ES201

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

Agathe BEUCHER, Magali CASAMAYOU, Cyprien GAUTHIER et Ethan ABIMELECH

Table des matières

Table des figures

1 Analyse théorique de cohérence de cache

1.1 Question 1

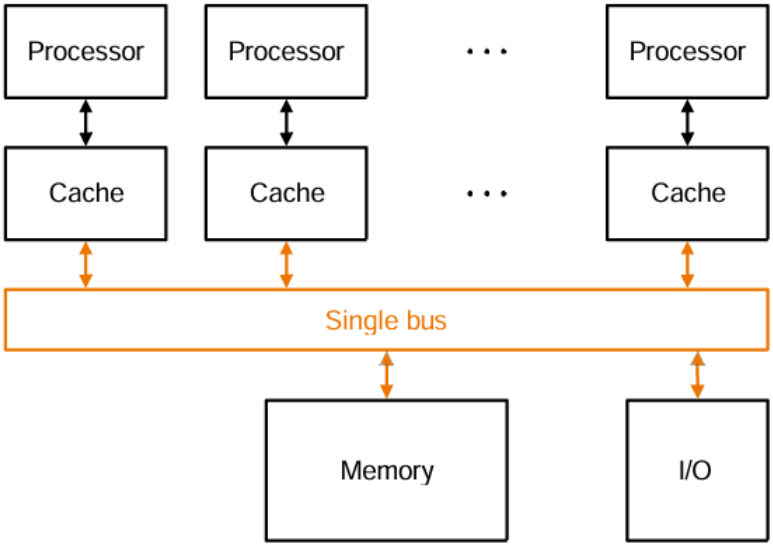

FIGURE 1 – Architecture multicoeurs à base de bus

Dans cette architecture, on considère que chaque thread s'exécute sur un processeur.

— **Comportement de la hiérarchie mémoire** : Pour chaque thread (soit chaque processeur), les lignes de la matrice A et les colonnes de la matrice B dont s'occupe le thread considéré sont chargés dans leur mémoire cache (donc modulo n le nombre de thread).

— **Cohérence des caches** : Comme les threads travaillent sur des parties différentes de la matrice, il y a peu de chances de conflits de cache directs entre eux

2 Paramètres de l'architecture multicoeurs

2.1 Question 2

On examine le fichier O3CPU.py afin de regarder les différents paramètres que l'on peut configurer dans la simulation.

Paramètre	Description	Valeur par défaut
cachePorts	Le nombre de ports sur la mémoire cache	200
fetchWidth	Nombre maximum d'instructions à récupérer à la fois	8
decodeWidth	Nombre maximum d'instructions à décoder simultanément	8
issueWidth	Nombre maximum d'instructions pouvant être émises à chaque cycle	8
wbwidth	largeur du bus utilisé pour écrire les instructions en mémoire	8

2.2 Question 3

On examine le fichier Options.py pour déterminer les différents paramètres demandés

Associativité	Taille du cache	Taille de la ligne
2-way	64 Ko	64 octets

TABLE 1 – **Valeurs des paramètres pour le Cache de données de niveau 1**

Associativité	Taille du cache	Taille de la ligne
2-way	32 Ko	64 octets

TABLE 2 – **Valeurs des paramètres pour Cache d'instructions de niveau 1**

Associativité	Taille du cache	Taille de la ligne
8-way	2 Mo	64 octets

TABLE 3 – **Valeurs des paramètres pour le Cache unifié de niveau 2**

3 Architecture multi coeurs avec des processeurs superscalaires in-order (Cortex A7)

Dans cette partie nous étudierons une architecture multiprocesseur de type CMP à base de cœurs équivalents au Cortex A7. Pour cela nous utiliserons un type de processeur "arm-detailed" en exécutant les programmes de la manière suivante [Ici la taille de la matrice est de 1 et le nombre de coeurs et de threads de 2]

```
$GEM5/configs/example/se.py --cpu-type=arm_detailed --caches -n 2 -c test_omp -o "2 1"
```

FIGURE 2 – Commande à exécuter pour utiliser le processeur en arm-detailled

On fixera la taille de la matrice m et on étudiera ici le comportement visibles en augmentant le nombre de threads en cours d'exécution de 1 à m. On propose de fixer m = 500 dans un premier temps et de regarder les résultats pour diverses valeurs de nombre de threads.

3.1 Question 4

Étudions les résultats obtenus par le simulateur, disponibles dans stats.txt. On remarque en modifiant le nombre de coeur/thread, c'est toujours le même CPU qui exécute le plus grand nombre de cycle : par exemple, en utilisant 3 coeurs et 3 threads :

```
system.cpu0.numCycles                          548518
```

FIGURE 3 – Nombre de cycle exécuté par le CPU0

```
system.cpu1.numCycles                                335020
```

FIGURE 4 – Nombre de cycle éxécuté par le CPU1

```
system.cpu2.numCycles                                334676
```

FIGURE 5 – Nombre de cycle exécuté par le CPU2

En étudiant ces mêmes résultats pour différents nombre de threads, on observe que le CPU0 a toujours le plus grand nombre de cycles exécutés. Les autres processeurs ont toujours un nombre de cycles exécutés inférieurs et de même valeur. Par conséquent, analyser le nombre de cycles exécutés sur le CPU0 renseignera sur la durée totale d'exécution du programme puisqu'il impose sa durée de calcul sur la durée totale (Les CPUs travaillant en parallèle) et donc sur le nombre total de cycles d'exécution de l'application.

3.2 Question 5

Pour les multiples exécutions du programme effectuées, nous garderons m = 50 (taille de la matrice). Après avoir exécuté le programme avec 1,2,4,8,10 et 16 threads (pour nb_thread>16, on a le message d'erreur *'memory out of bound'*), on obtient les résultats suivants :

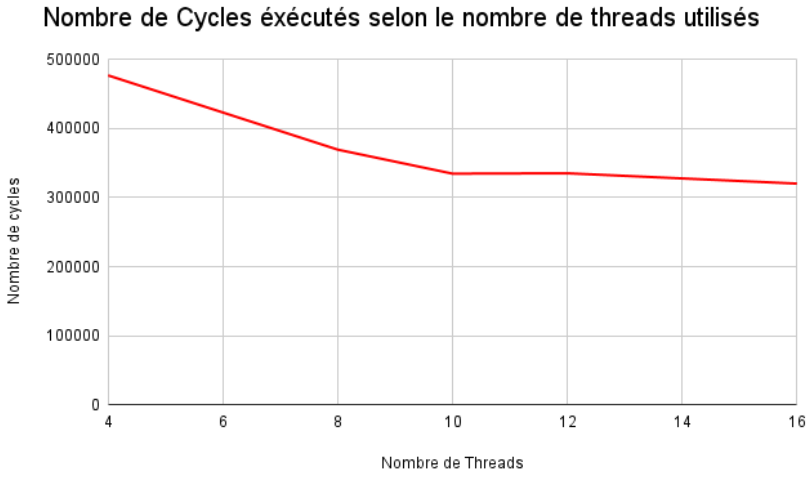

FIGURE 6

3.3 Question 6

En utilisant les résultats précédents, on peut calculer le speed-up pour chaque configuration en faisant le rapport entre le nombre de cycles exécutés dans la configuration à 1 thread et le nombre de cycle dans cette configuration. Ce qui donne :

Nombre de threads	Speedup
1	1
2	2.09
4	2.40
8	3.10
10	3.42
12	3.42
16	3.58

TABLE 4 – Speedup selon le nombre de threads

3.4 Question 7

Pour calculer la valeur de l'IPC pour chaque configuration, il faut effectuer le rapport entre le nombre de cycle exécuté par le CPU et le nombre total d'instructions simulées que l'on peut trouver en s'intéressant à ***system.cpu0.fetch.Insts***. D'où pour chaque configuration :

Nombre de Threads	Nombre de cycles	Nombre d'instructions simulées	IPC
1	2057720	1145446	1.797
2	781425	548518	1.423
4	628706	476904	1.318
8	394002	369450	1.067
10	316408	334680	0.945
12	294905	280313	0.910
16	320320	276571	0.863

TABLE 5 – Nombre de cycles, d'instructions simulées et valeur de l'IPC pour chaque configuration

On peut proposer un graphique résumant l'évolution de l'IPC selon le nombre de threads

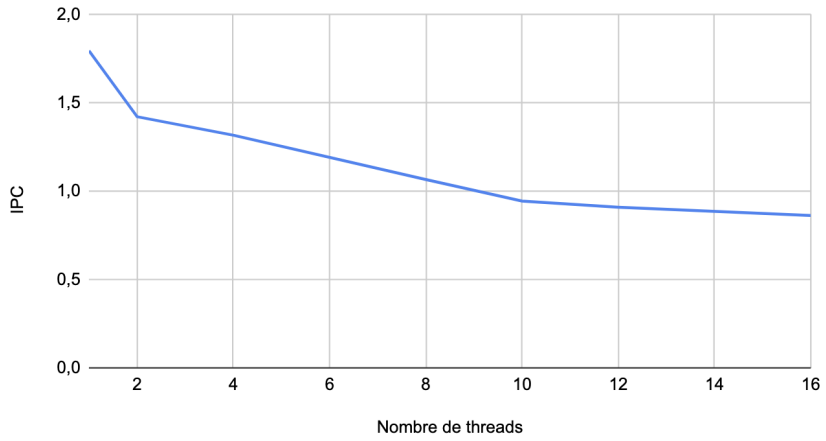

FIGURE 7 – Valeur de l'IPC selon le nombre de threads utilises

3.5 Question 8

Les questions précédentes nous ont permis d'offrir plusieurs outils d'analyse à notre étude, en particulier l'évolution de l'IPC (Instructions Par Cycle) et celle-du speed-up selon le nombre de threads/cœurs alloués à l'exécution du programme. On note qu'à partir de 10 coeurs, le speed-up et l'IPC stagnent presque complètement alors que la décroissance était linéaire.
On peut expliquer ce phénomène en regardant le nombre de cœurs physiques disponibles sur la machine. En effet, après vérification, et sans surprise, la machine ne comporte que 10 cœurs physiques, ce qui explique cette baisse notable de performance. En augmentant le nombre de thread mais non le nombre de coeur (plafonné à 10), on ne garde plus un seul thread par cœur comme on le devrait pour garder de bonnes performances!

4 Architecture multicoeurs avec des processeurs superscalaires out-of-order (Cortex A15)

Dans cette partie, il s'agit maintenant d'étudier une architecture multiprocesseur de type CMP à base de cœurs équivalents au Cortex A15. Pour cela, on fixera la taille de la matrice m = 50, comme dans la partie précédente et on fera varier le nombre de threads et la largeur du processeur superscalaire à l'aide de l'option -w de GEM5 de la manière suivante :

```
$GEM5/build/ARM/gem5.fast $GEM5/configs/example/se.py --cpu-type=detailed --caches -n 10 -w 1 -c test_omp -o '10 50'
```

FIGURE 8 – Commande à exécuter pour la suite de la partie [Ici la largeur superscalaire =1, nombre de threads = 10]

4.1 Question 9

De la même manière que pour la partie précédente on peut identifier le nombre de cycles exécutés par la simulation à l'aide de *system.cpu0.num.Cycles* dans stats.txt. En affichant un graphique selon 3 axes, on évalue le nombre de cycles selon le nombre de threads et la largeur superscalaire :

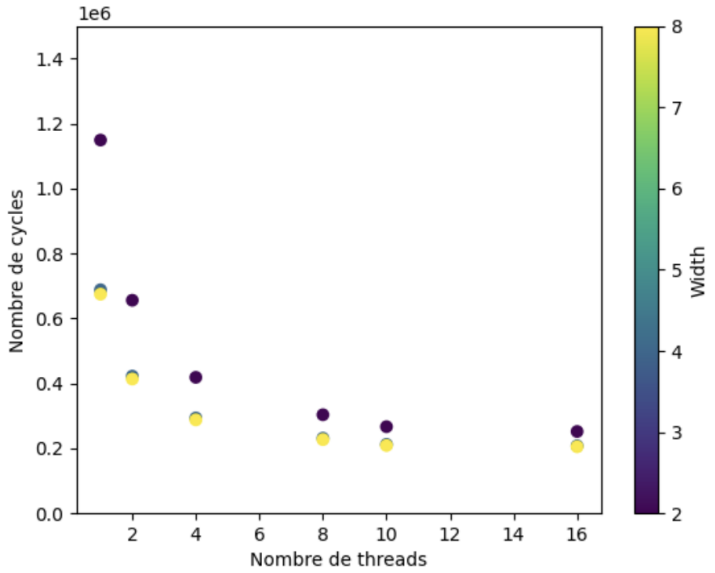

FIGURE 9 – Nombre de cycles selon le nombre de threads et selon la largeur superscalaire

4.2 Question 10

On peut déduire de la question précédente la valeur du speed-up dans chaque configuration :

Nombre de threads	Nombre de cycles	Speedup
1	1148813	1
2	655841	1,751663894
4	418699	2,743768196
8	303503	3,7851784
10	266783	4,306170183
16	251999	4,558799837

TABLE 6 – Speed-up pour chaque configuration avec width (Largeur superscalaire) = 2

Nombre de threads	Nombre de cycles	Speedup
1	688100	1
2	422400	1,629024621
4	293790	2,342149154
8	231402	2,97361302
10	212352	3,240374473
16	208030	3,307696005

TABLE 7 – Speed-up pour chaque configuration avec width (Largeur superscalaire) = 4

Nombre de threads	Nombre de cycles	Speedup
1	674734	1
2	413566	1,63150259
4	287812	2,344356733
8	227188	2,969936792
10	208972	3,228824914
16	205272	3,287024046

TABLE 8 – Speed-up pour chaque configuration avec width (Largeur superscalaire) = 8

Pour offrir une meilleure lisibilité, nous pouvons proposer d'inclure une courbe d'évolution du speed-up selon le nombre de threads pour chaque largeur superscalaire :

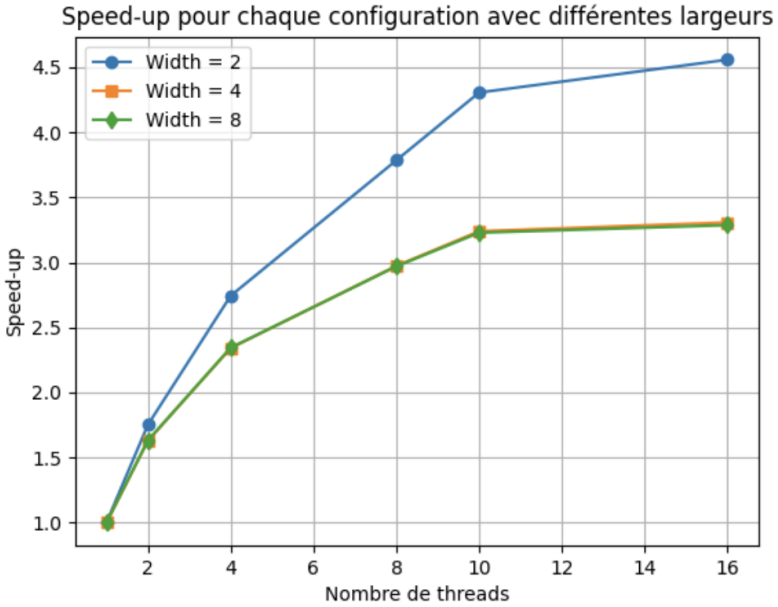

FIGURE 10 – Nombre de Cycles selon le nombre de threads pour chaque largeur superscalaire

On note qu'il y'a une différence notable entre les configuration où width = 2 et les 2 autres, qui sont presque confondus.

4.3 Question 11

De la même manière que la question 7, on peut calculer la valeur de l'IPC maximale dans chaque configuration en faisant le rapport entre le nombre d'insructions simulées et le nombre de cycles :

Nombre de Threads	Nombre de cycles	Nombre d'instructions simulées	IPC
1	1148813	2270832	1,976676796
2	655841	1198030	1,826707998
4	418699	681215	1,626980241
8	303503	422803	1,393076839
10	266783	337258	1,264166008
16	320320	276571	1,17549276

TABLE 9 – Nombre de cycles, d'instructions simulées et valeur de l'IPC pour chaque configuration avec width = 2

Nombre de Threads	Nombre de cycles	Nombre d'instructions simulées	IPC
1	688100	2101178	3,053593954
2	422400	1108656	2,624659091
4	293790	636552	2,166690493
8	231402	395201	1,707854729
10	212352	320999	1,51163634
16	208030	282436	1,357669567

TABLE 10 – Nombre de cycles, d'instructions simulées et valeur de l'IPC pour chaque configuration avec width = 4

Nombre de Threads	Nombre de cycles	Nombre d'instructions simulées	IPC
1	674734	2069566	3,06723242
2	413566	1092284	2,641135877
4	287812	627102	2,178859811
8	227188	389568	1,714738454
10	208972	317388	1,518806347
16	205272	279161	1,359956545

TABLE 11 – Nombre de cycles, d'instructions simulées et valeur de l'IPC pour chaque configuration avec width = 8

4.4 Question 12

En étudiant les différents résultats obtenus aux questions précédentes, on observe une différence nette de performance pour une largeur superscalaire de 2, par rapport à 4 et 8. Le speed-up est plus élevé, et en particulier, le seuil maximal de speed-up est plus important. Par ailleurs on note que la baisse d'IPC selon le nombre threads est plus importantes pour des largeurs superscalaires de 4 et 8.
Ces différences indiquent une limitation de performance pour des largeurs superscalaires plus élevées, probablement liées à l'ordinateur (et/ou le processeur) utilisé, ce qui explique pourquoi les résultats obtenus sont proches pour 4 et 8. On ne profite donc pas d'une amélioration de performance significative en fixant une largeur superscalaire à 8 ou plus dû à ces limitations.

5 Configuration CMP la plus efficace

5.1 Question 13

Lors du TP4, nous avons étudié en détail l'efficacité surfacique pour les processeurs A7 et A15 et avons conclu que le processeur A15 avait de meilleures performances en terme d'efficacité surfacique.
Nous proposons donc d'adopter une architecture CMP type Cortex A15. Utilisons maintenant les résultats de ce TP pour conclure quant aux options de configuration à utiliser. Nous avons constaté que lors de la compilation de *testomp* avec GEM5, on obtenait de meilleures performances pour une largeur superscalaire de 2 (meilleur speed-up notamment). Par ailleurs, on a vu qu'en terme d'efficacité,il était préférable d'exécuter le programme avec un nombre de threads inférieurs ou égal au nombre de coeurs physiques de la machine, soit ici 10.
Pour résumer, nous proposons d'utiliser une architecture CMP type A15 avec 10 threads et 10 coeurs, en utilisant une largeur super scalaire de 2.

5.2 Question 14

La taille du cache de niveau 1 pour les données est de 64ko. Les entiers étant généralement codé sur 4 octets, pour que la taille d'une ligne ou d'une colonne soit plus grande que la taille du cache de données, il faut un nombre de lignes ou de colonnes supérieur à 16k.