

# CMP Performance Analysis with gem5

**Javier Andres Tarazona Jimenez**

*Ingénieur Degree Programme  
STIC  
ENSTA Paris  
Paris, France  
javier-andres.tarazona@  
ensta-paris.fr*

**Jair Anderson Vasquez Torres**

*Ingénieur Degree Programme  
STIC  
ENSTA Paris  
Paris, France  
jair-anderson.vasquez@  
ensta-paris.fr*

**Maeva Noukoua**

*Ingénieur Degree Programme  
STIC  
ENSTA Paris  
Paris, France  
maeva-sandy.noukoua@  
ensta-paris.fr*

**Carlos adrian Meneses Gamboa**

*Ingénieur Degree Programme  
STIC  
ENSTA Paris  
Paris, France  
carlos-adrian.meneses@ensta.fr*

*Abstract—...*

*Index Terms—...*

A. Q4 : cœur critique (cycles maximaux) et lien avec le temps total

L'objectif de la Q4 est d'identifier le processeur qui exécute le plus grand nombre de cycles et d'expliquer pourquoi ce processeur caractérise le temps total d'exécution. Pour chaque exécution, nous lisons dans `stats.txt` la métrique par cœur `system.cpuXX.numCycles` et nous définissons le coût temporel de l'application par :

$$C(T) = \max_i C_i(T) \quad \text{avec}$$

$$C_i(T) = \text{system.cpu}i.\text{numCycles},$$

où  $T$  est le nombre de threads OpenMP (avec `nthreads=ncores`).

a) Résultats numériques (extraits de `results/images/A7/q4_cycles.csv`):

Threads	Cycles (max)	CPU critique	Tous égaux ?
1	4 140 593	cpu00	YES
2	2 147 905	cpu00	NO
4	1 151 585	cpu00	NO
8	653 929	cpu00	NO
16	406 212	cpu00	NO
32	284 352	cpu00	NO
64	227 503	cpu00	YES

TABLE I: Cycles maximaux et cœur critique pour Q4 (A7, size=64).

b) Pourquoi un cœur « critique » donne le temps total:

En exécution parallèle, l'application ne peut terminer qu'après la complétion de tous les threads et la synchronisation finale (barrière/join OpenMP). Le temps total correspond donc au *chemin critique* : il est gouverné par le cœur qui « finit en dernier ». En pratique, cette idée se traduit naturellement par la

métrique  $C(T) = \max_i C_i(T)$ , utilisée dans notre exploitation des résultats.

c) Cas particulier observé à  $T = 64$  : *numCycles identiques sur tous les cœurs*: Le résultat **Tous égaux ? = YES** à  $T = 64$  peut sembler contre-intuitif si l'on interprète `numCycles` comme une mesure de « travail ». Or `numCycles` mesure avant tout un *temps simulé* (nombre de cycles de l'horloge globale écoulés) jusqu'à l'événement d'arrêt de la simulation (ici l'appel `exit()` du programme, confirmé par le log `Done puis target called exit()`).

Nous avons vérifié que, pour  $T = 64$ , la distribution de `numCycles` est bien dégénérée :

$$\min(\text{numCycles}) = \max(\text{numCycles}) = 227\,503 \quad (64 \text{ occurrences}).$$

Autrement dit, tous les cœurs partagent le même horizon temporel global jusqu'à la fin.

d) Pourquoi cela ne signifie pas « même travail » : *instructions différentes*: Pour distinguer *temps* et *travail*, nous regardons `system.cpuXX.committedInsts` (instructions effectivement retirées/committed). À  $T = 64$ , ces compteurs varient fortement :

$$\min(\text{committedInsts}) = 63\,712,$$

$$\max(\text{committedInsts}) = 196\,793.$$

Le Tableau II illustre ce déséquilibre en montrant les 5 cœurs qui retirent le moins d'instructions et les 5 cœurs qui en retirent le plus.

CPU (min)	Committed insts	CPU (max)	Committed insts
cpu63	63 712	cpu04	73 299
cpu62	63 885	cpu03	73 467
cpu61	64 053	cpu02	73 635
cpu60	64 221	cpu01	73 803
cpu59	64 389	cpu00	196 793

TABLE II: Extrêmes de `committedInsts` à  $T = 64$  (size=64) : 5 plus faibles et 5 plus élevés.

Cela montre que même si les cœurs ont le même `numCycles` (même durée globale), ils n'exécutent pas la même quantité d'instructions : certains cœurs passent davantage de temps à attendre (synchronisation, contention mémoire, phases non uniformes), tandis que d'autres retirent plus d'instructions. Ainsi, le cas  $T = 64$  est cohérent : `numCycles` reflète un temps global partagé, alors que `committedInsts` reflète la charge effective par cœur.

e) *Conclusion pour Q4*: Pour  $T \in \{2, 4, 8, 16, 32\}$ , les valeurs de `numCycles` diffèrent entre cœurs (Tous égaux ? = NO), et le cœur critique correspond au maximum de `numCycles` (ici `cpu00`), ce qui caractérise directement le temps total par le chemin critique. Pour  $T = 64$ , `numCycles` est identique sur tous les cœurs (YES) : il n'existe alors pas de cœur plus long unique, et  $C(T)$  correspond à l'horizon temporel global de la simulation, tandis que la variabilité de charge se lit via `committedInsts` (Tableau II).

#### B. Q5 : cycles d'exécution en fonction du nombre de threads

La question Q5 demande de tracer un graphe 2D threads vs cycles. Pour rester cohérents avec la Q4, nous utilisons comme temps d'exécution applicatif :

$$C(T) = \max(\text{system.cpu}.*.\text{numCycles}),$$

où  $T$  est le nombre de threads OpenMP (avec `nthreads=ncores`).

a) *Résultats numériques* (extraits de `results/images/A7/q5_cycles.csv`):

Threads	Cycles
1	4 140 593
2	2 147 905
4	1 151 585
8	653 929
16	406 212
32	284 352
64	227 503

TABLE III: Cycles d'exécution obtenus pour Q5 (A7, size=64).

#### b) Visualisation 2D:

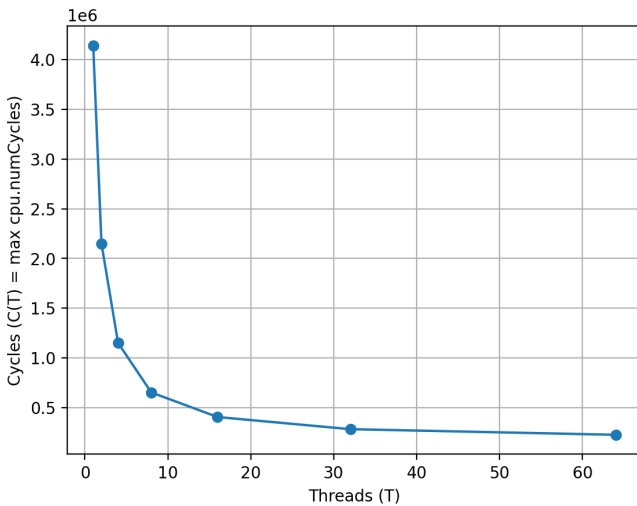


Fig. 1: Cycles d'exécution (A7, size=64) en fonction du nombre de threads, avec  $C(T) = \max(\text{system.cpu}.*.\text{numCycles})$ .

c) *Interprétation*: On observe une diminution monotone de  $C(T)$  lorsque  $T$  augmente, ce qui traduit l'exploitation du parallélisme au niveau des threads (TLP) sur une architecture CMP : la multiplication de matrices est répartie entre plusieurs cœurs, réduisant la quantité de travail par cœur et donc le temps total. Le gain reste toutefois sous-linéaire : (i) une partie des opérations et surcoûts est incompressible (initialisations, fork/join), (ii) les barrières OpenMP imposent des points de synchronisation, et (iii) les ressources partagées (hiérarchie mémoire, interconnexion, cohérence/cache) limitent l'accélération lorsque le nombre de cœurs/threads augmente.

### I. ARCHITECTURE MULTICOEURS AVEC DES PROCESSEURS SUPERSCAIRES OUT-OF-ORDER (CORTEX A15)

#### A. Stratégie adoptée pour traiter la Q9

Pour répondre à la Q9 (faire varier le nombre de threads et la largeur superscaire, puis produire un graphe 3D des cycles), nous avons mis en place une chaîne reproductible en quatre scripts :

- un script d'orchestration des simulations,
- un script `gem5 SE` adapté au modèle A15/o3,
- un script de post-traitement pour extraire les cycles et générer la visualisation,
- un script dédié à l'extraction/calcul de l'IPC.

Cette organisation permet de lancer une campagne complète, reprendre après erreur, tracer précisément chaque exécution, et générer automatiquement le CSV et la figure 3D demandés.

#### B. Script `run_q9_a15.sh`: orchestration de la campagne

##### Ce qu'il fait :

- lance toutes les combinaisons (size, width, threads) pour Q9 ;
- crée un répertoire de sortie par combinaison ;
- enregistre l'état d'avancement dans `state.tsv` (PENDING, DONE, FAILED) ;
- permet la reprise automatique après interruption/échec ;
- journalise chaque run dans un fichier de log dédié.

##### Comment il le fait :

- construit la commande `gem5` avec `-cpu-type=detailed, -o3-width, -num-cpus`, et les arguments du benchmark ;
- exécute les runs séquentiellement et s'arrête au premier échec pour conserver un diagnostic clair ;
- relit `state.tsv` au redémarrage pour ignorer les cas déjà DONE ;
- supporte un mode de mitigation OpenMP (`-omp-active-wait`) via un fichier d'environnement passé à `gem5`.

### C. Script `se_a15.py`: configuration `gem5` pour `A15/o3`

#### Ce qu'il fait :

- instancie un système `gem5` en mode `syscall emulation` (SE) ;
- configure des CPU de type `detailed` (modèle `o3`) ;
- applique la largeur superscalaire via `o3-width` ;
- exécute le binaire `test_omp` avec les paramètres `threads` et `size`.

#### Comment il le fait :

- s'appuie sur les options standard `gem5` (`Options.addCommonOptions`, `Options.addSEOptions`) ;
- crée `num_cpus` cœurs simulés et fixe `issueWidth` pour chaque cœur ;
- configure hiérarchie mémoire/caches et lance la simulation avec `Simulation.run()` ;
- prend en charge un fichier `-env` pour injecter des variables `OpenMP/libgomp` si nécessaire.

### D. Script `plot_q9_cycles.py`: extraction et visualisation

#### Ce qu'il fait :

- lit `state.tsv` et sélectionne les runs `DONE` valides ;
- extrait les cycles à partir des `stats.txt` de `gem5` ;
- génère un CSV consolidé ;
- produit le graphe 3D demandé (`threads`, largeur, cycles).

#### Comment il le fait :

- vérifie les colonnes attendues de `state.tsv` et la présence des fichiers `stats.txt` ;
- récupère la métrique `system.cpu*.numCycles` et conserve la valeur maximale par run ;
- écrit `q9_cycles.csv` puis trace `q9_cycles_3d.png` avec `Matplotlib` ;
- signale explicitement les combinaisons manquantes/invalides non incluses dans la figure.

### E. Script `extract_q9_ipc.py`: extraction de l'IPC

#### Ce qu'il fait :

- extrait `sim_insts` et `numCycles` des runs `DONE` ;
- calcule l'IPC pour chaque configuration (`width`, `threads`) ;
- exporte les résultats détaillés et les maxima.

#### Comment il le fait :

- lit `state.tsv`, filtre les runs valides et ouvre chaque `stats.txt` ;
- utilise `max(system.cpu*.numCycles)` en multi-cœur, avec fallback `system.cpu.numCycles` en mono-cœur ;
- écrit `results/images/A15/q9_ipc.csv` et `results/images/A15/q9_ipc_max.csv`.

### F. Expérimentation

L'expérimentation a été lancée avec le script `run_q9_a15.sh` en conservant les valeurs par défaut du script : `size=64`, `widths={2,4,8}`, `threads`

en puissances de 2, caches activés, et sorties dans `results/A15`.

Nous visions initialement une exploration jusqu'à 64 threads. En pratique, les exécutions à forte concurrence ont présenté des `SIGSEGV` de `gem5` (même après le message `Done` du benchmark), conformément au diagnostic détaillé dans `docs/report/sections/A15_boundary.md`.

Il est important de préciser que ce `Done` est imprimé par `test_omp` et signifie uniquement que le calcul applicatif est terminé ; il ne garantit pas la fin correcte de toute l'exécution `gem5`, puisque l'état `DONE` n'est validé que si le processus `gem5` se termine avec `exit=0`.

La mitigation `-omp-active-wait` (réduction de l'usage de la voie `futex/mutex`) a été déterminante pour stabiliser les cas en largeur 4, notamment à partir de `threads=16` et au-delà ; sans cette mitigation, plusieurs combinaisons échouaient.

a) Résultats numériques (extraits de `results/images/q9_cycles.csv`):

Width	Threads	Cycles
2	2	1282419
2	4	768565
2	8	515053
2	16	391465
2	32	341535
4	2	801594
4	4	520290
4	8	381832
4	16	318392
4	32	302483
8	2	785008
8	4	510238
8	8	376396
8	16	315224
8	32	299612

TABLE IV: Cycles d'exécution obtenus pour Q9 (size=64).

#### b) Visualisation 3D:

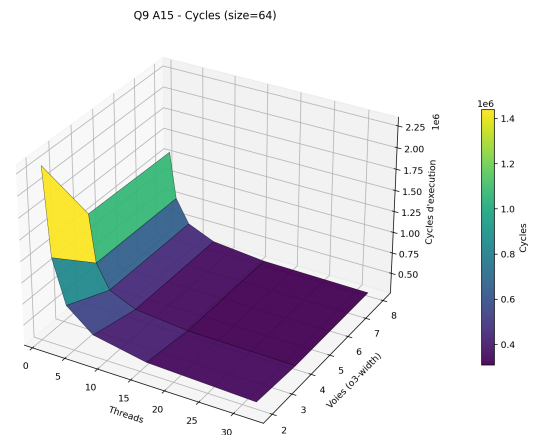


Fig. 2: Graphe 3D des cycles (X=threads, Y=voies/o3-width, Z=cycles).

c) Cycles de référence à 1 thread (extraits de `results/images/q9_speedup.csv`):

Width	Cycles (threads=1)
2	2308481
4	1365568
8	1334530

TABLE V: Cycles de référence utilisés pour le calcul du speedup.

d) *Calcul du speedup et résultats*: Le speedup est calculé, pour chaque largeur  $w$ , par rapport au cas `threads=1` de la même largeur :

$$S(w, t) = \frac{C_{w,1}}{C_{w,t}}$$

où  $C_{w,t}$  est le nombre de cycles de la configuration  $(w, t)$ . Dans notre cas, les valeurs obtenues sont celles de `results/images/q9_speedup.csv` :

Threads	Speedup (w=2)	Speedup (w=4)	Speedup (w=8)
1	1.000	1.000	1.000
2	1.800	1.704	1.700
4	3.004	2.625	2.616
8	4.482	3.576	3.546
16	5.897	4.289	4.234
32	6.759	4.515	4.454

TABLE VI: Speedup obtenu pour Q9 (size=64), calculé à partir de `q9_speedup.csv`.

e) *IPC maximal par configuration*: Pour traiter la question sur l'IPC, nous avons créé le script `scripts/extract_q9_ipc.py`. Ce script lit `state.tsv` et les `stats.txt`, calcule :

$$IPC(w, t) = \frac{\text{sim_insts}(w, t)}{\text{cycles}(w, t)}$$

et écrit les résultats dans :

- `results/images/A15/q9_ipc.csv`,
- `results/images/A15/q9_ipc_max.csv`.

Width	Threads au max	IPC max
2	32	14.696
4	32	19.221
8	32	23.301

TABLE VII: IPC maximal pour chaque largeur (size=64).

Le maximum global observé est **IPC = 23.301**, obtenu pour `width=8` et `threads=32`.

### G. Analyse de Résultats

a) *Limites d'exécution en gem5 (SE) : nombre de threads et stabilité*: Nous avons prévu d'explorer jusqu'à 64 threads, mais en pratique une limite nette de stabilité est apparue avec `gem5-stable` (mode SE) : au-delà d'un certain niveau de concurrence (dès `threads=40` dans nos essais), `gem5` termine en `SIGSEGV` (`exit=139`), parfois *après* que le benchmark ait affiché `Done` (le calcul applicatif est fini, mais la simulation n'est pas finalisée correctement). Cette limite est attribuée à une instabilité du simulateur plutôt qu'à un bug fonctionnel de `test_omp` (voir le diagnostic expérimental dans `docs/report/sections/A15_boundary.md`).

Un point important est que, dans ce TP, `nthreads = ncores` est imposé en mode SE (bibliothèque `pthread` en développement) : dans notre flot, `threads` est couplé à `-num_cpus` côté `gem5` (cf. sujet du TP `docs/consigne.pdf`). Autrement dit, augmenter `threads` augmente aussi le nombre de cœurs simulés, ce qui amplifie la pression sur la hiérarchie mémoire, la cohérence, et la synchronisation.

b) *Instabilité `futex/mutex` et mitigation `-omp-active-wait` (`width ≥ 4`)*: En plus de la limite « trop de threads », nous avons observé des échecs plus tôt pour des largeurs superscalaires plus élevées (à partir de `width=4`) quand le nombre de threads augmente. La cause la plus probable est liée au chemin de synchronisation `OpenMP/libgomp` en Linux : lors des barrières/verrous, `libgomp` utilise `futex` (\*fast userspace mutex\*) pour endormir/réveiller des threads sans consommer de CPU. Or, en `gem5 SE` (notamment sur des versions anciennes), la prise en charge de certains cas `futex` peut être incomplète/instable, et la fréquence accrue des synchronisations à forte concurrence augmente la probabilité de déclencher ce problème.

La mitigation utilisée est `-omp-active-wait` (décrite dans `scripts/A15/A15_commands.md`) qui injecte `OMP_WAIT_POLICY=ACTIVE` et un `GOMP_SPINCOUNT` très élevé : les threads attendent davantage en *spinning* en espace utilisateur, ce qui réduit les blocages/réveils via `futex`. Concrètement, cela a permis de faire passer des combinaisons qui échouaient auparavant (par exemple `width=4` avec plusieurs threads), même si, à très forte concurrence, une instabilité résiduelle peut encore persister (voir `docs/report/sections/A15_boundary.md`).

c) *Pourquoi les cycles diminuent quand on augmente les threads (TLP/CMP)*: La Table IV montre une tendance monotone : à largeur fixe, plus le nombre de threads est grand, plus le nombre de cycles diminue. Cela s'explique principalement par le parallélisme au niveau des threads (TLP) sur une architecture CMP : chaque thread `OpenMP` exécute une partie du travail (multiplication de matrices) sur un cœur distinct, et le temps total est gouverné par le cœur le plus long (métrique `max(system.cpu*.numCycles)`). En augmentant threads, on réduit la quantité de travail par cœur et on augmente le parallélisme global, donc le nombre de cycles d'exécution diminue.

Le gain reste cependant sous-linéaire (cf. Table VI) à cause (i) des portions sérielles incompressibles (création/fin des threads, initialisations), (ii) des surcoûts de synchronisation (barrières `OpenMP`), et (iii) des effets de hiérarchie mémoire/cohérence quand beaucoup de cœurs accèdent aux mêmes structures de données (bus/mémoire partagés).

d) *Pourquoi les cycles diminuent quand on augmente la largeur superscalaire (ILP/OoO)*: À nombre de threads donné, on observe aussi une baisse des cycles quand `width` augmente (Table IV). Ici, `width` correspond au degré de traitement superscalaire côté cœur (dans nos scripts, c'est `l'issueWidth` du modèle `o3`). Une largeur plus grande permet d'émettre davantage d'instructions par cycle quand le code présente suffisamment d'indépendances (ILP), et l'exécution out-of-order contribue à mieux cacher des latences (par exemple en chevauchant calculs et accès mémoire). Cela correspond au positionnement « hautes performances » du Cortex-A15, conçu pour exploiter agressivement l'ILP.

La différence entre `width=4` et `width=8` devient toutefois plus faible à forte concurrence : une partie des cycles est alors contrainte par la synchronisation et la mémoire partagée,

et non plus uniquement par la largeur d'émission d'instructions (rendements décroissants).

e) *Lecture du graphe 3D*: La Figure 2 illustre la même tendance sous forme de surface : on observe une « colline » pour `threads` faibles et `width` faible (beaucoup de cycles), puis une descente progressive quand on augmente l'un et/ou l'autre paramètre. Le « vallon » de cycles minimaux correspond à la zone de plus forte exploitation conjointe du parallélisme TLP (plus de cœurs) et de l'ILP (cœurs plus larges), dans la limite des surcoûts mémoire/synchronisation.

f) *Speedup : pourquoi le maximum est à `width=2` sans être le meilleur temps absolu*: Dans la Table VI, le meilleur speedup est obtenu pour `width=2` et `threads=32` ( $S = 6.759$ ), supérieur à `width=4` (4.515) et `width=8` (4.454). Ce résultat s'explique d'abord par la définition du speedup :  $S(w, t) = C_{w,1}/C_{w,t}$ . Or, le cas mono-thread à `width=2` est nettement plus lent ( $C_{2,1}$  est bien plus grand que  $C_{4,1}$  et  $C_{8,1}$ , Table V), ce qui donne mécaniquement davantage de “marge” pour augmenter le ratio.

Ensuite, avec des cœurs plus larges (`width=4/8`), chaque cœur produit plus de requêtes et atteint plus vite un régime limité par des ressources partagées (mémoire, cohérence, barrières OpenMP). On « plafonne » donc plus tôt en speedup relatif, même si le temps absolu continue de baisser. En performance pure (cycles minimaux), la meilleure configuration observée est `width=8`, `threads=32` (299 612 cycles, Table IV). Ainsi, le “gagnant” dépend du critère : `width=2` maximise le speedup relatif, tandis que `width=8` minimise les cycles (et correspond mieux à une configuration A15 performante).

g) *IPC : maximum à `threads=32` et notion de « configuration la plus efficace »*: La Table VII montre que, pour chaque largeur, l'IPC maximal est atteint à `threads=32`, et que le maximum global est obtenu pour `width=8`, `threads=32` avec  $IPC = 23.301$ . Dans notre extraction, l'IPC est calculé comme `sim_insts/cycles` avec `cycles = max(system.cpu*.numCycles)` : c'est donc un *IPC global* (débit d'instructions agrégé sur la durée critique), qui augmente naturellement quand on ajoute des cœurs/threads et quand les cœurs sont plus capables de retirer des instructions.

Dans le cadre de ce TP et de cette métrique, `width=8`, `threads=32` est bien la configuration la plus “efficace” au sens *débit par cycle* et aussi la plus performante en cycles. En revanche, cela ne préjuge pas de l'efficacité énergétique ou du coût matériel (non modélisés ici), et le fait que `threads=32` soit maximal reflète aussi notre limite opérationnelle de stabilité (au-delà, `gem5` devient instable).

## REFERENCES

- [1] TP5, “Analyse de performances de configurations de microprocesseurs multicœurs pour des applications parallèles,” document de travaux pratiques du cours.
- [2] O. Hamami, *Introduction à l'Architecture des Microprocesseurs*, ENSTA ParisTech, 828 Bvd des Maréchaux 91762 Palaiseau cedex, <https://www.ensta-paristech.fr>, ENSTA PARIS - Cours ES201, Année 2022.