UNIVERSITÉ DE TECHNOLOGIE DE COMPIÈGNE - MI01

Structure d'un calculateur informatique

Rapport TP n°5:7

Traitement d'image - Ladaube software

G : mi01a031

Deux experts en programmation x86-64 😉

- **Pillis Julien**
- **Galleze Rayane**

Introduction

Au cours de ce rapport nous allons présenter les résultats des exercices du TP 5 à 7 sur le traitement d'image. Ce TP a pour objectif d' accélérer la détection de contours (filtre de Sobel) dans une image traitée par un algorithme C, en passant par le langage assembleur. Pour cela, nous réaliserons et commenterons deux des six parties qui composent le sujet. La cinquième partie nous permettra de prendre en main le sujet et d'effectuer une première couche de traitement sur l'image donnée : la conversion en niveaux de gris. Nous aurons l'occasion de mieux comprendre comment se matérialise une image en mémoire (son stockage) et nous réaliserons notre premier calcul d'intensité des pixels à partir des données les composant.

Une fois cette cinquième partie effectuée, nous pourrons commencer la sixième partie qui traite de la détection de contours. Nous calculerons le gradient de chaque pixels à l'aide d'opérateurs que nous appliquerons sur ces mêmes pixels, et nous l'utiliserons en tant qu'indicateur de contours après certains traitements (conversion en valeur absolue, limitation des valeurs entre 0 et 255 et conversion en niveau de gris). On appelle ce traitement : le filtre de Sobel.

Première partie : conversion en niveaux de gris

5.1.1 Itération sur tous les pixels de l'image

[a] L'intérêt de réaliser la boucle en comptant le nombre de pixels restants :

Compter le nombre de pixels restants permet d'avoir un indice pour pouvoir manipuler les pixels de l'image, une fois cet indice à 0, on a fini le traitement.

En effet, nous pouvons initialiser un compteur à 0 et l'incrémenter jusqu'à atteindre `rdi` qui représente le nombre de pixels de notre image. Mais dans ce cas, nous devrons utiliser un nouveau registre et le comparer avec `rdi` (condition de sortie de boucle) ce qui n'est pas la meilleur option.

[b] Adresse du pixel en cours dans `img_src` et `img_temp1`

Rappel :
Les références mémoire se font selon la syntaxe suivante :

`[ rbase + rindex * ech + depl ]`

- **`rbase`** : registre de base
- **`rindex`** : registre d'index
- **`ech`** : facteur d'échelle (1, 2, 4 ou 8)
- **`depl`** : déplacement signé 32 bits constant.

Chacun de ces éléments peut être omis.

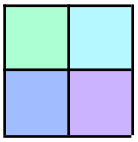
`rdi` : nombre de pixels restants, au départ = longueur * largeur = nombre de pixels total
`rdx` : adresse du pixel(0, 0) de l'image source (pointeur sur l'image source)
`rcx :` adresse du pixel(0, 0) de l'image tampon 1 (pointeur sur l'image tampon 1)

Chaque pixel est codé sur 4 octets (32 bits), pour passer d'un pixel à un autre le déplacement est de moins 4 octets (on commence par le tout dernier pixel, donc on décrémente du dernier jusqu'au premier pixel). De plus il faut noter que l'image est un tableau de pixels qui commence à 0, si on prenait juste **`[ rdx + rdi*4 ]`** on se retrouve hors de notre tableau de pixel. **`[ rdx + rdi*4 ]`** pointe en effet vers le premier octet qui suit notre tableau de pixels en mémoire.

Donc

- l'adresse du pixel en cours dans **`img_src`** est **`[ rdx + rdi*4 - 4 ]`**
- l'adresse du pixel en cours dans **`img_temp1`** est **`[ rcx + rdi*4 - 4 ]`**

Exemple :
Considérons une toute petite image de 4 pixels à l'adresse `0x004000`



rdx pointe donc vers l'adresse `0x004000` et **rdi** a pour valeur 4.
NB : Si on considère l'adresse du pixel en cours **[rdx + rdi*4]**, au début de notre boucle on se retrouve à l'adresse `0x004010` donc hors notre tableau de pixels.

```
&img :  0x004000 <Hex>
  Address   0 - 3      4 - 7      8 - B      C - F
  0x004000  ABFFD2FF   B5F8FFFF   A1BDFFFF   CCB3FFFF
  0x004010  01000000   00000000   900D1EF7   FF7F0000
```

1ère itération :
rdi = 4 → l'adresse du pixel en cours est **[rdx + rdi*4 - 4]** = 0x00400C
… traitement pixel …
rdi = rdi - 1
condition de boucle
2ème itération :
rdi = 3 → l'adresse du pixel en cours est **[rdx + rdi*4 - 4]** = 0x004008
… traitement pixel …
rdi = rdi - 1
condition de boucle
3ème itération:
rdi = 2 → l'adresse du pixel en cours est **[rdx + rdi*4 - 4]** = 0x004004
… traitement pixel …
rdi = rdi - 1
condition de boucle
4ème itération :
rdi = 1 → l'adresse du pixel en cours est **[rdx + rdi*4 - 4]** = 0x004000
… traitement pixel …
rdi = rdi - 1
sortie de boucle car rdi = 0

[c] Affectation d'une couleur unie à chaque pixel de `img_temp1`

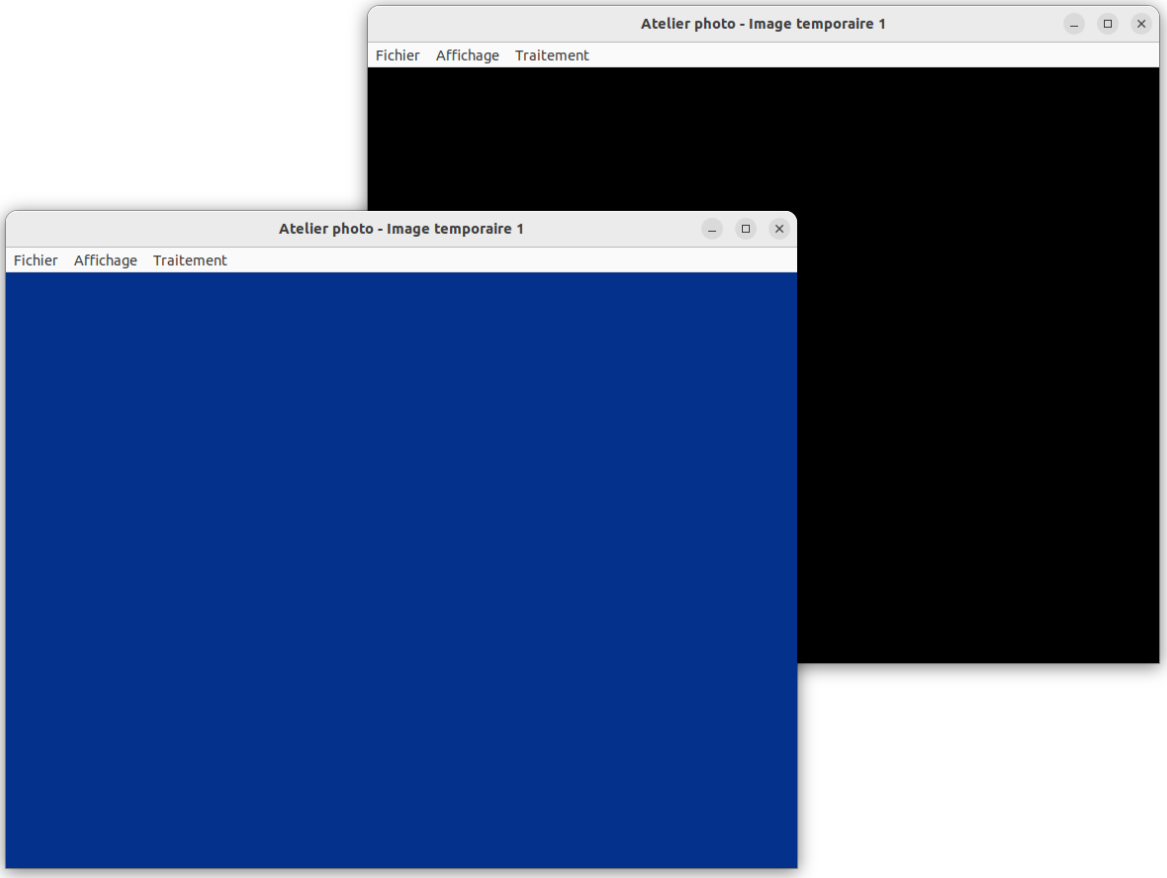
Une fois l'adresse du pixel en cours de traitement déterminée, nous pouvons implémenter une affectation de couleur à ces pixels pour vérifier le bon fonctionnement de la boucle. Puisqu'il est demandé d'affecter cette couleur aux pixels d'**img_temp1**, nous utiliserons le registre **rcx** correspondant au pointeur sur l'image tampon 1 (premier pixel).

Pour cela nous ajoutons, à la boucle **`loop_gs`** , l'instruction :

```
mov dword ptr [rcx + rdi*4 - 4], 0xff8C3103 ;
```

Nous affectons sur nos pixels une jolie couleur bleue avec un canal de transparence le plus opaque possible (`0xff`).
Un pixel étant défini sur 32 bits, il sera nécessaire de préciser la taille de l'emplacement mémoire à laquelle nous attribuons la couleur pour éviter toute ambiguïté.

Affichage de l'image tampon 1 :
(à droite : avant traitement assembleur, à gauche : après le traitement assembleur)



5.1.2 Calcul de l'intensité d'un pixel

[a] Représentation du produit : composant * coefficient

D'après l'énoncé, chaque coefficient (**0.2126** (R), **0.7152** (V), **0.0722** (B)) est représenté en code binaire non signé, à virgule fixe sur 16 bits. Le décalage de la virgule est de +8 bits.

De plus, les composants R,V,B, sont considérés comme entiers sans virgule . Sachant que leur valeur est inférieure ou égale à 255, nous en déduisons que leur représentation peut se faire sur 8 bits.

Règle de multiplication générale : le nombre de chiffres après la virgule du résultat est, au plus, égal à la somme du nombre des chiffres après la virgule de chaque opérande.

Exemple : 0,45 * 0,13 = 0,0585 (2+2 = 4)

⇒ **Règle de multiplication entier * nombre à virgule :** le nombre de chiffres après la virgule du résultat est, au plus, égal à celui du nombre à virgule.

Puisque dans la multiplication il y aura 4 chiffres après la virgule au plus (valeur des coefficients), il n'y aura que 4 décimales dans le résultat.

De plus, ces coefficients étant plus petits que 1 et la plus grande multiplication possible étant 255*0,7152 ≈ 182, seulement 8 bits seront nécessaires pour conserver la partie entière du résultat.

Nous pouvons donc conclure que 16 bits, avec le même décalage de 8 bits pour la virgule, sont suffisants pour stocker le résultat du produit.

[b] Représentations binaire et hexadécimale des coefficients

Coefficients représentés en base décimale, binaire et hexadécimale :

$$(0.7152)_{10} \Leftrightarrow (00000000.10110111)_2 \Leftrightarrow (00.B7)_{16}$$
$$(0.2126)_{10} \Leftrightarrow (00000000.00110110)_2 \Leftrightarrow (00.36)_{16}$$
$$(0.0722)_{10} \Leftrightarrow (00000000.00010010)_2 \Leftrightarrow (00.12)_{16}$$

Remarque :
La représentation en base 10 est la seule exacte. Celles des bases binaire et hexadécimale sont tronquées pour pouvoir être représentées sur 16 bits.

De plus, d'après la question [c](question suivante), la somme des coefficients vaut 1.
Dans notre cas 00.B7 + 00.36 + 00.12 = 00.FF ≠ 1.00 . (suite à la perte de précision induite par les troncatures).

Pour conserver cette propriété, nous rajouterons 00.01 au coefficient bleu (00.12).

Ainsi nous faisons l'équivalence suivante :

$$(0.0722)_{10} \Leftrightarrow (00.13)_{16}$$

Remarque : Nous aurions pu 00.01 à n'importe quel des trois coefficients, tant que la somme est égale à 1.

[c] Etude sur le débordement de l'intensité

Si nous majorons R, V et B par leurs valeurs max : 255, la partie entière de l'intensité est au maximum de 255, soit 8 bits :

I = B*coef.bleu + R*coef.rouge + V*coef.vert ≤ I = 255 * (coef.bleu + coef.rouge + coef.vert) = 255

Avec (coef.bleu + coef.rouge + coef.vert) = 1

Puisque la multiplication des composants par leurs coefficients respectifs ne donnera jamais plus de 4 chiffres après la virgule et la somme de ces produits non plus, nous n'aurons jamais plus de 4 chiffres après la virgule dans la valeur de l'intensité.
La valeur de I sera donc toujours représentable sur 16 bits (8 bits de partie entière, 8 bits de chiffres après la virgule).

Ceci nous permet d'affirmer qu'aucun débordement ne devrait être constaté.

[d] Algorithme du calcul de l'intensité

Nous posons BS, VS, RS et TS les composantes rouge, verte, bleue et la transparence des pixels de l'image source pour le calcul.
R représente une variable temporaire.

Décomposition du calcul de I en opérations élémentaires :

R <- $(BS)_{16}$ * $(00.13)_{16}$
I <- R
R <- $(VS)_{16}$ * $(00.36)_{16}$
I <- I + R
R <- $(RS)_{16}$ * $(00.B7)_{16}$
I <- I + R

Remarque :
Selon notre représentation, aucun décalage n'est nécessaire pour le produit. En effet, multiplier un entier par un chiffre à virgule ne nécessite pas de décalage spécifique. La virgule sera placée au même endroit que celle du chiffre à virgule dans le résultat.

Exemple :
$(2)_{16}$ * $(00.05)_{16}$ =$(00.0A)_{16}$ (le point représente une virgule théorique pour illustrer)
⇔ $(2)_{10}$ * $(0.0125)_{10}$ = $(0.0390625)_{10}$

De plus, il ne faut pas oublier que l'intensité récupérée est sur 8 bits car elle n'a que 256 valeurs possibles (de 0 à 255).

De cette décomposition, nous obtenons l'algorithme suivant :

```
Pour chaque pixel p de image_src :
      R1 <- bleu(p) * 0x0013
      R2 <- R1

      R1 <- vert(p) * 0x00B7
      R2 <- R2 + R1

      R1 <- rouge(p) * 0x0036
      R2 <- R2 + R1

      /********** Utile pour la partie suivante ******
      transparence(p) <- 0xff
      rouge(p) <- partie_entière(R2)
      ********************************************/
```

Notes :

- `bleu(p)`, `rouge(p)`, `vert(p)` sont les composantes du pixel sur 16 bits avec extension de 0 sur 8 bits (même nombre de bits nécessaire pour la multiplication).

- `transparence(p)` est sur 8 bits.

- R1 et R2 sont des registres temporaires 16 bits permettant de faire les calculs.
- À la fin de l'algorithme, il n'est pas nécessaire de faire un décalage de 8 bits si nous souhaitons récupérer la valeur entière de `R2`. Si nous utilisons `rax` en tant que `R2`, il suffira d'affecter l'octet du sous-registre ah à `rouge(p)` pour récupérer la valeur entière.

5.1.3 Calcul complet

[a] Implémentation de l'algorithme en assembleur

Rappel : Représentation du pixel en mémoire (voir partie 4 de l'énoncé)

L'ordre de stockage conventionnel étant le little endian, nous allons devoir le prendre en compte dans notre code assembleur pour ne pas modifier les mauvais composants des pixels à traiter.

Pour pouvoir implémenter le calcul entièrement dans la boucle, nous allons utiliser le sous registre `r11w`, `r10w` et `ax` pour nos calculs. Selon la convention Linux, le registre r11 est volatile (sauvegardé par l'appelant). Il n'est donc pas nécessaire d'effectuer une sauvegarde.

La structure de mémoire étant le Little Endian, l'accès aux composantes des pixel se fera selon les indices suivants :

Pixel n°rdi - 1	R	byte ptr [rcx + rdi*4 - 4]
	V	byte ptr [rcx + rdi*4 - 3]
	B	byte ptr [rcx + rdi*4 - 2]
	α	byte ptr [rcx + rdi*4 - 1]
Pixel n°rdi	R	byte ptr [rcx + rdi*4]
	…	

Remarque :
Pour des raisons d'optimisation du temps d'exécution les opérations pour calculer l'intensité seront écrites avec une certaine intercalation pour réduire au maximum les ruptures de pipeline.

Par exemple, étudions les 2 instructions successives suivantes :

```
movzx ax, byte ptr[rdx + rdi*4 - 2] /*composante bleue*/
imul ax, 0x0013 /*coef bleu*/
```

Faisons la supposition que le pipeline du processeur est découpé de la façon suivante :
PF : (Prefetch) recherche de l'instruction
D1 : (Decode 1) phase 1 du décodage de l'instruction
D2 : (Decode 2) phase 2 du décodage de l'instruction (calcul d'adresse & chargement des données en registres tampons)
EX : (Execute) exécution
WB : (Write Back) Ecriture du résultat en mémoire/dans les registres

(Pipeline type processeur 80486)

Le fonctionnement serait alors :

Programme	Cycles d'horloge					
	1	2	3	4	5	6
`movzx ax, byte ptr[rdx + rdi*4 - 2]`	PF	D1	D2	EX	WB	
`imul ax, 0x0013`		PF	D1	D2	EX	WB

Au cycle numéro 4, les instructions seraient en conflit. Nous voulons récupérer la valeur de ax dans la deuxième instruction alors qu'elle change de valeur pendant l'exécution de la première instruction.
Il faut alors que le processeur retarde la deuxième instruction pour contourner ce conflit et éviter une incohérence des données produites.

Voici l'implémentation assembleur :

Récupération de la composante bleue avec extension par zéro pour éviter les erreurs de calculs si ax n'est pas nul.

```
movzx ax, byte ptr[rdx + rdi*4 - 2] /*composante bleue*/
```

Récupération de la composante verte

```
movzx r10w, byte ptr[rdx + rdi*4 - 3] /*composante verte*/
```

Multiplication des coefficients bleu et vert et de leur composante. Puis déplacement vers le sous registre ax.

```
imul ax, 0x0013 /*coef bleu*/
imul r10w, 0x00B7 /*coef vert*/
```

Récupération de la composante verte

```
movzx r11w, byte ptr[rdx + rdi*4 - 4] /*composante rouge*/
```

On additionne V*V.coeff + B*B.coeff, le résultat sera stocké dans ax

```
add ax,r10w
```

Multiplication du coefficient rouge et la composante rouge.

```
imul r11w, 0x0036 /*coef rouge*/
```

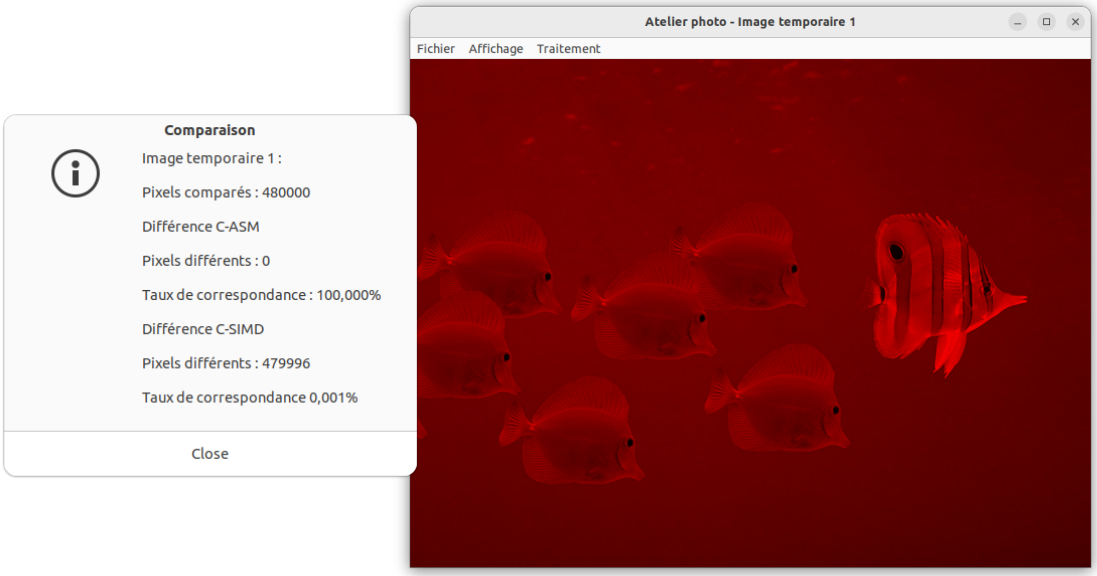
Puis dernière addition pour avoir l'intensité

```
add ax,r11w
```

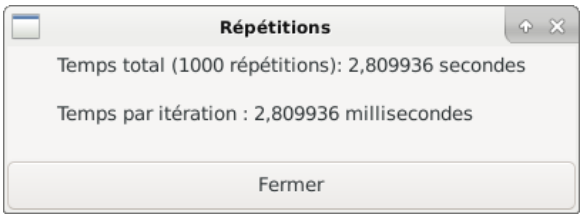
Enfin, attributions de l'opacité maximale à la composante correspondante du pixel et affectation de l'intensité à la composante rouge.

```
mov byte ptr[rcx + rdi*4 - 1], 0xff /*opacité maximale*/
mov byte ptr[rcx + rdi*4 - 4], ah /*valeur de I dans la composante rouge*/
```

Image obtenue après traitement :



[b] Performance et comparaison avec l'implémentation C



Implémentation C

Implémentation ASM

Après exécution du code sur la même image, sur 1000 répétitions, nous nous apercevons que le code assembleur est nettement plus performant que le code C : environ 2.3 fois plus rapide !
Et ce, pour un résultat équivalent (taux de correspondance = 100%)

Deuxième partie : filtre de Sobel

6.3.1 Construction de la double itération

[1] Calcul des adresses source et destination

Rappels :

- Pointeur sur img_temp1 : `rcx`
- Pointeur sur img_temp2 : `r8`
- L'instruction `pop rdi`, permet de récupérer la largeur de l'image en pixel dans ce registre. La taille en pixels d'une ligne correspond donc à `rdi`.

Pour le calcul des adresses source et destination, nous utiliserons les registres `r10` et `r11`. Ceux-ci étant volatiles selon la convention Linux, nous n'avons pas besoin de les sauvegarder.

L'adresse du premier pixel auquel appliquer le masque est le pixel pointé par `rcx`.

Cependant, le pixel de destination est lui le pixel de la deuxième ligne et de la deuxième colonne.

Pour le calculer, nous allons procéder ainsi :

- Nous devons traverser la première ligne : `r8 + rdi*4` (un pixel est défini sur 4 octets, le facteur d'échelle est donc de 4).
- Puis, aller sur le deuxième pixel de la deuxième ligne : `+ 4` octets.

Ainsi :

```
lea r10,[rcx] /* r10 : pointeur sur  img_temp1 */
lea r11,[r8 + rdi*4 + 4] /* r11 : pointeur sur  img_temp2 */
```

[2] Itération sur les lignes

[2.a] Initialisation du compteur de lignes

Sachant que les lignes des bords ne seront pas traitées, il suffit de de prendre la valeur de la hauteur (rsi) moins 2 (première ligne + dernière ligne). Ce compteur pourra donc être initialisé à valeur(rsi) - 2.

[2.b] Evolution des pointeurs de pixel

À la fin d'une ligne, le pointeur du pixel source doit prendre la valeur de l'adresse du premier pixel de la ligne suivante (en considérant qu'à la fin d'une ligne, r10 pointe sur un pixel qui n'est pas à traiter (bord)).
Cela signifie qu'il pointera vers 4 + 4 = 8 octets plus loin. Soit : `lea r10, [r10 + 8]`

De la même manière pour le pointeur du pixel de destination : `lea r11, [r11 + 8]`

[2.c] Implémentation de la boucle d'itération

Nous utiliserons rsi comme compteur de lignes. Nous faisons donc une sauvegarde préliminaire de ce registre non volatile et nous l'initialisons.

```
push rsi      /* compteur de lignes */
sub rsi, 2    /* élimination des bords */
```

Puis, nous démarrons la boucle :

```
loop_rows:
      lea r10, [r10 + 8]
      lea r11, [r11 + 8]
      sub rsi,1         /* décrémentation du nombre de lignes restantes à
                        traiter de l'image*/
      jg loop_rows      /* si nombre de lignes restantes > 0 on boucle */
      pop rsi           /* compteur de lignes */
```

[3] Itération sur les colonnes

[3.a] Initialisation du compteur de colonnes

Sachant que les colonnes des bords ne seront pas traitées, il suffit de de prendre la valeur de la largeur (rdi) moins 2 (première colonne + dernière colonne).
Ce compteur pourra donc être initialisé à valeur(rdi) - 2 à chaque nouvelle ligne.

[3.b] Evolution des pointeurs de pixel

À la fin d'une colonne, le pointeur du pixel source doit prendre la valeur de l'adresse du pixel suivant. Cela signifie qu'il pointera vers + 4 octets plus loin.
Soit : `lea r10, [r10 + 4]`

De la même manière pour le pointeur du pixel de destination : `lea r11, [r11 + 4]`

[3.c] Implémentation de la boucle d'itération

Nous utiliserons `rdi` comme compteur de colonnes. Nous faisons donc une sauvegarde préliminaire de ce registre non volatile et nous l'initialisons (sauvegarde à effectuer dans `loop_rows` pour pouvoir récupérer la hauteur à chaque saut de ligne.).

```
push rdi
sub rdi, 2  /* élimination des bords*/
```

Remarque : cette étape sera modifiée par la suite, pour les besoins du traitement de l'image (voir : 6.3.2 [b]).

Puis, nous démarrons la boucle :

```
loop_cols:
      lea r10, [r10 + 4]      /* incrémentation de la colonne (pixel
                              suivant) */
      lea r11, [r11 + 4]
      sub rdi,1               /* décrémentation du nombre de colonnes
                              restantes à traiter de la ligne */
      jg loop_cols            /* si nombre de colonnes restantes > 0 on
                              boucle */
      pop rdi                 /* récupération de la hauteur */
```

Imbriquée dans la boucle d'itération sur les lignes, nous obtenons :

```
            push rsi          /* compteur de lignes */
            sub rsi, 2        /* élimination des bords */

loop_rows:
            push rdi
            sub rdi, 2        /* élimination des bords*/

loop_cols:

          lea r10, [r10 + 4]  /* incrémentation de la colonne (pixel
                              suivant) */
          lea r11, [r11 + 4]
          sub rdi,1           /* décrémentation du nombre de colonnes
                              restantes à traiter de la ligne */

          jg loop_cols        /* si nombre de colonnes restantes > 0 on
```

```
                    boucle */
        pop rdi             /* récupération de la hauteur */

        lea r10, [r10 + 8]
        lea r11, [r11 + 8]
        sub rsi,1           /* décrémentation du nombre de lignes
                            restantes à traiter de l'image*/
        jg loop_rows        /* si nombre de lignes restantes > 0 on
                            boucle */
        pop rsi             /* compteur de lignes */
```

[4] Test des itérations

Lors des tests, l'image source devrait avoir deux colonnes de pixels noirs sur la droite de l'image et deux lignes de pixels noirs sur le bas de l'image.

Quant à l'image de destination, elle devrait être encadrée d'une ligne de pixels noirs.

Pour effectuer ce test voici le code implémenté :

```
            ...
            lea  r10, [rcx]
            lea  r11, [r8 + rdi*4 + 4]
            push rsi
            sub  rsi, 2

loop_rows:
            push rdi
            sub  rdi, 2
loop_cols:
            // Test de la boucle ( affectation de couleur )
            mov  dword ptr [r10],0xffFF920D
            mov  dword ptr [r11],0xffFF920D
            lea  r10, [r10 + 4]
            lea  r11, [r11 + 4]
            sub  rdi,1

            jg   loop_cols

            lea  r10, [r10 + 8]
            lea  r11, [r11 + 8]
            sub  rsi,1
            pop  rdi

            jg   loop_rows
            pop  rsi
            ...
```

Résultats :

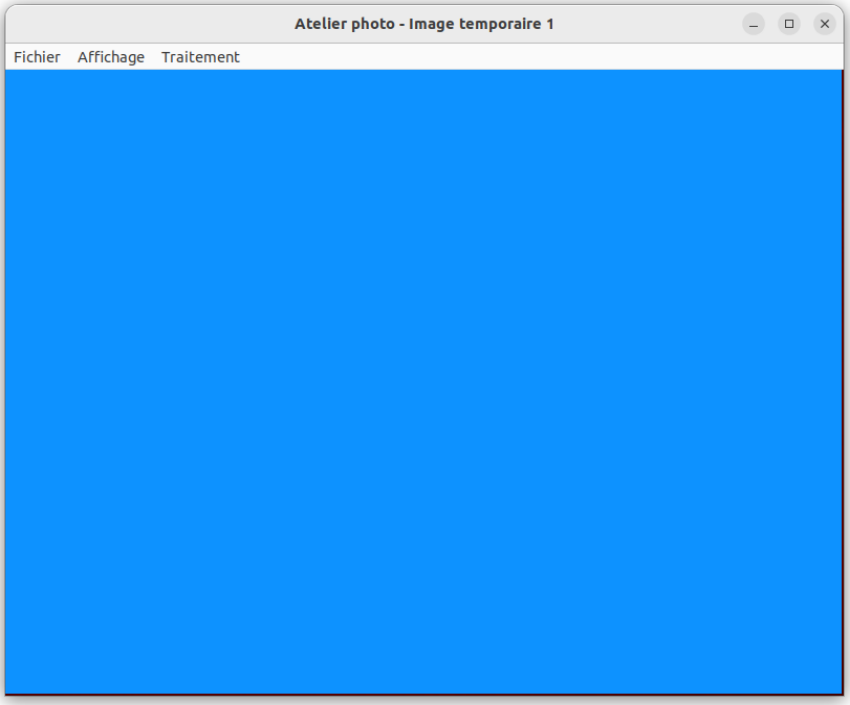


Image source post-traitement

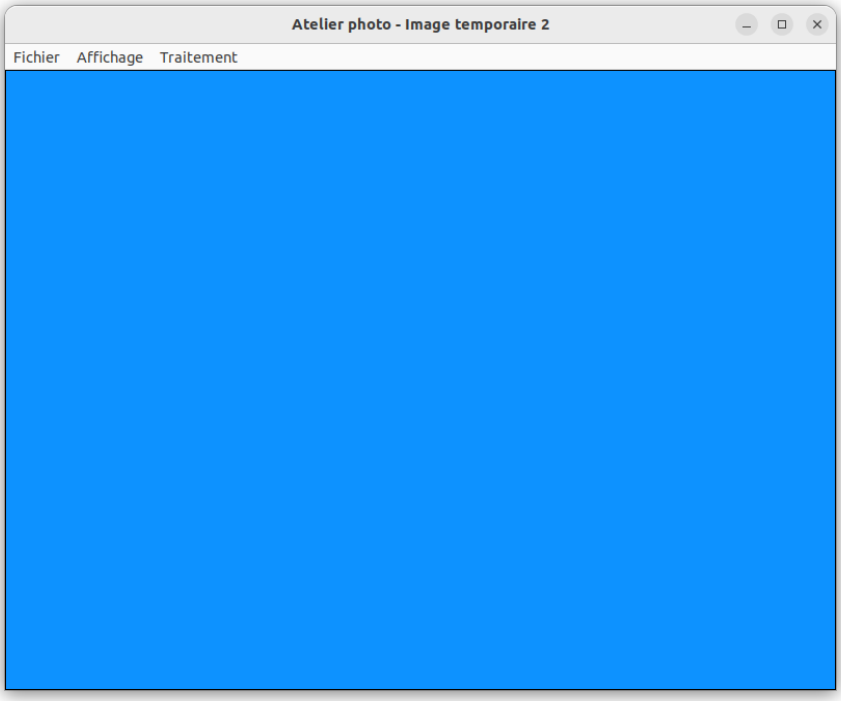


Image destination post-traitement

Les images traitées sont bien celles attendues !

6.3.2 Calcul du gradient de chaque pixel

[a] Calculs d'adresses

a_{11} correspond à l'adresse du premier pixel de img_temp1. Cette adresse est stockée dans le registre `rcx`. Le nombre de colonnes est quant à lui stocké dans le registre `rdi`.
A partir de ces registres, il n'est pas compliqué de calculer les adresses de a_{21}, a_{12} et a_{13}.

Calculer a_{21} signifie calculer le pixel de la seconde colonne sur la première ligne. Pour cela, il suffit simplement de reprendre le calcul de l'évolution du pointeur de pixel sur les colonnes : a_{21} = `rcx + 4`, car on saute de 4 octets pour passer au prochain pixel.

Pour calculer a_{12}, il suffit de sauter une ligne. Comme vu précédemment, cela se traduit par : adresse du pixel de départ + largeur de l'image. En utilisant les registres `rcx` et `rdi`, nous avons donc : a_{12} = `rcx + rdi*4` (rappel : le facteur d'échelle vaut 4 car un pixel est stockée sur 4 octets).

Enfin, le calcul de a_{13} est identique à celui de a_{12} . Il suffira simplement de sauter 2 lignes si on le calcul par rapport à a_{11} : c'est-à-dire `2*(rdi*4)` soit `rdi*8`

[b] Implémentation du calcul des gradients G_x et G_y

Bien qu'apparaissant lourd au premier abord, le calcul des gradients n'est en réalité pas compliqué puisque de nombreux éléments des matrices (masques) sont nuls (6 sur 18).
Il ne sera donc pas nécessaire d'effectuer de calcul sur les pixels concernés par ces éléments des masques (m_{21}, m_{22}, m_{23} dans S_x pour G_x, et m_{12}, m_{21}, m_{31} dans S_y pour G_y).

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Chaque masque s'appliquera sur les pixels déterminés par les boucles d'itérations sur les colonnes et lignes. Pour pouvoir récupérer les adresses des pixels (composantes rouges, car là où est stockée l'intensité), nous allons donc devoir utiliser le registre r10 (c.f partie 6.3.1)qui pointe vers le pixel à traiter, c'est-à-dire le pixel (0,0) sur masque. Nous devrons donc calculer les adresses des pixels auxquels appliquer les masques par rapport à celui-ci.

De plus, nous utiliserons le registre r13, qui prendra la valeur de rdi avant l'entrée dans les boucles des itérations puisque sa est modifiée dans celles-ci.

Ainsi, l'étape expliquée en 6.3.1 [2.c] devient alors :

```
                    …
                    mov    r13, rdi
loop_lignes:
                    mov    rdi, r13
                    sub    rdi, 2
                    …
```

Nous stockerons Gx dans dx ainsi que Gy dans bx. L'utilisation de sous-registre de la taille d'un mot permet de gagner en précision sur la suite des calculs, bien qu'un sous-registre 8 bits puisse aussi être utilisé.

Les multiplications par 2 et - 2 se feront, respectivement, à l'aide de deux instructions add et sub dans le registre du gradient correspondant.
La multiplication par 1 et -1 se fera de la même manière mais avec une seule instruction.

En utilisant la question précédente, il est facile de calculer les adresses des pixels impliqués dans le calcul :

```
// Calcul de G_x ( masque 1 Sx)
sub         ax, [r10]                   //    a11 * (-1)
add         ax, [r10 + 8]               // +  a31 * 1
sub         ax, [r10 + r13*4]           // +  a12 * (-2)
sub         ax, [r10 + r13*4]
add         ax, [r10 + r13*4 + 8]       // +  a32 * 2
add         ax, [r10 + r13*4 + 8]
sub         ax, [r10 + r13*8]           // +  a13 * (-1)
add         ax, [r10 + r13*8 + 8]       // +  a33 * 1


// Calcul de G_y ( masque 2 Sy)
add         bx, [r10]                   //    a11 * 1
add         bx, [r10+4]                 // +  a21 * 2
add         bx, [r10+4]
add         bx, [r10+8]                 // +  a31 * 1
sub         bx, [r10 + r13*8]           // +  a13 * (-1)
sub         bx, [r10 + r13*8 + 4]       // +  a23 * (-2)
sub         bx, [r10 + r13*8 + 4]
sub         bx, [r10 + r13*8 + 8]       // +  a33 * (-1)
```

[c] Valeur absolue

Pour calculer la valeur absolue de G_x et G_y, on pourrait procéder naïvement de la façon suivante :

```
      // Valeur abs(G_x) : si G_x < 0 on prendra - G_x
      cmp   ax, 0
      jge   g_x_positif
      neg   ax

g_x_positif:
      // Valeur abs(G_y) : si G_y < 0 on prendra - G_y
      cmp   bx, 0
      jge   g_y_positif
      neg   bx

g_y_positif:
      // suite
```

Mais étant experts en programmation assembleur x86 😉 , on cherche à tout prix à minimiser le temps d'exécution de notre programme.
Et pour cela on souhaite éviter les sauts pour le calcul de la valeur absolue car ces derniers provoquent des ruptures de pipelines (ruptures de séquence).
Une autre façon plus rapide pour calculer abs(x) serait de passer au complément à 2 lorsque x est négatif :

```
CWD               // dx:ax ← ax avec ext. de signe
xor ax, dx
sub ax, dx
```

Il faut noter d'abord les deux équations suivantes en complément à 2 :

```
            x xor 1 → not x
            x xor 0 → x

            not x - (- 1) → - x
            x - (- 1) → x

      ⇒     (x xor 1) - (-1) → -x
            (x xor 0) - (-1) → x
```

Si notre x est négatif dx sera égale à `0xFFFF` qui est - 1 en complément à 2.
`xor ax, dx` nous permet d'avoir `not ax` et enfin `not ax - (-1) = - ax`
Si x est positif dx sera égale à 0x0000 et donc `( ax xor 0) - (-1)` restera bien évidemment égale à ax.

Pour les besoins du calcul, nous utiliserons le sous-registre registre r14w qui sera sauvegardé avant les boucles à l'aide d'un push : `push r14`. Ce sous-registre stockera la valeur absolue de Gx.

L'implémentation en langage assembleur pour les deux calculs de valeur absolue est donc :

```
// Valeur abs(G_x)
CWD
xor ax, dx
sub ax, dx

mov    r14w, ax

// Valeur abs(G_y)
mov ax, bx
CWD
xor ax, dx
sub ax, dx
```

[d] Valeur finale du gradient

Selon l'algorithme de traitement :

```
G ← |Gx| + |Gy|
G ← 255 - G
si G < 0
     G ← 0
 fin si
```

Premièrement, effectuons la somme des gradients Gx (stocké dans r14w) et Gy (stocké dans ax) :

```
add          ax, r14w
```

Puis, soustrayons à 255 cette somme :

```
neg          ax
add          ax, 255
```

Remarque : nous effectuons ax = -ax +255 = 255 -ax, ce qui nous évite d'utiliser un registre supplémentaire.

Enfin, si G (ax) est supérieur à 0, il n'y a plus rien à faire. Sinon, on le borne à 0.

```
cmp          ax,0
jge          suite
xor          ax, ax
```

Soit :

```
 // Calcul final de G
              add          ax, r14w
              neg          ax
              add          ax, 255
              cmp          ax,0
              jge          suite
              xor          ax, ax
```

[e] Implémentation sur chaque pixel

Pour affecter le gradient au pixel, nous pouvons copier sa valeur dans sa composante rouge à l'aide de l'instruction :

```
                    mov byte ptr[r11], al
```

Voici le code complet dont les instructions ont été présentées précédemment :

```
/*********************************************************
Détecteur de contours de Sobel
*********************************************************/
              push     rsi
              push     rbx
              push     r13
              push     r14

              lea      r10, [rcx]    // adresse du pixel source à traiter
              lea      r11, [r8 + rdi*4 + 4]   // adresse du pixel de destination
              sub      rsi, 2
              mov    r13, rdi     // sauvegarde de la largeur de l'image dans r13
(constante)


loop_lignes:
              mov          rdi, r13
              sub     rdi, 2

loop_col:

              // Test de la boucle
              //mov        dword ptr[r10], 0xffFF920D
              //mov        dword ptr[r11], 0xffFF920D

              xor         rax, rax
              xor         rdx, rdx
              xor         rbx, rbx
```

```
                // Calcul de G_x ( masque 1 Sx)
                sub          ax, [r10]                //      a11 * (-1)
                add          ax, [r10 + 8]            // +     a31 * 1
                sub          ax, [r10 + r13*4]        // +     a12 * (-2)
                sub          ax, [r10 + r13*4]
                add          ax, [r10 + r13*4 + 8]    // +     a32 * 2
                add          ax, [r10 + r13*4 + 8]
                sub          ax, [r10 + r13*8]        // +     a13 * (-1)
                add          ax, [r10 + r13*8 + 8]    // +     a33 * 1


                // Calcul de G_y ( masque 2 Sy)
                add          bx, [r10]                //    a11 * 1
                add          bx, [r10+4]              // +     a21 * 2
                add          bx, [r10+4]
                add          bx, [r10+8]              // +     a31 * 1
                sub          bx, [r10 + r13*8]        // +     a13 * (-1)
                sub          bx, [r10 + r13*8 + 4]    // +    a23 * (-2)
                sub          bx, [r10 + r13*8 + 4]
                sub          bx, [r10 + r13*8 + 8]    // +     a33 * (-1)

               // Valeur abs(G_x)
                CWD
                xor ax, dx
                sub ax, dx

                mov    r14w, ax

               // Valeur abs(G_y)
                mov ax, bx
                CWD
                xor ax, dx
                sub ax, dx


                // Calcul final de G
                add          ax, r14w
                neg          ax
                add          ax, 255
                cmp          ax,0
                jge          suite
                xor     ax, ax

suite:


                mov byte ptr[r11], al


                lea     r10, [r10 + 4]
                lea     r11, [r11 + 4]
```

```
                sub     rdi, 1

                jg          loop_col

                lea     r10, [r10 + 8]
                lea     r11, [r11 + 8]
                sub     rsi, 1

                jg          loop_lignes

                pop    r14
                pop    r13
                pop    rbx
                pop    rsi

epilogue:       pop   rbp           # Dépiler le pointeur de cadre de pile sauvegardé
                ret                 # Retour à l'appelant
```

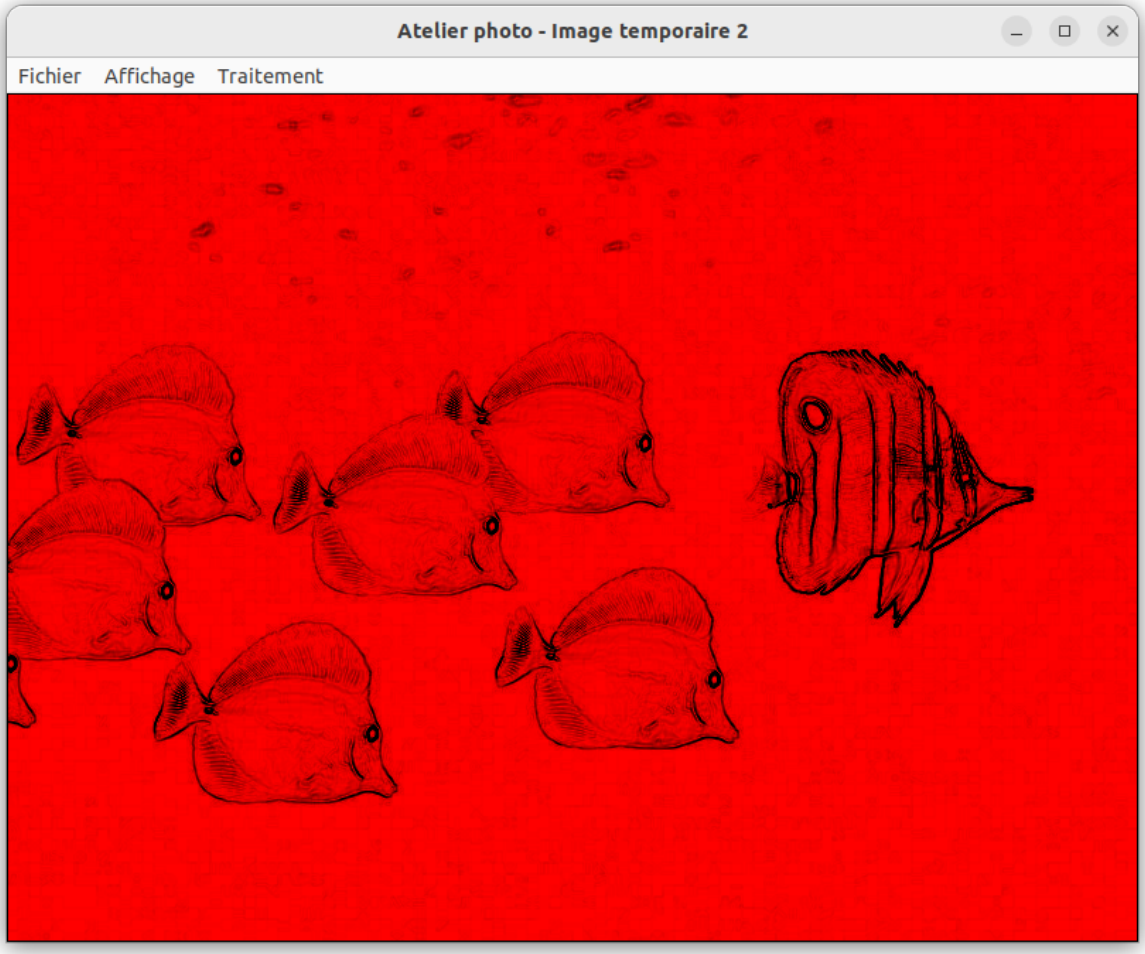


Image obtenue avec implémentation du calcul de gradient

6.3.3 Finalisation du programme

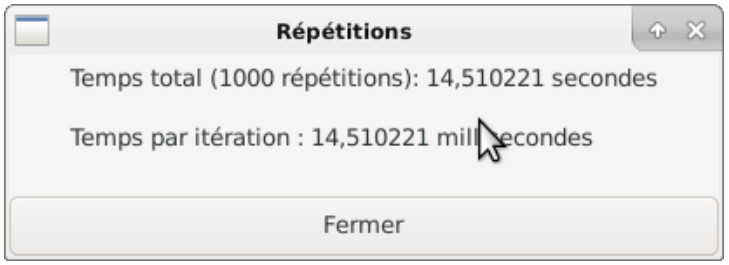
Pour que le résultat du calcul de chaque pixel s'affiche en niveaux de gris, il suffit simplement d'affecter à chaque composante couleur du pixel la même valeur que pour la composante rouge. Nous aurons donc :

```
mov byte ptr[r11], al          // composante rouge
mov byte ptr[r11 + 1], al      // composante verte
mov byte ptr[r11 + 2], al      // composante bleue
```

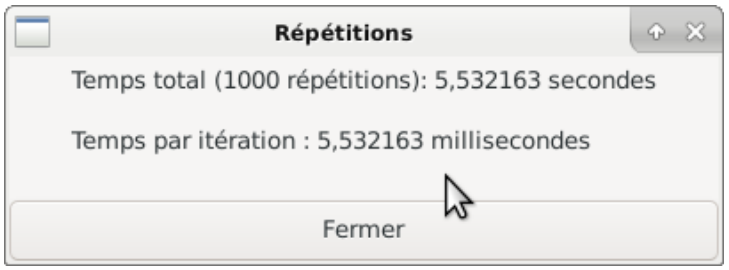
Remarque : la capture d'écran du test se trouve dans la question suivante.

6.3.4 Comparaison des performances

Pour finir, nous pouvons désormais tester et comparer la vitesse d'exécution de l'implémentation C et celle de l'implémentation assembleur.



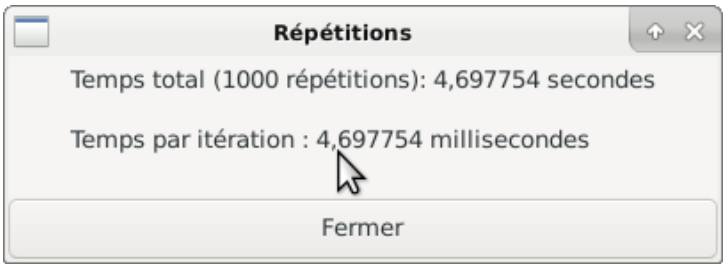
Implémentation de l'algorithme en C



Traitement avec calcul naïf de la valeur absolue (rupture de pipeline) (ASM)

Avec un calcul naïf (avec sauts) de la valeur absolue du gradient, nous remarquons ici un gain d'environ 9 millisecondes par itération ! Le code assembleur effectue le même traitement 2,6 fois plus rapidement que le code C. Il s'agit d'un gain non négligeable si le traitement est effectué de nombreuses fois (10 secondes gagnées pour 1000 itérations).

Cependant, comme énoncé précédemment, nous pouvons faire plus rapide car l'utilisation de sauts dans notre implémentation engendre des ruptures de pipeline.
Étudions la vitesse d'exécution du traitement avec un calcul de gradient sans rupture de pipeline.



Traitement avec calcul optimisé de la valeur absolue (sans rupture de pipeline) (ASM)

Une nouvelle fois, le code en langage assembleur est nettement plus rapide. Et encore plus qu'avec rupture de pipeline ! Pour environ 10 millisecondes de différence du temps d'exécution par itération, cela représente une exécution 3,2 fois plus rapide !

Notre objectif est donc rempli !

Conclusion

Pour conclure ce rapport, nous pouvons poser plusieurs constats. Le premier, et le plus frappant, est que nous nous sommes rendus compte que, bien que très performante, la compilation C n'est pas toujours équivalente et aussi efficace qu'un code en langage assembleur rédigé par le programmeur lui-même. La différence des temps d'exécution est significative et montre à quel point des optimisations peuvent être faites lorsque l'on traite un système critique nécessitant le temps d'exécution le plus rapide possible.
Deuxièmement, ce TP nous a permis, en plus du premier TP, de découvrir plus en profondeur les différentes instructions machines et de l'importance de bien les comprendre pour effectuer un code efficace et rapide. Au-delà d'un gain de temps d'exécution et de performance, cela permet également de réduire la consommation énergétique liée au traitement effectué. Il est évident, dans notre cas, qu'un gain de temps implique une réduction des ressources nécessaires.
Enfin, nous pensons avoir acquis de nouvelles compétences et connaissances nous permettant d'être plus à l'aise avec le langage assembleur et le traitement d'image.
Le résultat final a été très gratifiant, d'autant plus que le sujet du TP était très intéressant pour bien comprendre les différents aspects du langage assembleur évoqués au cours de ce rapport et dans cette même conclusion.