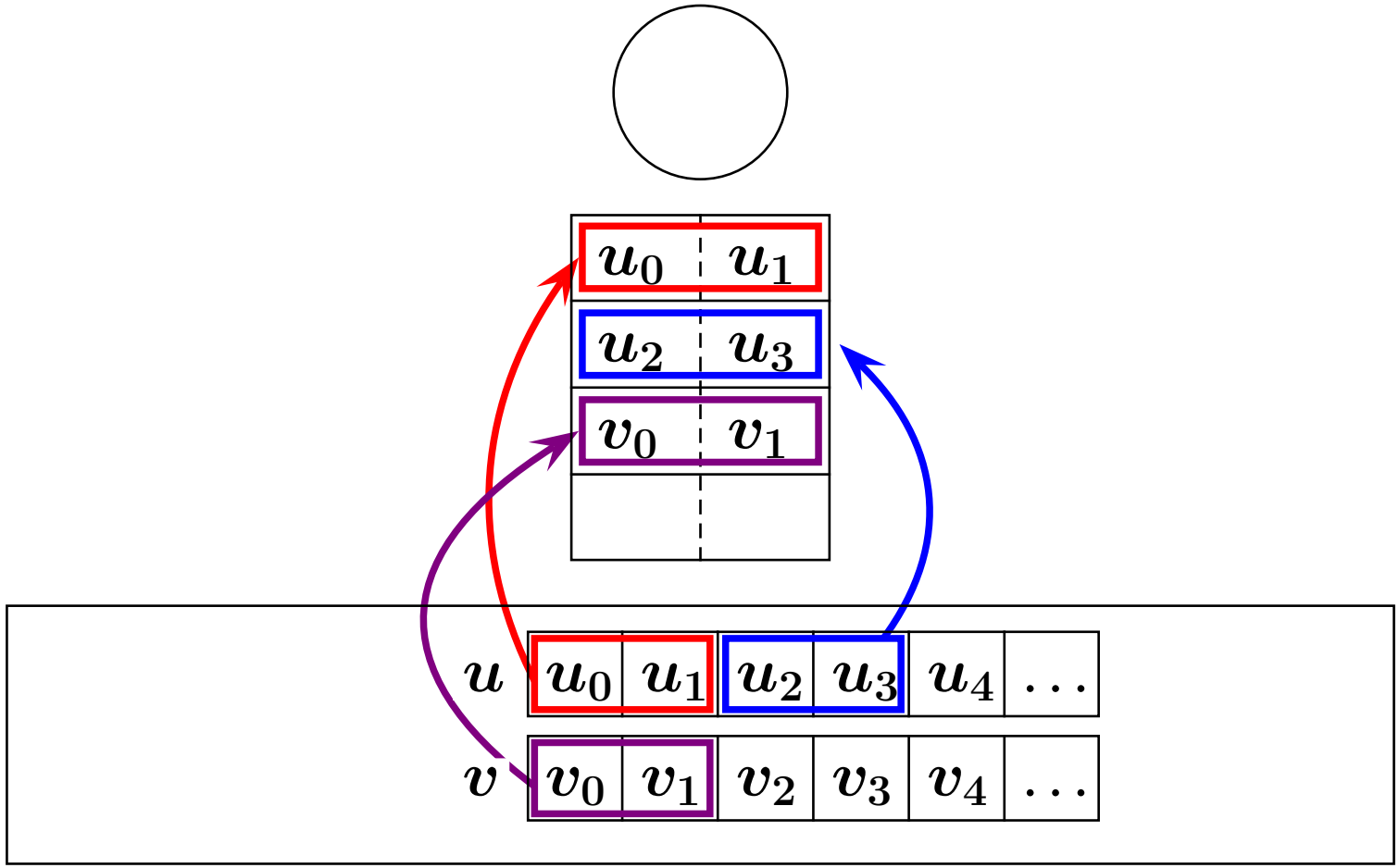


1. Avant d'exécuter $v_1 = (u_0 + 2 * u_1 + u_2)/4$:

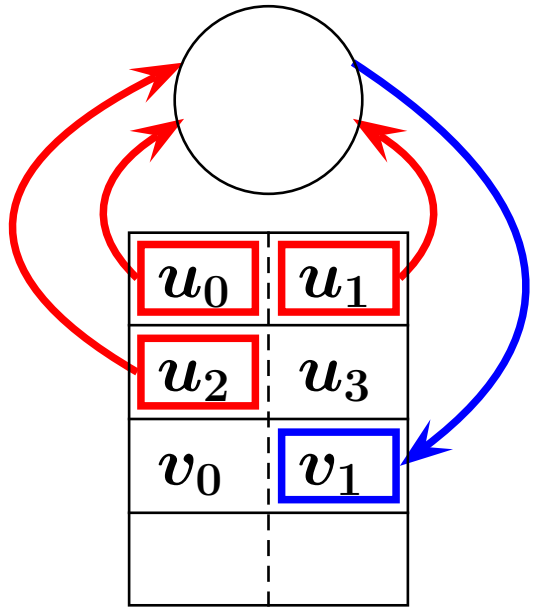


$\boldsymbol{u}$	$\boldsymbol{u_0}$	$\boldsymbol{u_1}$	$\boldsymbol{u_2}$	$\boldsymbol{u_3}$	$\boldsymbol{u_4}$	$\boldsymbol{\ldots}$
$\boldsymbol{v}$	$\boldsymbol{v_0}$	$\boldsymbol{v_1}$	$\boldsymbol{v_2}$	$\boldsymbol{v_3}$	$\boldsymbol{v_4}$	$\boldsymbol{\ldots}$

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

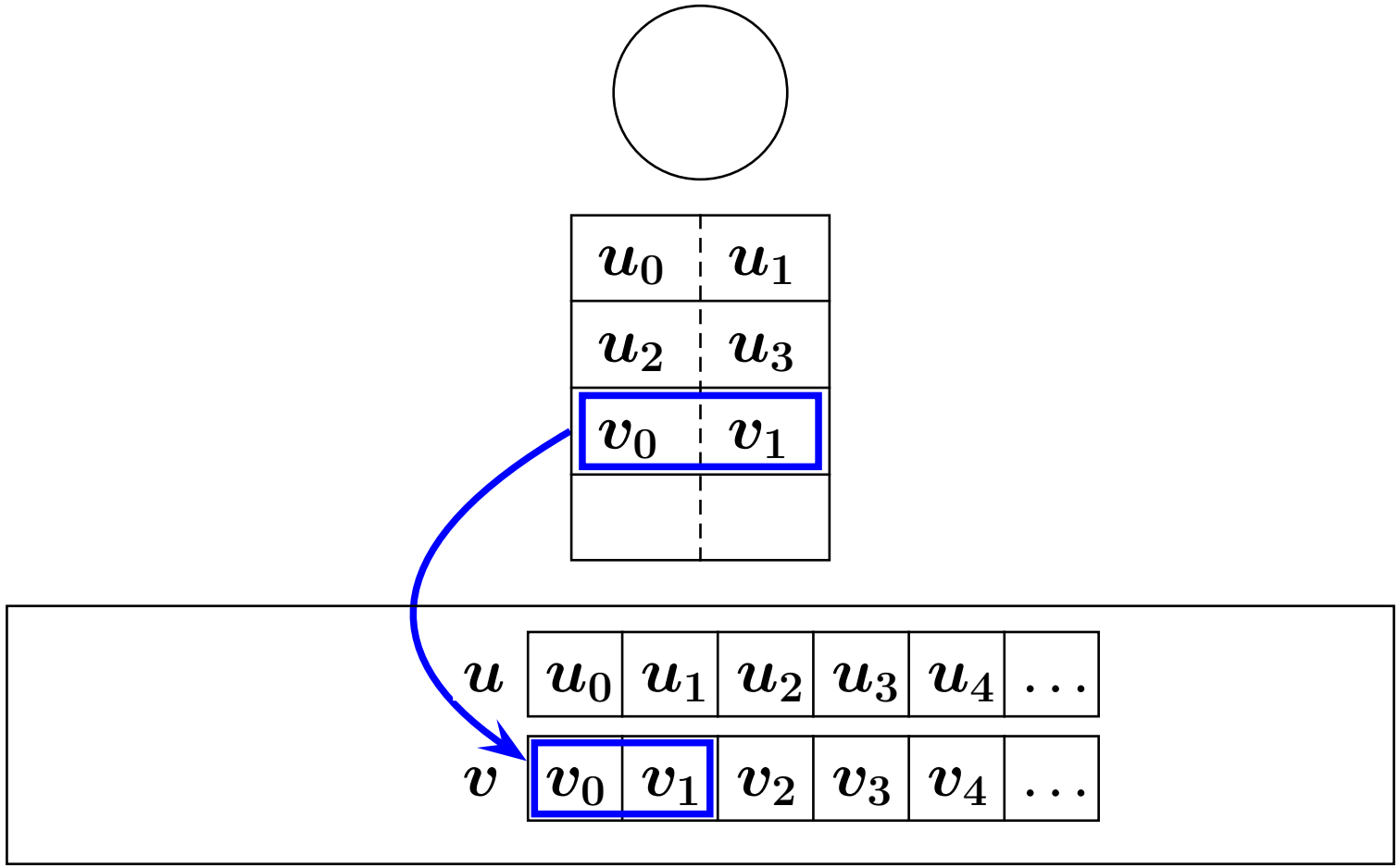


3. Le cœur utilise les copies de u_0, u_1, u_2 de la mémoire cache, calcule l'expression et place le résultat dans la mémoire cache :

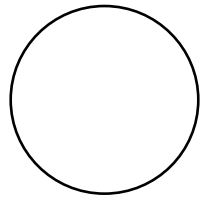


u	u_0	u_1	u_2	u_3	u_4	$\ldots$
v	v_0	v_1	v_2	v_3	v_4	$\ldots$

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



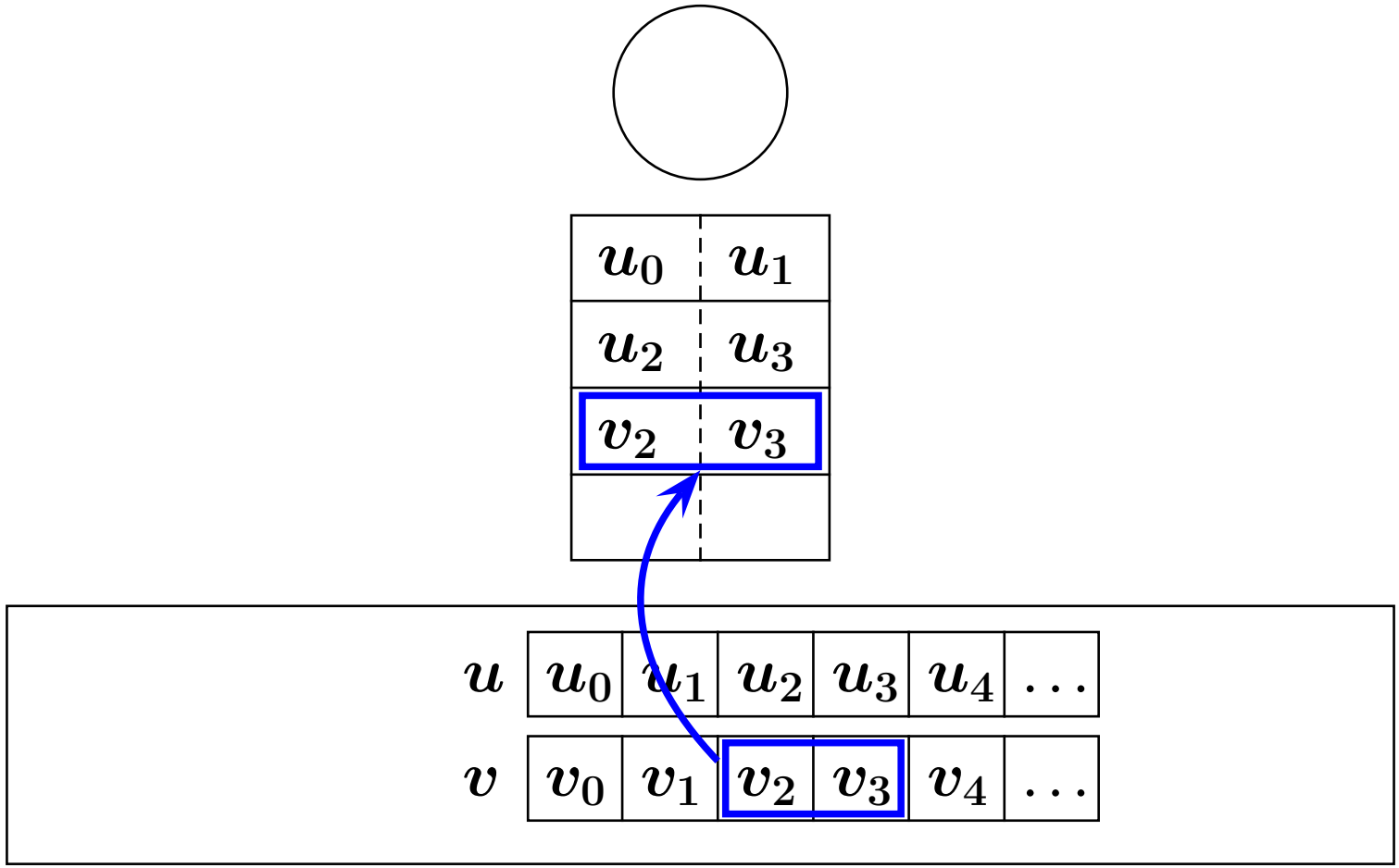
5. Le calcul de l'instruction suivante $v_2 = (u_1 + 2 * u_2 + u_3)/4$ peut commencer:



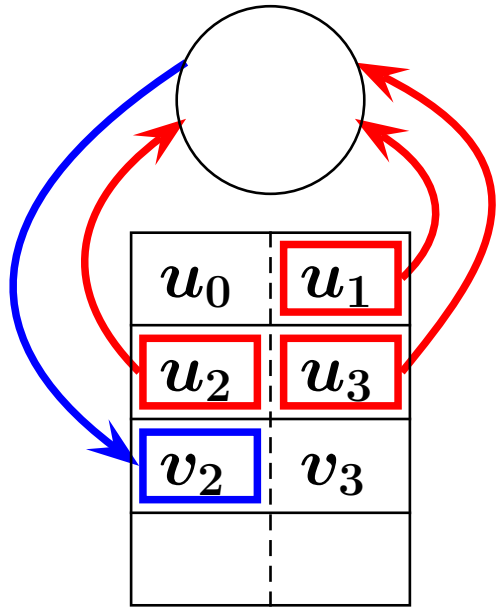
u_0	u_1
u_2	u_3
v_0	v_1

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 :

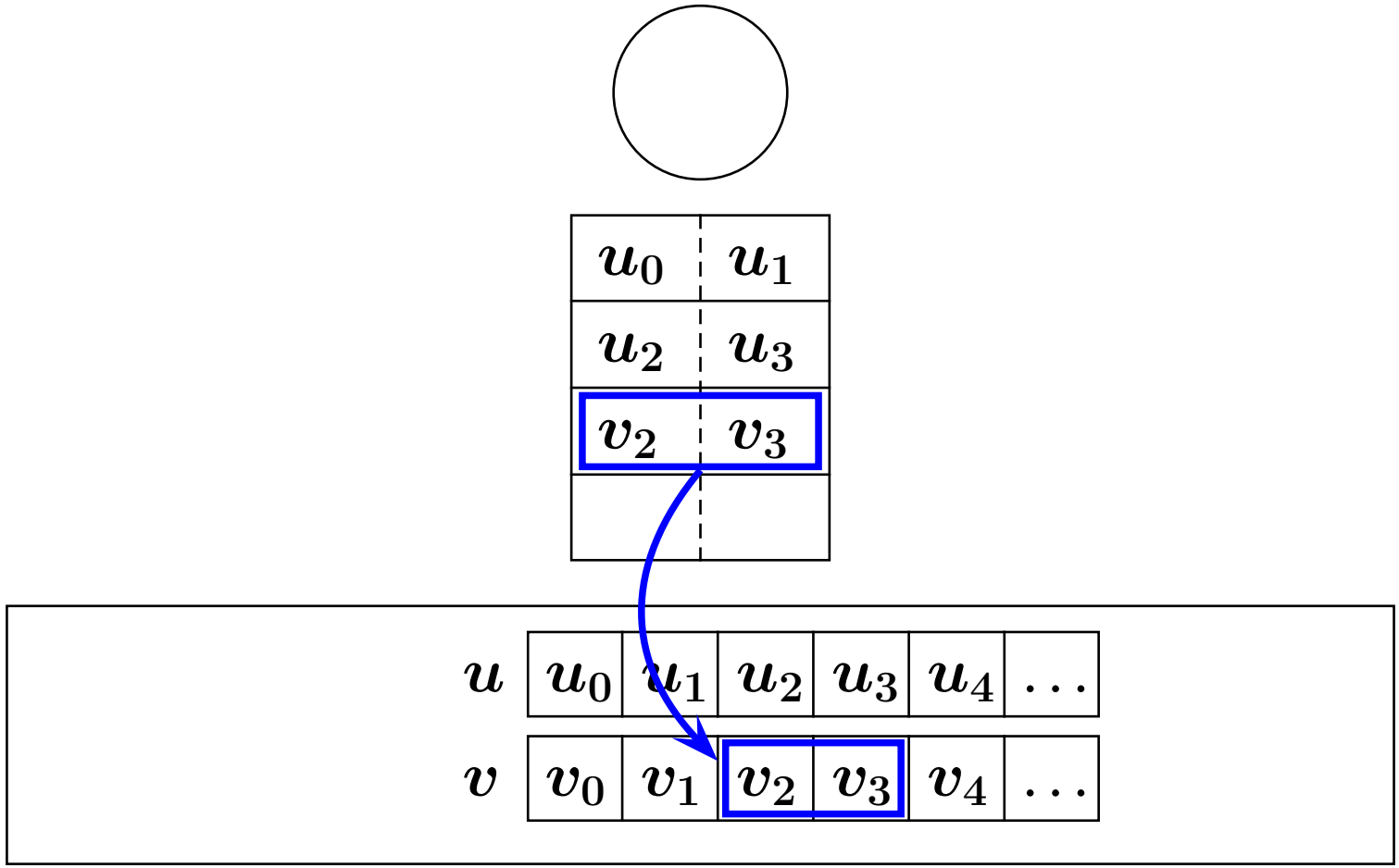


6. Le cœur utilise les copies de u_1, u_2, u_3 de la mémoire cache, calcule l'expression et place le résultat dans la mémoire cache :



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



En résumé :

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

Pour utiliser au mieux la mémoire cache, deux règles sont déduites de ce comportement : 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.

Exemple: voir TP 1

Localité temporelle

Règle: autant que possible, pour une zone mémoire, les instructions qui l'utilisent 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)

Exemple: voir TP 1

Utilisation du // interne au processeur

Règles:

- essayer de rassembler plusieurs instructions simples en une seule (quand cela a un sens),

  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).

Exemple: remplacer

```
for (i=0; i<n; i++) {
  u[i] = u0[i];
  if (terme1) u[i] += a*v[i];
  if (terme2) u[i] += b*w[i];
}
```

par:

```
if (terme1) aa = a; else aa = 0.0;
if (terme2) bb = b; else bb = 0.0;
for (i=0; i<n; i++) {
   u[i] = u0[i] + aa*v[i] + bb*w[i];
}
```