Laboratoire 3 : Optimisation des programmes pour les DSP TMS320C6x

Le but de ce laboratoire est de se familiariser avec l'optimisation en assembleur sur la carte TMS320C6713. Il fait suite au cours et permet de rentrer concrètement dans le sujet. Code Composer Studio a été installé sur http://processors.wiki.ti.com/index.php/Download_CCS et il s'agit de la version 5.2.1.00018.
Nous verrons différentes méthodes d'optimisation en 1ère partie suivi de l'optimisation Pipeline en 2ème partie.

I. Méthodes d'optimisation

Un programme qui marche n'est pas forcément le résultat final, il est souvent nécessaire d'optimiser son programme pour qu'il soit le plus performant possible. C'est le but de cette partie, on verra au total 4 programmes différents :

1. Non optimisé
2. Optimisation par parrallélisme ||
3. Optimisation des NOPs
4. Optimisation World - Wide

Les temps de process seront ensuite analysés.

I.1 Non optimisé : $sum1.asm$

Il faut commencer par créer le « main », j'ai déclaré 2 tableaux *short* statiques x et y respectivement de taille 40, et ceci de façon globale. Ces tableaux sont initilialisés dans la boucle principale du programme.

J'ai décidé pour la suite de ne faire rentrer qu'un seul paramètre dans les fonctions, *short* n qui correspond à la taille des tableaux.

main.c | sum1.asm | sum2.asm | sum3.asm | sum4.asm

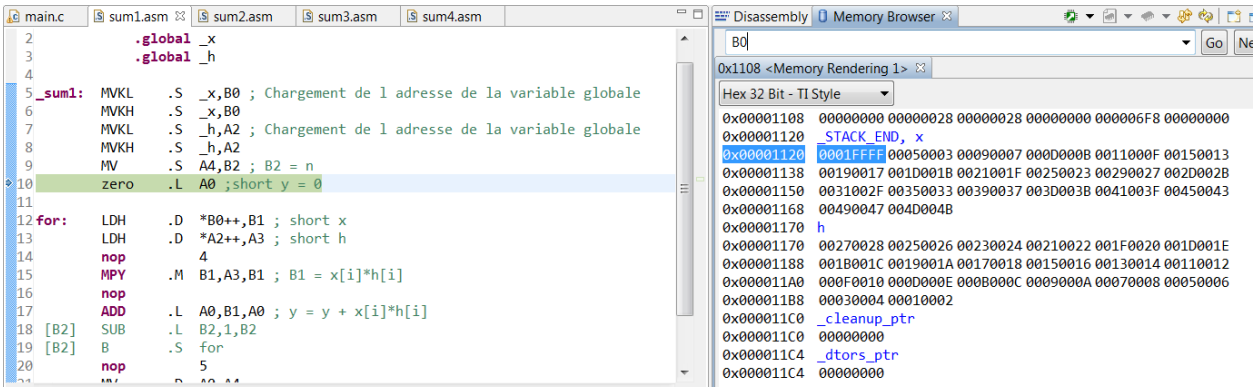
```c
/*
 * main.c
 */
short sum1(short);
short sum2(short);
short sum3(short);
short sum4(short);

short x[40], h[40];

void main(void) {

    unsigned short i;
    short y = 0, n = 40;

    for(i=0; i<n; i++)
    {
        x[i] = 2*i-1;
        h[i] = 40-i;
    }

    y = sum1(n);
    y = sum2(n);
    y = sum3(n);
    y = sum4(n);

}
```

Pour accéder aux contenus des tableaux, j'utilise donc les variables globales. Là où cette méthode rajoute plus de cycles (comme nous le verrons par la suite), elle permet d'économiser du temps dans la transistion entre le *C* et l'assembleur. En effet car on ne fait rentrer qu'un seul paramètre n dans les fonctions.

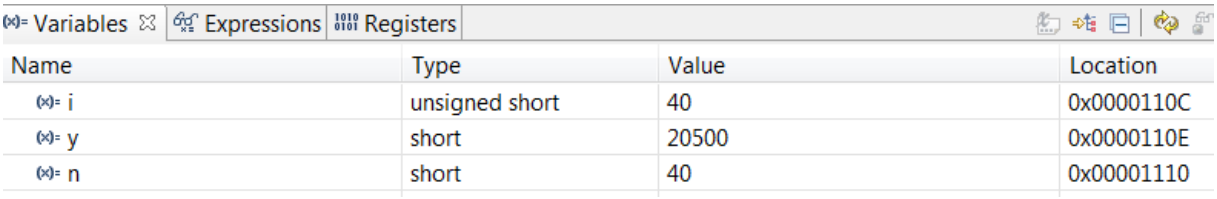
main.c | sum1.asm | sum2.asm | sum3.asm | sum4.asm

```
            .global _sum1
            .global _x
            .global _h

_sum1:  MVKL    .S  _x,B0 ; Chargement de l adresse de la variable globale
        MVKH    .S  _x,B0
        MVKL    .S  _h,A2 ; Chargement de l adresse de la variable globale
        MVKH    .S  _h,A2
        MV      .S  A4,B2 ; B2 = n
        zero    .L  A0 ;short y = 0

for:    LDH     .D  *B0++,B1 ; short x
        LDH     .D  *A2++,A3 ; short h
        nop         4
        MPY     .M  B1,A3,B1 ; B1 = x[i]*h[i]
        nop
        ADD     .L  A0,B1,A0 ; y = y + x[i]*h[i]
[B2]    SUB     .L  B2,1,B2
[B2]    B       .S  for
        nop         5
        MV      .D  A0,A4
        B       .S2 B3
        nop         5
```

En mode debug, j'ai commencé par vérifier les variables globales (les tableaux h et x), ainsi que le paramètre $n = 40$ (dans le registre A4).



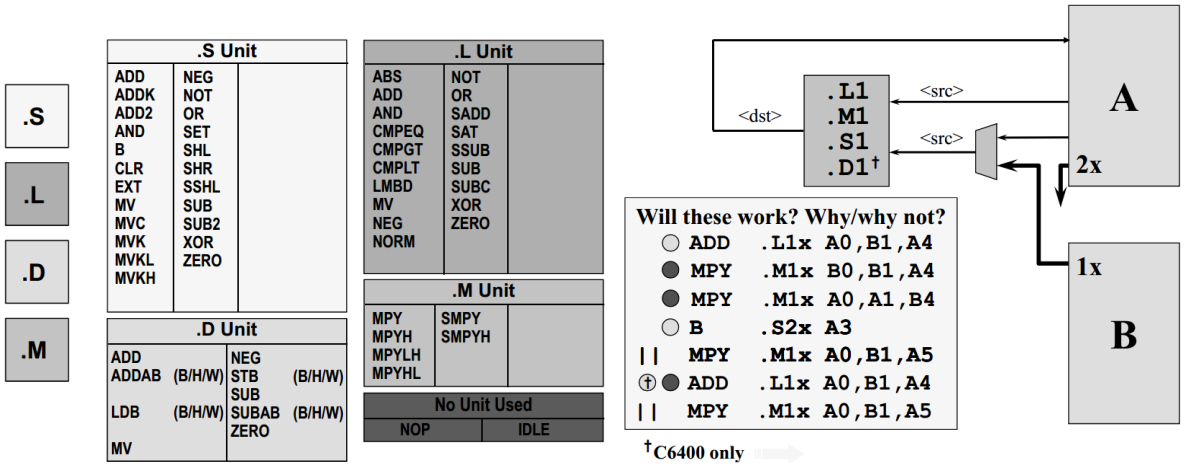
J'ai ensuite executé le programme pour vérifier la valeur de y qui doit être égal à 20500



Name	Type	Value	Location
i	unsigned short	40	0x0000110C
y	short	20500	0x0000110E
n	short	40	0x00001110

I.2 Optimisation par parrallélisme : $sum2.asm$

La parrallélisation permet de faire travailler plusieurs unités de calcul en même temps. Lorsque l'on veut parralléliser les tâches, il faut bien définir les unités ainsi que leur numéro. Rajouter un « x » à la fin des unités indique que l'on fait un cheminement croisé, les opérandes de registre passent d'un côté à un autre.



J'en ai profité pour mettre en parrallèle les *MVK* nécessaires pour le chargement des variables globales.

main.c | sum1.asm | sum2.asm | sum3.asm | sum4.asm

```
1              .global _sum2
2              .global _x
3              .global _h
4
5  _sum2:  MVKL    .S2     _x,B0 ; Chargement de l adresse de la variable globale
6      ||  MVKL    .S1     _h,A2 ; Chargement de l adresse de la variable globale
7          MVKH    .S2     _x,B0
8      ||  MVKH    .S1     _h,A2
9          MV      .S2x    A4,B2 ; B2 = n
10         zero    .L1     A0 ;short y = 0
11
12 for:    LDH     .D2     *B0++,B1 ; short x
13     ||  LDH     .D1     *A2++,A3 ; short h
14         nop             4
15         MPY     .M2     B1,A3,B1 ; B1 = x[i]*h[i]
16         nop
17         ADD     .L1     A0,B1,A0 ; y = y + x[i]*h[i]
18 [B2]    SUB     .L2     B2,1,B2
19 [B2]    B       .S1     for
20         nop             5
21         MV      .D1     A0,A4
22         B       .S2     B3
23         nop             5
```

La procédure de debug est exactement la même que précedemment,

Variables | Expressions | Registers

Name	Type	Value	Location
i	unsigned short	40	0x0000110C
y	short	20500	0x0000110E
n	short	40	0x00001110

I.3 Optimisation des NOPs : $sum3.asm$

Pour cette partie, il faut bien analysé les temps de calcul de chaque instruction :

- Les opérations de calcul simples *zero, ADD* ou encore *SUB* prennent 1 cycle
- Les transferts simples *MV, MVKL* ou *MVKH* prennent 1 cycle
- Les opérations de multiplication terme à terme *MPY* prennent 2 cycles
- Les opérations de chargement *LD* prennent 5 cycles d'horloges du CPU
- Les branch *B* sont les plus coûteux et prennent 6 cyles

Le but est de réduire au minimum l'utilisation des NOP. On remarque par exemple que le temps que le branching se fasse, on a le temps d'executer des opérations.

main.c | sum1.asm | sum2.asm | sum3.asm | sum4.asm

```
1             .global _sum3
2             .global _x
3             .global _h
4
5  _sum3:  MVKL    .S2     _x,B0 ; Chargement de l adresse de la variable globale
6      ||  MVKL    .S1     _h,A2 ; Chargement de l adresse de la variable globale
7          MVKH    .S2     _x,B0
8      ||  MVKH    .S1     _h,A2
9          MV      .S2x    A4,B2 ; B2 = n
0          zero    .L1     A0 ;short y = 0
1
2  for:    LDH     .D2     *B0++,B1 ; short x
3      ||  LDH     .D1     *A2++,A3 ; short h
4   [B2]   SUB     .L2     B2,1,B2
5   [B2]   B       .S1     for
6          nop             2
7          MPY     .M2     B1,A3,B1 ; B1 = x[i]*h[i]
8          nop
9          ADD     .L1     A0,B1,A0 ; y = y + x[i]*h[i]
0          B       .S2     B3
1          MV      .D1     A0,A4
2          nop             4
```

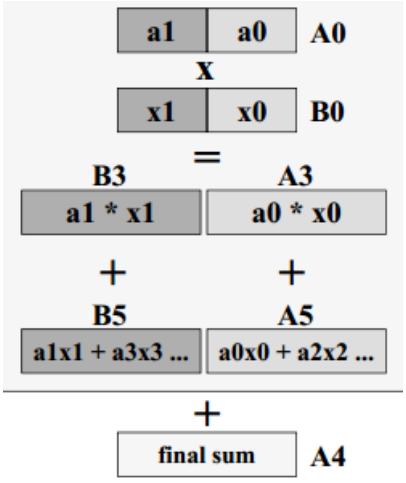
Avec la même procédure de debug,

Variables | Expressions | Registers

Name	Type	Value	Location
i	unsigned short	40	0x0000110C
y	short	20500	0x0000110E
n	short	40	0x00001110

I.4 Optimisation World - Wide : $sum4.asm$

Dans cette optimisation, il est question de réduire le temps d'éxecution de la boucle principale. Pour cela on va utiliser des *LDW* au lieu de *LDH* qui permet de déplacer un mot de 32 bits en une seule fois. Voici l'idée générale de l'algorithme :



main.c | sum1.asm | sum2.asm | sum3.asm | sum4.asm

```
1              .global _sum4
2              .global _x
3              .global _h
4  _sum4:
5           MVKL     .S2     _x,B0 ; Chargement de l adresse de la variable globale
6       ||  MVKL     .S1     _h,A2 ; Chargement de l adresse de la variable globale
7           MVKH     .S2     _x,B0
8       ||  MVKH     .S1     _h,A2
9           MV       .S2x    A4,B2 ; B2 = n
10          zero     .L1     A0
11      ||  zero     .L2     B7
12
13 for:     LDW      .D2     *B0++,B1 ; double { h[i+1], h[i] }
14      ||  LDW      .D1     *A2++,A3 ; double { x[i+1], x[i] }
15  [B2]    SUB      .L2     B2,2,B2
16  [B2]    B        .S1     for
17          nop              2
18          MPY      .M2x    B1,A3,B5 ; B5 = x[i]*h[i]
19      ||  MPYH     .M1x    B1,A3,A1 ; A1 = x[i+1]*h[i+1]
20          nop
21          ADD      .L2x    A1,B7,B7 ; B7 = B7 + x[i+1]*h[i+1]
22      ||  ADD      .L1x    B5,A0,A0 ; A0 = A0 + x[i]*h[i]
23
24          B        .S2     B3
25          ADD      .L      B7,A0,A0
26          MV       .D1     A0,A4
27          nop              3
```

Au début j'ai déroulé la boucle pour économiser des ressources au niveau de la gestion de la boucle. Mais il s'est avéré que le résultat n'est pas si significatif que ça par apport à la taille du programme (qui commence à deveir très gros). De plus ce déroulement n'est vraiment pas pratique si l'on veut changer la taille de la boucle (il faut recopier le code à chaque fois..). Au final même en enlevant le branch *B*, les *LD* au début prennent du temps à se finir donc on ne gagne pas là-dessus. On gagne au total 2 cyles d'instructions par rapport à une optimisation world wide « normale » ce qui ne vaut vraiment pas le coup.

I.5 Mesure de performances

J'ai mesuré les performances en mode Debug et Release. Le premier taleau correspond à la mesure de performance lorsque l'on lance le programme pour la première fois :

Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
main()	1	-	-	1419.00	1419	-	-	3224.00	3224
sum1	1	684	684	684.00	684	684	684	684.00	684
sum2	1	621	621	621.00	621	621	621	621.00	621
sum3	1	330	330	330.00	330	330	330	330.00	330
sum4	1	170	170	170.00	170	170	170	170.00	170

Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
main()	1	1459	1459	1459.00	1459	3223	3223	3223.00	3223
sum1	1	653	653	653.00	653	653	653	653.00	653
sum2	1	611	611	611.00	611	611	611	611.00	611
sum3	1	330	330	330.00	330	330	330	330.00	330
sum4	1	170	170	170.00	170	170	170	170.00	170

Ainsi l'optimisation World Wide permet de diviser par plus de 3 le nombre de cycles !

Mode Release

Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
main()	1	-	-	170.00	170	-	-	1986.00	1986
sum1	1	690	690	690.00	690	690	690	690.00	690
sum2	1	616	616	616.00	616	616	616	616.00	616
sum3	1	340	340	340.00	340	340	340	340.00	340
sum4	1	170	170	170.00	170	170	170	170.00	170

Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
main()	1	70	70	70.00	70	1834	1834	1834.00	1834
sum1	1	653	653	653.00	653	653	653	653.00	653
sum2	1	611	611	611.00	611	611	611	611.00	611
sum3	1	330	330	330.00	330	330	330	330.00	330
sum4	1	170	170	170.00	170	170	170	170.00	170

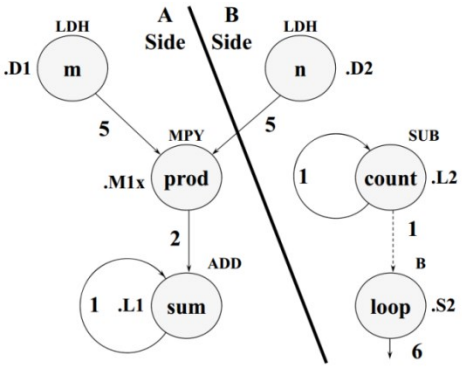
Le mode Release est surtout utile pour le language C, en effet en assembleur c'est en quelque sorte à nous de faire l'optimisation.

II. Pipeline

L'optimisation par Pipeline est une approche qui permet de maximiser le travail des unités fonctionnels, en implémentant une bonne parrallélisation des tâches et en remplissant les délais. Il y a plusieurs étapes pour faire du pipeline :

- Ecrire le programme en C
- Ecrire le programme en assembleur linéaire
- Créer le graphe de dépendance
- Allouer de manière optimale les registres
- Ecrire la table de temps
- Programmer

Après avoir créer le programme en C et en assembleur, il est nécessaire de créer ce que l'on appelle un « graphe de dépendance ». J'ai décidé d'utiliser des *LDW* au lieu de *LDH* pour diminuer le temps de cycle. **Si l'on se concentre sur la boucle**, le graphe et l'allocation des registres sont de la forme suivante :



Register File A	#	#	Register File B
&x	A0	B0	&h
x	A1	B1	h
n	A2	B2	
	A3	B3	
sum[i] / y	A4	B4	sum[i+1]
prod[i]	A5	B5	prod[i+1]
	...	...	

On peut découper le programme en 3 séquences le prologue (initialisation nécessaire pour que la boucle fonctionne), la boucle principale et l'épilogue (qui évite de faire des instructions inutiles en trop comme des *LDH*).

La taille du prologue est determiné par la plus longue séquence d'instructions (ici *LDW, MPY, ADD* pour 7 cycles). L'épilogue revient à terminer les calculs sans faire des *LDW* en trop dans la boucle qui seraient inutiles. Il ne faut pas oublier de soustraire n pour avoir le bon nombre de boucles.

	Prolog							Loop		Epilog							
	1	2	3	4	5	6	7	8	...	22	23	24	25	26	27	28	29
.L1								ADD	...	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD
.L2								ADD	...	ADD	ADD	ADD	ADD	ADD	ADD	ADD	
.S1		SUB	SUB	SUB	SUB	SUB	SUB	SUB	...								
.S2			B	B	B	B	B	B	...			B B3					
.M1						MPY	MPY	MPY	...	MPY	MPY	MPY	MPY	MPY			
.M2						MPYH	MPYH	MPYH	...	MPYH	MPYH	MPYH	MPYH	MPYH			
.D1	LDW x[0]	LDW x[2]	LDW x[4]	LDW x[6]	LDW x[8]	LDW x[10]	LDW x[12]	LDW x[14]	...								
.D2	LDW y[0]	LDW y[2]	LDW y[4]	LDW y[6]	LDW y[8]	LDW y[10]	LDW y[12]	LDW y[14]	...								

```
1             .global _sum5
2             .global _x
3             .global _h
4  _sum5:
5          MVKL     .S1     _x,A0
6     ||   MVKL     .S2     _h,B0
7          MVKH     .S1     _x,A0
8     ||   MVKH     .S2     _h,B0
9     ||   sub      .D1     A4,12,A4
10         MV       .S1     A4,A2
11    ||   zero     .L1     A4
12    ||   zero     .L2     B4
13
14 Prolog_1:   LDW      .D1      *A0++,A1
15         ||  LDW      .D2      *B0++,B1
16
17 Prolog_2:        LDW      .D1      *A0++,A1
18    ||            LDW      .D2      *B0++,B1
19    ||  [A2]      SUB      .S1      A2,2,A2
20
21 Prolog_3:        LDW      .D1      *A0++,A1
22    ||            LDW      .D2      *B0++,B1
23    ||  [A2]      SUB      .S1      A2,2,A2
24    ||  [A2]      B        .S2      Loop
25
26 Prolog_4:        LDW      .D1      *A0++,A1
27    ||            LDW      .D2      *B0++,B1
28    ||  [A2]      SUB      .S1      A2,2,A2
29    ||  [A2]      B        .S2      Loop
30
31 Prolog_5:        LDW      .D1      *A0++,A1
32    ||            LDW      .D2      *B0++,B1
33    ||  [A2]      SUB      .S1      A2,2,A2
34    ||  [A2]      B        .S2      Loop
35
36 Prolog_6:        LDW      .D1      *A0++,A1
37    ||            LDW      .D2      *B0++,B1
38    ||  [A2]      SUB      .S1      A2,2,A2
39    ||  [A2]      B        .S2      Loop
40    ||            MPY      .M1x     A1,B1,A5
41    ||            MPYH     .M2x     A1,B1,B5
42
43 Prolog_7:        LDW      .D1      *A0++,A1
44    ||            LDW      .D2      *B0++,B1
45    ||  [A2]      SUB      .S1      A2,2,A2
46    ||  [A2]      B        .S2      Loop
47    ||            MPY      .M1x     A1,B1,A5
48    ||            MPYH     .M2x     A1,B1,B5
```

```
50 Loop:          LDW     .D1     *A0++,A1 ; double { x[i+1], x[i] }
51     ||         LDW     .D2     *B0++,B1 ; double { h[i+1], h[i] }
52     ||  [A2]   SUB     .S1     A2,2,A2
53     ||  [A2]   B       .S2     Loop
54     ||         MPY     .M1x    A1,B1,A5 ; A5 = x[i]*h[i]
55     ||         MPYH    .M2x    A1,B1,B5 ; B5 = x[i+1]*h[i+1]
56     ||         ADD     .L1     A4,A5,A4 ; A4 = A4 + x[i]*h[i]
57     ||         ADD     .L2     B4,B5,B4 ; B4 = B4 + x[i+1]*h[i+1]
```

```
59 Epilog_1:  MPY     .M1x    A1,B1,A5
60     ||     MPYH    .M2x    A1,B1,B5
61     ||     ADD     .L1     A4,A5,A4
62     ||     ADD     .L2     B4,B5,B4
63
64 Epilog_2:  MPY     .M1x    A1,B1,A5
65     ||     MPYH    .M2x    A1,B1,B5
66     ||     ADD     .L1     A4,A5,A4
67     ||     ADD     .L2     B4,B5,B4
68
69 Epilog_3:  B       .S2     B3
70     ||     MPY     .M1x    A1,B1,A5
71     ||     MPYH    .M2x    A1,B1,B5
72     ||     ADD     .L1     A4,A5,A4
73     ||     ADD     .L2     B4,B5,B4
74
```

```
75 Epilog_4:  MPY     .M1x    A1,B1,A5
76     ||     MPYH    .M2x    A1,B1,B5
77     ||     ADD     .L1     A4,A5,A4
78     ||     ADD     .L2     B4,B5,B4
79
80 Epilog_5:  MPY     .M1x    A1,B1,A5
81     ||     MPYH    .M2x    A1,B1,B5
82     ||     ADD     .L1     A4,A5,A4
83     ||     ADD     .L2     B4,B5,B4
84
85 Epilog_6:  ADD     .L1     A4,A5,A4
86     ||     ADD     .L2     B4,B5,B4
87
88 Epilog_7:  ADD     .L1     A4,A5,A4
89     ||     ADD     .L2     B4,B5,B4
90
91 Epilog_8:  ADD     .L1x    A4,B4,A4
```

Pour débuger, j'ai utilisé le « Memory browser » et les informations des registres. Ceci pour vérifier quand est-ce que les registres sont chargés et calculés pour chaque itération. Voici les résultats du profiler en mode debug :

1ère fois

Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
main()	1	-	-	1416.00	1416	-	-	3295.00	3295
sum1	1	681	681	681.00	681	681	681	681.00	681
sum2	1	621	621	621.00	621	621	621	621.00	621
sum3	1	335	335	335.00	335	335	335	335.00	335
sum4	1	175	175	175.00	175	175	175	175.00	175
sum5	1	67	67	67.00	67	67	67	67.00	67

Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
main()	1	1466	1466	1466.00	1466	3262	3262	3262.00	3262
sum1	1	653	653	653.00	653	653	653	653.00	653
sum2	1	611	611	611.00	611	611	611	611.00	611
sum3	1	330	330	330.00	330	330	330	330.00	330
sum4	1	170	170	170.00	170	170	170	170.00	170
sum5	1	30	32	32.00	32	32	32	32.00	32

J'atteint un minimum de 30 boucles pour le calcul d'un produit scalaire, ce qui est un très bon résultat quand on compare au nombre de cycles en C du lab 2 (169 cycles de CPU en mode Release).

Release :

Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
main()	1	-	-	175.00	175	-	-	2048.00	2048
sum1	1	690	690	690.00	690	690	690	690.00	690
sum2	1	616	616	616.00	616	616	616	616.00	616
sum3	1	335	335	335.00	335	335	335	335.00	335
sum4	1	170	170	170.00	170	170	170	170.00	170
sum5	1	62	62	62.00	62	62	62	62.00	62

Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
main()	1	76	76	76.00	76	1872	1872	1872.00	1872
sum1	1	653	653	653.00	653	653	653	653.00	653
sum2	1	611	611	611.00	611	611	611	611.00	611
sum3	1	330	330	330.00	330	330	330	330.00	330
sum4	1	170	170	170.00	170	170	170	170.00	170
sum5	1	30	32	32.00	32	32	32	32.00	32

Pour conclure ces laboratoires, voici un petit schéma qui résume bien la programmation :

Ca a en effet été très rapide pour la programmation en C (quelques minutes) mais assez long pour l'optimisaton Pipeline qui nécessite beaucoup de rigueur (une à deux heures). A voir si les heures en plus sont justifiés par rapport au nombre de cycles.

Quelques minutes ➔ 169 cycles

1 à 2 heures ➔ 30 cycles