ECE_4ES01

TP5 — Microprocesseurs

Zhe CHEN
José Daniel CHACÓN GÓMEZ
Gustavo de MACENA BARRETOS
Lucas Emanuel BARBOSA VASCONCELOS

Encadrant : SIDEM Antoine
Responsable matière : HAMMAMI Omar

École Nationale des techniques avancées (ENSTA Paris)
Institut Polytechnique de Paris
2026

Table des matières

1 Résumé

2 Introduction

3 Méthodologie

4 Réponses aux questions (Q1–Q14)

4.1 Q1 — Hiérarchie mémoire et cohérence (multiplication de matrices)

> **Énoncé (Q1).**
> En considérant que chaque thread s'exécute sur un processeur dans une architecture de type multicoeurs à base de bus et 1 niveau de cache (comme décrit Figure 21), décrivez le comportement de la hiérarchie mémoire et de la cohérence des caches pour l'algorithme de multiplication de matrices. On supposera que le thread principal se trouve sur le processeur d'indice 1.

Comportement global (bus + 1 niveau de cache)

On considère une architecture à mémoire partagée avec un bus commun, et un seul niveau de cache privé par cœur (les accès manqués vont directement en mémoire). La multiplication $C = A \times B$ implique principalement : (i) beaucoup de lectures sur A et B, et (ii) des écritures sur C.

Hiérarchie mémoire : misses et localité

— **Démarrage (cold misses).** Au début, chaque processeur subit des *cold misses* car aucune ligne n'est encore présente dans son cache. Les données sont transférées depuis la mémoire par **lignes de cache** (blocs).
— **Localité spatiale.** Quand un thread parcourt des éléments contigus (par exemple une ligne de A ou des éléments proches de C), le chargement d'une ligne de cache apporte plusieurs éléments voisins, ce qui réduit les accès mémoire pour les itérations suivantes.
— **Localité temporelle.** Dans la boucle interne, certains éléments peuvent être réutilisés (accumulateur de $C[i, j]$ et éléments de A selon l'ordonnancement). En pratique, la réutilisation dépend fortement de l'implémentation (ordre des boucles et éventuel *blocking/tiling*).

Cohérence : protocole snoopy *write-invalidate*

Les contrôleurs de cache « écoutent » le bus (snooping). Les lignes transitent typiquement entre des états du type **Invalid**, **Shared** et **Exclusive** (selon le protocole simplifié du cours).

— **Lecture de A et B (données majoritairement en lecture).** Lorsqu'un cœur lit une ligne de A ou B absente de son cache, il émet une transaction de lecture sur le bus. La ligne est récupérée (depuis la mémoire, ou depuis un autre cache selon le protocole) et placée dans

le cache local. Comme plusieurs threads peuvent lire les mêmes zones de A ou B, ces lignes tendent à se retrouver en **Shared** dans plusieurs caches, ce qui est cohérent et ne nécessite pas d'invalidation.

— **Écriture de C (nécessite l'exclusivité).** Quand un cœur veut écrire dans C, il doit obtenir la ligne correspondante en état **Exclusive**. Pour cela, il réalise une requête sur le bus (type *upgrade* ou *read-for-ownership*), ce qui **invalide** les copies éventuelles de cette même ligne dans les autres caches. Ceci garantit qu'il n'existe qu'un seul auteur actif de la ligne au moment de l'écriture, et donc que les autres processeurs ne lisent pas une valeur obsolète.

Rôle du thread principal sur le processeur 1

Le thread principal étant sur le **processeur 1** :

— S'il **initialise** A, B et/ou C, il va charger et/ou écrire de nombreuses lignes dans son cache en premier (état plutôt **Exclusive** au départ).

— Lorsque les autres processeurs commencent le calcul, ils vont provoquer des **misses** sur A et B et placer ces lignes en **Shared** (lecture partagée).

— À la fin, si le processeur 1 **relit** C (assemblage des résultats, vérification), il peut générer des misses supplémentaires, car les lignes écrites par d'autres processeurs ne sont pas forcément dans son cache.

Point important : trafic sur le bus et faux-partage

Avec un bus partagé et un seul niveau de cache, les performances peuvent être limitées par :

— **Contention du bus** : tous les misses et toutes les transactions de cohérence passent par le bus.

— **Faux-partage (false sharing)** : si deux threads écrivent dans des éléments différents de C mais situés sur la **même ligne de cache**, ils vont s'invalider mutuellement (ping-pong de lignes), même sans partager la même case. En pratique, une décomposition par blocs/lignes bien alignée (chaque thread écrit des zones de C séparées par lignes de cache) réduit fortement ce problème.

4.2 Q2 — Paramètres configurables du CPU O3 (DerivO3CPU)

> **Objectif.**
> Identifier des paramètres configurables du processeur *out-of-order* de gem5 (`DerivO3CPU`) et préciser, pour chacun, sa valeur par défaut ainsi que son rôle.

Méthode (où chercher les paramètres)

Sur les machines ENSTA, les paramètres du CPU O3 sont définis dans le fichier Python `O3CPU.py` (répertoire `src/cpu/o3` de gem5). Pour lister rapidement les paramètres et leurs valeurs par défaut, on se place dans le dossier du CPU O3 puis on filtre les lignes contenant `Param.` :

```
cd /auto/g/gbusnot/ES201/tools/TP5/gem5-stable/src/cpu/o3
grep -n "Param\." O3CPU.py
```

Le `grep -n` affiche les numéros de ligne, ce qui permet de retrouver facilement la définition exacte des paramètres dans `O3CPU.py`.

Sélection de 5 paramètres (valeur par défaut + impact)

Nous avons choisi des paramètres liés (i) à la fenêtre OoO (ROB / IQ), (ii) au sous-système mémoire spéculatif (Load/Store Queues), et (iii) à la prédiction de branchement, car ce sont des éléments déterminants pour l'IPC et la performance globale.

TABLE 1 – Paramètres `DerivO3CPU` (extraits de `O3CPU.py`)

Paramètre	Valeur par défaut	Rôle / impact (résumé)
`numROBEntries`	`192`	Taille du *Reorder Buffer* : nb. d'instructions "en vol". Plus grand ⇒ meilleure exploitation de l'ILP et masquage de latences.
`numIQEntries`	`64`	Taille de l'*Issue Queue* : instructions prêtes à être émises. Limite la fenêtre effective (même si le ROB est grand).
`LQEntries`	`32`	*Load Queue* : nb. de loads suivis/pendants en OoO. Important pour le recouvrement mémoire et la gestion des dépendances.
`SQEntries`	`32`	*Store Queue* : nb. de stores en vol avant écriture en cache/mémoire. Impact sur le débit mémoire et les dépendances load-after-store.
`branchPred`	`TournamentBP(numThreads= Parent.numThreads)`	Prédicteur de branchements par défaut. Une mauvaise prédiction provoque des *flush/squash* et dégrade l'IPC.

4.3 Q3 — Valeurs par défaut des caches (L1I, L1D, L2)

Objectif.
Retrouver et reporter les valeurs par défaut des paramètres de cache (tailles, associativités et taille de ligne) utilisées par `se.py` lorsque aucune option n'est passée en ligne de commande.

Méthode

D'après l'énoncé, les valeurs par défaut sont définies dans `$GEM5/configs/common/Options.py`. Dans ce fichier, elles apparaissent dans les lignes `parser.add_option(... default=...)`. Ces `default=` correspondent aux paramètres effectivement utilisés par `se.py` si l'on ne fournit pas d'options explicites au lancement.

Exemples de recherche (optionnel) :

```
grep -n "l1d_size"   $GEM5/configs/common/Options.py
grep -n "l1i_size"   $GEM5/configs/common/Options.py
grep -n "l2_size"    $GEM5/configs/common/Options.py
grep -n "cacheline_size" $GEM5/configs/common/Options.py
```

Valeurs par défaut (Options.py)

TABLE 2 – Paramètres de cache par défaut (extraits de `Options.py`)

Niveau	Option gem5	Valeur par défaut
L1D	`-l1d_size`	`64kB`
L1D	`-l1d_assoc`	`2` (2-way)
L1I	`-l1i_size`	`32kB`
L1I	`-l1i_assoc`	`2` (2-way)
L2	`-l2_size`	`2MB`
L2	`-l2_assoc`	`8` (8-way)
Global	`-cacheline_size`	`64B`

Remarque

La taille de ligne est donnée par un paramètre global (`cacheline_size`). Par défaut, elle s'applique donc à la L1D, la L1I et la L2 (sauf si une configuration spécifique la surcharge ailleurs).

4.4 Q4 — Processeur critique : maximum de cycles et cycles totaux de l'application

Énoncé (Q4).
Déterminez quel est le processeur exécutant toujours le plus grand nombre de cycles. Expliquez pourquoi. Expliquez également pourquoi l'analyse du nombre de cycles sur ce processeur revient à analyser le nombre total de cycles d'exécution de l'application.

Note méthodologique (deux séries de mesures)

Dans le TP, l'activation de la hiérarchie de cache se fait avec `-caches` (L1) et, pour ajouter une L2, avec `-l2cache`. Nous avons réalisé une série **avec L2** jusqu'à $T \leq 8$ (`-caches -l2cache`). En revanche, pour $T = 16$ avec L2, la simulation n'a pas produit de statistiques exploitables (`stats.txt` invalide / exécution non finalisée). Pour pouvoir inclure $T = 16$ (avec la contrainte du TP : *n threads* = *n cœurs*), nous avons donc réalisé une seconde série **sans L2** ($T \leq 16$) en gardant uniquement `-caches`. Les deux séries sont reportées séparément (elles ne sont pas directement comparables en cycles absolus puisque la hiérarchie mémoire change).

Méthode (extraction des cycles)

Pour chaque exécution, nous avons extrait depuis `m5out/stats.txt` la statistique `system.cpu<i>.numCycles` (nombre de cycles simulés par cœur). Le processeur « critique » est identifié comme celui qui maximise `numCycles` sur l'ensemble des cœurs. En mono-cœur ($T = 1$), gem5 reporte `system.cpu.numCycles` (sans indice), que l'on interprète comme le cœur `cpu0`.

Résultat : cœur ayant le plus grand nombre de cycles

TABLE 3 – Cœur critique (max `numCycles`) — Série avec L2

Threads	**CPU max**	$cycles_{app} = \max_i(numCycles_i)$
1	cpu0	2 092 404
2	cpu0	1 128 158
4	cpu0	646 916
8	cpu0	408 798

Série avec L2 (jusqu'à 8 threads).

TABLE 4 – Cœur critique (max `numCycles`) — Série sans L2

Threads	**CPU max**	$cycles_{app} = \max_i(numCycles_i)$
1	cpu0	2 597 586
2	cpu0	1 508 970
4	cpu0	970 012
8	cpu0	705 494
16	cpu0	582 578

Série sans L2 (jusqu'à 16 threads). Conclusion : sur toutes les configurations mesurées, le processeur accumulant le plus grand nombre de cycles est toujours `cpu0`.

Pourquoi est-ce souvent `cpu0` ?

Dans un programme OpenMP, un **thread maître** (master) gère typiquement l'initialisation, le lancement du parallélisme (*fork/join*) et une partie des synchronisations (barrières, fin de région parallèle, etc.). Dans notre configuration, ce thread maître est naturellement associé au premier cœur (`cpu0`), ce qui tend à lui faire cumuler légèrement plus de travail global (ou des phases d'attente/synchronisation) et donc un `numCycles` maximal.

Pourquoi le max des cycles correspond aux cycles totaux de l'application ?

L'application parallèle se termine quand le **dernier thread** termine (le plus lent). Autrement dit, le temps d'exécution global (*makespan*) est déterminé par le cœur qui exécute le plus de cycles. On peut donc estimer les cycles totaux de l'application par :

$$cycles_{app} \;=\; \max_i \big(numCycles(\mathrm{cpu}_i)\big).$$

Les autres cœurs peuvent finir plus tôt puis attendre (barrière, join), mais la fin globale est imposée par le cœur critique ; analyser ses cycles revient donc à analyser la durée totale d'exécution de l'application.

4.5 Q5 — Nombre de cycles d'exécution de l'application

Énoncé (Q5).
Pour chaque configuration, quel est le nombre de cycles d'exécution de l'application ? Vous pourrez présenter vos résultats sous forme de graphe 2 axes.

Définition de la métrique

Nous définissons le nombre de cycles d'exécution de l'application comme le nombre de cycles du **cœur critique** (le plus lent), c'est-à-dire :

$$cycles_{app} = \max_i \left(\texttt{system.cpu}\langle i \rangle\texttt{.numCycles}\right).$$

Cette valeur est extraite de `m5out/stats.txt`. Contexte : Cortex-A7 (`-cpu-type=arm_detailed`), `size=64` et `nthreads = ncores`.

Valeurs numériques (référence à Q4)

Les valeurs de `cycles_app` pour chaque configuration correspondent exactement au maximum de `numCycles` identifié en Q4. Elles sont donc déjà reportées dans les Tables 3 (série avec L2) et 4 (série sans L2).

Graphe 2 axes

Les Figures 1 et 2 représentent `cycles_app` en fonction du nombre de threads (avec `nthreads = ncores`). Nous séparons les deux séries (avec/sans L2), car la hiérarchie mémoire n'est pas la même.

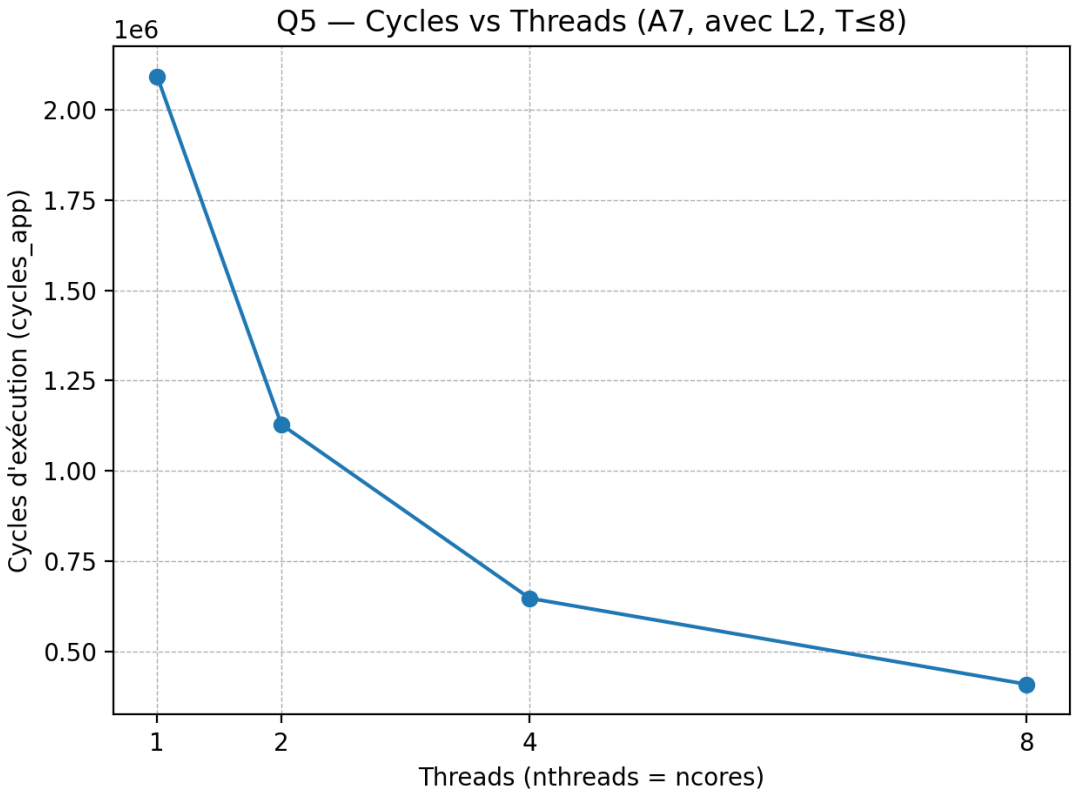

FIGURE 1 – Q5 — `cycles_app` vs threads (A7, avec L2, $T \leq 8$).

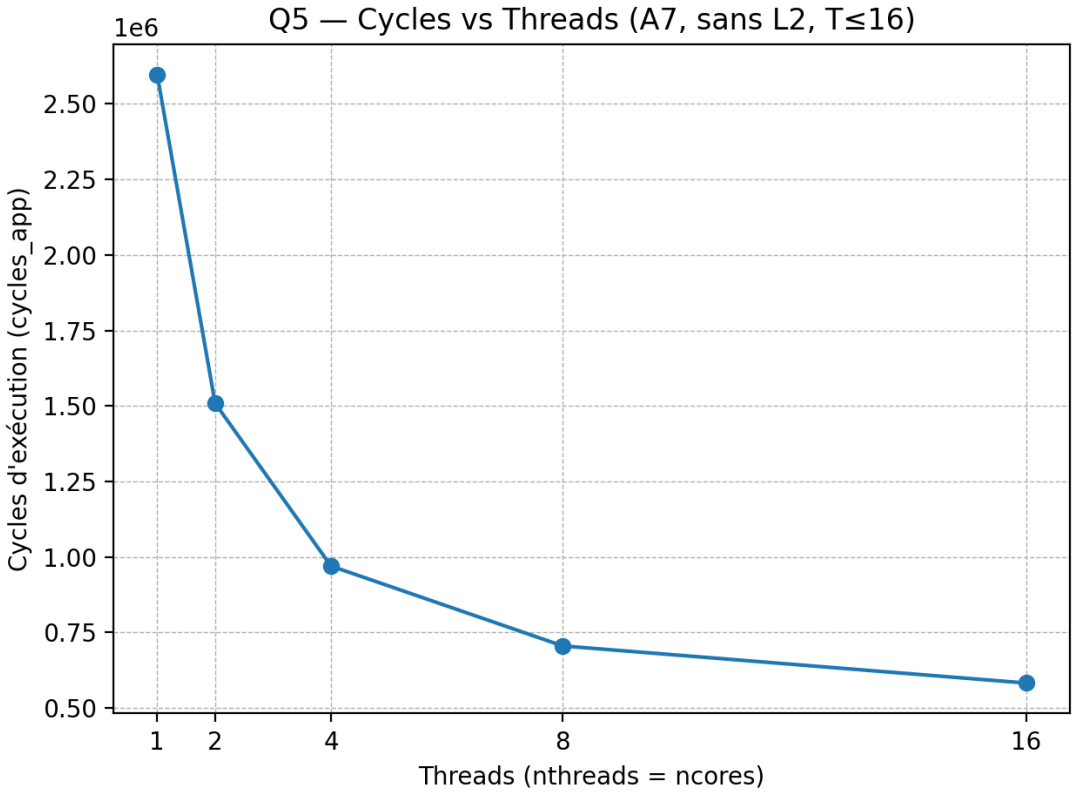

FIGURE 2 – Q5 — `cycles_app` vs threads (A7, sans L2, $T \leq 16$).

Observation (sans interprétation avancée)

Dans les deux séries, `cycles_app` diminue lorsque le nombre de threads augmente, ce qui traduit une exécution plus rapide avec davantage de cœurs. La diminution est très forte entre 1→2 et 2→4, puis devient moins marquée lorsque l'on continue à augmenter le parallélisme. Enfin, comme la hiérarchie mémoire change (avec/sans L2), on interprète chaque courbe séparément, sans comparer directement les valeurs absolues entre séries.

4.6 Q6 — Speedup par rapport à la configuration à 1 thread

> **Énoncé (Q6).**
> Déduire le speedup par rapport à la configuration à 1 thread.

Définition

À fréquence constante, le temps d'exécution est proportionnel au nombre de cycles. On en déduit le speedup par rapport à 1 thread :

$$\text{Speedup}(T) \;=\; \frac{cycles_{app}(1)}{cycles_{app}(T)},$$

où $cycles_{app}(T)$ est défini en Q5 comme le maximum de `numCycles` (cœur critique). Les valeurs de $cycles_{app}(T)$ sont celles déjà reportées en Q4 (Tables 3 et 4).

Résultats

Nous calculons le speedup séparément pour les deux séries (avec/sans L2), car les bases ($cycles_{app}(1)$) ne sont pas identiques.

TABLE 5 – Speedup vs 1 thread — A7 avec L2

Threads (T)	$cycles_{app}(T)$	**Speedup**(T)
1	2 092 404	1.000
2	1 128 158	1.855
4	646 916	3.234
8	408 798	5.118

Série A — A7 avec L2 (base : $cycles_{app}(1) = 2\,092\,404$**).**

TABLE 6 – Speedup vs 1 thread — A7 sans L2

Threads (T)	$cycles_{app}(T)$	**Speedup**(T)
1	2 597 586	1.000
2	1 508 970	1.721
4	970 012	2.678
8	705 494	3.682
16	582 578	4.459

Série B — A7 sans L2 (base : $cycles_{app}(1) = 2\,597\,586$**).**

4.7 Q7 — Valeur maximale de l'IPC (à partir de `sim_insts`)

Énoncé (Q7).
En utilisant le nombre total d'instructions simulées, déterminez quelle est la valeur maximale de l'IPC pour chaque configuration ?

Définition et calcul

Pour chaque exécution, gem5 fournit dans `m5out/stats.txt` :

— `sim_insts` : le nombre total d'instructions simulées,

— $cycles_{app}$: les cycles d'exécution de l'application (définis en Q5 comme $\max_i$(`system.cpu`$\langle i \rangle$`.numCycles`), voir aussi Q4–Q5).

On calcule alors l'IPC global par :

$$IPC(T) \;=\; \frac{\texttt{sim\_insts}(T)}{cycles_{app}(T)}.$$

Résultats

Comme précédemment, on reporte deux séries séparées (avec/sans L2).

TABLE 7 – IPC — A7 avec L2

Threads	$cycles_{app}$	`sim_insts`	**IPC**
1	2 092 404	4 107 655	1.963127
2	1 128 158	4 132 898	3.663404
4	646 916	4 158 816	6.428680
8	408 798	4 216 901	10.315366

Série A — A7 avec L2 ($T \leq 8$**).** **IPC maximal (avec L2) :** $IPC_{max} = 10.315366$ (configuration $T = 8$).

Table 8 – IPC — A7 sans L2

Threads	$cycles_{app}$	sim_insts	IPC
1	2 597 586	4 107 655	1.581336
2	1 508 970	4 233 827	2.805773
4	970 012	4 369 018	4.504087
8	705 494	4 570 580	6.478553
16	582 578	5 008 208	8.596631

Série B — A7 sans L2 ($T \leq 16$). IPC maximal (sans L2) : $IPC_{max} = 8.596631$ (configuration $T = 16$).

Observation courte

Dans nos mesures, l'IPC global augmente avec le nombre de threads car $cycles_{app}$ diminue fortement lorsque l'on parallélise l'exécution. Les résultats sont reportés séparément pour les séries avec L2 et sans L2.

4.8 Q8 — Discussion et interprétation (max. 10 lignes)

Énoncé (Q8).
Discussion et interprétation (max. 10 lignes).

On observe une cohérence entre les métriques : quand le nombre de threads augmente, $cycles_{app}$ diminue, donc le speedup (Q6) augmente. L'IPC global (Q7), calculé comme `sim_insts`/$cycles_{app}$, a aussi tendance à croître, principalement parce que l'exécution se termine en moins de cycles.

En revanche, le speedup reste clairement sous-linéaire et les gains marginaux deviennent plus faibles à grand T (rendements décroissants). C'est attendu : une partie du programme reste séquentielle et il existe des coûts incompressibles liés au parallélisme (fork/join OpenMP, synchronisations/barrières). Le fait que `cpu0` soit toujours le cœur critique suggère que le thread maître et/ou les points de synchronisation jouent un rôle important. Enfin, l'IPC doit être interprété avec prudence car c'est une métrique globale sur tout le run et `sim_insts` peut varier légèrement avec T (overhead).

4.9 Q9 — Titre de la question

Énoncé (Q9).

Réponse

4.10 Q10 — Titre de la question

Énoncé (Q10).

Réponse

4.11 Q11 — Titre de la question

Énoncé (Q11).

Réponse

4.12 Q12 — Titre de la question

Énoncé (Q12).

Réponse

4.13 Q13 — Titre de la question

Énoncé (Q13).

Réponse

4.14 Q14 — Titre de la question

Énoncé (Q14).

Réponse

5 Conclusion

A Annexe A — Reproductibilité