



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	2
2	Introduction	2
3	Analyse théorique de cohérence de cache	2
4	Paramètres de l'architecture multicœurs	3
4.1	Paramètres micro-architecturaux configurables (extrait du fichier de configuration)	3
4.2	Paramètres cache configurables (extrait du fichier de options)	4
5	Architecture multicœurs avec des processeurs superscalaires in-order (Cortex A7)	4
5.1	Chemin critique, synchronisation et cycles dominants en exécution OpenMP . . .	4
5.2	Cycles per configuration	4

1 Resume

Ce rapport analyse l'influence de la hiérarchie mémoire et des contraintes énergétiques sur les performances des processeurs. Dans un premier temps, nous évaluons l'impact des caches L1/L2 sur des calculs matriciels, en mettant en évidence le rôle clé de la localité et de l'organisation des données. Dans un second temps, nous menons une exploration de l'espace de conception d'une architecture *big.LITTLE* (Cortex-A7/A15) afin d'optimiser les configurations pour Dijkstra et Blowfish. Les performances sont mesurées avec Gem5 (IPC, taux de *miss*) et les coûts physiques des caches sont estimés avec CACTI, compilé en version 7.0 pour fiabiliser les résultats en 32 nm.

2 Introduction

Dans le contexte actuel des systèmes embarqués et du calcul haute performance, la conception de processeurs est confrontée à deux contraintes majeures : le *mur mémoire* (*Memory Wall*), qui limite les gains de performance lorsque la hiérarchie mémoire ne suit pas le rythme du calcul, et l'efficacité énergétique, devenue déterminante pour les plateformes modernes. Ce rapport de travaux pratiques s'articule autour de deux études complémentaires visant à caractériser et quantifier ces problématiques à travers des expérimentations reproductibles.

La première partie (Exercice 3) analyse l'impact de la hiérarchie mémoire (caches L1/L2) sur des algorithmes de calcul matriciel. L'objectif est de montrer que les performances ne dépendent pas uniquement de l'algorithme, mais aussi de la manière dont les données sont organisées et accédées en mémoire, notamment via la localité spatiale et temporelle.

La seconde partie (Exercice 4) porte sur une exploration de l'espace de conception (*Design Space Exploration*, DSE) appliquée à une architecture hétérogène *big.LITTLE*. Nous dimensionnons un cluster *LITTLE* (Cortex-A7) et un cluster *big* (Cortex-A15) pour des charges applicatives distinctes (Dijkstra et Blowfish), en mettant en évidence les compromis entre performance, coût en surface et consommation.

Pour mener ces analyses, nous nous appuyons sur deux outils principaux : Gem5, utilisé en mode *Syscall Emulation* pour simuler l'exécution et extraire des indicateurs tels que l'IPC et les taux de *miss*, et CACTI, utilisé pour estimer la surface et l'énergie des caches. Enfin, afin de garantir la fiabilité des résultats de l'Exercice 4, nous avons automatisé les campagnes de simulation et pris l'initiative de compiler CACTI 7.0, ce qui a permis de corriger des erreurs critiques observées avec la version standard lors des estimations en technologies fines (notamment 32 nm).

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 μ ops déjà présentes lorsque le fetch est bloqué, par exemple à cause d'un *miss* du cache d'instructions. Sa capacité est de 32 μ ops/thread. Le paramètre *DecodeWidth* correspond au nombre maximal de micro-opérations décodées par cycle, ici 8 μ ops/cycle, tandis que *IssueWidth* représente le nombre maximal de μ ops émises par cycle depuis la file de scheduling vers les unités fonctionnelles, également 8 μ ops/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) :

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

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.

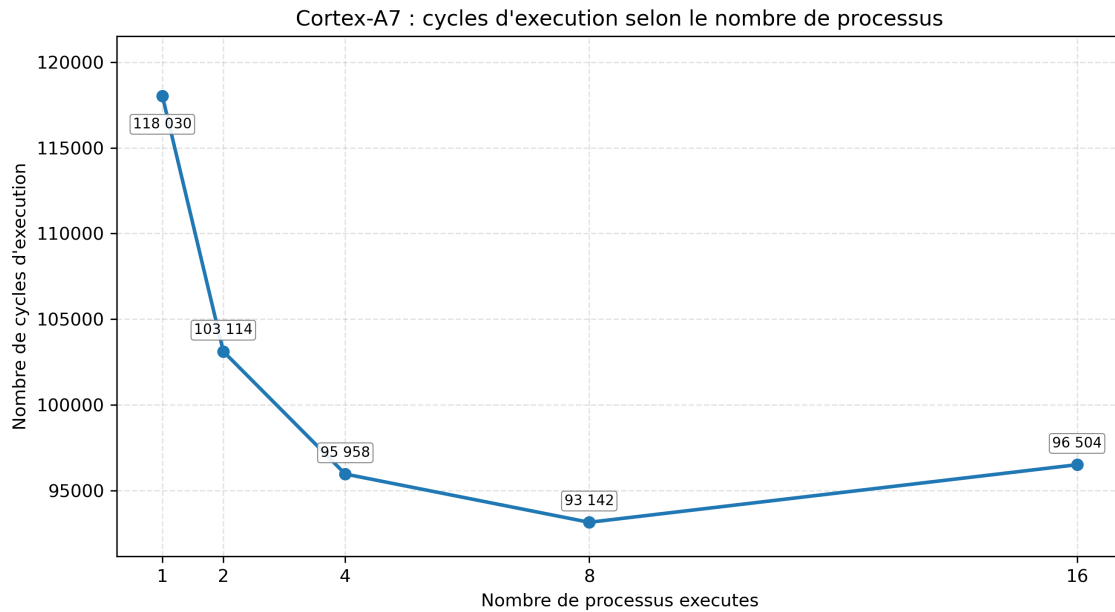


FIGURE 1 – Cortex-A7 : cycles d'exécution de l'application selon le nombre de processus.

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.

Références

- [1] M. J. P. (University of York), “Profiling,” *Lecture Notes (4th Year HPC), University of York*. [Online]. Available : https://www-users.york.ac.uk/~mijp1/teaching/4th_year_HPC/lecture_notes/Profiling.pdf. Accessed : Feb. 9, 2026.