Architecture des ordinateurs

Département Informatique

Erwan LEBAILLY — Vilavane LY — Vincent TRÉLAT — Benjamin ZHU

5 avril 2022

***

Table des matières

1 Chapitre 1

1.1 Exercice 1

Avec la convention $0 \leftrightarrow$ `faux` et $1 \leftrightarrow$ `vrai`, $0 \wedge 1 =$ `faux`.

1.2 Exercice 2

On donne la table de c_0 :

c_0 :

$a_0 \backslash b_0$	0	1
0	0	1
1	1	0

On peut interpréter cette table comme la table de vérité du "ou exclusif", le *xor*. Ainsi, c_0 coincide avec $a_0 \oplus b_0 = (a_0 \vee b_0) \wedge (\neg(a_0 \wedge b_0))$.

1.3 Exercice 3

a. Montrons que l'opérateur xor $\oplus$ est associatif et commutatif :
Soient $a, b, c \in \{0, 1\}$.

Associativité : on donne ci-dessous la table de vérité de $(a \oplus b) \oplus c$ et $a \oplus (b \oplus c)$:

a	b	c	$a \oplus b$	$(a \oplus b) \oplus c$	$a \oplus (b \oplus c)$
0	0	0	0	0	0
0	0	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	0
0	1	0	1	1	1
0	1	1	1	0	0
1	1	0	0	0	0
1	1	1	0	1	1

Commutativité : on donne ci-dessous la table de vérité de $a \oplus b$ et $b \oplus a$:

a	b	$a \oplus b$	$b \oplus a$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

Enfin, $a \oplus a = 0$ et $a \oplus 0 = a$.

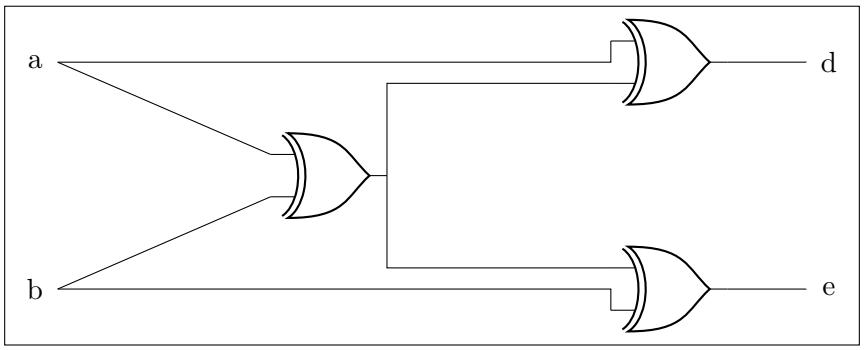
On peut maintenant montrer le résultat demandé :

$$d = a \oplus c = a \oplus (a \oplus b) = a \oplus a \oplus b = b$$

$$e = b \oplus c = b \oplus (a \oplus b) = b \oplus a \oplus b = a$$

■

b. On donne ci-dessous le circuit correspondant :



1.4 Exercice 4

a. On écrit le code suivant :

```c
int main()
    {
        printf("Sizeof int: %lu octets\n", sizeof(int));
        printf("Sizeof short: %lu octets\n", sizeof(short));
        printf("Sizeof char: %lu octets\n", sizeof(char));
        return 0;
    }
```

La sortie est la suivante :

```
    Sizeof int: 4 octets
    Sizeof short: 2 octets
    Sizeof char: 1 octets
```

b. On écrit le code suivant :

```c
int main()
    {
        int a = pow(2, 31);
        int b = pow(2, 31);
        int c = a + b;
        printf("%d\n", c);
        return 0;
    }
```

La sortie affiche 0, ce qui correspond bien à $2^{32} \mod (2^{32})$

1.5 Exercice 5

On donne ci-dessous l'écriture binaire sur 4 et 8 bits de 0, 1, -1 et -2 :

x	4 bits	8 bits
0 :	0000	0000 0000
1 :	0001	0000 0001
-1 :	1111	1111 1111
-2 :	1110	1111 1110

1.6 Exercice 6

a. $m_1 = 0001$ et $m_{-1} = 1001$.
b. En abusant de la notation + pour des mots : $m_0 = m_1 + m_{-1} = 1010$.
c. En suivant la règle de signes, 1010 est l'encodage de -2.

1.7 Exercice 7

Soit b un nombre de bits. Soit x un entier relatif qu'on souhaite représenter sur b bits.
Si $x \geq 0$, alors l'encodage de x correspond à une écriture dans $[0, 2^{b-1} - 1]$, alors cette écriture commence par un zéro (de 00...0 à 01...1). Si $x < 0$, alors $2^b - x \in [2^{b-1}, 2^b - 1]$ (soit de 10...0 à 11...1), son écriture commence par un 1.

■

1.8 Exercice 8

Dans le premier code, on dispose de 2 cases mémoires différentes. Le résultat affiché est -106 pour la valeur de d, ce qui est normal puisque d est signé.
Dans le deuxième code, on utilise une seule case mémoire à travers l'utilisation de deux pointeurs, un signé et un non signé. Le résultat affiché est identique au premier code.
Cela permet de montrer que la mémoire est "non typée", l'interprétation de la valeur mémoire dépend directement du type de l'objet qui lit cette valeur.

1.9 Exercice 9

a. 10 s'écrit 2×5 et toute puissance de 2 s'écrit 2^k où $k \in \mathbb{N}$.
Ainsi, si $x \in 2^{\mathbb{N}}$ (par abus de langage) est divisible par 10, alors x contient au moins 2 et 5 dans sa décomposition en facteurs premiers, ce qui donne une contradition avec la propriété précédemment énoncée.

■

b. Supposons que 0.1 soit représentable sur kl bits. Alors, d'après le résultat du cours, $2^l \times 0.1$ est un entier, autrement dit 2^l est divisible par 10. D'après la question précédente, c'est impossible.

■

1.10 Exercice 10

L'écriture binaire approchée de 0.1 est $0.0001\ 1001_2$, de valeur décimale 0.09765625.

1.11 Exercice 11

a. On écrit la fonction suivante en C :

```c
int main() {
    for (int i = 0; i<10; i++){
        for (int j = 0; j < 10; j++){
            float a = i/10.0, b = j/10.0, c = (i+j)/10.0;
            printf("(%d, %d) : %s\n", i, j, (a+b == c)?"true":"false");
        }
    }
    return 0;
}
```

La sortie affichée contient, entre autres, les résultats suivants :

- (1, 4) : true
- (1, 5) : true
- (1, 6) : false
- (1, 7) : true
- (1, 8) : false
- (3, 4) : false
- (3, 5) : true
- (3, 6) : false
- (3, 7) : true
- (3, 8) : true

b. On modifie seulement la ligne d'affichage dans le code [1] :

```c
printf("%.16f + %.16f = %.16f\n", a, b, c);
```

On donne seulement les résultats pour les 5 premiers couples ci-dessus :

```
0.1000000014901161 + 0.4000000059604645 = 0.5000000000000000
0.1000000014901161 + 0.5000000000000000 = 0.6000000238418579
0.1000000014901161 + 0.6000000238418579 = 0.6999999880790710
0.1000000014901161 + 0.6999999880790710 = 0.8000000119209290
0.1000000014901161 + 0.8000000119209290 = 0.8999999761581421
```

c. On remarque que pour une addition, l'égalité `a+b==c` est vérifiée lorsque `a` et `b` sont représentables, ou quand l'un des deux seulement l'est. Dans le second cas, l'erreur de représentation n'a pas eu d'impact sur le résultat puisque c'est la seule erreur du calcul. Ainsi l'égalité reste vraie.
En revanche, dès que les deux flottants ne sont pas représentables, les erreurs s'accumulent et alors la représentation de `c` peut différer de la valeur de `a+b`.

Dans un cas général si on prend $x, y \in \mathbb{R}$ tels que $x = y$, on aura `x==y` quand x et y sont représentables sur un nombre de bits donné [2]. Dans les autres cas, il est possible

1. Comme un `int` est codé sur 4 octets, on donne 16 caractères à chaque affichage.
2. Il faut tout de même faire attention à la précision. Augmenter la précision ne rendra pas les calculs exacts pour autant.

d'obtenir un résultat correct mais cela résulte plutôt du hasard.

d. On modifie la fonction précédente :

```c
int main() {
    for (int i = 0; i<10; i++){
        for (int j = 0; j < 10; j++){
            float a = i/10.0, b = j/10.0, c = (i+j)/10.0;
            if(a+b!=c){
                int m1 = a+b>c;
                int m2 = a+b+.000001>c+.000001;
                printf("(%d, %d) : %d %d\n", i, j, m1, m2);
            }
        }
    }
    return 0;
}
```

On obtient la sortie suivante :

```
// a+b > c is tested before and after adding 1e-6
(1, 6) : 1 1
(1, 8) : 1 1
(3, 4) : 1 1
(3, 6) : 1 1
(4, 3) : 1 1
(6, 1) : 1 1
(6, 3) : 1 1
(6, 8) : 1 1
(7, 9) : 0 0
(8, 1) : 1 1
(8, 6) : 1 1
(9, 7) : 0 0
```

On constate que l'ordre est conservé à chaque fois. Comme 10^{-6} n'est pas représentable sur 16 bits ($10^{-6} \approx 2^{-20}$), on simule l'effet de la propagation d'une erreur.
On en déduit que des erreurs successives d'arrondi ne bousculent pas l'ordre sur des valeurs arrondies. Toutefois ici on effectue le même calcul des deux côtés. On peut donc toujours les comparer [3], même après plusieurs calculs, puisque l'ordre est conservé. On peut également penser que cela ne provoquera pas d'évolution chaotique ou aléatoire de ces valeurs dans les calculs. En revanche, pour une suite d'opérations inconnue, cela risque de devenir insignifiant de vouloir comparer deux flottants.

1.12 Exercice 12

On note $s_1 = \sum_{i=1}^{k} \frac{1}{i}$ et $s_2 = \sum_{i=k}^{1} \frac{1}{i}$.
On écrit la fonction suivante :

3. Pas l'égalité.

```c
int main() {
    float s1 = 0.0, s2 = 0.0;
    long k = 1000000000;
    for (double i = 1; i<k+1; i++){
        s1 += 1/i;
        s2 += 1/(k+1-i);
    }
    printf("k=%ld\n    s1 = %.16f\n    s2 = %.16f\n", k, s1, s2);
    return 0;
}
```

Les résultats sont les suivants :

```
k=1000
    s1 = 7.4854784011840820
    s2 = 7.4854717254638672
k=1000000
    s1 = 14.3573579788208008
    s2 = 14.3926515579223633
k=1000000000
    s1 = 15.4036827087402344
    s2 = 18.8079185485839844
```

On constate que les résultats diffèrent [4] d'autant plus que le nombre d'erreurs successives est grand, ce qui n'est pas étonnant. Là où le résultat interpelle, c'est que l'ordre de parcours de la somme a un impact conséquent sur le résultat.
Cette erreur est due au fait que lorsque le parcours est décroissant, on finit pas les petits nombres. Ces derniers causent alors des erreurs d'arrondi car leur ordre de grandeur est faible par rapport aux premiers nombres.

1.13 Exercice 13

a. La taille d'un `float` en C est de 4 *bytes*, soit 32 bits :
- 1 bit de signe
- 8 bits d'exposant
- 23 bits de mantisse

b. Convertissons d'abord `0x414BD000` en binaire :

$$414BD000_{16} = 01000001010010111101000000000000_2$$

On identifie ensuite les bits de signe, d'exposant et de mantisse :

$$\underbrace{0}_{S}\ \underbrace{10000010}_{E=130}\ \underbrace{10010111101000000000000}_{T}$$

4. Même plus que ça, on dirait que la série converge ! On connait l'équivalent pour la série harmonique : $H_n \underset{n\to+\infty}{\sim} \gamma + \ln(n)$ où $\gamma \approx 0.5$ est la constante d'Euler. Pour $n = 10^9$, on devrait donc plutôt être autour de $H_n \approx 21$.

Le biais b valant 127, notre exposant ici vaut $E - b = 3$. Donc, en "écriture binaire à virgule", on obtient :

$$1.10010111101 \times 2^3 = \underbrace{1100}_{=12}.\underbrace{10111101}_{=0.73828125}$$

Soit finalement :

$$\texttt{0x414BD000}_{16} \text{ encode } 12.73828125$$

c. En suivant le raisonnement inverse, on peut trouver l'exposant et la mantisse de l'encodage de 0.1. On commence par l'écrire en "binaire à virgule" :

$$0.1 = 0.0001100110011001100110011001101_2$$

On décale la virgule pour trouver l'exposant :

$$0.1 = 1.10011001100110011001101_2 \times 2^{-4}$$

Cela donne donc un exposant de $E = b - 4 = 123 = 01111011_2$ sur 8 bits. Enfin, $0.1 > 0$ donc on met un premier bit à 0. On obtient donc l'encodage suivant – dont on donne également la valeur décimale réelle – pour le nombre 0.1 :

$$\underbrace{0}_{\text{signe}}\ \underbrace{01111011}_{\text{exposant}}\underbrace{10011001100110011001101_2}_{\text{mantisse}} = 0.100000001490116119384765625$$

2 Chapitre 2

2.1 Exercice 1

a. Lorsqu'on crée par exemple un tableau de taille un milliard, on obtient une erreur similaire à la suivante :

```
[1]    54133 segmentation fault  ./a.out
```

b. En expérimentant à la main, on trouve qu'on peut créer un tableau de taille maximale 2 096 286.

2.2 Exercice 2

On vérifie par exemple qu'on peut créer un tableau de taille un milliard.

2.3 Exercice 3

La machine utilisée pour ce TD utilise la convention *little endian*.

2.4 Exercice 4

On écrit déjà le code suivant :

```c
#include <stdio.h>
#include <x86intrin.h>

unsigned long int squareSum(int n){
    unsigned long int tic, toc;
    unsigned int ui;
    int a = 0;
    tic = __rdtscp(&ui);
    for (int i = 0; i < n; ++i){
        a = i * i;
    }
    toc = __rdtscp(&ui);
    return toc - tic;
}
int main(){
    printf("n=%d: %lu tics\n", 1000, squareSum(1000));
    printf("n=%d: %lu tics\n", 10000, squareSum(10000));
    printf("n=%d: %lu tics\n", 1000000, squareSum(1000000));
    printf("n=%d: %lu tics\n", 10000000, squareSum(10000000));
    printf("n=%d: %lu tics\n", 100000000, squareSum(100000000));
    return 0;
}
```

On obtient les résultats suivants :

n	10^3	10^4	10^6	10^7	10^8
mesure	3310	42456	7134866	67719612	637584056

On note qu'on semble bien obtenir une relation qui a l'air linéaire, hormis la dernière mesure.

2.5 Exercice 5

On obtient les résultats suivants :

appel\n	10^3	10^5	10^7	10^8	10^9
1er	9108	835910	52244848	532734614	5350923602
2ème	3238	685744	31633110	317468584	3161147144
3ème	3088	322012	32683670	341747932	3100426690
$\frac{\text{écart max.}}{\text{moyenne}}$ en %	117	83	53	53	58

2.6 Exercice 6

a. On teste la fonction avec la fonction `test` calculant la somme des premiers carrés. On obtient des résultats cohérents (croissance linéaire). La fonction `print_timing` comprend une amorce qui exécute 10 fois la fonction à tester et moyenne les mesures sur 100 appels.

b. Voici la fonction `print_timing` :

```
void print_timing(int arg, void (*func)(int), int nb_boot, int nb_call)
{
    // boot up
    for (int i = 0; i < nb_boot; i++)
    {
        func(arg);
    }

    // measure
    unsigned long int tic, toc;
    unsigned int ui;
    tic = __rdtscp(&ui);
    for (int i = 0; i < nb_call; i++)
    {
        func(arg);
    }
    toc = __rdtscp(&ui);

    printf("average time : %lu\n", (toc - tic) / n);
}
```

c. Oui.

d. On obtient les résultats suivants :

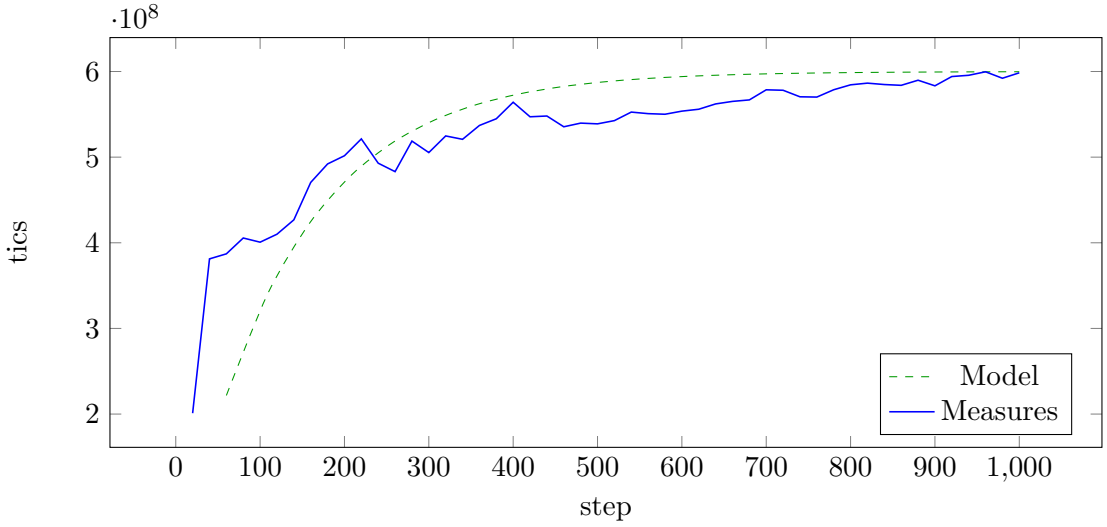
appel\n	10^3	10^5	10^7	10^8	10^9
mesure `print_timing`	4214	432308	31686176	308206789	3176004203

2.7 Exercice 7

On obtient les résultats suivants (arrondis) :

pas	1	2	3	4	8	16	32
mesure ($\times 10^6$ tics)	59	61	63	66	83	102	197

On peut ainsi tracer les résultats obtenus pour les pas situés entre 1 et 1000 :



On remarque que les temps d'accès semblent converger [5] ! Pour mieux l'illustrer, on trace par dessus les mesures la fonction suivante (en vert sur le graphique) :

$$x \mapsto 6 \times 10^8 (1 - e^{-\frac{x}{130}})$$

2.8 Exercice 8

a. On écrit d'abord les deux fonctions suivantes [6] :

```
void access_seq(int *tab, int n)
{
    int tmp;
    for (int i = 0; i < n; i++)
    {
        tmp = tab[i];
    }
}
```

5. Plus précisément – car on sait qu'il est impossible que cela converge – on semble observer une croissance logarithmique.

6. Il faut inclure les bibliothèques `stdlib` et `time` au début du code, et initialiser le *seed* avec `srand(unsigned int)time(NULL)`.

```c
void access_rand(int *tab, int n)
{
    int tmp;
    for (int i = 0; i < n; i++)
    {
        tmp = tab[(unsigned long int)rand() % 1000000000];
    }
}
```

On obtient les temps suivants :

n	10^3	10^6	10^7	10^8	10^9
`access_seq`	5111	5316032	44992606	450441905	4609590108
`access_rand`	3855618	139739559	1386369206	14181946914	148526968620

b. On constate que la fonction `access_rand` est (très largement, d'un facteur 30 environ) plus lente que la fonction `access_seq` ! Est-ce réellement le cas ? Pour l'instant, on ne peut pas répondre à cette question, toutefois on peut facilement se rendre compte qu'on mesure bien trop de choses. En effet, à chaque accès aléatoire, on appelle – donc on mesure – la fonction `rand`, qui prend visiblement un temps non négligeable.

c. On modifie légèrement les fonctions de mesure [7] et on écrit la fonction suivante :

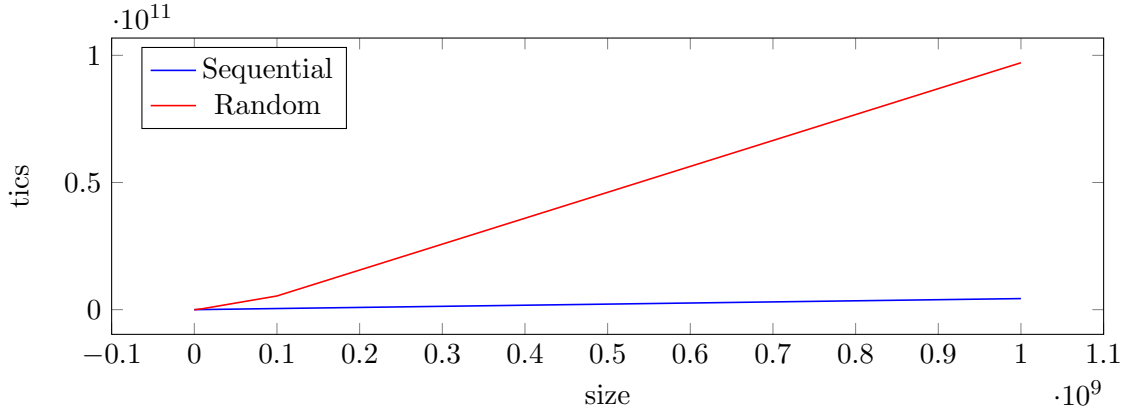
```c
void access_aux(int *tab, unsigned long int *aux, unsigned long int n)
{
    int tmp;
    for (register unsigned long int i = 0; i < n; i++)
    {
        tmp = tab[aux[i]];
    }
}
```

On obtient les temps suivants :

n	10^3	10^6	10^7	10^8	10^9
`access_seq`	4298	4263336	42850857	444757816	4356153500
`access_rand`	3463	8848371	377748557	5385496993	97091291039

On obtient les courbes suivantes :

7. Tous les codes sont disponibles sur GitHub. Attention toutefois, l'exécution du code correspondant à cette mesure peut prendre plusieurs dizaines de minutes et nécessite au moins une dizaine de Go de RAM.



Passons rapidement sur l'étude de ces courbes.
Accès séquentiel : on peut définir la fontion

$$t_{\text{seq}} : n \mapsto \tau_{\text{seq}} n$$

donnant une estimation temps mis pour réaliser des accès séquentiels en mémoire, où $\tau_{\text{seq}} \approx 4.298$. Le temps d'un accès séquentiel en mémoire, *i.e.* sur des cases mémoires successives, est donc d'environ 4 *tics*[8]. C'est très rapide !

Accès aléatoire : on peut de même définir la fontion

$$t_{\text{rand}} : n \mapsto \tau_{\text{rand}} n$$

où $\tau_{\text{rand}} \approx 101.9$. Ainsi, un accès aléatoire dans la mémoire prendra environ 102 *tics*. C'est approximativement 20 fois plus long que pour un accès séquentiel, toutefois cela reste du temps constant ! La différence peut déjà s'expliquer par le fait que pour "sauter" d'une position à une autre, le pointeur qui fait office de tête de lecture doit faire un calcul arithmétique pour déterminer l'adresse mémoire de la case sur laquelle il doit se rendre. Cela peut nécessiter une soustraction, par exemple, ou encore un modulo, qui est tout de même assez coûteux (quelques dizaines de *tics*).

Conclusion : Néanmoins, la vraie différence vient d'un autre phénomène qui est la mise en cache de données[9]. Lorsqu'on fait des accès séquentiel, on charge des blocs en cache, et on fait une suite d'accès en cache, ce qui est rapide. Lorsqu'on fait une suite d'accès aléatoires, on fait des accès hors-cache presque à chaque nouvel accès, puisque la probabilité que deux accès consécutifs concernent des données présentes dans un même bloc est beaucoup plus faible.

8. Le temps de la mesure augmente linéairement avec le nombre d'accès mémoire. En moyennant, on obtient bien un temps constant par mesure.

3 Chapitre 3

3.1 Exercice 1

`0xAE01D500` et `0xAE01D501`	Oui
`0xAE01D500` et `0xAE01D4FF`	Non
`0xAE01D500` et `0xAE01D580`	Non
`0xAE01D53D` et `0xAE01D542`	Oui
`0xAD01F506` et `0xAE01D508`	Non
`0xAE01D55F` et `0xAE01D560`	Oui

3.2 Exercice 2

On vérifie déjà qu'une instance de type `noeud` occupe bien 24 octets avec la commande :

```
1 printf("sizeof noeud: %lu o", sizeof(struct noeud));
```

Le fait que la somme des tailles diffère de la taille de la somme (de la structure) provient de la compilation : le compilateur fait de l'alignment (*padding*) sur des multiples de 8 afin qu'un élément se trouve toujours en début de bloc. Ainsi, un `noeud` occupe une taille de $20 + 4 = 24$ octets.

a. On peut stocker $\frac{32\times1024}{64} = 512$ blocs. Le noeud 14 par exemple commence à l'adresse `0xAE01D53C` et finit à l'adresse `0xAE01D54F`. Or, le premier bloc se termine à l'adresse `0xAE01D540`[10], soit au "milieu" du noeud 14.

b. On commence par lire le noeud 1, qui charge – c'est donc un accès hors-cache – le bloc contenant la fin du noeud 20, le 1, le 25 et le début du noeud 22. Ensuite, on lit le noeud 2 qui charge le bloc – accès hors-cache – qui contient les noeuds 2, 9 et le début du noeud 14. On fait de même pour le noeud 3, puis pour le 4, on doit charger deux blocs car le noeud 4 est à cheval sur ces deux blocs : on charge donc les noeuds 27 (fin), 10, 24, 4, 21 et 13. On continue jusqu'au noeud 8, qui est dans le bloc du noeud 7, donc pas d'accès hors-cache.
Pour résumer, sur les 27 accès, les accès hors-cache sont : 1, 2, 3, 4, 5, 6, 7 et 16.

3.3 Exercice 3

On obtient les résultats suivants[11] :

10. `0xAE01D500` décalé de 64 octets
11. Sous Mac, on peut accéder à ce genre d'informations par la commande `sysctl`. En particulier, ici on peut utiliser la commande `sysctl -a | grep hw`.

	L1 instruction	L1 données	L2	L3
Taille mémoire	32768	32768	262144	16777216
Ligne de cache	64	64	64	64
Associativité	8	8	4	16

3.4 Exercice 4

On cherche à mettre en évidence la taille du cache L1 et du cache L2. Pour cela, on va regarder les temps mis pour parcourir un tableau en colonnes et si on observe un pic pour une taille N de tableau, cela signifie que le cache est de taille $64 \times N$ ko. En l'occurrence, on s'attend à voir un pic autour de $N = \frac{32768}{64} = 512$ pour L1 et $N = \frac{262144}{64} = 4096$ pour L2.

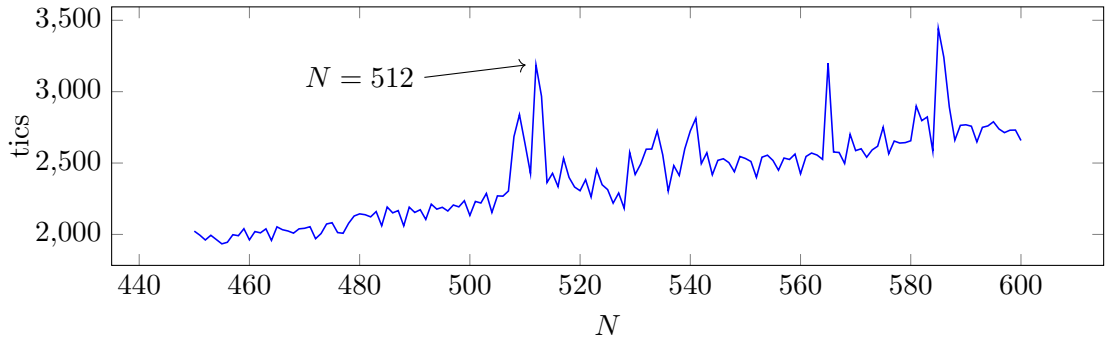


FIGURE 1 – Mise en évidence du cache L1

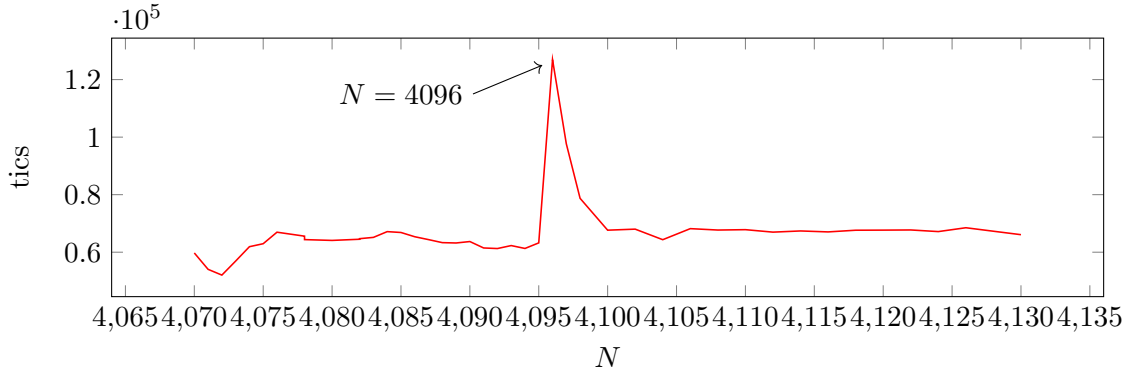


FIGURE 2 – Mise en évidence du cache L2

On donne quelques valeurs numériques autour de $N = 4096$ ci-dessous :

n	4094	4095	4096	4097	4098	4099
en ligne	59737	54042	52042	56912	61912	62952
en colonne	61302	63239	126933	97693	78726	67656

3.5 Exercice 5

Voir le tableau de l'Exercice 3 [12].

3.6 Exercice 6

On redonne ci-dessous les blocs de l'exercice 2 :

p	addresse	p	addresse	p	addresse
2	0xAE01D500	3	0xAF01D900	11	0xBF01DD00
9	0xAE01D514	17	0xAF01D914	5	0xBF01DD14
19	0xAE01D528	12	0xAF01D928	18	0xBF01DD28
14	0xAE01D53C	20	0xAF01D93C	27	0xBF01DD3C
14	0xAE01D53C	20	0xAF01D93C	27	0xBF01DD3C
6	0xAE01D550	1	0xAF01D950	10	0xBF01DD50
15	0xAE01D564	25	0xAF01D964	24	0xBF01DD64
26	0xAE01D578	22	0xAF01D978	4	0xBF01DD78
26	0xAE01D578	22	0xAF01D978	4	0xBF01DD78
23	0xAE01D58C	8	0xAF01D98C	21	0xBF01DD8C
16	0xAE01D5A0	7	0xAF01D9A0	13	0xBF01DDA0

Si on dispose d'un cache de 1 Ko et d'un niveau d'associativité 2, on a $\frac{1024}{64} = 16$ blocs, 128 octets par sous-cache, soit 2 blocs et $\frac{1024}{128} = 8$ sous-caches.
Les couleurs dans le tableau représentent les blocs qui sont dans le même sous-cache. Par exemple, pour 2, 3, 11, 9, 17, 5, etc :

- `(0xAE01D500 >> 6) mod 8 = 4`
- `(0xAF01D900 >> 6) mod 8 = 4`
- `(0xAE01D500 >> 6) mod 8 = 4`
- ...

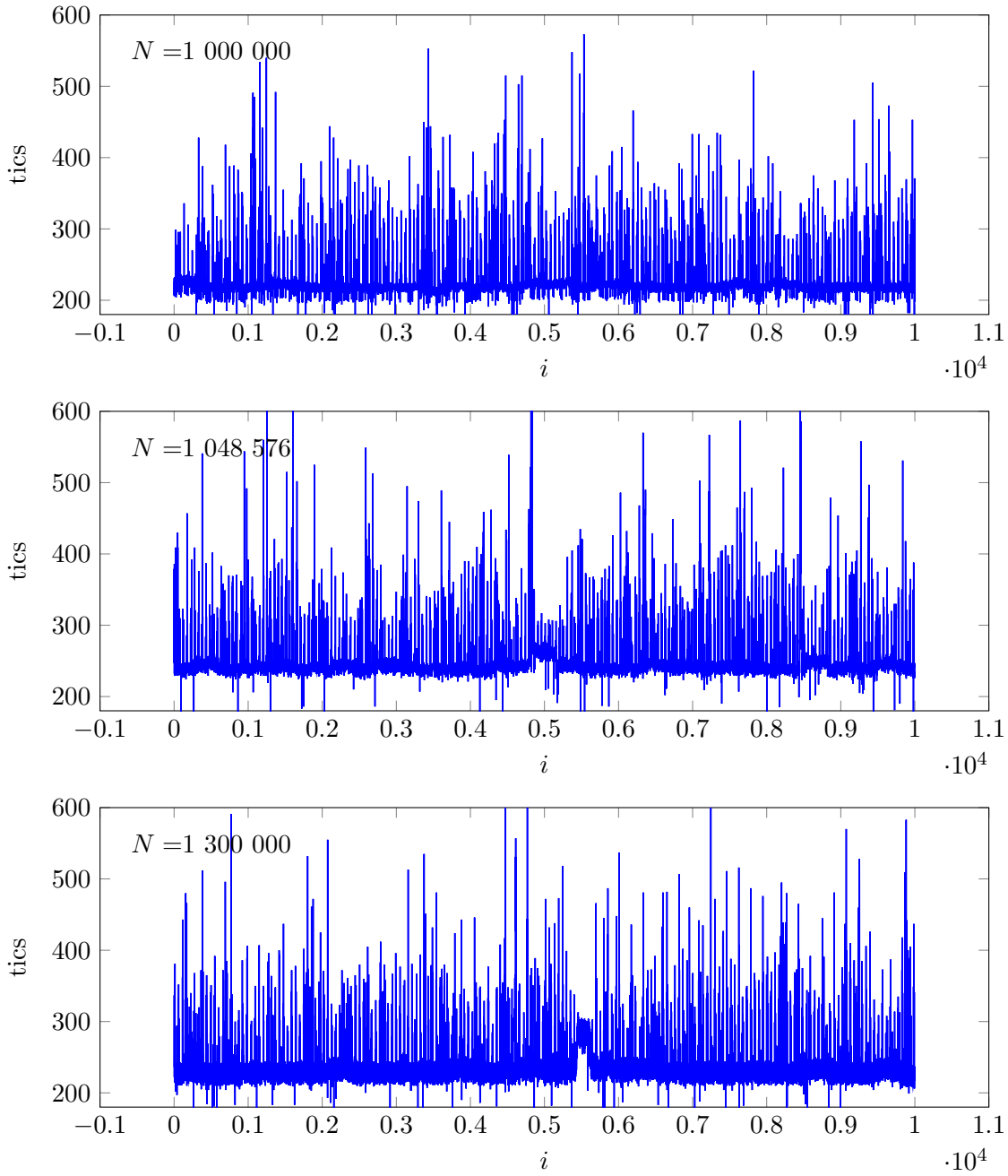
Le premier appel qui est fait en cache est l'accès de 8, qui est dans le bloc chargé par 7. Le premier appel hors-cache non trivial est l'accès à 9, qui est bien dans le même bloc que 2, mais les accès à 3 et à 5 ont écrasé le premier bloc entre temps.
En résumé :

- *cache-miss* : 1, 2, 3, 4, 5, 6, 7, 9, 12, 13, 16, 18, 19, 20, 22, 25, 27
- *cache-hit* : 8, 10, 11, 14, 15, 17, 21, 23, 24, 26

12. Note : sous Mac, on obtient seulement la valeur de l'associativité pour le cache L2 : `machdep.cpu.cache.L2_associativity: 4` . Toutes les autres valeurs ont été obtenues sur la fiche technique du processeur fournie par Intel, en l'occurrence un `Intel(R) Core(TM) i9-9980HK CPU @ 2.40GHz`.

3.7 Exercice 7

On trace les temps successifs obtenus pour les trois expériences (N valant 1000000, 1048576 puis 1300000) :



On constate que pour $N = 2^{20}$, les temps sont globalement toujours supérieurs aux deux autres cas. En effet, on peut calculer les temps moyens (*i.e.* le temps total cumulé

divisé par le nombre d'accès R) $T(N)$ dans chaque cas :

$$T(1000000) = 227.726$$
$$T(1048576) = 252.717$$
$$T(1300000) = 239.417$$

Pourquoi une recherche dichotomique dans un tableau de taille 2^{20} met un temps environ 10% plus long qu'un tableau de taille 1 300 000 ?
Ce phénomène est lié au principe d'associativité dans le système de cache de l'architecture de la machine. En effet, pour un tableau dont la taille est une puissance de 2, la taille de chaque sous-tableau est encore une puissance de 2. Or, le nombre de sous-caches ici est aussi une puissance de 2 – mais beaucoup plus petite [13], d'où le 2^{20} – donc à chaque accès au milieu du tableau tombe sur une puissance de 2. Or, on sait que les blocs mis en cache sont mis dans le sous-cache dont le numéro est calculé "modulo le nombre de sous-cache". Ainsi, on écrase à chaque étape la valeur précédemment mise en cache. On ne fait donc (presque) que des accès hors-cache, d'où le temps plus long.

3.8 Exercice 8

b. On écrit la fonction `transpose_naive` suivante :

```
void transpose_naive(int n, int *mat)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i; j++)
        {
            int tmp = mat[i * n + j];
            mat[i * n + j] = mat[j * n + i];
            mat[j * n + i] = tmp;
        }
    }
}
```

On obtient les résultats suivants :

n	1024	4096	8192	16384	65536
temps	7244426	373965628	1895640055	9090680074	171798691840

c. On écrit ensuite une fonction `transpose_blocs` qui transpose une matrice en 4 blocs égaux :

```
void transpose_blocs(int n, int *mat)
{
    for (int I = 0; I < n; I += n / 2)
    {
        for (int J = 0; J < n; J += n / 2)
        {
```

13. En l'occurrence pour le cache L1 cette puissance est 3 et pour L2 elle vaut 2 !

```
            int i_upper = min(I + n / 2, n);
            int j_upper = min(J + n / 2, n);
            for (int i = I; i < i_upper; i++)
            {
                for (int j = J; j < j_upper; j++)
                {
                    int tmp = mat[i * n + j];
                    mat[i * n + j] = mat[j * n + i];
                    mat[j * n + i] = tmp;
                }
            }
        }
    }
}
```

On obtient les résultats suivants :

n	1024	4096	8192	16384	65536
temps	14578497	643229848	3429862750	17595195907	457658083617

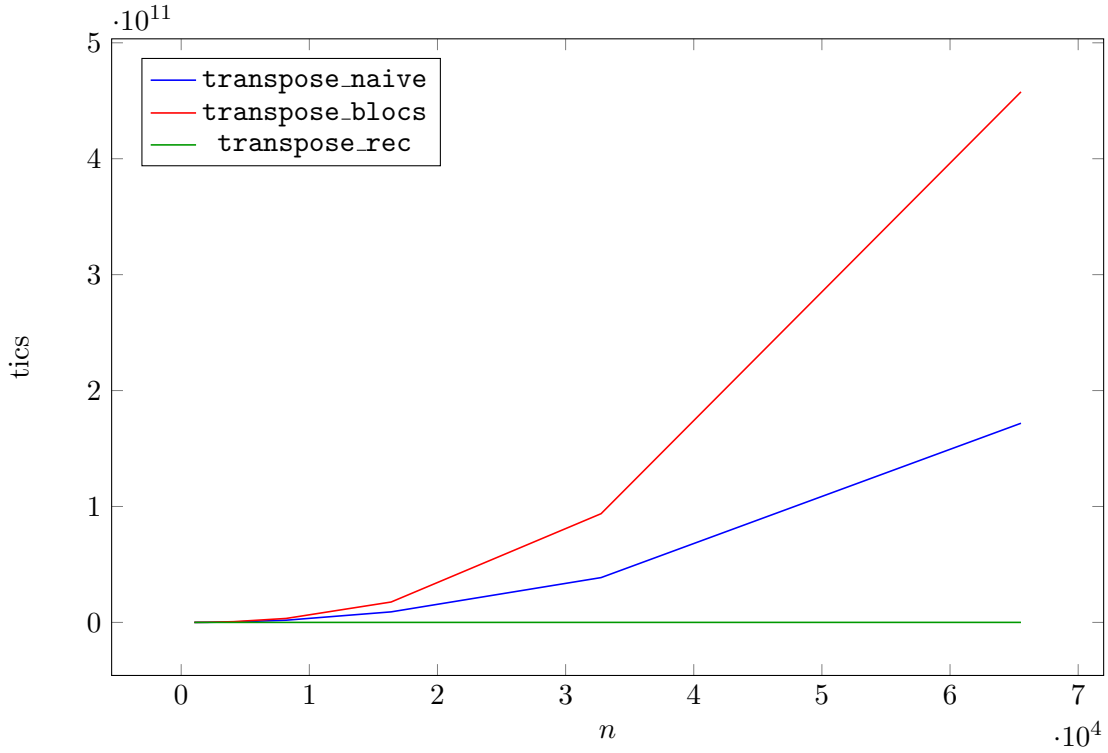
d. Enfin, on définit la fonction `_transpose_rec` comme suit :

```
void _transpose_rec(int *A, int mAb, int mAe, int nAb, int nAe, int m,
    int n, int S)
{
    int nblines = mAe - mAb;
    int nbcols = nAe - nAb;

    if ((nblines <= S) && (nbcols <= S))
    {
        int iA, jA;
        for (iA = mAb; iA < mAe; ++iA)
        {
            for (jA = nAb; jA < nAe; ++jA)
            {
                A[jA * m + iA] = A[iA * n + jA];
            }
        }
    }

    else if (nblines > nbcols)
    {
        int mid = nblines / 2;
        _transpose_rec(A, mAb + mid, mAe, nAb, nAe, m, n, S);
        _transpose_rec(A, mAb, mAb + mid, nAb, nAe, m, n, S);
    }
    else
    {
        int mid = nbcols / 2;
        _transpose_rec(A, mAb, mAe, nAb + mid, nAe, m, n, S);
        _transpose_rec(A, mAb, mAb, nAb, nAb + mid, m, n, S);
    }
}
```

Et on l'appellera sur `_transpose_rec(mat, 0, n, 0, n, n, n, 4)` [14].
On obtient les résultats suivants :

n	1024	4096	8192	16384	65536
temps	72610	627212	1604796	3357688	17023348

On regroupe toutes ces valeurs dans une seule figure :



14. Le dernier argument S, qui vaut 4 ici, est une valeur de seuil sous laquelle on ne procède plus récursivement mais itérativement comme dans la fonction `transpose_naive`.

4 Chapitre 4

4.1 Exercice 1

On obtient le code assembleur suivant (avec `gcc`) :

```
_sum: ## @sum
## %bb.0:
    push rbp
    mov rbp, rsp
    mov dword ptr [rbp - 4], edi
    mov dword ptr [rbp - 8], 0
    mov dword ptr [rbp - 12], 0
LBB0_1: ## =>This Inner Loop Header: Depth=1
    mov eax, dword ptr [rbp - 12]
    cmp eax, dword ptr [rbp - 4]
    jge LBB0_4
## %bb.2: ## in Loop: Header=BB0_1 Depth=1
    mov eax, dword ptr [rbp - 12]
    add eax, dword ptr [rbp - 8]
    mov dword ptr [rbp - 8], eax
## %bb.3: ## in Loop: Header=BB0_1 Depth=1
    mov eax, dword ptr [rbp - 12]
    add eax, 1
    mov dword ptr [rbp - 12], eax
    jmp LBB0_1
LBB0_4:
    mov eax, dword ptr [rbp - 8]
    pop rbp
    ret
                                    ## -- End function
    .globl _main ## -- Begin function main
    .p2align 4, 0x90
_main: ## @main
## %bb.0:
    push rbp
    mov rbp, rsp
    sub rsp, 16
    mov dword ptr [rbp - 4], 100000000
    mov edi, dword ptr [rbp - 4]
    call _sum
    xor eax, eax
    add rsp, 16
    pop rbp
    ret
                                    ## -- End function
```

4.2 Exercice 2

Dans la fonction `sum`, on commence par sauvegarder la dernière valeur du pointeur `rbp` et empile cette valeur. Puis, tout est stocké dans la RAM : `edi` représente l'argument

de la fonction (de manière informelle, c'est `count`), puis on stocke les valeurs de `s` et `i`, qui valent 0, au adresses qui suivent. Pour le reste, on travaille sur des registres.

4.3 Exercice 3

a. Les trois instructions `mov`, `cmp` et `jge` correspondent aux instructions `for(int i=0; i<count; i++)` (la plupart du temps, `eax` représente la valeur de `i`, sauf à la toute fin où `eax` prend la valeur de `s`).
L'initialisation (`int i = 0`) est l'instruction `mov eax, dword ptr [rbp - 12]`. En particulier, le saut conditionnel ne se produit que si l'instruction `cmp eax, dword ptr [rbp - 4]` est vérifiée, *i.e.* lorsque `eax` $\geq$ `dword ptr [rbp - 4]`. Autrement dit, on sort de la boucle dès que `i` $\geq$ `count`.
Enfin, l'incrémentation se produit avec l'instruction `add eax, 1`.

b. L'appel à la fonction `sum` se fait avec l'instruction `call _sum` .

c. L'instruction `call` va chercher l'argument de la fonction dans le registre `edi`.

d. L'instruction `ret` va chercher la valeur de retour dans le registre `eax`.

4.4 Exercice 4

Premier code : `rax` vaut 47 à la fin et est équivalent au code C suivant :

```
int i = 12;
i = i + i;
i = i + i;
i--;
```

Deuxième code : `rax` vaut 123 à la fin et est équivalent au code suivant :

```
int a = 12;
for(int i = 0; i<10; i++){
    int a += 10;
}
```

Troisième code : `rax` vaut 143 à la fin et est équivalent au code suivant :

```
int a = 12;
int b;
for(b=13; b>0; b--){
    a += 10;
}
```

4.5 Exercice 5

Premier code : `rax` vaut 15 au niveau de `nop`. Deuxième code : `rax` vaut 12. Troisième code : `rax` vaut `rbx`.

4.6 Exercice 6

a. On obtient le code assembleur suivante (avec `objdump`) :

```
0000000000000000 <_main>:
    0: 55                                  push    rbp
    1: 48 89 e5                            mov     rbp, rsp
    4: c7 45 fc 00 00 00 00                mov     dword ptr [rbp - 4], 0
    b: c7 45 f4 00 00 00 00                mov     dword ptr [rbp - 12], 0
   12: c7 45 f0 00 00 00 00                mov     dword ptr [rbp - 16], 0
   19: 83 7d f0 64                         cmp     dword ptr [rbp - 16], 100
   1d: 0f 8d 17 00 00 00                   jge     0x3a <_main+0x3a>
   23: 8b 45 f0                            mov     eax, dword ptr [rbp - 16]
   26: 03 45 f4                            add     eax, dword ptr [rbp - 12]
   29: 89 45 f4                            mov     dword ptr [rbp - 12], eax
   2c: 8b 45 f0                            mov     eax, dword ptr [rbp - 16]
   2f: 83 c0 01                            add     eax, 1
   32: 89 45 f0                            mov     dword ptr [rbp - 16], eax
   35: e9 df ff ff ff                      jmp     0x19 <_main+0x19>
   3a: 8b 45 fc                            mov     eax, dword ptr [rbp - 4]
   3d: 5d                                  pop     rbp
   3e: c3                                  ret
```

b. Il y a donc 17 instructions et le programme occupe `0x3e`+1 soit 63 octets en mémoire.

c. Il suffit de lire :

```
1: 48 89 e5  mov rbp, rsp
```

d. Les instructions qui occupent le plus de mémoire sont les instructions `mov` correspondant aux initialisations des variables `i` (2 fois) et `j` à 0. Comme ce sont des `int`, on réserve immédiatement 4 octets (32 bits) : c'est de l'adressage immédiat.

4.7 Exercice 7

a. Le stockage en RAM des valeurs de `i` et `s` initialisées à 0 est enlevé. À la place, on trouve la commande `test edi, edi` qui agit comme un `bitwise and` ("et" bit à bit). La valeur de véracité de `edi` $\wedge$ `edi` est toujours *vrai* (donc le test vaut 1), sauf quand `edi` $= 0$, auquel cas la valeur est *faux* (et le test vaut 0). Cela permet de mettre à jour tous les flags concernés (par exemple ZF vaudra 0 si `edi` est non nul), mais également de stocker les valeurs de `i` et `s` dans des registres.

b. Dans le `main`, on enlève des opérations inutiles telles que `sub rsp, 16` et `add rsp, 16` et l'assignation `int count = 100000000` est transcrite en une seule commande. Suivant les architectures, l'appel de la fonction `sum` est oublié aussi, puisque le résultat est inutilisé, de même que l'instanciation de `count`.

c. On modifie le `main` :

```c
int main()
{
    int a = 0;
    int count = 100000000;
    a += sum(count);
}
```

d. Un changement est l'utilisation de la commande `lea` qui fait presque la même opération que `mov` : `lea dest source` met l'adresse de `source` dans `dest`. Ainsi, `lea dest [source]` est équivalent à `mov dest source`, et c'est ce qui est utilisé dans le code optimisé. `lea` est pratique car elle permet de faire directement les calculs à la volée dans l'argument, comme par exemple `lea ecx, [edx + eax*4]`.
On note aussi l'utilisation du registre *destination index* `rdi` dans le code optimisé, qui n'apparait pas dans le code non optimisé.
Concernant la boucle, elle est donnée par le code suivant :

```asm
lea eax, [rdi - 1]
lea ecx, [rdi - 2]
imul rcx, rax
shr rcx
lea eax, [rcx + rdi]
add eax, -1
pop rbp
ret
```

Le compilateur effectue directement l'opération [15] :

$$\texttt{rcx} = \frac{(\texttt{rdi}-1)(\texttt{rdi}-2)}{2} + (\texttt{rdi}-1) = \sum_{i=1}^{\texttt{rdi-1}} i$$

Enfin, grâce au test sur `edi`, on saute directement à la fin de la fonction si `edi` $= 0$, ce qui n'est pas fait dans le code non optimisé, puisqu'on utilise un `cmp` peu importe la valeur de `edi`.

e. La seule différence obtenue [16] est la suppression de l'appel à la fonction `sum` et l'instanciation de `count` par `edi`.

f. Comme `count` n'est plus instancié dans le code assembleur avec un niveau 2 d'optimisation, on n'observe aucun changement (`essai avec count=5`).

g. L'assembleur semble utiliser des méthodes complexes similaires au cas de la somme des entiers. On sait par exemple que la somme des carrés peut s'écrire de la manière suivante :

$$\sum_{i=1}^{n-1} i^2 = \frac{n(n-1)(2n-1)}{6}$$

15. La division par 2 se fait avec l'opération *shift right*, `shr`, parfois notée `>>` dans d'autres langages.
16. Tous les codes sont sur GitHub.

De même, la somme des cubes s'écrit :

$$\sum_{i=1}^{n-1} i^2 = (\frac{(n-1)(n-2)}{2})^2$$

On reconnait des opérations similaires au cas de la somme des entiers, néanmoins les expressions ci-dessus ne ressortent pas explicitement.

4.8 Exercice 8

a. On souhaite faire l'opération `rax` $\leftarrow \min(4$ `rax` $+ 5, 8$ `rbx` $+ 7)$:

```
; rax <- 4*rax+5
mul rax, 4
add rax, 5

; rbx <- 8*rbx+7
mul rbx, 8
add rbx, 7

mov dword ptr [rbp - 4], rax
mov dword ptr [rbp - 8], rbx
mov rax, dword ptr [rbp - 4]

cmp rax, dword ptr [rbp - 8] ; performs 4*rax+5 - 8*rbx+7
jge LBB0_1 ; jump if >= 0

jmp LBB0_2

LBB0_1 ; case 4*rax+5 >= 8*rbx+7
mov rax, dword ptr [rbp - 8] ; rax <- 8*rbx+7

LBB0_2 ; case 4*rax+5 < 8*rbx+7 but rax already holds 4*rax+5
skip ahead
```

b. On souhaite faire l'opération suivante :

```
while(rax < rcx){rax += rbx;}
```

On écrit le code assembleur suivant :

```
LBB0_1
cmp rax, rcx
jge LBB0_2 ; rax >= rcx

add rax, rbx ; rax <- rax + rbx
jmp LBB0_1

LBB0_2
skip ahead
```

c. On veut effectuer l'opération suivante :

```
if(rax % 4 == 0){rcx = 1;}
```

On écrit le code assembleur suivant :

```
mov edi, 4
div edi
cmp rdx, 0
jne LBB0_1
mov rcx, 1

LBB0_1
skip ahead
```

Il est nécessaire de préciser comment fonctionne `div`. Lorsqu'on effectue l'opération `div` a où a est un registre, on divise l'entier représenté par le mot écrit sur le segment `rdx:rax` par la valeur dans a. Le quotient est stocké dans `rax` et le reste dans `rdx`, d'où la comparaison directe sur `rdx`.

5 Chapitre 5

5.1 Exercice 1

a. Le premier saut conditionnel (l.4-5 en assembleur) correspond au *if* (l.5 en C). Le deuxième saut conditionnel (l.8-9 en assembleur) correspond à la comparaison `i < n` dans la boucle *for* (l.4 en C).

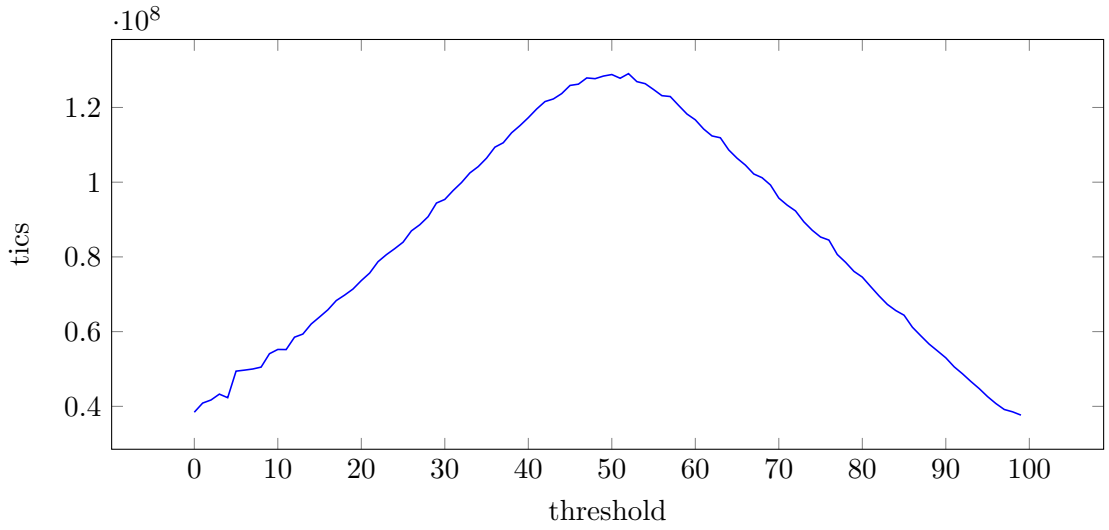
b.

(a) Les erreurs de prédiction se font aux indices 3, 4, 8, 9, 11, 13, 15, 16, 17, 18, 19

(b) Les erreurs de prédiction se font aux indices 0, 1, 3, 4, 8, 9, 11, 13, 15, 16, 17, 18, 19

5.2 Exercice 2

a. On obtient les résultats suivants :



(a) Oui !

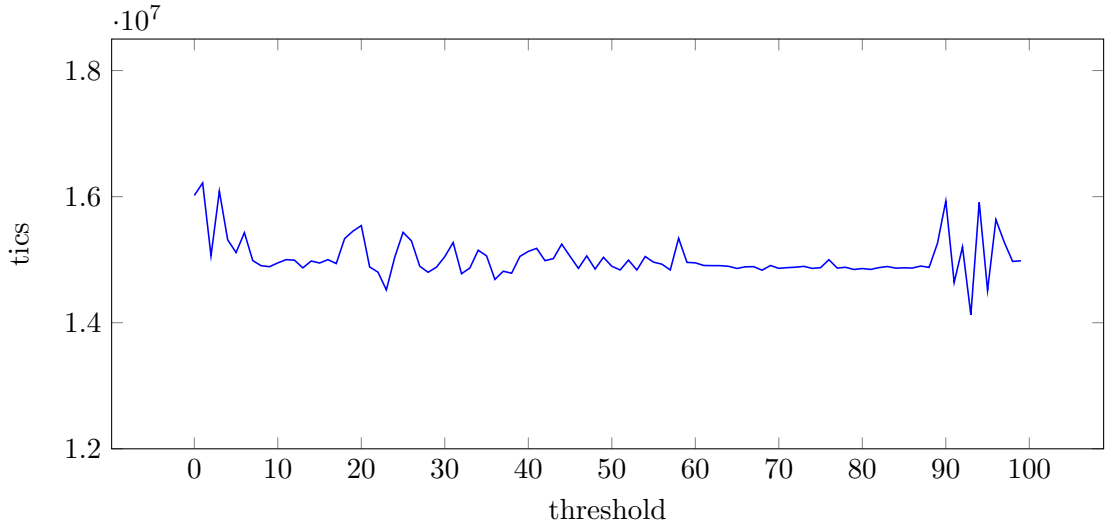
(b) Le facteur multiplicatif est d'environ 3.

(c) Le fait que le pire cas se trouve pour des valeurs de seuil autour de 50 est dû au fait que pour ces valeurs, le prédicteur a une chance sur deux de se tromper. Le nombre de pénalité est donc plus élevé et l'exécution est plus longue.

b. On obtient les résultats suivants :

Le compilateur s'est débarassé [17] du saut conditionnel (comparaison avec le seuil). Ainsi, on réduit énormément la bulle par rapport au code non optimisé. On obtient donc

17. En utilisant un `setl`



une exécution quasi-constante.

5.3 Exercice 3

a. On peut réaliser deux tests successifs avec deux sauts conditionnels. Si le premier n'est pas vérifié, on saute directement à la suite. Sinon, on effectue le second test.

b. C'est concluant :

```
count1: 152644587 tics
count2: 183140942 tics
```

c. Il y a un facteur multiplicatif de 1.2 (soit 20% de différence).

d. Dans le premier cas, le prédicteur a environ une chance sur quatre de se tromper. Puis, il a environ une chance sur trois de se tromper au deuxième test, soit au total une chance sur douze. Dans le deuxième cas on a environ une chance sur deux de se tromper au premier test, puis encore une chance sur deux au deuxième test, soit environ une chance sur quatre. On se trompe donc qualitativement trois fois plus souvent dans le deuxième cas.

e. Avec `O1` ou `O2` on obtient presque les mêmes temps pour les deux cas.

5.4 Exercice 4

On donne ci-dessous les deux codes obtenus pour les niveaux O0, O1 et O2 :

Listing 1 – Optimisation O0

```
main:
    push rbp
    mov rbp, rsp
    sub rsp, 32
    mov DWORD PTR -16[rbp], 0
    mov DWORD PTR -12[rbp], 0
    mov DWORD PTR -8[rbp], 0
    mov DWORD PTR -4[rbp], 0
    mov DWORD PTR -20[rbp], 1
    jmp .L2
.L3:
    mov eax, DWORD PTR -20[rbp]
    imul eax, eax
    imul eax, DWORD PTR -20[rbp]
    imul eax, DWORD PTR -20[rbp]
    add DWORD PTR -16[rbp], eax
    mov eax, DWORD PTR -16[rbp]
    imul eax, eax
    imul eax, DWORD PTR -16[rbp]
    imul eax, DWORD PTR -16[rbp]
    add DWORD PTR -12[rbp], eax
    mov eax, DWORD PTR -16[rbp]
    imul eax, eax
    imul eax, DWORD PTR -12[rbp]
    imul eax, DWORD PTR -12[rbp]
    add DWORD PTR -8[rbp], eax
    mov eax, DWORD PTR -16[rbp]
    cdq
    idiv DWORD PTR -12[rbp]
    add DWORD PTR -4[rbp], eax
    mov eax, DWORD PTR -8[rbp]
    add DWORD PTR -4[rbp], eax
    add DWORD PTR -20[rbp], 1
.L2:
    cmp DWORD PTR -20[rbp], 9
    jle .L3
    mov eax, DWORD PTR -4[rbp]
    mov esi, eax
    lea rdi, .LC0[rip]
    mov eax, 0
    call printf@PLT
    mov eax, 0
    leave
    ret
```

Listing 2 – Optimisation O1

```
main:
    sub rsp, 8
    mov r9d, 0
    mov r8d, 0
    mov esi, 0
    mov ecx, 0
    mov edi, 1
.L2:
    mov eax, edi
    imul eax, edi
    imul eax, eax
    add ecx, eax
    mov eax, ecx
    imul eax, ecx
    mov edx, eax
    imul edx, ecx
    imul edx, ecx
    add esi, edx
    imul eax, esi
    imul eax, esi
    add r8d, eax
    mov eax, ecx
    cdq
    idiv esi
    add eax, r9d
    lea r9d, [r8+rax]
    add edi, 1
    cmp edi, 10
    jne .L2
    mov edx, r9d
    lea rsi, .LC0[rip]
    mov edi, 1
    mov eax, 0
    call __printf_chk@PLT
    mov eax, 0
    add rsp, 8
    ret
```

Listing 3 – Optimisation O2

```
main:
    sub rsp, 8
    xor r9d, r9d
    xor r8d, r8d
    xor esi, esi
    xor ecx, ecx
    mov edi, 1
.L2:
    mov eax, edi
    imul eax, edi
    add edi, 1
    imul eax, eax
    add ecx, eax
    mov eax, ecx
    imul eax, ecx
    mov edx, eax
    imul edx, ecx
    imul edx, ecx
    add esi, edx
    imul eax, esi
    imul eax, esi
    add r8d, eax
    mov eax, ecx
    cdq
    idiv esi
    add eax, r9d
    lea r9d, [r8+rax]
    cmp edi, 10
    jne .L2
    mov edx, r9d
    lea rsi, .LC0[rip]
    mov edi, 1
    xor eax, eax
    call __printf_chk@PLT
    xor eax, eax
    add rsp, 8
    ret
```

- Première différence entre O0 et O1 : le code assembleur utilise plus de registres et ne fait pas d'accès en RAM.

- Deuxième différence : au niveau O2, on utilise plutôt des instructions comme `xor rax, rax` pour remplacer les instructions `mov rax, 0` qui sont plus rapides et prennent moins de place en mémoire.

- Troisième différence : le niveau O2 déplace des instructions afin de réduire les bulles. Par exemple, la ligne 11 du code O2 correspond à la ligne 27 du code O1.

Type	Superscalar, Superpipeline
OoOE	Yes
Speculative	Yes
Reg Renaming	Yes
Stages	14-19

FIGURE 3 – Description du pipeline de l'architecture skylake.

OoOE signifie *Out-of-Order Execution* : les instructions ne sont pas forcément exécutées dans l'ordre dans lequel elles sont écrites dans le programme. *Speculative*, pour *Speculative Execution*, désigne le lancement anticipé d'instructions. *Reg Renaming* signifie *Register Renaming* : les registres sont renommés afin de réordonner les dépendances entre les instructions. Tous ces termes sont des aspects d'une architecture superscalaire basée sur un fonctionnement avec plusieurs *pipelines* en parallèle. Le terme *superpipeline* désigne quant à lui une amélioration du modèle de *pipeline* qui traite les instructions sur sept niveaux [18] ou plus (certaines architectures utilisent des *pipelines* à 10 voire 20 ou 21 niveaux) et non quatre comme décrit dans le cours pour un pipeline classique.

6 Chapitre 7

6.1 Exercice 1

a. On donne le code assembleur suivant :

```
int sum(int n)
{
    cur = 0,1,2,3;
    w = 4,4,4,4;
    s = 0,0,0,0;
    for(int i = 0; i < n/4; i++)
    {
        s = s + cur;
        cur = cur + w;
    }
    return sum_comp(s);
}
```

b. Si n n'est pas multiple de 4, on ajoute une instruction qui calcule la somme des `n % 4` dernières valeurs.

c. On commence par tester si `edi = 0` et si c'est le cas, on sort directement. Ensuite, on associe `(n-4)/4+1` à `eax` et `n-1` à `ecx`. On compare `ecx` à 8, qui est la valeur de seuil. Autrement dit, on traitera les entrées modulo 4 et décalées de 4, car le compilateur considère que cette valeur de seuil est la plus rentable. Il restera donc au plus 7 entrées

18. Pré-chargement, premier décodage, deuxième décodage, première génération d'adresse, deuxième génération d'adresse de retour, exécution de l'instruction, écriture des données via les adresses prégénérées.

à traiter après la boucle (ce sont les données non vectorisables).
La vectorisation se fait dans les lignes qui suivent : `xmm0` représente la variable `s` et vaut `0,0,0,0` initialement. `xmm2` représente la variable `w` et vaut `4,4,4,4`, et `xmm1` représente la variable `cur` et vaut `0,1,2,3` initialement.
Les opérations effectuées ensuite sont les suivantes :

- `add ecx, 1` $\longrightarrow$ `i += 1`
- `xmm0 += xmm1` $\longrightarrow$ `s += cur`
- `xmm1 += xmm2` $\longrightarrow$ `cur += w`

Les opérations correspondant à `sum_comp` sont coûteuses et correspondent aux commandes :

```
movdqa xmm1, xmm0
cmp edi, edx
psrldq xmm1, 8
paddd xmm0, xmm1
movdqa xmm1, xmm0
psrldq xmm1, 4
paddd xmm0, xmm1
movd eax, xmm0
```

Enfin, les potentiels restes sont traités par la suite (de $\frac{n-4}{4} + 1$ à n) et la somme est effectuée en même temps.

6.2 Exercice 2

(i) C'est vectorisable et `gcc` y parvient.

(ii) C'est vectorisable et `gcc` n'y parvient pas.

(iii) C'est vectorisable et `gcc` n'y parvient pas.

(iv) C'est vectorisable et `gcc` y parvient.

Le compilateur n'arrive pas à vectoriser les codes dans lesquels il y un décalage à gérer, car la structure de la boucle ne répond pas à un certain motif prédéfini. Ainsi, si intuitivement un cerveau humain peut facilement modifier le code dans le but de le rendre vectorisable, le compilateur n'en est pas (encore) capable.

6.3 Exercice 3

a. Le code a été vectorisé.

b.

`pcmpgtd` : effectue une comparaison signée en SIMD avec la condition *greater than* pour *double word*. Assigne au premier opérande un masque binaire correspondant au résultat de la comparaison composante par composante.

`pand` : opérateur et logique pour des registres XMM (128 bits).

`pandn` : opérateur de non et logique pour des registres XMM (128 bits). `pandn` p $q \equiv p \leftarrow p \wedge \neg q$.

On donne ci-dessous le code permettant d'avoir le minimum terme à terme de deux vecteurs :

```
1 movdqa xmm2, xmm1
2 pcmpgtd xmm2, xmm0
3 pand xmm1, xmm2
4 pandn xmm0, xmm2
5 por xmm0, xmm1
6 ;xmm0 contient le minimum composante par composante de xmm0 et xmm1
```

c. On donne ci-dessous le code produit par le compilateur :

```
1  _min:
2  ## %bb.0:
3      push rbp
4      mov rbp, rsp
5      mov eax, dword ptr [rdi]
6      cmp esi, 2
7      jl LBB0_11
8  ## %bb.1:
9      mov ecx, esi
10     lea r8, [rcx - 1]
11     mov edx, 1
12     cmp r8, 8
13     jb LBB0_10
14 ## %bb.2:
15     mov rdx, r8
16     and rdx, -8
17     movd xmm0, eax
18     pshufd xmm0, xmm0, 0
19     lea rax, [rdx - 8]
20     mov r9, rax
21     shr r9, 3
22     add r9, 1
23     test rax, rax
24     je LBB0_3
25 ## %bb.4:
26     mov rax, r9
27     and rax, -2
28     neg rax
29     mov esi, 1
30     movdqa xmm1, xmm0
31 LBB0_5:
32     movdqu xmm2, xmmword ptr [rdi + 4*rsi]
33     pminsd xmm2, xmm0
34     movdqu xmm3, xmmword ptr [rdi + 4*rsi + 16]
35     pminsd xmm3, xmm1
36     movdqu xmm0, xmmword ptr [rdi + 4*rsi + 32]
37     pminsd xmm0, xmm2
38     movdqu xmm1, xmmword ptr [rdi + 4*rsi + 48]
39     pminsd xmm1, xmm3
40     add rsi, 16
41     add rax, 2
42     jne LBB0_5
```

```
43 ## %bb.6:
44     test r9b, 1
45     je LBB0_8
46 LBB0_7:
47     movdqu xmm2, xmmword ptr [rdi + 4*rsi]
48     pminsd xmm0, xmm2
49     movdqu xmm2, xmmword ptr [rdi + 4*rsi + 16]
50     pminsd xmm1, xmm2
51 LBB0_8:
52     pminsd xmm0, xmm1
53     pshufd xmm1, xmm0, 238
54     pminsd xmm1, xmm0
55     pshufd xmm0, xmm1, 85
56     pminsd xmm0, xmm1
57     movd eax, xmm0
58     cmp r8, rdx
59     je LBB0_11
60 ## %bb.9:
61     or rdx, 1
62 LBB0_10:
63     mov esi, dword ptr [rdi + 4*rdx]
64     cmp esi, eax
65     cmovl eax, esi
66     add rdx, 1
67     cmp rcx, rdx
68     jne LBB0_10
69 LBB0_11:
70     pop rbp
71     ret
72 LBB0_3:
73     mov esi, 1
74     movdqa xmm1, xmm0
75     test r9b, 1
76     jne LBB0_7
77     jmp LBB0_8
```

Tout d'abord, on observe que la vectorisation se fait sur des vecteurs de taille 8 (L.19-22), et donc les données sont espacées en mémoire de 2 octets (16 bits [19]). La commande utilisée pour trouver le min est `pminsd`.

19. Les registres xmm peuvent stocker 128 bits.

6.4 Exercice 4

a. On dispose de 1024[20] blocs et $\frac{64*1024}{64} = 1024$ sous-caches contenant chacun un bloc. On sait également que les entiers sont codés sur 4 octets, un bloc peut donc contenir 16 entiers. On peut donc stocker en cache $1024 \times 16 = 16384$ entiers simultanément. De plus, on accède aux données dans le tableau (ou dans chaque sous-tableau) de manière consécutive.

Le tableau est divisé en 3 sous-tableaux de taille `LARGEUR*HAUTEUR`. On stocke les entiers par blocs de 16, donc pour un sous-tableau, on fera donc $\frac{\texttt{LARGEUR} \times \texttt{HAUTEUR}}{16}$ accès hors-caches.

(a) Pour un total de $1600 \times 1000 = 16 * 100000$ pixels, on fera donc 100 000 accès hors-chache par sous-tableau, donc 300 000 accès hors-cache au total.

(b) Pour un total de $1024 \times 1024 = 16 \times 65536$ pixels, on fera donc 65536 accès hors-chache par sous-tableau, donc 196 608 accès hors-cache au total.

b. Pour une associativité 8, on dispose de $\frac{64 \times 1024}{8 \times 64} = 128$ sous-caches, contenant chacun $\frac{1024}{128} = 8$ blocs. À chaque accès hors-cache, on charge donc un total de $8 \times 16 = 128$ entiers en cache simultanément. Si `LARGEUR` = `HAUTEUR` = 1024, on fera $\frac{1024 \times 1024}{128} = 8192$ accès hors-cache.

20. $\frac{64 \text{ ko}}{64 \text{ o par ligne}}$