M2 E3A - SETI

TP T1 - Compte-rendus

Saša Radosavljevic Quentin Rousset
Etienne Stransky

20 décembre 2022

Table des matières

1 TP1 - Modélisation et évaluation de mémoires caches

Ce TP va nous permettre d'étudier l'impact des paramètres de configuration d'une mémoire cache sur ses caractéristiques techniques. En effet, le cours nous a montré que la taille du cache, son niveau d'associativité ou la technologie utilisée ont une influence sur le temps d'accès à la mémoire ou encore l'énergie utilisée par celle-ci. Une fois l'étude de ces paramètres réalisée, nous aborderons l'analyse au niveau des ports entrée-sortie et enfin des modes d'accès du cache.

Pour cela, on utilisera CACTI 6.5, un logiciel développé par HP permettant l'étude de l'impact de l'architecture d'un cache sur ses caractéristiques. En optimisant le travail avec deux scripts de génération des résultats et de parsing des résultats, on est capables de tracer les multiples figures disponibles dans les sections suivantes.

Durant le TP, on développe un script Python permettant de configurer la simulation, de la lancer, puis de récupérer les résultats sous forme de graphiques. Ainsi, la suite du présent compte-rendu présentera les résultats obtenus grâce à ce script.

1.1 - Temps d'accès, de la surface et de l'énergie

Dans un premier temps, nous cherchons à observer l'impact qu'ont la taille du cache ainsi que son associativité sur ses performances. Les critères retenus ici sont le temps d'accès, la consommation électrique (considérée ici en nJ pour faire un accès en lecture) ainsi que l'aire du cache. On lance donc plusieurs simulations et nous obtenons les courbes en figure 1.1.

Nous observons sur cette figure que l'associativité a un impact linéaire sur les paramètres du cache, ce qui peut s'expliquer par le besoin de composants supplémentaires en fonction du degré d'associativité. En revanche, l'augmentation du temps d'accès, de la consommation et de la taille se fait moins rapidement qu'une évolution linéaire et semble se rapprocher d'une évolution logarithmique. Cela peut s'expliquer par le fait que, plus la taille augmente, moins le besoin de composants de contrôle par octet supplémentaire est important, et cela se répercute donc sur les paramètres. Nous constatons donc qu'augmenter ces paramètres tend à dégrader les performances du cache.

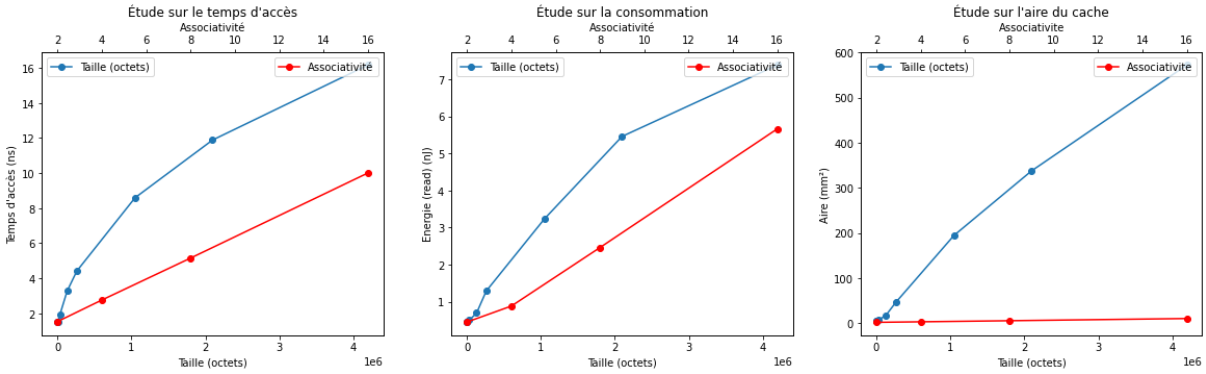

FIGURE 1.1 – Résultats de l'étude de l'impact de l'associativité et de la taille du cache sur les performances de ce dernier

1.2 - Impact du nombre de ports lecture/écriture

Nous étudions dans un second temps l'impact qu'a le nombre de ports d'entrée et de sorties du cache sur ses performances. Nous utilisons pour cela le même script, mais en modifiant d'autres paramètres. Nous faisons donc de nouveau tourner la simulation, et nous obtenons les résultats en figure 1.2.

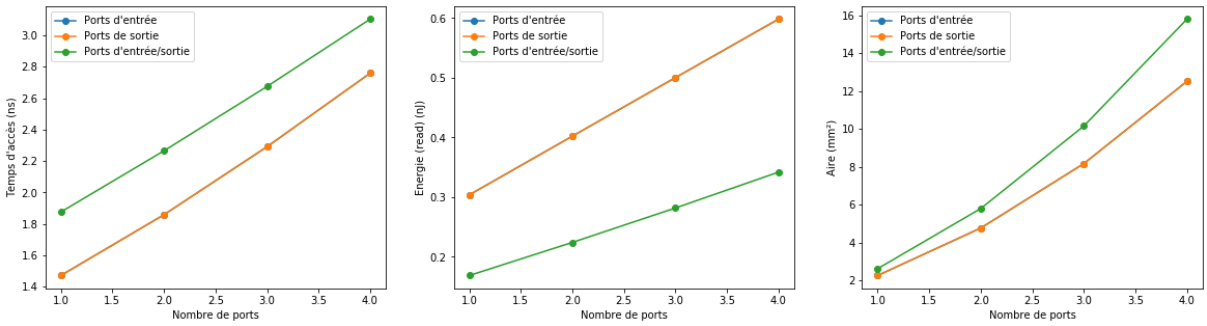

FIGURE 1.2 – Étude de l'impact des ports d'entrée et de sortie du cache sur ses performances

Les courbes donnant l'évolution en fonction des ports d'entrée ont été entièrement recouvertes par celles donnant l'évolution en fonction du nombre de ports de sorties : nous en déduisons que ces ports ont le même impact. Pour les temps d'accès et la consommation, l'évolution se fait de façon linéaire. Concernant l'impact de ces ports sur l'aire du cache, celle-si se fait de façon plus rapide. Cela semble cohérent, étant donné qu'une augmentation du nombre de ports résulte nécessairement en un besoin très accru de différents composants avec les chemins d'accès associés, ce qui se répercute grandement sur l'aire du cache.

1.3 - Analyse de l'impact du mode d'accès du cache

On peut également étudier l'impact qu'a le mode d'accès au cache sur ses performances. En prenant un cache de taille 8Mo avec une associativité de 2, on arrive aux résultats en

figure 1.3.

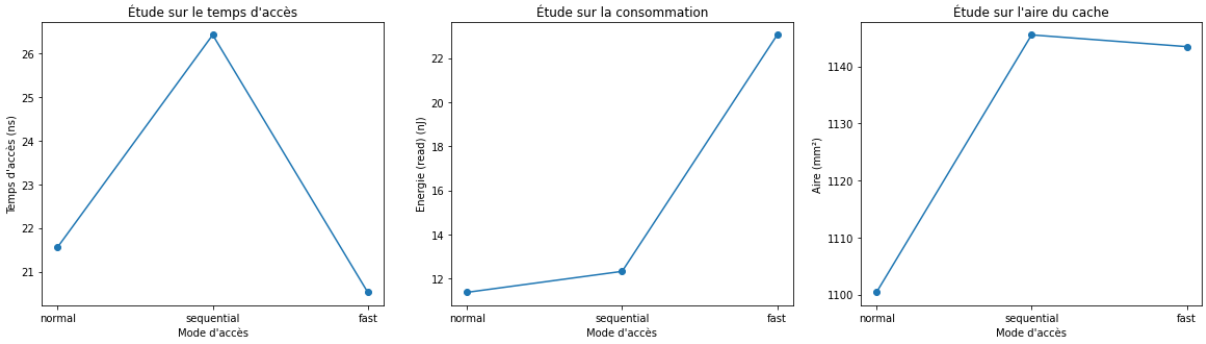

FIGURE 1.3 – Étude de l'impact du mode d'accès au cache sur ses performances

Nous constatons sur ces courbes que, pour cette configuration de la mémoire, le temps d'accès est le plus élevé avec un mode séquentiel, mais le plus faible avec le mode rapide. Au contraire, la consommation du mode séquentiel est réduite et très élevée sur le mode rapide. Les résultats de ces évaluations évoluent également selon les configurations du cache, notamment sa taille mémoire, ce qui fait qu'il est complexe de généraliser les résultats obtenus ici.

1.4 - Temps d'accès moyen TAMM

On cherche dans cette partie à évaluer les temps d'accès moyens du processeur aux données. On utilise pour cela une table de référence d'accès mémoire d'un processeur Intel, en figure 1.4.

Characteristic	L1	L2	L3
Size	32 KB I/32 KB D	256 KB	2 MB per core
Associativity	4-way I/8-way D	8-way	16-way
Access latency	4 cycles, pipelined	10 cycles	35 cycles
Replacement scheme	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with an ordered selection algorihtm

FIGURE 1.4 – Évaluation des caractéristiques des cahces du processeur i7 d'Intel
Source : https://icl.utk.edu/~luszczek/teaching/courses/fall2013/cosc530/Cosc530Report_ARM_Cortex-A.pdf

On utilise alors la formule suivante :

$$TAMM = t_{aL1} + \tau_{eL1}(t_{aL2} + \tau_{eL2} * t_M) \tag{1.1}$$

En considérant un temps d'accès mémoire de 100 cycles, avec un taux d'échec de 0,1 pour chacun des caches, on arrive à un temps d'accès moyen à la mémoire :

$$TAMM = 6 \text{ cycles}$$

1.5 - Conclusion

Grâce au simulateur Cacti, nous avons pu évaluer l'impact qu'ont différents paramètres d'un cache sur ses caractéristiques. Nous avons notamment pu évaluer l'impact qu'ont la taille du cache, le nombre de ports d'entrée/sortie du système ainsi que les modes d'accès au cache. Cela nous a permis de déterminer des paramètres de sortie tels que sa surface, sa consommation ou son temps d'accès qui sont impactés de façon positive ou négative selon les paramètres. Si une chose était à retenir de ce TP, c'est qu'il n'existe pas de solution parfaite pour la conception du cache. En effet, lorsqu'un paramètre améliore une caractéristique, une autre est généralement dégradée. Il y a donc un compromis à trouver selon l'application considérée, et c'est ce qu'aident à faire les outils de simulation tels que Cacti : en permettant d'avoir une idée de l'évolution des caractéristiques du système sans avoir à le concevoir entièrement, ils permettent de trouver les configurations adéquates et ainsi de connaître les objectifs à se fixer avant de réaliser une conception matérielle.

2 | TP2 - Analyse de performances

2.1 - Préparation du TP

Ce TP a pour objectifs de permettre d'analyser l'importance des options de compilation et d'optimisation sur un programme, ainsi que le lien avec l'influence du cache sur les temps d'exécution. Dans un permier temps, nous allons rechercher les caractéristiques des composants des PC que nous allons utiliser avant de faire une étude de l'optimisation. Ensuite, on travaillera sur l'influence du cache en fonction des implémentations d'algorithme et leur manière d'accéder à la mémoire.

2.1.1 - Caractéristiques

Premier ordinateur : MSI Prestige 15 A10SC Intel Core i7-10710U (Hexa-Core 1.1 GHz / 4.7 GHz Turbo - 12 Threads - Cache 12 Mo)

En calcul, la fréquence du CPU est en moyenne à 4,2 GHz. Le problème étant que lors d'un calcul intensif, le Boost est activé et augmente la fréquence jusqu'à 4.7 GHz. Pour cela, on désactive la fonction Boost avec la commande suivante :

```
echo "1" | sudo tee /sys/devices/system/cpu/intel_pstate/no_turbo
```

- **Fréquence** : 1.1 GHz
- **RAM** : 16 Go 2400MT/s
- **bogoMIPS** : 3200
- **Caches** :
  - L1d : 192 KB
  - L1i : 192 KB
  - L2 : 1,5 MB
  - L3 : 12 MB

Deuxième ordinateur : MacBook Pro Intel Core i5 Dual-Core, fréquence de 2GHz.

Les informations sur le processeur s'obtiennent sur MacOs avec la commande `sysctl -a`, que l'on peut formater à l'aide de `grep`.

- **RAM** : 8Go

— **Caches** :
  — L1d : 32 KB
  — L1i : 32 KB
  — L2 : 262 MB
  — L3 : 4 MB

2.1.2 - Mesures

On dispose de plusieurs fonctions de test dans le fichier tp2.c. On dispose également de la fonction print_res() pour normaliser les mesures à laquelle on peut rajouter le calcul de la moyenne et un critère pour enlever les mesures aberrantes qui peuvent fausser la moyenne.

```
void print_res(char *funcname, double nb_iter){
    double normalize = 1.0/nb_iter;
    double moy = 0;
    int correct = 0;
    double min = resultats[0]*normalize;

    // calcul du minimum pour enlever les mesures aberrantes
    for(int i=1; i<M; i++){
        if(resultats[i]*normalize < min)
            min = resultats[i]*normalize;
    }

    printf("%s\t", funcname);
    for(int i=0; i<M; i++){
        printf("%g%s",resultats[i]*normalize, (i==M-1 ? "\n" : "\t"));
        if(resultats[i]*normalize < min + 1){
            moy += resultats[i]*normalize;
            correct++;
        }
    }
    moy /= correct;
    printf("Moyenne %f \n\r",moy);
}
```

2.2 - Optimisation du compilateur

Nous utiliserons le compilateur gcc. Il comprend de nombreuses options de compilation, notamment pour l'optimisation. Nous pouvons comparer les résultats pour différentes options de compilation, ici sur un exemple de programme de multiplication de matrices, en figure 2.1. Ces résultats ont été obtenus sur le MacBook.

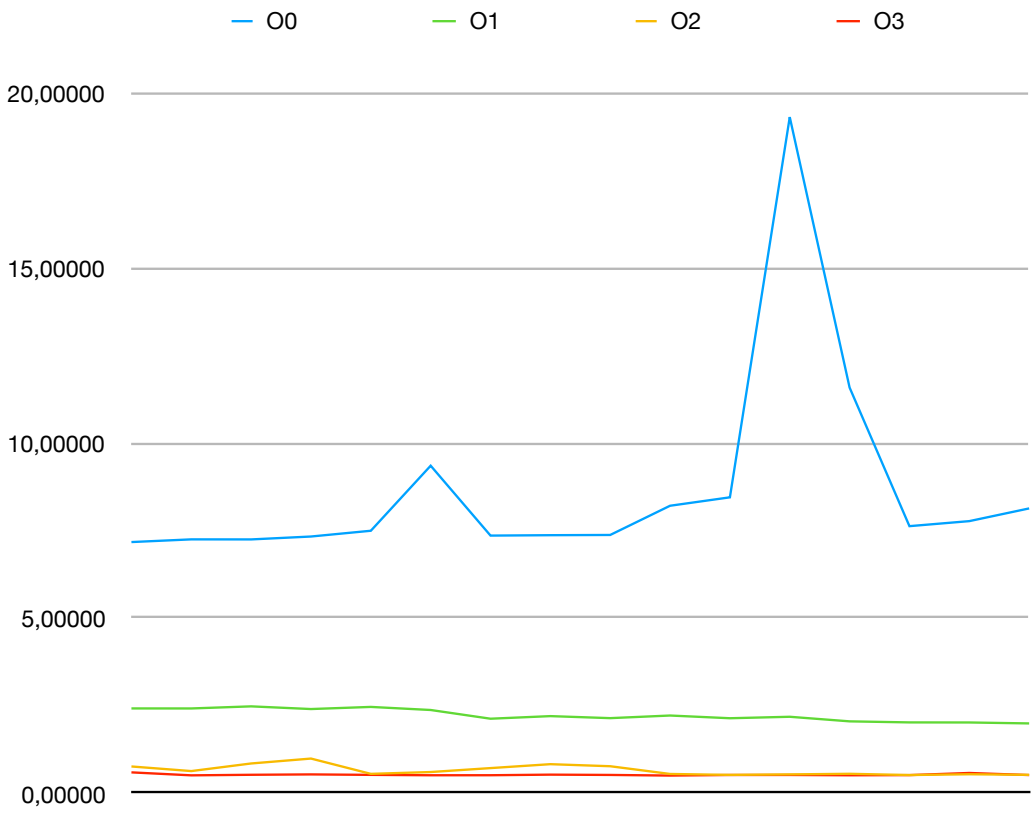

FIGURE 2.1 – Comparaison du nombre de cycles par itération de la fonction mm_ikj()

On remarque la différence notable entre `O0` et les autres niveaux d'optimisation. Cependant, il est dur ici de différencier `O2` et `O3`. En effet, le programme utilisé ne présente pas plus d'opportunités d'amélioration. C'est pourquoi nous allons étudier d'autres fonctions et comparer plus précisément ces options.

2.3 - Étude de diverses fonctions

Pour l'étude des fonctions, l'optimisation utilisée pour le compilateur sera -O2 par défaut, nous pourrons changer ensuite ce niveau pour effectuer quelques remarques.

On pourra également modifier les paramètres des fonctions à la compilation avec : `gcc -O2 -DN=500 -DTYPE=int tp2.c -o tp2` où `DN` définit la taille $N \times N$ des matrices et `DTYPE` le type des variables pour l'estimation du débit d'écriture en mémoire.

2.3.1 - Mise à zéro

On mesure la moyenne des cycles par itérations pour le i7-10710U pour l'évolution de temps d'exécution pour différents N, pour différents types de variables et pour différentes optimisations.

N	O	char	short	int	float	double
100	O1	2.93	2.93	2.93	2.93	2.93
	O2	1.47	1.49	1.60	1.57	1.65
	O3	0.10	0.20	0.46	0.47	0.94
500	O1	2.93	2.95	2.93	2.94	2.93
	O2	1.76	1.80	1.76	1.79	1.84
	O3	0.14	0.29	0.59	0.60	1.38
1000	O1	2.94	3.42	3.94	2.95	2.9
	O2	1.80	1.76	1.79	1.79	1.89
	O3	0.15	0.30	0.60	0.62	1.47

On constate une évolution quasi linéaire entre les cycles par itérations en fonction de la taille des données pour une optimisation O3. En prenant 1.47 cycles / 8 octets dans le cas d'un double avec $N = 1000$, on obtient 5.44 octets/cycle. Soit, pour un Intel i7-10710 @ 1.10 GHz, un débit de **5984 Mo/s**.

2.3.2 - Copie de matrices

On étudie maintenant les fonctions de copie d'une matrice $[N][N]$ vers une autre. Il y a deux manières de faire pour coder une telle fonction : ligne-colonne ou colonne-ligne. On a mesuré la performance des deux méthodes sur la figure 2.2.

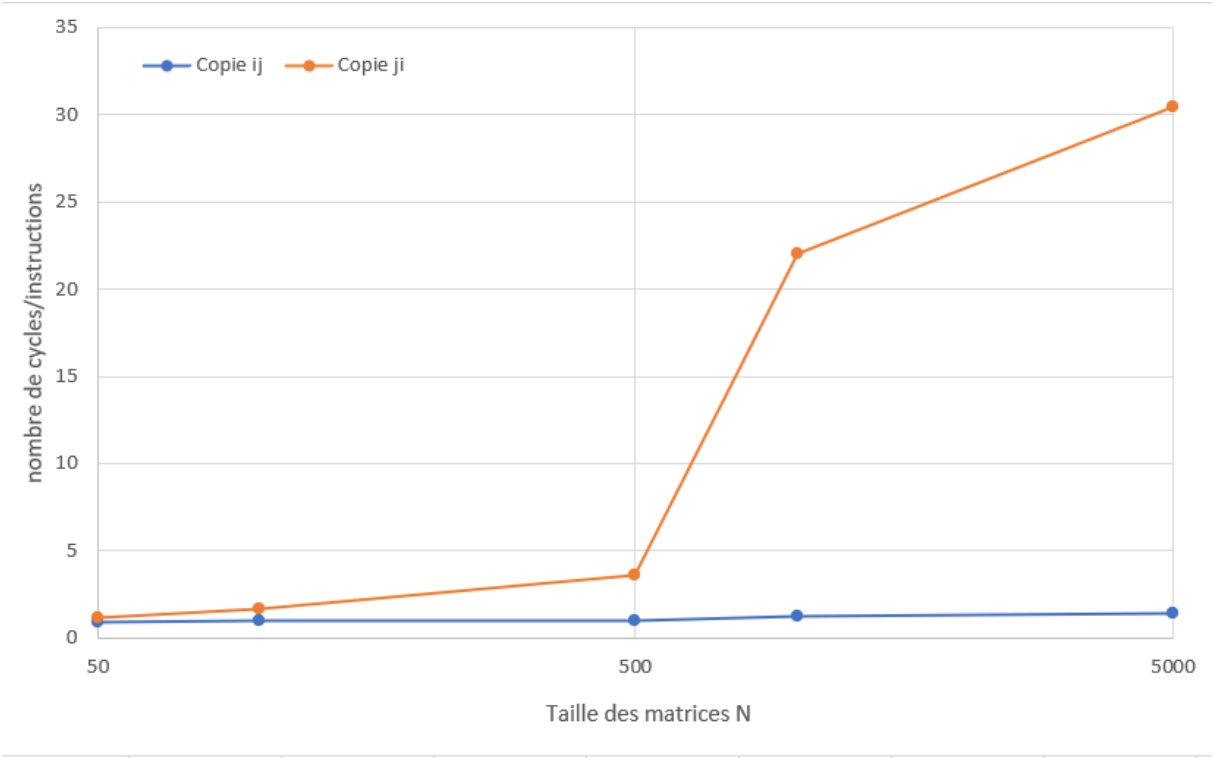

FIGURE 2.2 – Performances des deux fonctions copie en -02

On remarque notamment une nette différence pour une taille de matrice élevée sur la vitesse d'exécution. En effet, les matrices étant stockées comme des tableaux statiques

à deux dimensions, les données rangées en mémoire se suivent ligne par ligne. Le cache charge donc ces lignes à la suite, ce qui n'est pas le cas lorsque l'on copie colonne par colonne puisqu'il est obligé de faire des sauts mémoires.

Cependant, ce résultat est nettement amélioré lorsqu'on passe à un niveau d'optimisation O3. Le facteur pour $N = 1000$ passe de 10 à 2 sur la vitesse. Des hypothèses sur les améliorations que le compilateur fournit seraient qu'il parallélise pour une utilisation SIMD ou alors qu'il permute les deux boucles et donc rapproche les deux fonctions.

2.3.3 - Addition de matrices

Lorsqu'on teste les fonctions d'addition de matrices de tailles N*N, on tombe sur des résultats similaires à ceux obtenus avec la copie d'une matrice, à savoir que l'exécution est bien plus rapide avec l'addition ij plutôt que l'addition ji. Les résultats sont visibles en figure 2.3.

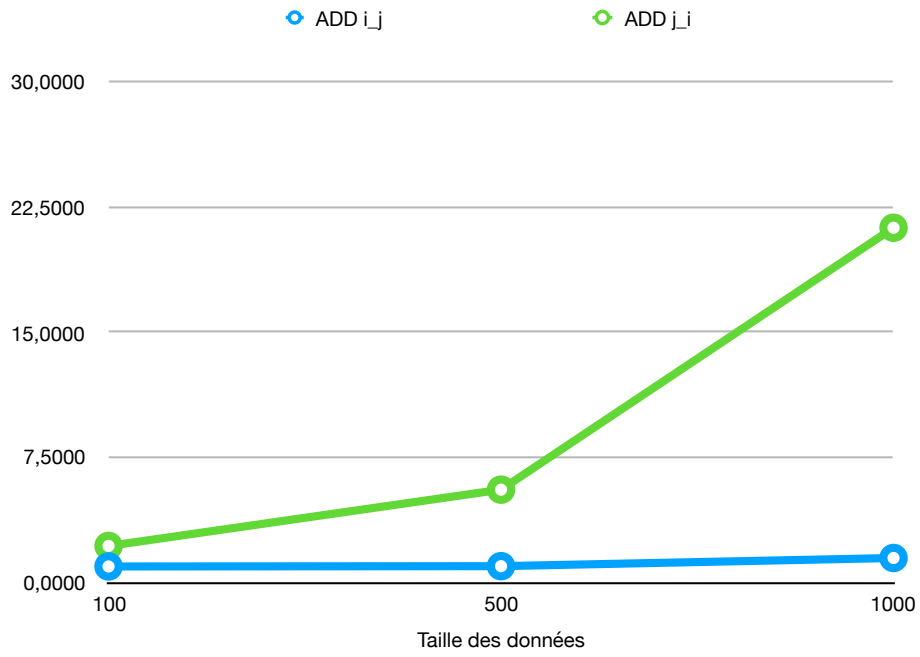

FIGURE 2.3 – Comparaison des performances de l'addition avec les fonctions add_ij et add_ji

Cette disparité s'explique par l'organisation des données dans la mémoire, puisque les éléments situés dans une même ligne à des colonnes adjacentes se trouvent à des emplacements adjacents dans la mémoire. Ainsi, le cache peut charger facilement ces données, ce qui accélère le processus. Dans le cas contraire, on observe des ruptures du cache, ce qui ralentit l'exécution du programme. C'est d'autant plus cohérent avec la copie de matrices qu'une copie peut s'assimiler à une addition avec un élément nul.

2.3.4 - Produit scalaire de deux vecteurs

Pour le produit scalaire, on effectue des mesures pour un ensemble de N501050010005000. On se retrouve avec une vitesse d'exécution moyenne de 1.6 cycles/itération en O1 et 1.1 cycles/itération en O3 pour $N = 1000$. La vitesse varie de la même manière que pour les autres méthodes en fonction de N mais on remarque cependant que le niveau d'optimisation ne permet pas vraiment d'améliorer le programme.

2.3.5 - Produit de matrices

Produit de matrices ijk de flottants 32 bits

On s'intéresse au produit matriciel. On teste le programme pour plusieurs valeurs de N pour comparer les performances. On relève également le nombre de cycles par itération à chaque étape du calcul. On obtient alors l'évolution en figure 2.4.

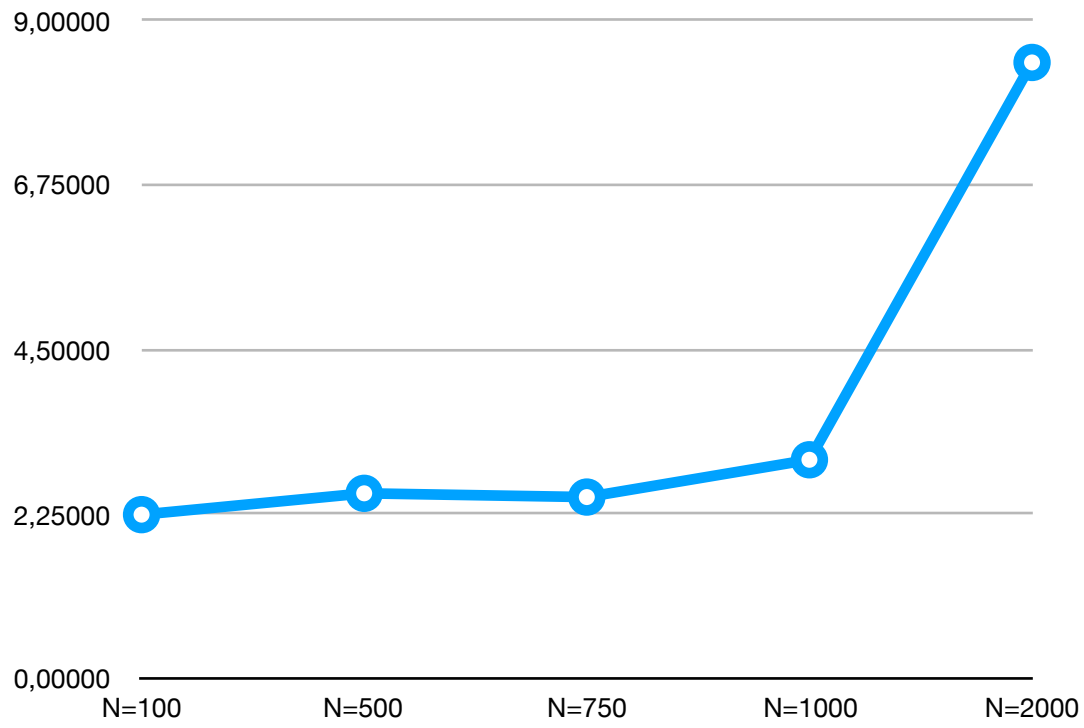

FIGURE 2.4 – Comparaison du nombre de cycles par itération de la multiplication matricielle en fonction de N

Nous constatons un saut important aux alentours de la taille de matrice N=1000. Ce saut correspond au moment où les matrices ne peuvent plus être stockées dans les caches, mais ne peuvent être présentes que dans la mémoire. Les accès doivent donc se faire dans la mémoire centrale, ce qui ralentit énormément les performances. Pour des matrices de

flottants de taille 1000*1000, la mémoire occupée a une taille de 4Mo, sachant que ce calcul nécessite trois matrices (les deux entrées et le résultat), ce qui occupe au total 12Mo. On en déduit que le cache a approximativement cette taille.

Produit de matrices ikj

N	char	short	int	float	double
100	3.37	3.24	3.29	3.35	3.33
500	3.05	3.09	3.06	3.03	3.11
1000	3.01	3.00	3.00	3.01	3.02

Lorsque l'on effectue la multiplication de matrices dans un autre ordre qui favorise plutôt un accès mémoire colonne par colonne puis ligne par ligne, on se retrouve avec une nette amélioration de la performance du programme puisque le cache est mieux utilisé. Sur certains ordis après nos tests, on peut voir une amélioration allant d'un facteur 2 en O2 à 10 en O3.

Produit de matrices ijk par blocs

En effectuant la multiplication par petits blocs en O2, on se rend compte que des tailles de blocs de 16 permettent une exécution plus rapide. On peut supposer que la taille des lignes dans le cache L1d s'approche de $16 \times 16 = 256$ octets, ce qui permet un accès rapide aux blocs étudiés sur le moment.

Produit de matrices avec transposition

Une autre manière de faire est de transposer l'une des matrices avant une multiplication avec une implémentation ijk. Cela permet d'accèder aux deux matrices en même temps colonne par colonne. Les résultats ne sont pas aussi efficaces qu'une ikj mais permettent tout de même d'améliorer le temps d'exécution comme on le voit dans le tableau ci-dessous :

ijk	ikj	ijk_b	ijk_t
2.50	1.73	2.61	2.33

Le code utilisé est proprosé ci-dessous.

```
1  for (i = 0; i < N; i++)
2      for (j = 0; j < N; j++)
3          XTF[i][j] = XF[j][i];
4
5
```

```
6  for (i = 0; i < N; i++)
7  {
8      for (j = 0; j < N; j++)
9      {
10         SF = ZERO;
11         for (k = 0; k < N; k++)
12             SF += AF[i][k] * XTF[j][k];
13         YF[i][j] = SF;
14     }
15 }
```

2.4 - Questions additionnelles

2.4.1 - Multiplication de matrices en entier et flottant

En effectuant une multiplication de matrice avec des variables de même taille de représentation, on se rend compte qu'il y a une différence entre entiers et flottants. Cela est sûrement lié à la difficulté pour un processeur d'effectuer une multiplication sur des flottants, qui est une opération plus coûteuse généralement en cycles qu'avec des entiers.

int	float
2.08	3.01

2.4.2 - Produit scalaire en entier et en flottant

On voit le même problème dans le cas d'un produit scalaire. On remarque un facteur 2 entre les temps d'exécution. De la même manière ici, les pipelines utilisé pour les calculs en flottant sont moins rapides que leur équivalent entier.

2.5 - Conclusion

Ce TP nous a permis de comprendre l'intérêt des options d'optimisation à la compilation d'un code, ainsi que l'influence de notre manière de coder vis-à-vis des accès mémoires et du cache. En effet, des accès mémoires contiguës sont essentiels pour utiliser correctement le cache, qui peut de toute manière être surchargé à cause de la taille des données que l'on utilise. Du côté des optimisations de compilation, on se rend compte que l'option O3 a une influence importante dans les seuls cas où le SIMD est utilisable. Les options O1 et O2 elles sont toujours efficaces quelque soit le code.

3 | TP3

3.1 - Objectifs

Ce TP a pour objectif d'étudier le comportement d'un processeur pipeline simple sur des programmes écrits en assembleur. Nous utiliserons le simulateur de processeur MIPS eduMIPS http://www.edumips.org/

3.2 - Etude du pipeline sur des programmes élémentaires

Prog	Forwarding	Cycles	CPI	Cycles d'attente
sum.s	Non	13	2.6	4
mul.s	Non	16	2.66	6
sumf.s	Non	16	3.2	7
divf.s	Non	36	7.2	27
abs.s	Non	19	2.375	7
double.s	Non	90	3.6	61 (Raw + Structural)
double.s	Oui	83	3.32	54 (Raw + Structural)

Latences des différentes opérations :

Opération	Latence (cycles)	Type
ld	5	Entier
l.d	5	Flottant
dadd	6	Entier
add.d	8	Flottant
sd	5	Entier
s.d	5	Flottant
dmul	5	Entier
mul.d	11	Flottant
div.d	30	Flottant
mflo	5	Entier
halt	5	x

Si on exécute le programme *double.s*, on obtient les performances suivantes :

Type	Nombre de cycles	CPI
Non-forwardé	90	3.6
Forwardé	83	3.32

On observe que, même si le forwarding permet de réduire le nombre d'instructions en permettant d'accélérer les accès mémoire, notamment au niveau des instructions d'addition et de multiplication, le principal facteur ralentissant le programme est le nombre d'instructions nécessaire pour l'opération de division. Celle-ci nécessite en effet 24 instructions, sachant qu'il y a deux divisions dans ce programme. Ainsi, il n'est pas possible d'accélérer le programme au-delà de cette limite.

On pourrait accélérer ces programmes en ré-agençant les instructions pour séparer au maximum les instructions occupant les mêmes sections du processeur, et ainsi réduire l'attente qui existe pour la section.

3.3 - Comportement des programmes complexes

On teste les programmes fibs.s et daxpy.s et on évalue leurs nombres d'instructions lors de l'exécution. On arrive aux résultats suivants :

Prog	Forwarding	Cycles	CPI	Cycles d'attente	Rapport Nb cycles
fib.s	Non	153	1.961	62	-
fib.s	Oui	101	1.29	10	1.5
daxpy.s	Non	309	3.059	193	-
daxpy.s	Oui	224	2.217	108	1.37

On constate que le forwarding permet d'améliorer les performances du système en supprimant un nombre important de cycles d'attente. Cependant, cela ne fait pas de miracles, et une quantité importante d'instructions d'attente (notamment sur le programme daxpy.s) subsiste. Il faut donc optimiser le code asssembleur pour supprimer ces instructions d'attente.

Prog	Forwarding	Cycles	CPI	Cycles d'attente	Rapport Nb cycles
fib.s	Non	153	1.961	62	-
fib.s	Oui	101	1.29	10	1.5
fib.s réarrangé	Non	123	1.576	32	1.24
fib.s réarrangé	Oui	91	1.16	0	1.68
fib.s réarrangé + rec	Non	114	1.652	32	1.34
fib.s réarrangé + rec	Oui	82	1.188	0	1.86
daxpy.s	Non	309	3.059	193	-
daxpy.s	Oui	224	2.217	108	1.37
daxpy.s opti	Non	273	2.2.702	157	1.13
daxpy.s opti	Oui	212	2.0999	96	1.05
daxpy.s déroulé	Oui	106	1.341	21	2

Réarrangement du addi dans le code fib.s

```
1    addi $t1, $t1, -1      ; decrement loop counter
2    add  $t2, $t3, $t4     ; $t2 = F[n] + F[n+1] 2 RAW
3    sw   $t2, 8($t0)       ; Store F[n+2] = F[n] + F[n+1] in array 2 RAW
4    addi $t0, $t0, 4       ; increment address of Fib. number source
```

Réduction d'une instruction dans la boucle par récursivité des résultats (on réutilise F[n+2] en tant que F[n+1] (t4) de la boucle suivante)

```
1  ; Compute first twelve Fibonacci numbers and put in array, then print
2        .data
3  fibs: .space 48          ; "array" of 12 32_bits words to contain fib
    ↪  values
4  size: .word  12          ; size of "array"
5        .text
6        addi $t0, r0, fibs    ; load address of array
7        lw   $t5, size(r0)    ; load array size
8        addi $t2, r0,1        ; 1 is first and second Fib. number
9        sw   $t2, 0($t0)      ; F[0] = 1
10       sw   $t2, 4($t0)      ; F[1] = F[0] = 1
11       addi $t1, $t5, -2     ; Counter for loop, will execute (size-2)
    ↪  times
12       lw   $t4, 4($t0)      ; Get value from array F[n+1] -> $t4
13 loop: lw   $t3, ($t0)       ; Get value from array F[n] -> $t3
14       addi $t1, $t1, -1     ; decrement loop counter
15       add  $t4, $t3, $t4    ; $t4 (F[n+1(+1)]) = F[n] + F[n+1]
16       sw   $t4, 8($t0)      ; Store F[n+2] = F[n] + F[n+1] in array
17       addi $t0, $t0, 4      ; increment address of Fib. number source
18       bnez $t1, loop        ; repeat if not finished yet.
19       nop
20       halt
```

Optimisations pour le programme daxpy.s

Pour améliorer le fonctionnement du programme daxpy, une première approche est de réarranger les instructions. En effet, dans le programme initial, une multiplication à virgule flottante est directement suivie d'une addition utilisation le résultat de cette opération. Cela résulte en un temps de traitement accru, puisque ces opérations prennent un temps important. L'idée est donc de modifier l'ordre des instructions afin qu'il n'y ait plus ce type de dépendances. On arrive donc au programme suivant :

```
1         ;; MIPS program double ax plus y (daxpy)
2
3         .data
4  A:             .double 10.0
5  X:             .double 8.0,
    ↪  1.0,2.0,3.0,4.0,5.0,7.0,8.0,9.0,10.0,11.0,12.0
6  Y:             .double 3.0,14.0,13.0,4.0,8.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0
7  N:             .word 12
8
9         .text
10 main:
11        l.d    f0,A(r0)                ; f0 = A
12        dadd   r1,r0,r0                ; r1=i=0
13        ld     r2,N(r0)                ; r2=N
14 loop:       l.d   f1,X(r1)                 ; f1=x[i]
15        mul.d  f1,f1,f0                ; f1=axi
16        l.d    f2,Y(r1)                ; f2=y[i]
17        daddi  r2,r2,-1                ; N++
18        add.d  f2,f2,f1                ; f2=axi+yi
19        s.d    f2,Y(r1)                ; y[i]=axi+yi
20        daddi  r1,r1,8                ; i++
21        bne    r2,r0,loop        ;
22        nop
23        halt
```

L'idée de ce programme est de commencer par réaliser la multiplication de X[i] par A pour ensuite l'ajouter à Y[i]. On profite des moments où le processeur n'est pas utilisé pour effectuer les incréments adéquats.

Pour améliorer les performances du système, nous pouvons faire appel à un déroulage de boucle. Pour ce faire, on commence par faire une préparation du code de la manière suivante :

```
1         ;; MIPS program double ax plus y (daxpy)
2       .data
3  A:   .double 10.0
4  X:   .double 8.0, 1.0,2.0,3.0,4.0,5.0,7.0,8.0,9.0,10.0,11.0,12.0
```

```
Y:    .double 3.0,14.0,13.0,4.0,8.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0
N:    .word 12

      .text
main:
        l.d   f0,A(r0)                ; f0 = A
        dadd  r1,r0,r0                ; r1=i=0
        daddi r2,r0,X                 ; r2=X
        daddi r3,r0,Y                 ; r3=Y
        daddi r3,r3,-8                ; r3 pointe Y[-1]
        ld    r4,N(r0)                ; r4=N
loop:       l.d   f1,(r2)                   ; f1=x[i]
        daddi r3,r3,8                 ; Y[i] pointe sur Y[i+1]
        mul.d f1,f1,f0                ; f1=axi
        l.d   f2,(r3)                 ; f2=Y[i]
        add.d f2,f2,f1                ; f2=axi+yi
        daddi r4,r4,-1                ; N--
        s.d   f2,(r3)                 ; y[i]=axi+yi
        daddi r2,r2,8                 ; X[i] pointe sur X[i+1]
        bne   r4,r0,loop        ;
        nop
        halt
```

Cette préparation consiste à stocker les pointeurs vers les tableaux X et Y dans des registres plutôt que d'utiliser des constantes de décalage. Ainsi, on pourra utiliser des constantes lors des accès suivants. Dans ce code, on a également changé la manière d'accéder à Y[i] pour pouvoir faire la mise à jour du pointeur au début, plutôt qu'à la fin de la boucle.

Une fois ces modifications effectuées, nous pouvons dès lors utiliser le code déroulé suivant :

```
            ;; MIPS program double ax plus y (daxpy)
      .data
A:    .double 10.0
X:    .double 8.0, 1.0,2.0,3.0,4.0,5.0,7.0,8.0,9.0,10.0,11.0,12.0
Y:    .double 3.0,14.0,13.0,4.0,8.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0
N:    .word 12

      .text
main:
        l.d   f0,A(r0)                ; f0 = A
        dadd  r1,r0,r0                ; r1=i=0
        daddi r2,r0,X                 ; r2=X
        daddi r3,r0,Y                 ; r3=Y
        ld    r4,N(r0)                ; r4=N
loop:       l.d   f1,(r2)                   ; f1=x[i]
```

```
16          l.d   f3,8(r2)                ; f1=x[i+1]
17          l.d   f5,16(r2)          ; f1=x[i+2]
18          l.d   f7,24(r2)          ; f1=x[i+3]
19          mul.d f1,f1,f0                 ; f1=axi
20          mul.d f3,f3,f0                 ; f3=ax(i+1)
21          mul.d f5,f5,f0                 ; f5=ax(i+2)
22          mul.d f7,f7,f0                 ; f7=ax(i+3)
23          l.d   f2,(r3)                ; f2=Y[i]
24          l.d   f4,8(r3)                 ; f4=Y[i]
25          l.d   f6,16(r3)          ; f6=Y[i]
26          l.d   f8,24(r3)          ; f8=Y[i]
27          add.d f2,f2,f1                 ; f2=axi+yi
28          add.d f4,f4,f3                 ; f4=ax(i+1)+y(i+1)
29          add.d f6,f6,f5                 ; f6=ax(i+2)+y(i+2)
30          add.d f8,f8,f7                 ; f8=ax(i+3)+y(i+3)
31          daddi r4,r4,-4                 ; N-=4
32          s.d   f2,(r3)                ; y[i]=axi+yi
33          s.d   f4,8(r3)                 ; y[i+1]=axi+yi
34          s.d   f6,16(r3)          ; y[i+2]=axi+yi
35          s.d   f8,24(r3)          ; y[i+3]=axi+yi
36          daddi r2,r2,32                 ; X[i] pointe sur X[i+4]
37          daddi r3,r3,32                 ; Y[i] pointe sur Y[i+4]
38          bne   r4,r0,loop          ;
39          nop
40          halt
```

Ce code présente l'avantage de diminuer le nombre de boucles (ce qui supprime certaines instructions) et que, grâce au pipeline, les dépendances des données sont réduites, ce qui optimise le pipeline. Nous arrivons à un fonctionnement en seulement 106 cycles dont 21 d'attente. Ce sont des attentes de type Str (structurelles) liées au fait que, lorsque les instructions de multiplication ont terminé l'opération, il y a collision avec les instructions suivantes qui utilisent l'étage MEM du pipeline. Ce typre de collisions semble difficile à résoudre, en tout cas en gardant un code relativement bien organisé.

Nous constatons donc que le déroulage de boucle permet une très forte amélioration des performances du système. Nous constatons cependant aussi que cela rend le système moins flexible, puisque, pour bien fonctionner, il faut que le nombre de boucles soit un multiple du facteur de déroulage (ici 4). Néanmoins, les performances sont significatives, ce qui permet de démontrer l'intérêt de ce procédé.

3.4 - Écriture de programmes assembleur

Dans cette partie, on va écrire plusieurs programmes en assembleur à partir de leur code C. Pour toute cette partie, le simulateur sera en mode forward.

3.4.1 - Parite.s

On souhaite faire une opération sur une condition de parité entre deux variables.

```
#define N 12
double A[N], B[N] ;
for(int i=0 ; i<N; i++){
    if ( i&0x1) {   // i impair
        B[i]=B[i]+A[i] ;
    } else {        // i pair
        B[i]=B[i]-A[i] ;
    }
}
```

Le programme en équivalent en assembleur devient :

```
;; MIPS program parity check
    .data
N:  .word  12
A:  .double 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
B:  .double 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

    .text
main:
    dadd r1,r0,r0       ; r1=i=0
    lw r2, N(r0)        ; r2=N
loop:
    l.d f1, A(r1)       ; f1=A[i]
    l.d f2, B(r1)       ; f2=B[i]
    andi $t0, r1, 1     ; i&1
    bnez $t0, impair    ; if i impair
    sub.d f2, f2, f1    ; f2=B[i]-A[i]
    beq r0,r0,end
    nop
impair:
    add.d f2, f2, f1    ; f2=B[i]+A[i]
end:
    s.d f2, B(r1)       ; B[i]=f2
    daddi r1,r1,8       ; i++
    daddi r2,r2,-1      ; N--;
    bne r2, r0, loop
    nop
    halt
```

Il peut y avoir deux approches pour le code en assembleur. La première serait d'avoir des sauts conditionnels. La deuxième serait d'exécuter $B = B - A$ puis $B = B + 2A * (i\&1)$.

Après essai, les sauts conditionnels sont bien plus performants que de faire plusieurs instructions de calculs.

Il est exécuté en 163 cycles dont 12 suspensions pour un CPI de 1,314.

L'intérêt de cette fonction est de montrer qu'une condition alternant l'exécution d'un code peut être fortement optimisé simplement en exécutant les deux branches de la condition à chaque itération (i=i+2).

La version déroulée donne :

```
;; MIPS program parity check
    .data
N:  .word  12
A:  .double 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
B:  .double 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

    .text
main:
    dadd r1,r0,r0       ; r1=i=0
    lw r2, N(r0)        ; r2=N
loop:
    l.d f1, A(r1)       ; f1=A[i]
    l.d f2, B(r1)       ; f2=B[i]
    sub.d f2, f2, f1    ; f2=B[i]+A[i]
    s.d f2, B(r1)       ; B[i] = f2
    daddi r1,r1,8       ; i+=2
    l.d f3, A(r1)       ; f3=A[i+1]
    l.d f4, B(r1)       ; f4=B[i+1]
    add.d f4, f4, f3    ; f4=B[i+1]-A[i+1]
    s.d f4, B(r1)       ; B[i+1] = f4
    daddi r1,r1,8       ; i+=2
    daddi r2,r2,-2      ; N--;
    bne r2, r0, loop
    nop
    halt
```

Sans optimisations, on arrive déjà à avoir 133 cycles dont 48 cycles d'attente. Malheureusement, nous n'avons pas la possibilité d'ajouter d'offset lors de chargement de valeur et ainsi compenser les les cycles d'attentes entre les instructions add et sub. Cependant, on peut jouer des indices en profitant du faible nombre de cycle de l'instruction addi.

On optient la version optimisée :

```
;; MIPS program parity check unrolled + optimized
    .data
N:  .word  12
```

```
A:   .double 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
B:   .double 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

     .text
main:
     dadd r1,r0,r0        ; r1=i=0
     lw r2, N(r0)         ; r2=N
loop:
     l.d f1, A(r1)        ; f1=A[i]
     l.d f2, B(r1)        ; f2=B[i]
     daddi r1,r1,8        ; i++
     sub.d f2, f2, f1     ; f2=B[i]+A[i]
     l.d f3, A(r1)        ; f3=A[i+1]
     l.d f4, B(r1)        ; f4=B[i+1]
     add.d f4, f4, f3     ; f4=B[i+1]-A[i+1]
     daddi r2,r2,-2       ; N--;
     s.d f4, B(r1)        ; B[i+1] = f4
     daddi r1,r1,-8       ; i--
     s.d f2, B(r1)        ; B[i] = f2
     daddi r1,r1,16       ; i+=2
     bne r2, r0, loop
     nop
     halt
```

On obtient maintenant une exécution en 109 cycles et 18 cycles d'attente pour un CPI de 1,329. On pourrait profiter de la variable de r1 (i) pour arrêter la boucle, mais cela ne change pas le nombre de cycles (daddi r2 sous l'addition).

Prog	Cycles	CPI	Cycles d'attente	Rapport Nb cycles
parite.s	163	1.314	12	-
parite.s + deroule	133	1.43	48	1.22
dparite.s + deroule + opti	109	1.329	18	1.49

3.4.2 - Somme.s

On va étudier cette fois-ci une boucle qui fait une somme de chaque composante :

```
#define N 12
double A[N], somme=0.0 ;
for(int i=0 ; i<N; i++){
     somme += A[i] ;
}
```

Le programme équivalent :

```
    .data
N:  .word  96
A:  .double 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
S:  .double 0

    .text
main:
    dadd r1,r0,r0       ; r1=i=0
    lw r2, N(r0)        ; r2=N
    l.d f1, S(r0)       ; f1=S
loop:
    l.d f2, A(r1)       ; f2=A[i]
    add.d f1, f1, f2    ; f1=f1+A[i]
    addi r1, r1, 8      ; i++
    bne r1, r2, loop
    s.d f1, S(r0)       ; S=somme(A)
    halt
```

Il s'exécute en 81 cycles et seulement 12 cycles d'attente. Cela va être compliqué à optimiser.

Après déroulage :

```
    .data
N:  .word  96
A:  .double 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
S:  .double 0

    .text
main:
    dadd r1,r0,r0       ; r1=i=0
    lw r2, N(r0)        ; r2=N
    l.d f1, S(r0)       ; f1=S
loop:
    l.d f2, A(r1)       ; f2=A[i]
    add.d f1, f1, f2    ; f1=f1+A[i]
    addi r1, r1, 8      ; i++
    l.d f3, A(r1)       ; f2=A[i+1]
    add.d f1, f1, f3    ; f1=f1+A[i+1]
    addi r1, r1, 8      ; i++
    bne r1, r2, loop
    s.d f1, S(r0)       ; S=somme(A)
    halt
```

On obtient 75 cycles et 13 cycles d'attente RAW et 6 WAW. On ne gagne pas beaucoup car les instructions add s'attendent entre elles. Pour optimiser encore un peu plus, il

faudrait revenir une étape avant et profiter des cycles d'instructions du saut de la boucle pour éviter les cycles d'attente. De plus, pour pouvoir réorganiser les instructions, il faut remonter une instruction load avant la boucle.

Après optimisation :

```
    .data
N:  .word  96
A:  .double 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
S:  .double 0

    .text
main:
    dadd r1,r0,r0       ; r1=i=0
    lw r2, N(r0)        ; r2=N
    l.d f1, S(r0)       ; f1=S
    l.d f2, A(r1)       ; f2=A[i]
loop:
    addi r1, r1, 8      ; i++
    add.d f1, f1, f2    ; f1=f1+A[i]
    l.d f2, A(r1)       ; f2=A[i+1]
    bne r1, r2, loop
    nop
    s.d f1, S(r0)       ; S=somme(A)
    halt
```

Prog	Cycles	CPI	Cycles d'attente	Rapport Nb cycles
somme.s	81	1.528	13	-
somme.s + deroule	75	1.595	13+6	1.08
somme.s + opti	71	1.29	0	1.14

Il faut donc faire attention aux boucles simples, qui ne peuvent pas forcément êtres déroulées sinon elles introduisent des attentes mémoires.

3.4.3 - power.s

On veut optimiser le code C suivant :

```
double x=2.0, w=1.0 ;
int n=5 ;
for(int i=0 ; i<n; i++)
    w *= x;
```

On obtient en Assembleur :

```
;; MIPS program pow(x,n)
    .data
N:  .word 5
X:  .double 2
W:  .double 1

    .text
    dadd r1,r0,r0       ; r1=i=0
    lw r2, N(r0)        ; r2=n
    l.d f1, W(r0)       ; f1=w
    l.d f2, X(r0)       ; f2=x
loop:
    mul.d f1, f1, f2    ; w*=x
    addi r1, r1, 1      ; i++
    bne r1, r2, loop
    s.d f1, W(r0)
    halt
```

Il s'exécute en 50 cycles, dont 17 RAW et 4 WAW. Il peut normalement être optimisé. Pour cela, on va utiliser le calcul des x^{2^k} successifs :

```
;; MIPS program pow(x,n)
    .data
N:  .word 5
X:  .double 2
W:  .double 1

    .text
    lw r2, N(r0)        ; r2=n
    l.d f1, W(r0)       ; f1=w
    l.d f2, X(r0)       ; f2=x
loop:
    andi $t0, r2, 1     ; n&1
    beqz $t0, pair      ; if n pair
    mul.d f1, f1, f2    ; w*=x sinon
pair:
    mul.d f2, f2, f2
    dsrl r2, r2, 1      ; n>>1
    bnez r2, loop       ; while(n)
    s.d f1, W(r0)
    halt
```

Il s'exécute maintenant en 37 cycles dont 4 RAW. On a gagné des cycles en enchaînant deux multiplications à la suite n'utilisant pas le registre f1 et donc de cycles d'attentes. On a toujours des traces d'attentes entre deux itérations mais on ne pourrait pas dérouler

le programme d'un facteur 2 car on réintroduirait le problème précédemment réglé avec l'attente sur l'instruction de mul.d utilisant f2.

On a un rapport de nombre de cycle = 1,35.

3.5 - Conclusion

On a vu au cours de ce TP, l'importance des optimisations présentes sur une architecture. Nous avons notamment vu la capacité du pipeline à diminuer les cycles d'instructions pour l'exécution d'un programme. Il est alors important de connaître son système pour pouvoir profiter des optimisations présentes qui peuvent faire gagner beaucoup de temps (voir section 3.3). De plus, l'expérience apportée par l'optimisation de cas particuliers permettent de prédire les comportements ou les optimisations possible sur des boucles. Nous conseillons donc à tous architectes programme de prendre connaissance de l'architecture sur laquelle leur programme sera déployé. Vous ne serez pas déçus des résultats.