Analyse de performances de configurations de microprocesseurs multicoeurs pour des applications parallèles

Rapport de Travaux Pratiques

Luiz Gariglio Dos Santos
Helena Guachalla De Andrade
Santiago Florido Gomez
Franck Ulrich Kenfack Noumedem

Programme : Ingénieur STIC

Ecole Nationale des Techniques Avancees
ENSTA Paris
Fevrier 2026

Table des matières

1 Resume

Ce rapport présente une analyse expérimentale des performances d'une architecture multicœurs exécutant un produit matriciel parallélisé avec OpenMP, en mettant l'accent sur l'impact de la hiérarchie mémoire, de la cohérence de cache et des paramètres architecturaux. L'étude utilise deux types de processeurs superscalaires in-order, modélisant des configurations de type Cortex-A7 et Cortex-A15, simulées sous Gem5. Nous analysons l'évolution du nombre de cycles, du speedup, de l'efficacité parallèle et de l'IPC par rapport le nombre de threads et la taille du problème. Une étude complémentaire compare les performances avec et sans cache L2 afin de quantifier l'impact d'un niveau intermédiaire de mémoire sur la scalabilité. Les résultats montrent que, bien que la parallélisation améliore significativement les performances pour des tailles de matrices importantes, le gain reste limité par la fraction séquentielle du code, la contention mémoire et le trafic de cohérence. De cette façon, l'ensemble des résultats met en évidence le compromis fondamental entre parallélisme, hiérarchie mémoire et coûts de synchronisation dans les architectures multicœurs.

2 Introduction

L'amélioration des performances des systèmes informatiques modernes repose largement sur le parallélisme multicœur, mais l'augmentation du nombre de cœurs ne garantit pas une accélération proportionnelle des applications. Dans ce travail, nous étudions le comportement d'une application de multiplication de matrices parallélisée avec OpenMP sur une architecture multicœurs simulée. L'objectif est d'analyser les facteurs limitant la scalabilité et de quantifier l'impact des choix architecturaux et de la hiérarchie mémoire.

L'étude se structure autour de trois axes principaux. Premièrement, une analyse du mécanisme de cohérence de cache dans une architecure à bus partagé, afin de comprendre le trafic induit par les lectures et écritures concurrentes. Ensuite, une évaluation expérimentale des performances sur un processeur de type Cortex-A7 (in-order), en étudiant l'évolution des cycles d'exécution, du speedup, de l'efficacité et de l'IPC selon le nombre de threads et la taille des matrices. Enfin, ne comparaison avec un processeur de type Cortex-A15, incluant l'analyse de l'impact de la largeur du pipeline (issue width) sur les performances parallèles. Une étude complémentaire examine l'effet de la suppression du cache L2 afin d'isoler le rôle de ce niveau intermédiaire dans la réduction de la contention mémoire et l'amélioration de la scalabilité.

À travers ces expérimentations, réalisées sous Gem5, nous mettons en évidence les limites du parallélisme OpenMP : poids des sections séquentielles, surcoût des synchronisations, trafic de cohérence et saturation de la hiérarchie mémoire. Les résultats illustrent concrètement la manière dont l'architecture et l'organisation mémoire conditionnent les gains réels obtenus par la parallélisation.

3 Analyse théorique de cohérence de cache

On considère que chaque thread s'exécute sur un processeur dans une architecture de type multicoeurs à base de bus et 1 niveau de cache pour l'algorithme de multiplication de matrices. Les caches privés stockent les données locales, pour réduire le temps d'accès à la mémoire, et toutes les transactions passent par le bus. Pour la cohérence de cache, le Snooping est donc utilisé pour surveiller ce bus et effectuer les mises-à-jour correspondantes.

Dans la phase d'initialisation, le thread principal remplit A et B sur le processeur 1, et les données montent dans son cache. Lorsque les autres processeurs commencent le calcul, ils émettent des requêtes de lecture sur le bus, et les lignes de cache contenant A et B passent à l'état partagé dans tous les caches.

Dans la phase d'écriture, chaque thread calcule un élément $C[i][j]$ et tente de l'écrire. Pour réussir, il doit obtenir l'exclusivité, et s'il possède la ligne en état partagé, il doit envoyer un signal d'invalidation sur le bus. De cette manière, tous les autres caches marquent la ligne comme invalide et le processeur écrivain la passe en état modifié.

Ce système pose des problèmes lorsque le nombre de processeurs augmente. Le bus est de plus en plus sollicité pour les lectures et les signaux d'invalidation, ce qui peut saturer le bus et limiter le gain de performance.

4 Paramètres de l'architecture multicœurs

4.1 Paramètres micro-architecturaux configurables (extrait du fichier de configuration)

En plus de la configuration des caches d'instructions (I-cache) et du cache L2 unifié (présents dans le fichier d'options), le fichier de configuration du microprocesseur permet aussi d'identifier d'autres éléments micro-architecturaux importants. Par exemple, le prédicteur de branchement indiqué est un *Tournament Branch Predictor*, c'est-à-dire un prédicteur « composé » qui combine plusieurs prédicteurs internes (par exemple un prédicteur local et un prédicteur global) et s'appuie sur un méta-prédicteur pour décider, selon le contexte, lequel des deux doit être privilégié. En pratique, ce mécanisme fournit généralement une meilleure précision qu'un prédicteur plus simple, au prix d'une complexité matérielle et d'un coût énergétique potentiellement plus élevés. On trouve également une *fetch queue*, située entre les étages *Fetch* et *Decode* (ou très proche de cette frontière), dont le rôle est de stocker les instructions ou micro-opérations déjà fetch mais pas encore décodées ; elle permet de poursuivre le fetch tant que la file n'est pas pleine lorsque le decode est ralenti par backpressure, et inversement de maintenir le decode actif en consommant les μops déjà présentes lorsque le fetch est bloqué, par exemple à cause d'un *miss* du cache d'instructions. Sa capacité est de 32 μops/thread. Le paramètre *DecodeWidth* correspond au nombre maximal de micro-opérations décodées par cycle, ici 8 μops/cycle, tandis que *IssueWidth* représente le nombre maximal de μops émises par cycle depuis la file de scheduling vers les unités fonctionnelles, également 8 μops/cycle. De même, *CommitWidth* (ou *RetireWidth*) fixe le nombre maximal d'instructions validées par cycle, soit 8 instructions/cycle. Le *Reorder Buffer* (ROB) conserve les instructions en vol afin de garantir un commit en ordre (et la gestion des exceptions précises) et comporte ici 192 entrées. L'*Instruction Queue* (IQ), aussi appelée *Issue Queue*, est la structure où attendent les instructions renommées jusqu'à ce que les opérandes soient prêts et qu'une unité fonctionnelle soit disponible, avec une capacité de 64 entrées. Enfin, la *Load Queue* (LQ) et la *Store Queue* (SQ) suivent respectivement les loads et les stores en vol pour gérer la spéculation mémoire, détecter les violations de dépendances et respecter les contraintes d'ordre interne ; chacune dispose de 32 entrées.

Les cinq paramètres configurables de référence pour la description de l'architecture du microprocesseur sont présentés dans le Tableau 1.

TABLE 1 – Paramètres micro-architecturaux configurables (extrait du fichier de configuration).

Paramètre	Valeur
`branchPred`	`TournamentBP(numThreads = Parent.numThreads)`
`fetchQueueSize`	32
`decodeWidth`	8
`issueWidth`	8
`numROBEntries`	192

4.2 Paramètres cache configurables (extrait du fichier de options)

Les configurations des caches de données et d'instructions de niveau 1, ainsi que la configuration du cache unifié de niveau 2, sont extraites du fichier d'options en considérant l'associativité, la taille du cache et la taille des lignes, comme présenté dans la Table 2.

TABLE 2 – Configurations des caches (extrait du fichier d'options).

Niveau	Type	Associativité	Taille du cache	Taille de ligne
L1	D-cache	2	64 kB	64 B
L1	I-cache	2	32 kB	64 B
L2	Unifiée	8	2 MB	64 B

5 Architecture multicœurs avec des processeurs superscalaires in-order (Cortex A7)

5.1 Chemin critique, synchronisation et cycles dominants en exécution OpenMP

En analysant l'architecture multicœur proposée et la manière dont l'algorithme est développé afin de garantir la cohérence des caches, on observe que le processus *master* réalise un travail séquentiel indispensable à l'obtention du résultat final, non parallélisable et donc obligatoire. Ce travail apparaît en une étapes principale , l'initialisation de A et B au moyen de boucles complètes, impliquant davantage d'opérations et, par conséquent, un plus grand nombre d'accès mémoire . Ce traitement n'est pas réparti : il repose sur le cœur exécutant le thread principal. Ainsi, même si les autres cœurs exécutent des calculs en parallèle, ce cœur supporte une « file » de travail supplémentaire. Il convient aussi de noter qu'OpenMP, à la fin d'une région parallèle, impose des barrières de synchronisation : lorsqu'un *worker* termine plus tôt, il reste en attente.

Dans un programme parallèle, le temps total n'est pas « la somme » des temps de tous les threads ; il correspond au temps nécessaire pour que l'application puisse se terminer. Or, l'application ne peut se terminer que lorsque tous les threads ont fini leur travail . On peut formaliser cette idée par l'équation (1) :

$$T_{\text{app}} = T_{\text{init}}^{(\text{master})} + \max_i \left(T_{\text{calc}}^{(\text{thread } i)} \right). \tag{1}$$

Le terme déterminant est le maximum : même si 15 threads terminent rapidement, si un seul thread prend plus de temps, tous les autres restent bloqués à la synchronisation (*barrier*/*join*) en attendant le dernier. . Ainsi, le master accumule souvent davantage de cycles, car il exécute $T_{\text{init}}^{(\text{master})}$ en plus du calcul parallèle, et il peut aussi inclure des temps d'attente dus aux synchronisations (*barrier*/*join*) ainsi qu'aux effets mémoire et de cohérence. Cela s'accorde avec l'analyse du bus et de la cohérence : lorsque le nombre de cœurs augmente, on observe davantage

de lectures sur le bus (matrices partagées A et B) et davantage d'invalidations lors des écritures sur C, ce qui induit des *stalls* (temps d'arrêt en attente de bus ou d'exclusivité de ligne).

5.2 Cycles per configuration

La Figure 1 présente le nombre de cycles d'exécution en fonction du nombre de processus exécutés pour les configurations Cortex-A7 disponibles.

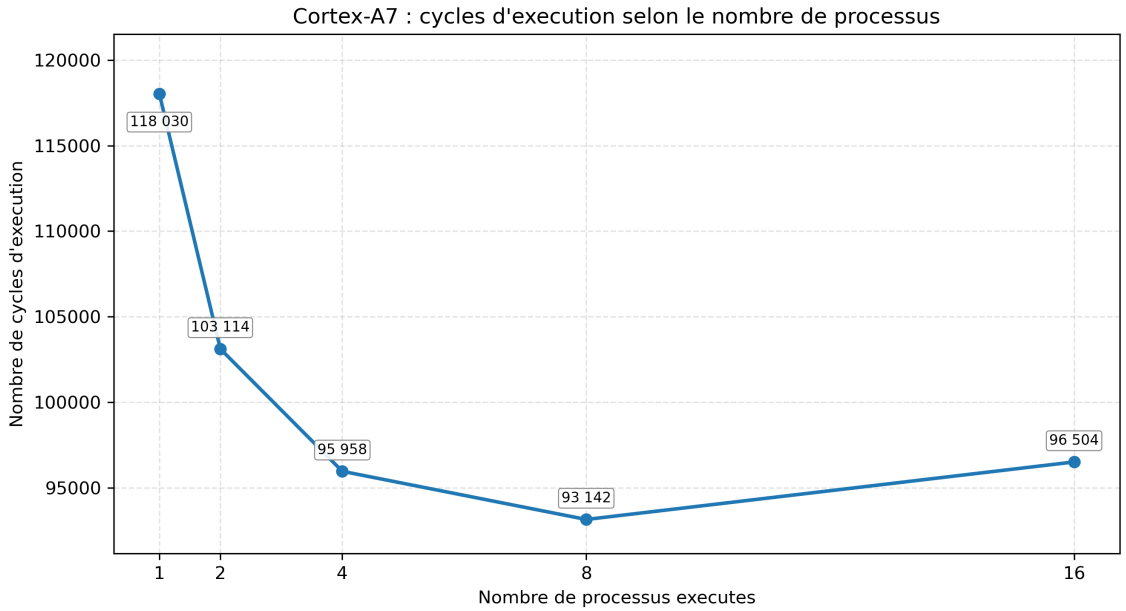


FIGURE 1 – Cortex-A7 : cycles d'exécution de l'application selon le nombre de processus.

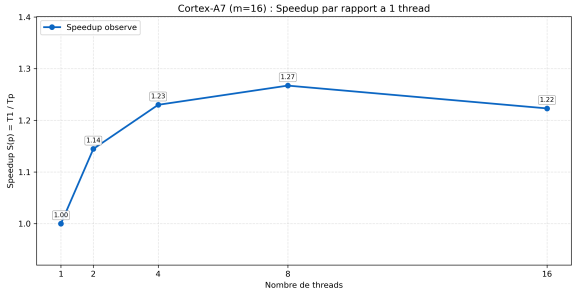
Il est proposé de poursuivre l'analyse avec l'estimation du nombre minimal de cycles d'exécution de l'application pour une taille de matrice $M = 16$, et ce pour chacune des configurations proposées. Les résultats correspondants sont présentés dans la Figure 1. On observe que, même si l'exécution devient plus rapide, le gain de performance n'est pas proportionnel au nombre de threads utilisés, principalement à cause de l'impact des accès mémoire et de la présence des étapes du traitement qui ne sont pas parallélisables et dont le coût devient prépondérant lorsque le temps de calcul parallèle diminue. En particulier, les phases d'initialisation, exécutées séquentiellement par le *master*, constituent une fraction croissante du temps total à mesure que l'optimisation accélère la région de calcul en parallèle. De plus, lorsque le nombre de threads augmente, la quantité d'accès concurrents aux caches partagés (L2/L3) et à la RAM croît également, ce qui génère de la contention et un plus grand nombre de défauts de cache, réduisant ainsi le gain marginal apporté par l'ajout de threads. Par ailleurs, l'écriture concurrente dans la matrice C induit un trafic supplémentaire de cohérence de cache (invalidations et mises à jour de lignes), introduisant des surcoûts absents de la version séquentielle. Enfin, il existe un surcoût propre à OpenMP (création et gestion des threads, ainsi que barrières implicites à la fin du `parallel for`) qui devient relativement plus significatif lorsque la charge de travail par thread n'est pas suffisamment élevée ou lorsque le *blocking* pour le cache n'est pas mis en œuvre de manière efficace. Par conséquent, le *speedup* observé reflète la combinaison de l'amélioration due au parallélisme et des limites pratiques imposées par la hiérarchie mémoire, ainsi que des coûts de coordination.

Il convient de souligner que, dans le cas où m correspond au nombre de threads, on observe une augmentation du nombre de cycles en passant à 16 threads par rapport à l'exécution avec 8

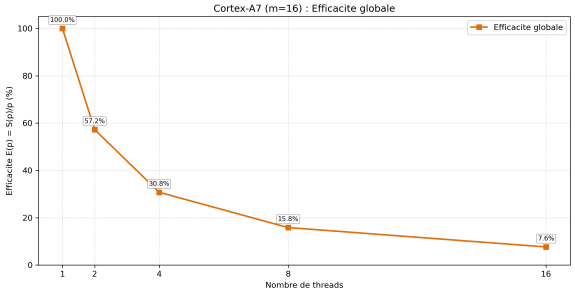
threads. Ce comportement peut s'expliquer par le fait qu'avec 16 threads, la charge de travail utile par thread devient si faible que le surcoût OpenMP domine l'exécution de la phase parallèle. Cette dégradation est notamment amplifiée par l'augmentation du trafic de cohérence (invalidations et transferts de lignes lors des écritures) ainsi que par une pression accrue sur la mémoire partagée, en particulier le cache L2.

5.3 Speed-Up per configuration

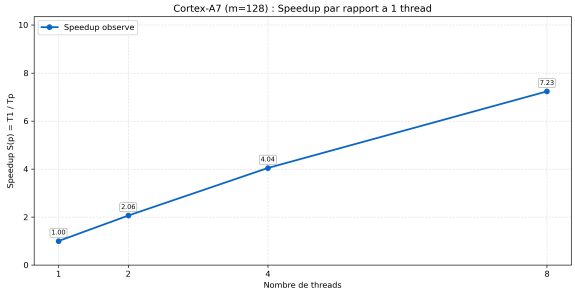
Les résultats de *speedup* et d'efficacité globale, calculés par rapport à la configuration à 1 thread, sont présentés pour $m = 16$ et $m = 128$ dans la Figure 2.



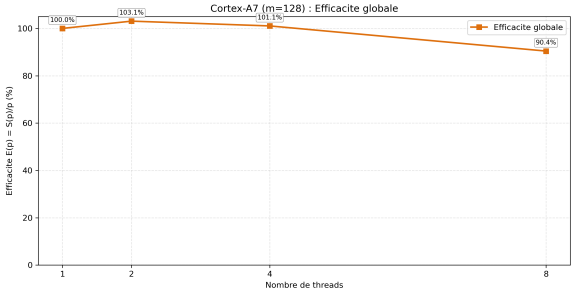
(a) $m = 16$: *Speedup*



(b) $m = 16$: Efficacité globale



(c) $m = 128$: *Speedup*



(d) $m = 128$: Efficacité globale

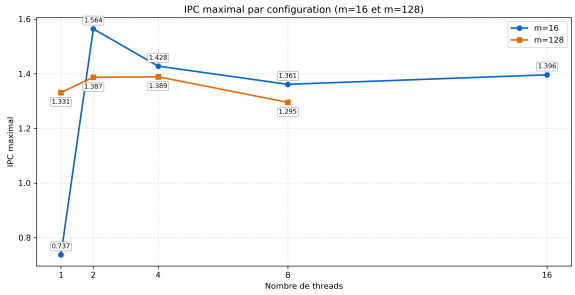
FIGURE 2 – Comparaison du *speedup* et de l'efficacité globale pour les configurations Cortex-A7 ($m = 16$ et $m = 128$).

En analysant les *speedups* pour $m = 16$, on constate que la quantité d'opérations réellement parallélisables n'est pas suffisamment représentative par rapport aux surcoûts de parallélisation d'OpenMP, voire par rapport à la section linéaire du programme. Cette observation est cohérente avec le fait que les *speedups* entre configurations restent inférieurs à 1.3 dans le meilleur des cas, et qu'ils demeurent éloignés de la courbe idéale de *speedup*. De plus, si l'on évalue l'efficacité liée à l'ajout de nouveaux threads, on observe qu'elle diminue fortement à mesure que le nombre de threads augmente. En revanche, lorsque l'on accroît la dimension de la matrice, l'augmentation de la complexité et du volume des opérations parallélisables se traduit par des *speedups* plus significatifs lors de l'augmentation du nombre de threads. C'est notamment le cas pour une matrice de taille $m = 128$, où les *speedups* obtenus se rapprochent davantage du comportement idéal, ce qui se reflète par des efficacités proches de 1 pour les différentes configurations de cette application avec $m = 128$.

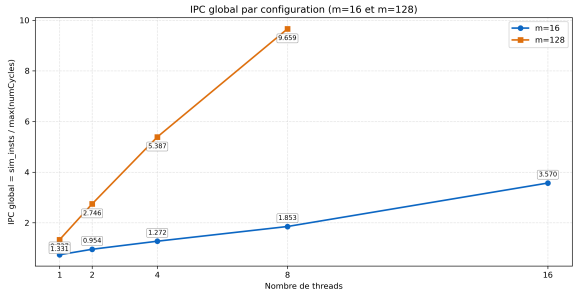
5.4 valeur maximale de l'IPC pour chaque configuration

En termes d'IPC, il est proposé d'analyser l'IPC maximal par CPU pour les configurations considérées ; on met alors en évidence qu'il n'existe pas de relation directe avec le nombre de threads employés. En revanche, si l'on considère l'IPC global de chaque configuration, on observe que le nombre d'instructions traitées par cycle augmente de manière quasi linéaire avec l'ajout de chaque thread, en s'appuyant sur l'expression donnée à l'équation (2). Il convient également de mentionner que, lorsque la taille de la matrice est plus grande, l'IPC global obtenu est sensiblement supérieur à celui mesuré pour une valeur de m plus faible, ce qui se traduit par une pente plus marquée de la droite reliant les valeurs d'IPC estimées pour chaque configuration.

$$\mathrm{IPC}_{\mathrm{global}} = \frac{\mathrm{sim\_insts}}{\max(\mathrm{numCycles})}. \tag{2}$$



(a) IPC maximal par configuration (par CPU)



(b) IPC global par configuration (sim_insts/ max(numCycles))

FIGURE 3 – Comparaison des métriques IPC pour les simulations Cortex-A7 ($m = 16$ et $m = 128$).

5.5 Hiérarchie mémoire sans L2

5.5.1 Cycles per configuration

Il est proposé, en tant que composante d'analyse supplémentaire, d'étudier les résultats de la même configuration mais en l'absence d'une mémoire cache L2 pour l'application, en ne considérant dans la hiérarchie mémoire que les deux caches L1 (cache de données et cache d'instructions). Dans ce contexte, il a été choisi d'évaluer le comportement pour une taille de matrice $m = 128$, afin de garantir une comparaison directe avec les résultats présentés précédemment.

Pour la hiérarchie mémoire avec cache L2 activé, les résultats Cortex A7 (*in-order*) pour $m = 128$ sont résumés dans la Table 3.

TABLE 3 – Résultats Cortex A7 (*in-order*), $m = 128$.

Threads	sim_ticks	Cycles max	sim_insts
1	45 387 718 500	90 775 438	32 000 412
2	23 542 334 500	47 084 670	32 016 938
4	14 050 578 500	28 101 158	32 068 570
8	13 601 720 500	27 203 442	32 866 791
16	11 374 642 500	22 749 286	35 160 170

5.5.2 Speed-Up per configuration

Le *speedup* augmente jusqu'à 4 threads puis tend à se stabiliser. Les valeurs observées sont résumées dans la Table 4.

TABLE 4 – Speedup observé par rapport à 1 thread (Cortex-A7, sans L2, $m = 128$).

Threads	Speedup
2	1.93
4	3.23
8	3.34
16	3.99

Ces résultats sont conformes à ce qui était attendu : on observe un nombre de cycles considérablement plus élevé pour exécuter le produit matriciel, alors même que le nombre d'instructions reste comparable. Cela met en évidence non seulement l'effet bénéfique d'une mémoire partagée de type L2 placée avant le bus reliant les cœurs à la RAM, mais aussi l'impact de L2 sur la dynamique de parallélisation. En effet, l'analyse des *speedups* lors de l'augmentation du nombre de threads montre que l'optimisation des mêmes opérations est fortement dégradée en l'absence de L2 : la réduction du nombre de cycles s'écarte de manière bien plus marquée de la droite idéale que dans le cas du processeur disposant d'une hiérarchie mémoire complète. Ce comportement est cohérent avec la contention accrue sur les niveaux de mémoire restants, qui induit davantage de *stalls* et des accès plus coûteux, entraînant des instructions effectives prenant plus de cycles. Ainsi, l'absence de L2 apparaît comme l'un des principaux facteurs limitants de l'amélioration via la parallélisation OpenMP, ce qui explique la dynamique non linéaire et la saturation du *speedup* lorsque le nombre de threads augmente de manière significative.

5.5.3 valeur maximale de l'IPC pour chaque configuration

L'IPC maximal augmente avec le parallélisme : de 0.352 à 1 thread jusqu'à 1.546 à 16 threads. Cette progression reflète un meilleur recouvrement des latences mémoire grâce au parallélisme.

5.6 Discussion et interprétation

En conclusion, le produit matriciel profite de la parallélisation OpenMP, car une part significative du calcul est parallélisable. Cependant, le *speedup* n'est pas proportionnel au nombre de threads à cause des phases séquentielles (initialisation et validation) exécutées par le *master*. Quand le nombre de threads augmente, les accès concurrents aux caches partagés et à la RAM augmentent aussi, ce qui crée de la contention. Cette contention se traduit par davantage de défauts de cache et de *stalls*, limitant le gain marginal des threads supplémentaires. Les écritures parallèles dans la matrice C accroissent le trafic de cohérence (invalidations et transferts de lignes). De plus, l'overhead OpenMP (gestion des threads et barrières implicites) devient plus visible lorsque la charge par thread diminue. Cela peut expliquer une saturation, voire une dégradation, en passant de 8 à 16 threads sur certains cas. Ainsi, la hiérarchie mémoire est un facteur déterminant des performances et de la scalabilité. Les gains sont plus proches de l'idéal lorsque la taille du problème augmente et amortit les surcoûts. Les résultats observés reflètent donc un compromis entre parallélisme, mémoire et coûts de coordination.

6 Architecture multicœurs avec des processeurs superscalaires in-order (Cortex A15)

6.1 Chemin critique, synchronisation et cycles dominants en exécution OpenMP

De la même manière que dans le cas du Cortex-A7, le goulot d'étranglement se situe dans l'exécution des sections non parallélisables du code ; par conséquent, c'est le *master* qui détermine le maximum de cycles pour chaque exécution, selon les configurations considérées. Cela reste vrai même si ce cœur n'est pas nécessairement celui qui passe le plus de temps dans l'exécution des sections strictement parallèles. On retrouve ainsi la même conclusion que pour l'autre microprocesseur et l'ensemble de ses configurations.

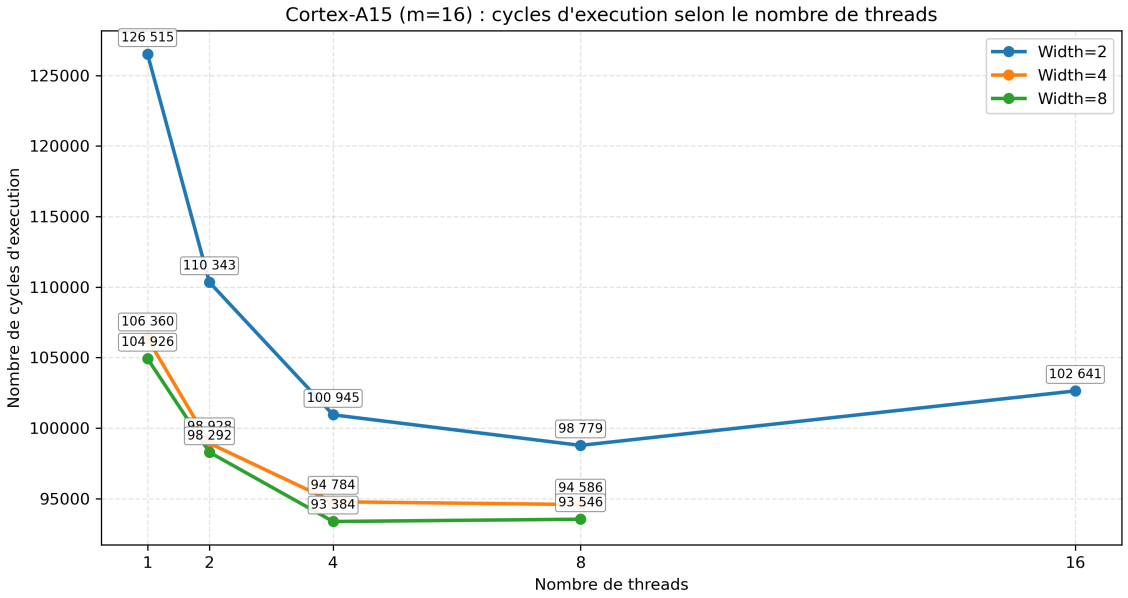


FIGURE 4 – Cortex-A15 ($m = 16$) : nombre de cycles d'exécution selon le nombre de threads et la largeur du pipeline.

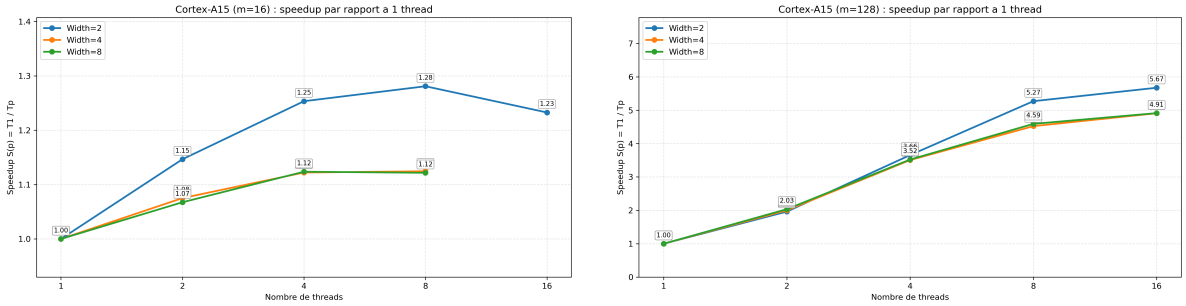
6.2 Speed-Up per configuration

Dans le but de disposer d'une perspective de comparaison valide, il est proposé d'analyser le rendement du système en parallélisation dans des conditions comparables à celles du Cortex-A7. Ainsi, on vérifie le comportement pour 1 à 16 threads avec des matrices de taille $m = 16$, en considérant ici l'effet du *largeur de voie* (*issue width*). Il apparaît que, comme dans le cas du A7, pour ces configurations la fraction de calcul parallélisable n'est pas suffisamment représentative au regard des pénalités mémoire (défauts et latences), des surcoûts liés à l'augmentation du nombre de threads (overhead OpenMP) et du poids des sections linéaires non parallélisables. Par conséquent, les *speedups* ne dépassent pas 1.3 et, de manière attendue, on observe des *speedups* légèrement plus élevés pour les configurations à largeur de voie plus importante, celles-ci permettant de traiter davantage d'instructions par cycle et donc d'améliorer le débit d'exécution lorsque l'application n'est pas strictement bornée par la mémoire.

Afin de comparer ces résultats à un cas où la complexité de calcul augmente, il a également été proposé, comme pour l'étude du Cortex-A15, d'évaluer la parallélisation de 1 à 16 threads pour des largeurs de voie de 2, 4 et 8 sur le produit matriciel avec $m = 128$. Les résultats

mettent en évidence une dynamique comparable à celle observée pour $m = 128$ sur le A7 : un comportement initial relativement proche de l'idéal lorsque l'on augmente le nombre de threads, puis une saturation à partir de 8 threads, avec un écart croissant à la courbe idéale. Cette saturation s'explique par l'augmentation des opérations mémoire et des défauts de cache, par l'overhead associé à un plus grand nombre de threads, ainsi que par le coût relatif des sections linéaires non parallélisables qui devient plus visible lorsque la partie parallèle se réduit. Enfin, on peut conclure que la configuration à largeur de voie 2 présente un nombre de cycles supérieur aux autres pour l'ensemble des nombres de threads, et qu'on observe globalement une relation inverse entre le nombre de threads et le nombre de cycles requis par l'application. Néanmoins, cette configuration (voie 2), étant la moins performante en absolu, se révèle aussi plus sensible aux gains de la parallélisation, ce qui se traduit par des *speedups* plus marqués lorsqu'on augmente le nombre de threads, par rapport aux configurations à 4 et 8 voies qui, elles, présentent des dynamiques plus proches l'une de l'autre.

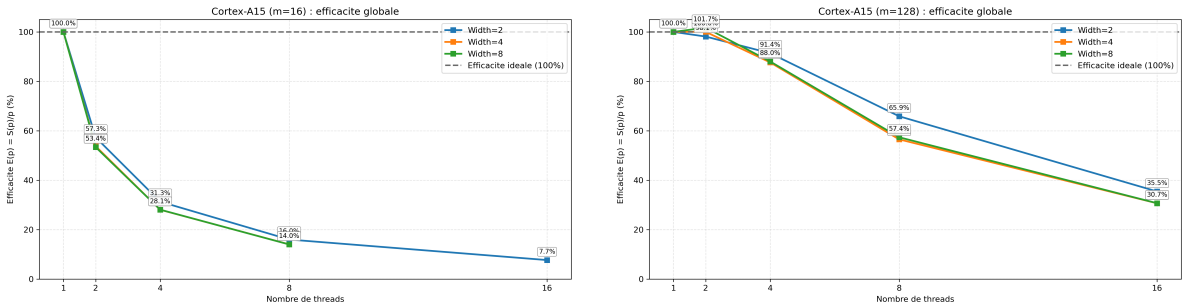
Les courbes de *speedup* (référence à 1 thread) pour Cortex-A15 sont présentées pour $m = 16$ et $m = 128$ dans la Figure 5.



(a) $m = 16$: *Speedup* (b) $m = 128$ (jusqu'à 16 threads) : *Speedup*

FIGURE 5 – Comparaison du *speedup* pour Cortex-A15 selon la largeur du pipeline ($m = 16$ et $m = 128$).

Les courbes d'efficacité globale associées aux mêmes configurations sont présentées dans la Figure 6.

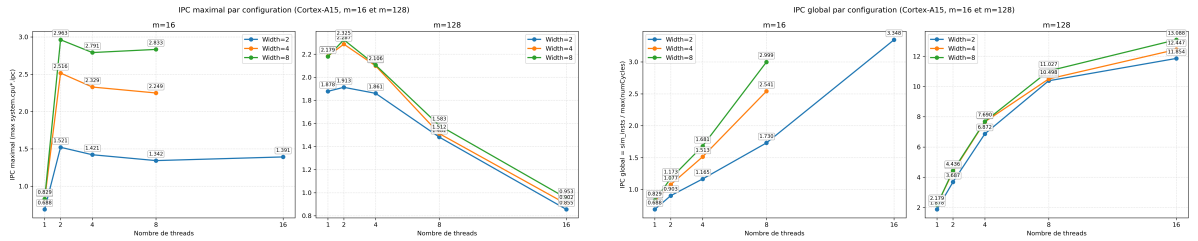


(a) $m = 16$: Efficacité globale (b) $m = 128$ (jusqu'à 16 threads) : Efficacité globale

FIGURE 6 – Comparaison de l'efficacité globale pour Cortex-A15 selon la largeur du pipeline ($m = 16$ et $m = 128$).

6.3 valeur maximale de l'IPC pour chaque configuration

Pour Cortex-A15, l'IPC peut être analysé selon deux indicateurs complémentaires : l'IPC maximal observé sur un cœur (*max system.cpu*.ipc*) et l'IPC global de la configuration (sim_insts/ max(numCycles)). Les résultats pour $m = 16$ et $m = 128$, en distinguant les largeurs de voie, sont présentés dans la Figure 7.



(a) IPC maximal par configuration

(b) IPC global par configuration

FIGURE 7 – Comparaison des métriques IPC pour Cortex-A15 ($m = 16$ et $m = 128$).

En termes d'IPC maximal par thread dans la simulation, on observe pour $m = 16$ une dynamique similaire à celle discutée précédemment : l'IPC maximal atteint par chacun des threads ne dépend pas directement du nombre de threads utilisés par l'application. En revanche, pour une dimension de matrice $m = 128$, plusieurs effets peuvent conduire à une diminution de l'IPC maximal des threads. D'une part, la contention en L2 et au niveau de la mémoire augmente lorsque davantage de cœurs demandent simultanément des données, ce qui se traduit par plus de *misses*, davantage de files d'attente et une latence effective plus élevée ; ainsi, malgré la présence de plus de cœurs, chacun passe proportionnellement plus de temps à attendre les données. D'autre part, la cohérence de cache et le trafic associé (et, dans certains cas, le *false sharing*) peuvent dégrader l'IPC lorsque plusieurs threads accèdent à des données proches partageant les mêmes lignes de cache, provoquant des invalidations et un effet de *ping-pong* coûteux. Enfin, l'overhead d'OpenMP augmente avec le nombre de threads : barrières, répartition du travail et synchronisations représentent un coût relatif plus important, en particulier lorsque la charge utile par thread diminue ; avec 16 threads et $m = 128$, la quantité de travail assignée à chaque thread peut devenir suffisamment faible pour que cet overhead pèse fortement sur l'exécution. Néanmoins, du point de vue de l'IPC global, la dynamique reste comparable à celle observée sur le A7, avec des augmentations quasi linéaires de l'IPC lorsque le nombre de threads croît. De plus, lorsque la dimension de la matrice atteint $m = 128$, on obtient des IPC globaux sensiblement plus élevés, ce qui traduit une meilleure exploitation du parallélisme et une contribution plus marquée de l'optimisation des opérations de calcul lorsque le nombre de threads employés augmente.

6.4 Discussion et interprétation

Dans un scénario où la charge de calcul est faible, l'intégration de multiples threads produit bien un *speedup*, mais la dynamique de réponse du système devient rapidement dominée par les accès mémoire et par la synchronisation. Cela implique que l'effet d'agrégation de threads est moins significatif en termes de nombre de cycles nécessaires pour réaliser la multiplication matricielle. C'est pourquoi, lorsque la taille de la matrice augmente et que le problème devient plus *compute bound*, on commence à obtenir de meilleurs *speedups*, comme on l'observe également avec le microprocesseur Cortex-A15.

Par ailleurs, l'augmentation de la largeur des voies (par exemple de 2 à 4 puis 8) peut être reliée à

une diminution du nombre de cycles requis pour exécuter le produit matriciel, puisque davantage d'instructions peuvent être traitées par cycle. Toutefois, précisément parce que cette optimisation réduit le temps d'exécution de base, les gains relatifs associés à la parallélisation ont tendance à diminuer pour les configurations à 4 ou 8 voies : une fraction plus importante du temps total est alors expliquée par des coûts non idéaux (attentes mémoire, cohérence et synchronisation) plutôt que par le calcul utile.

Enfin, on constate que, pour 8 threads et une matrice de taille $M = 128$, les IPC globaux obtenus ne sont que légèrement supérieurs à ceux mesurés sur les configurations de type A7. Cela s'explique par le fait que, compte tenu de la nature de l'opération (produit matriciel) et de la taille considérée, l'exécution devient rapidement *memory bound* et dépend fortement des surcoûts liés à la synchronisation et à l'exécution OpenMP. En conséquence, les avantages micro-architecturaux du A15, mis en évidence lors du premier TP, se trouvent nettement atténués dans ce contexte, ce qui rend la différence entre l'utilisation d'un A7 et d'un A15 relativement peu significative pour cette application.

7 Configuration and conclusion

7.1 Configuration CMP la plus efficace

Se propose donc la sélection d'un microprocesseur **Cortex-A7 (ARM)**, avec une configuration à **8 threads**, car cette option offre un compromis pertinent entre le nombre de threads, le *speedup* et l'IPC global observé. Ce choix est principalement justifié par le fait que, dans les conditions de l'application étudiée, l'exécution cesse rapidement d'être *compute bound* pour devenir fortement contrainte par la mémoire ; dans ce contexte, l'emploi d'un microprocesseur A15 n'apporte pas d'amélioration significative. En effet, les résultats montrent non seulement que les *speedups* obtenus avec les A15 pour une matrice de taille $M = 128$ restent proches de ceux mesurés sur A7, mais aussi que, pour le même nombre de threads (8), les valeurs d'IPC global sont très similaires à celles de la configuration proposée. Par ailleurs, la configuration à 8 threads est retenue de préférence à celle à 16 threads, car l'on observe que, pour des tailles de matrice inférieures à $M = 128$, le *speedup* tend à saturer, notamment en raison des phénomènes déjà identifiés : défauts de cache et pénalités mémoire, surcoûts OpenMP, coûts de synchronisation, ainsi que la présence de sections linéaires non parallélisables du code qui limitent le gain global.

7.2 speedup supra-linéaire

Un *speedup* supra-linéaire peut apparaître lorsque l'augmentation du nombre de threads améliore fortement l'utilisation des caches. En effet, les données peuvent être réparties entre plusieurs caches (notamment L1 et, selon l'architecture, L2), ce qui réduit la fréquence des accès à la mémoire principale et diminue la latence effective des lectures/écritures. Dans ce cas, le temps total peut baisser plus vite que la simple division du travail entre threads, car on gagne simultanément en parallélisme et en localité mémoire. Ce phénomène est généralement favorisé par des matrices de grande taille, des caches relativement petits (rendant la localité plus sensible), et un nombre de threads modéré, pour lequel l'augmentation de capacité de cache agrégée ne s'accompagne pas encore d'une contention excessive ni de surcoûts de synchronisation dominants.

7.3 Conclusion

Ce TP met en évidence l'impact du parallélisme et du type de cœur sur les performances. Les cœurs *in-order* offrent un bon compromis pour concevoir des CMP efficaces en surface et en consommation, tandis que les cœurs *out-of-order* (o3) permettent d'augmenter l'IPC grâce à une meilleure exploitation du parallélisme au niveau des instructions. Néanmoins, le *speedup* observé reste souvent limité par la hiérarchie mémoire (contention, défauts de cache, latences) et par les surcoûts de synchronisation et de coordination imposés par l'exécution parallèle, ce qui empêche d'atteindre un gain proportionnel au nombre de threads.