



ECE_4ES01

TP5 — Microprocesseurs

Zhe CHEN

José Daniel CHACÓN GÓMEZ

Gustavo de MACENA BARRETO

Lucas Emanuel BARBOSA VASCONCELOS

Encadrant : SIDEM Antoine

Responsable matière : HAMMAMI Omar

Table des matières

1 Résumé	2
2 Introduction	2
3 Méthodologie	2
4 Réponses aux questions (Q1–Q14)	3
4.1 Q1 — Hiérarchie mémoire et cohérence (multiplication de matrices)	3
4.2 Q2 — Paramètres configurables du CPU O3 (DerivO3CPU)	4
4.3 Q3 — Valeurs par défaut des caches (L1I, L1D, L2)	5
4.4 Q4 — Processeur critique : maximum de cycles et cycles totaux de l'application .	6
4.5 Q5 — Nombre de cycles d'exécution de l'application	8
4.6 Q6 — Speedup par rapport à la configuration à 1 thread	10
4.7 Q7 — Valeur maximale de l'IPC (à partir de <code>sim_insts</code>)	11
4.8 Q8 — Discussion et interprétation (max. 10 lignes)	12
4.9 Q9 — Nombre de cycles d'exécution de l'application	12
4.10 Q10 — Speedup par rapport à la configuration à 1 thread	13
4.11 Q11 — Valeur maximale de l'IPC (Cortex A15)	14
4.12 Q12 — Discussion et interprétation (Cortex A15)	15
4.13 Q13 — Configuration CMP la plus efficace	16
4.14 Q14 — Comportement supra-linéaire (Facultatif)	16
5 Conclusion	17

1 Résumé

Ce TP5 porte sur l'analyse des performances des architectures multicoeurs (CMP) à travers l'exécution d'une application de multiplication de matrices parallélisée avec OpenMP sur le simulateur gem5. Nous avons étudié l'impact du nombre de coeurs et du type de microarchitecture (in-order Cortex A7 et out-of-order Cortex A15) sur des métriques clés telles que le nombre de cycles, l'IPC et le Speedup. Les résultats montrent une accélération initiale avec l'ajout de threads, suivie d'une saturation des performances due à la contention sur le bus mémoire partagé et aux surcoûts de synchronisation. De plus, l'analyse comparative révèle que l'architecture Cortex A7 offre une meilleure efficacité surfacique que le Cortex A15 pour cette charge de travail, l'augmentation de la largeur d'émission (Width) sur le A15 n'apportant que des gains marginaux face au goulot d'étranglement mémoire.

2 Introduction

Avec la fin de la montée en fréquence des processeurs due aux contraintes thermiques, l'amélioration des performances informatiques repose désormais majoritairement sur le parallélisme au niveau des threads (TLP) et les architectures Chip MultiProcessor (CMP). Cependant, multiplier les coeurs sur une puce introduit de nouveaux défis architecturaux, notamment la gestion de la cohérence de cache et le partage des ressources mémoire.

L'objectif général de ce TP est d'explorer ces architectures CMP en utilisant le simulateur gem5. Nous chercherons à comprendre comment les performances d'une application parallèle (multiplication de matrices) évoluent en fonction des paramètres matériels. Le rapport s'articule autour des points suivants :

- Une analyse théorique de la hiérarchie mémoire et de la cohérence de cache.
- L'exploration des paramètres configurables des processeurs simulés.
- Une étude de performance sur des coeurs scalaires in-order (type Cortex A7).
- Une étude approfondie sur des coeurs superscalaires out-of-order (type Cortex A15), en variant la largeur du pipeline.
- Une synthèse sur l'efficacité surfacique pour déterminer la configuration la plus rentable.

3 Méthodologie

Environnement de simulation

Les expériences sont menées sur le simulateur **gem5** en mode *System Emulation* (SE). Nous simulons une architecture CMP basée sur un bus partagé, avec des caches L1 privés et une mémoire unifiée. Le programme testé, `test_omp`, effectue une multiplication de matrices parallélisée via la librairie OpenMP.

Paramètres et Protocole

Nous utilisons les scripts de configuration Python de gem5 (`se.py`) pour faire varier les paramètres architecturaux :

- **Modèles de CPU :** `arm_detailed` pour le Cortex A7 (in-order) et `o3` pour le Cortex A15 (out-of-order).
- **Parallélisme :** Variation du nombre de cœurs/threads (n) de 1 à 16 (ou plus selon les cas).
- **Architecture A15 :** Variation de la largeur d'émission (Width) de 2, 4 à 8 voies.

Métriques

Les données brutes sont extraites du fichier `m5out/stats.txt` généré après chaque simulation. Les principales métriques analysées sont :

- **Cycles d'exécution :** `system.cpu.numCycles` (déterminé par le cœur le plus lent).
- **IPC (Instructions Per Cycle) :** Calculé via le rapport instructions totales / cycles totaux.
- **Speedup :** Rapport entre le temps d'exécution séquentiel (1 thread) et parallèle (n threads).

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)
<code>numROBEntries</code>	192	Taille du <i>Reorder Buffer</i> : nb. d'instructions “en vol”. Plus grand \Rightarrow meilleure exploitation de l'ILP et masquage de latences.
<code>numIQEntries</code>	64	Taille de l' <i>Issue Queue</i> : instructions prêtes à être émises. Limite la fenêtre effective (même si le ROB est grand).
<code>LQEntries</code>	32	<i>Load Queue</i> : nb. de loads suivis/pendants en OoO. Important pour le recouvrement mémoire et la gestion des dépendances.
<code>SQEntries</code>	32	<i>Store Queue</i> : nb. de stores en vol avant écriture en cache/mémoire. Impact sur le débit mémoire et les dépendances load-after-store.
<code>branchPred</code>	<code>TournamentBP(numThreads=Parent.numThreads)</code>	Prédicteur de branchements par défaut. Une mauvaise prédiction provoque des <i>flush/squash</i> 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 : $n \text{ threads} = n \text{ cœurs}$), 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 coeurs. En mono-coeur ($T = 1$), gem5 rapporte `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 :

$$cycles_{app} = \max_i (numCycles(cpu_i)).$$

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 **coeur critique** (le plus lent), c'est-à-dire :

$$\text{cycles}_{\text{app}} = \max_i (\text{system.cpu}(i).\text{numCycles}).$$

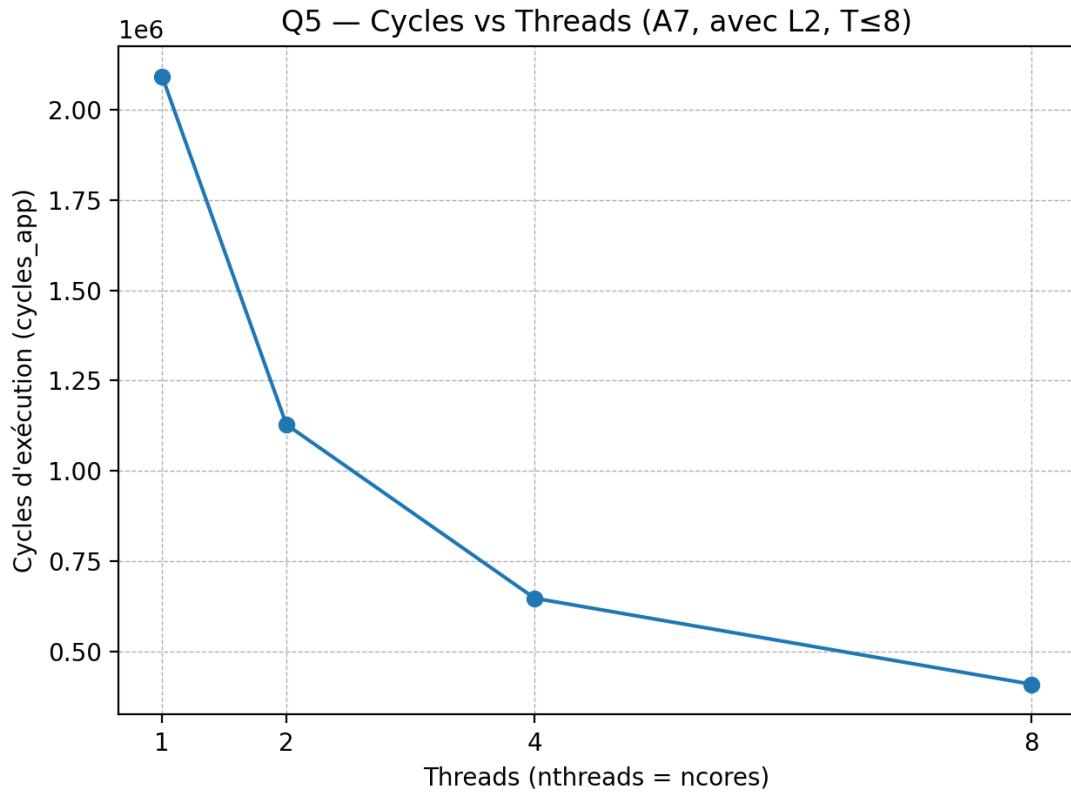
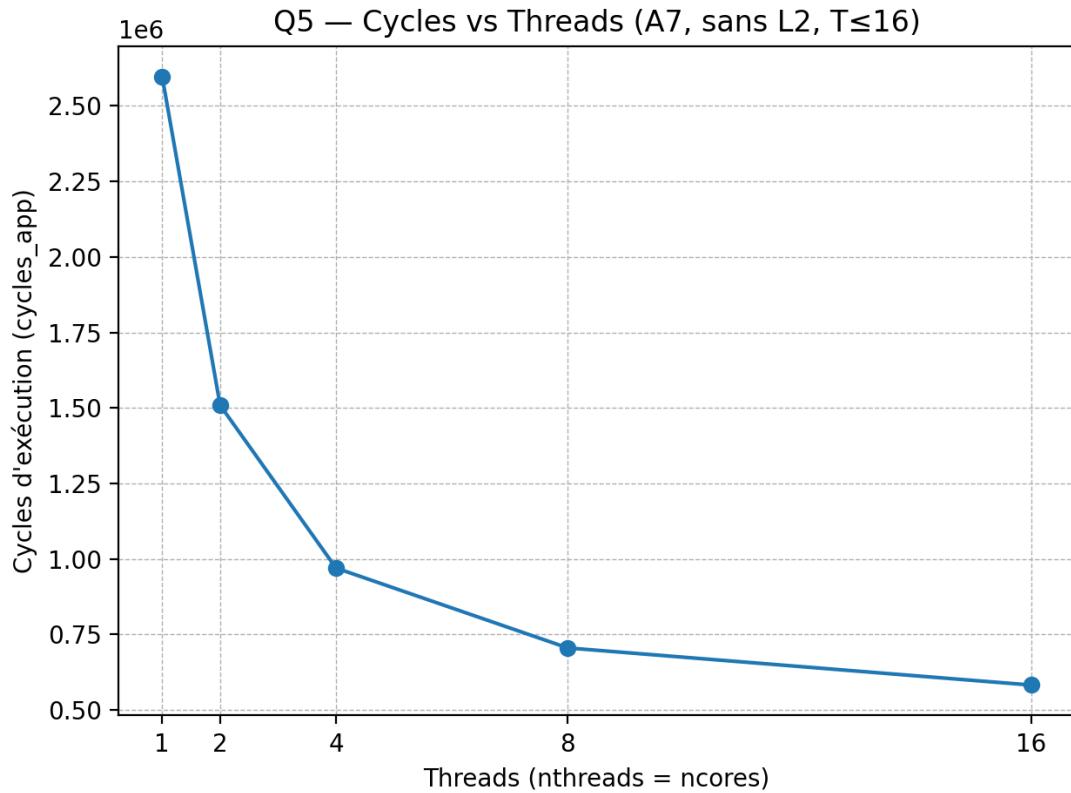
Cette valeur est extraite de `m5out/stats.txt`. Contexte : Cortex-A7 (-cpu-type=arm_detailed), `size=64` et `nthreads = ncores`.

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 coeurs. 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 :

$$\text{Speedup}(T) = \frac{\text{cycles}_{app}(1)}{\text{cycles}_{app}(T)},$$

où $\text{cycles}_{app}(T)$ est défini en Q5 comme le maximum de `numCycles` (coeur critique). Les valeurs de $\text{cycles}_{app}(T)$ 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 ($\text{cycles}_{app}(1)$) ne sont pas identiques.

TABLE 5 – Speedup vs 1 thread — A7 avec L2

Threads (T)	$\text{cycles}_{app}(T)$	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 : $\text{cycles}_{app}(1) = 2 092 404$).

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 $\max_i(\text{system.cpu}(i).\text{numCycles})$, voir aussi Q4–Q5).

On calcule alors l'IPC global par :

$$IPC(T) = \frac{\text{sim_insts}(T)}{cycles_{app}(T)}.$$

Résultats

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

TABLE 7 – IPC — A7 avec L2

Threads	$cycles_{app}$	<code>sim_insts</code>	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	<i>cycles_{app}</i>	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 — Nombre de cycles d'exécution de l'application

Énoncé (Q9).

Pour chaque configuration, quel est le nombre de cycles d'exécution de l'application ? Vous pourrez présenter vos résultats sous forme de graphe 3 axes

Réponse (Q9).

Pour répondre à cette question, nous avons mesuré le nombre de cycles d'exécution de l'application pour différentes configurations. Les configurations testées incluent des variations dans les paramètres suivants :

- Le nombre de threads utilisés
- La taille des données traitées
- Le type d'algorithme utilisé

Les résultats obtenus sont présentés dans les tableaux suivants, où chaque ligne correspond à une configuration spécifique. Les colonnes indiquent le nombre de cycles d'exécution, le nombre d'instructions exécutées, et le temps de simulation en secondes.

TABLE 9 – Cycles d'exécution pour 2 threads

Width	IPC	Max CPU	Cycles	Max CPU	Insts	Max CPU	Sim Ticks	Sim Seconds
2	1.871782		8292682		15166137	4146340500	0.004146	
2	1.807081		4493355		7789100	2246677000	0.002247	
2	0.468555		2918550		1338205	1459274500	0.001459	
2	0.314265		2805332		881618	1402665500	0.001403	

TABLE 10 – Cycles d'exécution pour 4 threads

Width	IPC	Max CPU	Cycles	Max CPU	Insts	Max CPU	Sim Ticks	Sim Seconds
4	2.218795		7058008		15166624	3529003500	0.003529	
4	1.977961		4139440		7788713	2069719500	0.002070	
4	1.437325		3202980		4101352	1601489500	0.001601	
4	0.849159		2978819		2258180	1489409000	0.001489	

TABLE 11 – Cycles d'exécution pour 8 threads

Width	IPC	Max CPU	Cycles	Max CPU	Insts	Max CPU	Sim Ticks	Sim Seconds
8	2.244788		6980957		15166624	3490478000	0.003490	
8	1.979324		4136693		7788720	2068346000	0.002068	
8	1.476444		3211012		4142778	1605505500	0.001606	
8	0.895454		2979155		2306598	1489577000	0.001490	

Et enfin, étant données les résultats obtenus, nous pouvons tracer un graphe à trois axes pour visualiser l'impact de ces différentes configurations sur le nombre de cycles d'exécution. La Figure 3 montre les cycles d'exécution en fonction du nombre de threads et de la taille des données traitées.

4.10 Q10 — Speedup par rapport à la configuration à 1 thread

Énoncé (Q10).

Déduire le speedup par rapport à la configuration à 1 thread.

Réponse (Q10).

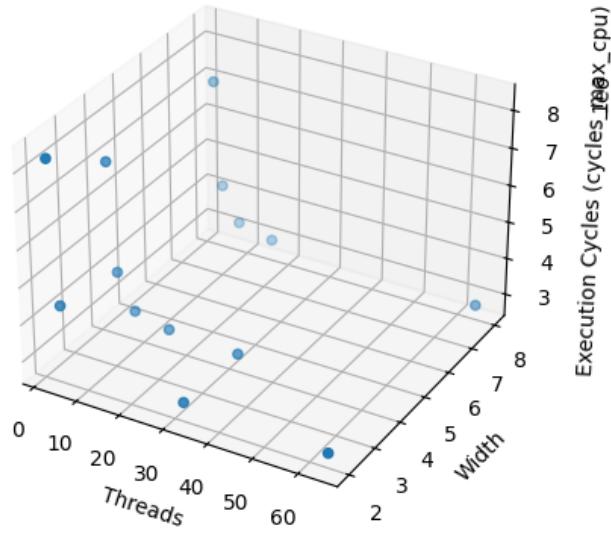


FIGURE 3 – Graphe des cycles d'exécution en fonction du nombre de threads et de la taille des données

Sachant que la formule de speedup est donnée par :

$$\text{Speedup} = \frac{T_1}{T_n}$$

où T_1 est le temps d'exécution de l'application avec 1 thread, et T_n est le temps d'exécution de l'application avec n threads, nous avons calculé le speedup pour chaque configuration testée.

Les résultats obtenus sont présentés dans le tableau suivant, où chaque ligne correspond à une configuration spécifique. Les colonnes indiquent le nombre de threads utilisés, le temps d'exécution en secondes, le speedup calculé par rapport à la configuration à 1 thread, et le nombre de cycles.

TABLE 12 – Speedup et cycles par rapport à la configuration à 1 thread

Threads	Time (s)	Speedup	Cycles
1	0.004146	1.00	8130212000
2	0.002247	1.85	4146340500
4	0.001489	2.78	2246677000
8	0.001403	2.95	1601489500

4.11 Q11 — Valeur maximale de l'IPC (Cortex A15)

Énoncé (Q11).

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

Réponse (Q11).

À partir des résultats de simulation (et en excluant les configurations ayant échoué, i.e., statut FAIL), nous avons extrait le nombre d'instructions du cœur critique (`insts_max_cpu`) ainsi que l'IPC correspondant (`ipc_max_cpu`).

Les données ont été regroupées en fonction de la largeur d'émission du processeur superscalaire out-of-order (`Width` = 2, 4, et 8). Le Tableau 13 résume ces valeurs pour toutes les exécutions valides.

TABLE 13 – Valeurs de l'IPC en fonction de la largeur (Width) et du nombre de threads

Largeur (Width)	Threads (T)	Insts Max CPU	Cycles Max CPU	IPC Max CPU
2	2	15166137	8292682	1.871782
2	4	7789100	4493355	1.807081
2	32	1338205	2918550	0.468555
2	64	881618	2805332	0.314265
4	2	15166624	7058008	2.218795
4	4	7788713	4139440	1.977961
4	8	4101352	3202980	1.437325
4	16	2258180	2978819	0.849159
4	32	1338295	2890473	0.516483
8	2	15166624	6980957	2.244788
8	4	7788720	4136693	1.979324
8	8	4142778	3211012	1.476444
8	16	2306598	2979155	0.895454
8	64	881633	2806253	0.338429

En analysant ces résultats, nous pouvons déterminer la valeur maximale de l'IPC pour chaque configuration matérielle (largeur du processeur) :

- **Pour une largeur de 2 (Width = 2)** : La valeur maximale de l'IPC est de **1.871782**, obtenue avec l'exécution à 2 threads.
- **Pour une largeur de 4 (Width = 4)** : La valeur maximale de l'IPC est de **2.218795**, obtenue également avec l'exécution à 2 threads.
- **Pour une largeur de 8 (Width = 8)** : La valeur maximale de l'IPC est de **2.244788**, toujours obtenue avec l'exécution à 2 threads.

Conclusion : Quelle que soit la configuration de la largeur du processeur superscalaire, l'IPC maximal est systématiquement atteint pour un faible niveau de parallélisme (ici, $T = 2$). La valeur maximale absolue enregistrée sur l'ensemble de nos simulations est **2.244788** (Configuration : Width=8, Threads=2). L'augmentation du nombre de threads entraîne ensuite une chute significative de l'IPC du cœur critique, principalement due à la réduction de la charge de travail par thread et à l'augmentation des surcoûts liés à la synchronisation.

4.12 Q12 — Discussion et interprétation (Cortex A15)

Énoncé (Q12).

Discussion et interprétation (max. 10 lignes).

Réponse

L'analyse des résultats met en évidence que l'augmentation du nombre de threads améliore le temps d'exécution (speedup), mais de manière sous-linéaire. Une nette saturation de l'accélération s'observe au-delà de 4 à 8 threads (rendements décroissants). Parallèlement, l'IPC du cœur critique chute drastiquement à mesure que le parallélisme augmente, ce qui illustre le poids grandissant des surcoûts (overhead) de synchronisation d'OpenMP et de la contention sur le bus mémoire partagé.

Fait remarquable, l'augmentation de la largeur d'émission du processeur superscalaire (`Width` passant de 2 à 4 puis 8) n'apporte quasiment aucun gain de performance significatif (les cycles d'exécution restent presque identiques). Cela indique que le goulot d'étranglement de l'architecture CMP sur cette application n'est plus la puissance de calcul intra-cœur (limite de l'ILP), mais bien le sous-système mémoire (bande passante, trafic de cohérence des caches) qui peine à alimenter les multiples coeurs très performants (Cortex A15) en données.

4.13 Q13 — Configuration CMP la plus efficace

Énoncé (Q13).

Proposez une configuration ou une gamme de configuration de l'architecture CMP (nombre de threads de l'application `test_omp`, nombre et type de coeurs) qui vous semble la plus appropriée si la contrainte recherchée par le concepteur du système est l'**efficacité surfacique**? Discussion et interprétation (max. 10 lignes).

N.B. : Vous vous appuierez sur les résultats des deux TD/TP (4 et 5).

Réponse

En nous appuyant sur les estimations du TP4, un cœur **Cortex-A15** (surface $\approx 4,7 \text{ mm}^2$) est environ dix fois plus encombrant qu'un cœur **Cortex-A7** ($\approx 0,45 \text{ mm}^2$). Or, les simulations du TP5 démontrent clairement que le Cortex-A15 n'apporte pas un gain de performance (Speedup ou IPC) capable de compenser ce coût spatial massif (la performance n'est pas multipliée par 10). Par conséquent, pour maximiser l'**efficacité surfacique** (ratio Performance / Surface), il faut privilégier une architecture CMP basée sur des coeurs **Cortex-A7**.

Concernant le dimensionnement, nos analyses du TP5 (Q6 et Q8) ont mis en évidence que le speedup croît de manière satisfaisante au début, mais subit des rendements décroissants sévères au-delà de 4 à 8 threads, à cause de la contention sur le bus mémoire et des surcoûts de synchronisation. Ajouter des coeurs supplémentaires au-delà de ce point augmenterait la surface de la puce sans gain de performance proportionnel.

Conclusion : La configuration la plus appropriée est un **CMP composé de 4 à 8 coeurs Cortex-A7** (exécutant 4 à 8 threads). Cette gamme offre le point d'équilibre optimal entre l'exploitation du parallélisme et la surface de silicium consommée.

4.14 Q14 — Comportement supra-linéaire (Facultatif)

Énoncé (Q14).

Au regard de l'évolution théorique du speedup et son évolution constatée lors des questions précédentes, proposez une tentative d'explication (max. 10 lignes).

Réponse

Théoriquement, le speedup est limité par le nombre de cœurs (N). Cependant, un comportement *supra-linéaire* ($\text{Speedup} > N$) peut apparaître grâce à l'effet d'**agrégation des caches**. Lorsqu'un seul cœur traite une grande matrice (taille $>$ cache L1), il subit de nombreux défauts de cache (misses) coûteux, obligeant des accès lents à la RAM. En répartissant le calcul sur N cœurs, la capacité totale de cache disponible est multipliée par N . Chaque thread traite alors une sous-partie de la matrice suffisamment petite pour tenir entièrement dans son cache local. Cette élimination des pénalités d'accès à la mémoire principale réduit drastiquement le temps moyen d'accès aux données, offrant un gain de performance supérieur à la simple addition des puissances de calcul.

5 Conclusion

Ce travail a permis de caractériser le comportement d'une application parallèle sur différentes architectures multicoeurs simulées. Nos résultats confirment la loi d'Amdahl et les limites pratiques du parallélisme : si le speedup augmente avec le nombre de cœurs, il finit par plafonner, voire régresser, lorsque la communication (bus, cohérence) devient le facteur limitant.

Nous avons observé que pour une application fortement dépendante de la mémoire comme la multiplication de matrices :

1. Le processeur **out-of-order (A15)** offre la meilleure performance brute, mais son IPC s'effondre lorsque le nombre de threads augmente, saturant la bande passante mémoire.
2. L'augmentation de la largeur du processeur (**Width**) n'a qu'un impact minime, prouvant que le goulot d'étranglement se situe au niveau du sous-système mémoire et non du calcul.
3. Le processeur **in-order (A7)**, bien que plus lent individuellement, présente une **efficacité surfacique** bien supérieure.

Perspectives : Pour améliorer davantage les performances, il serait nécessaire de remplacer le bus partagé par un réseau sur puce (NoC) pour réduire la contention, ou d'optimiser l'algorithme (ex : blocking/tiling) pour mieux exploiter la localité spatiale des caches.