

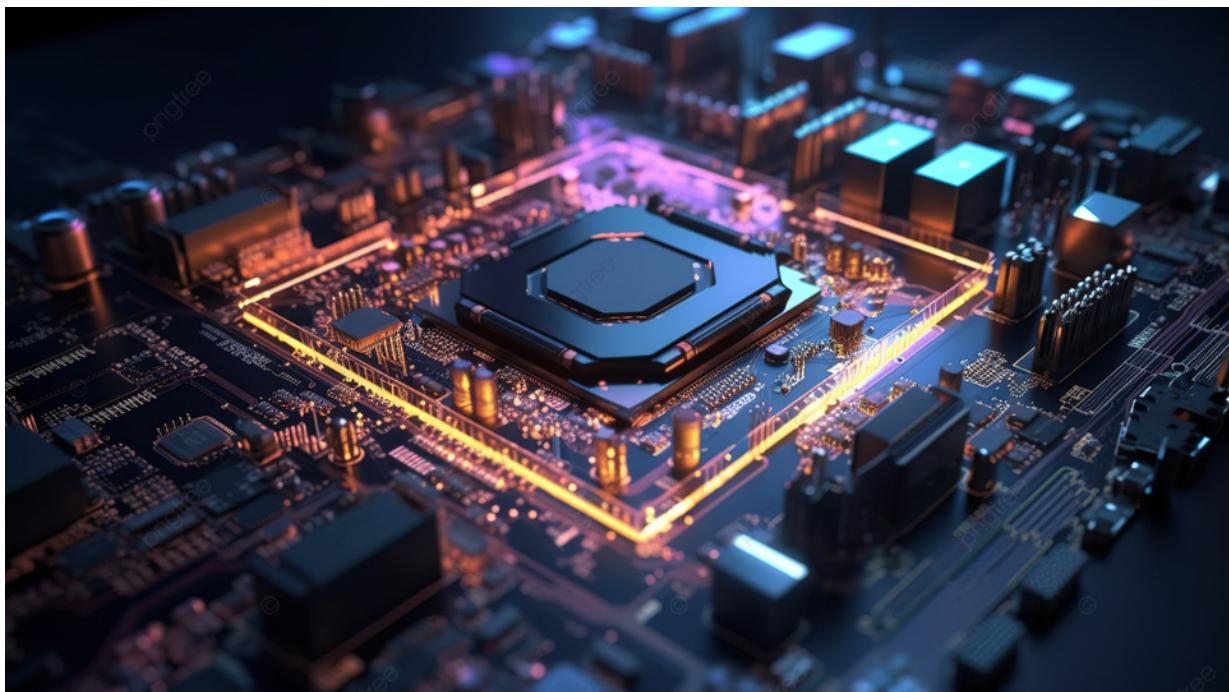
UNIVERSITÉ DE TECHNOLOGIE DE COMPIÈGNE

MI01 - STRUCTURE D'UN CALCULATEUR

Responsable
Mr. Shawky Marc

TP 5 À 7 – TRAITEMENT D'IMAGE

Responsable TP
Mr. Sanahuja Guillaume



TP1 - Groupe n°6

Sommaire

1	Introduction	4
2	Première partie : conversion en niveaux de gris	5
2.1	ITÉRATION SUR TOUS LES PIXELS DE L'IMAGE	5
2.1.1	QUESTION A - INTÉRÊT DE LA DÉCRÉMENTATION	5
2.1.2	QUESTION B - ADRESSE DU PIXEL EN COURS DANS IMG_SRC ET IMG_TEMP1	5
2.1.3	QUESTION C - AFFECTATION D'UNE COULEUR UNIE À CHAQUE PIXEL DE IMG_TEMP1	6
2.2	CALCUL DE L'INTENSITÉ D'UN PIXEL	7
2.2.1	QUESTION A - REPRÉSENTATION DU PRODUIT .	7
2.2.2	QUESTION B - REPRÉSENTATION BINAIRE ET HEXA- DÉCIMALE DES COEFFICIENTS	7
2.2.3	QUESTION C - DÉBORDEMENT DE L'INTENSITÉ .	8
2.2.4	QUESTION D - ALGORITHME DE CALCUL DE L'IN- TENSITÉ	8
2.3	CALCUL COMPLET	9
2.3.1	QUESTION A - IMPLÉMENTATION DE L'ALGORITHME EN ASSEMBLEUR	9
2.3.2	QUESTION B - PERFORMANCE ET COMPARAISON AVEC LE C	10
3	Deuxième partie : filtre de Sobel	11
3.1	CONSTRUCTION DE LA DOUBLE ITÉRATION	11
3.1.1	CALCUL DES ADRESSES SOURCE ET DESTINATION	11
3.2	ITÉRATION SUR LES LIGNES	11
3.2.1	QUESTION A - COMPTEUR DE LIGNES	11
3.2.2	QUESTION B - ÉVOLUTION DES POINTEURS DE PIXEL	11
3.2.3	QUESTION C - IMPLÉMENTATION DE LA BOUCLE D'ITÉRATION	12
3.3	ITÉRATION SUR LES COLONNES	12
3.3.1	QUESTION A - COMPTEUR DE COLONNES	12

3.3.2	QUESTION B - ÉVOLUTION DES POINTEURS DE PIXEL	12
3.3.3	QUESTION C - IMPLÉMENTATION DE LA BOUCLE D'ITÉRATION	12
3.4	TEST DES ITÉRATIONS	13
3.5	CALCUL DU GRADIENT DE CHAQUE PIXEL	15
3.5.1	QUESTION A - CALCULS D'ADRESSES	15
3.5.2	QUESTION B - IMPLÉMENTATION DU CALCUL DES GRADIENTS G_x et G_y	15
3.5.3	QUESTION C - VALEUR ABSOLUE	16
3.5.4	QUESTION D - VALEUR FINALE DU GRADIENT G	17
3.5.5	QUESTION E - IMPLÉMENTATION SUR CHAQUE PIXEL	18
3.6	FINALISATION DU PROGRAMME	20
3.7	COMPARAISON DES PERFORMANCES	21
4	Conclusion	22

Table des figures

1	Image source à traiter	4
2	Image tampon 1 avant traitement assebleur	6
3	Image tampon 1 après traitement assebleur	6
4	Structure mémoire niveaux de gris	9
5	Niveaux de rouge traité en assebleur	10
6	Implémentation ASM	10
7	Implémentation C	10
8	Image source après traitement	14
9	Image destination après traitement	14
10	Masques de convolutions	15
11	Image obtenue en niveaux de rouge avec implémentation du calcul de gradient	20
12	Image obtenue en niveaux de gris avec implémentation du calcul de gradient	21
13	Implémentation ASM	21
14	Implémentation C	21

1 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 parties qui composent le sujet. Une première partie qui 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 partie effectuée, nous pourrons commencer la dernière 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.

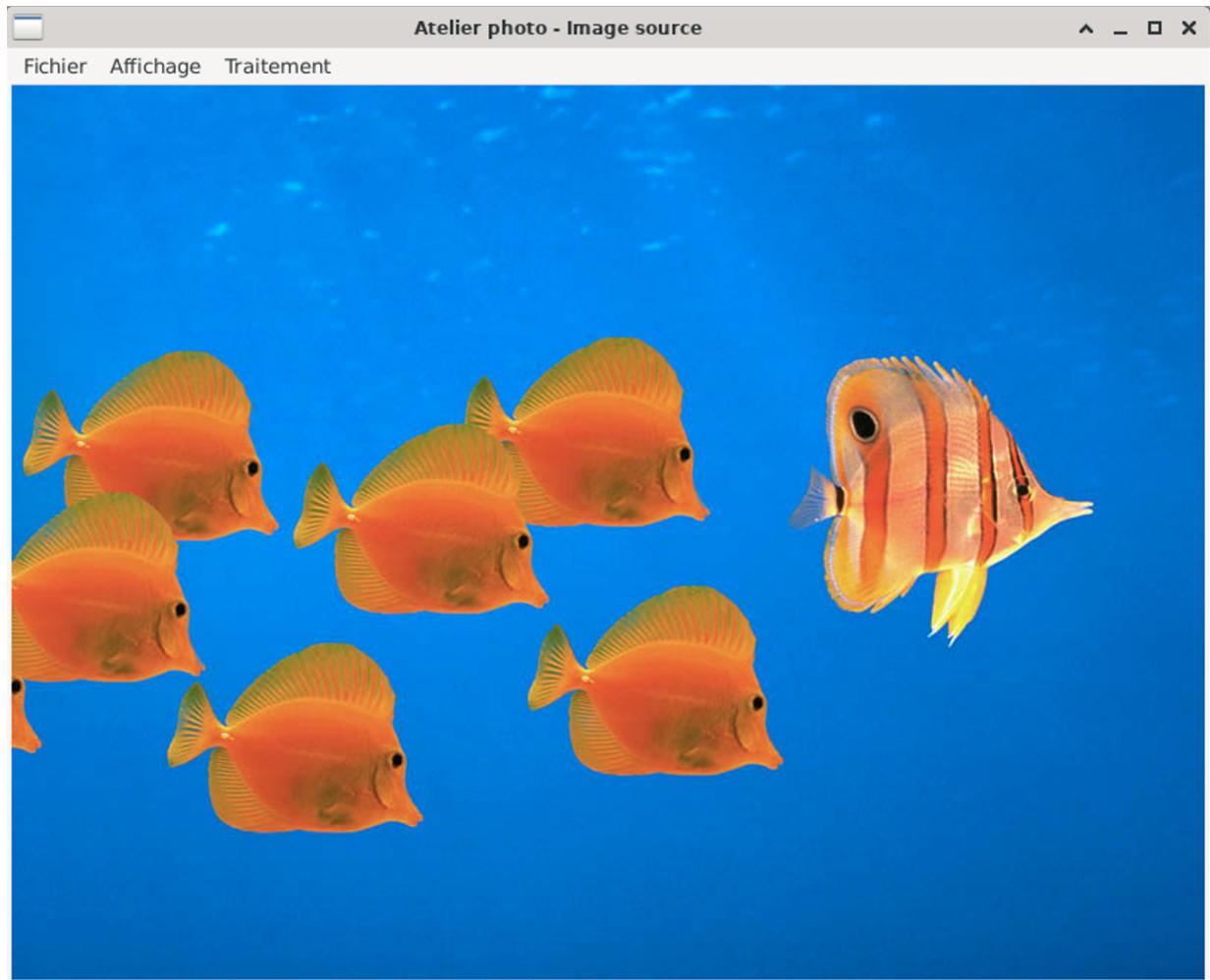


FIGURE 1 – Image source à traiter

2 Première partie : conversion en niveaux de gris

2.1 Itération sur tous les pixels de l'image

Dans le sous-programme process_image_asm, on nous donne une structure itérative qui parcourt tous les pixels de l'image. Dans cette structure, le registre *rdi* contient en permanence le nombre de pixels restant à traiter dans l'image. Quand il arrive à 0, on a terminé le traitement.

2.1.1 Question a - Intérêt de la décrémentation

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 meilleure option.

De plus, nous n'avons plus qu'à comparer le compteur à 0 plutôt qu'au nombre de pixels total puisque nous avons chargé qu'une seule fois le nombre de pixels dans la mémoire (la hauteur puis la largeur puis en faire le produit).

2.1.2 Question b - Adresse du pixel en cours dans img_src et img_temp1

Pour répondre à cette question nous rappelons que les références mémoires se font selon la syntaxe suivante :

$$[\text{rbase} + \text{rindex} * \text{ech} + \text{depl}],$$

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

De plus dans ce TP, nous savons à l'aide du programme process_image_asm que les registres :

- **rdi** correspond au nombre de pixel restants (longueur*largeur = nombre de pixels total),
- **rdx** correspond à l'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) et pour passer d'un pixel à un autre le déplacement est de -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 retrouverait 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**].

2.1.3 Question 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 de **img_temp1**, nous utiliserons le registre **rcx** correspondant au pointeur sur l'image tampon 1 (premier pixel).

Pour cela, nous ajoutons une instruction dans la boucle **loop_gs** du programme :

```
loop_gs:  
    sub     rdi, 1  
  
    /*ajout instruction*/  
    mov     dword ptr [rcx + rdi*4 - 4], 0xff0000ff  
  
    ja     loop_gs  
    pop    rdi  
    jmp    epilogue
```

Nous affectons sur nos pixels une couleur rouge 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. Nous obtenons le résultat suivant :

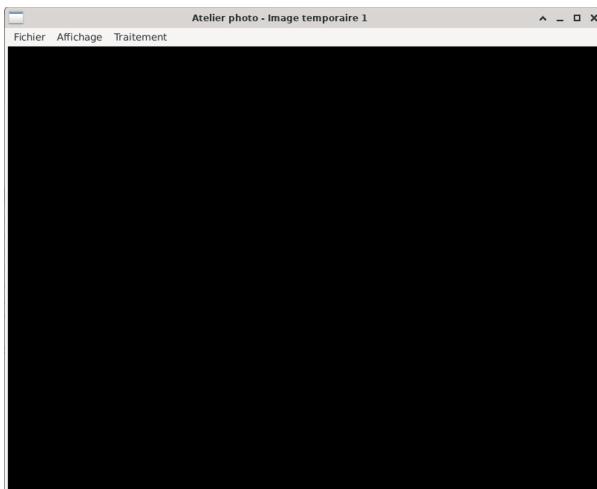


FIGURE 2 – Image tampon 1 avant traitement assebleur

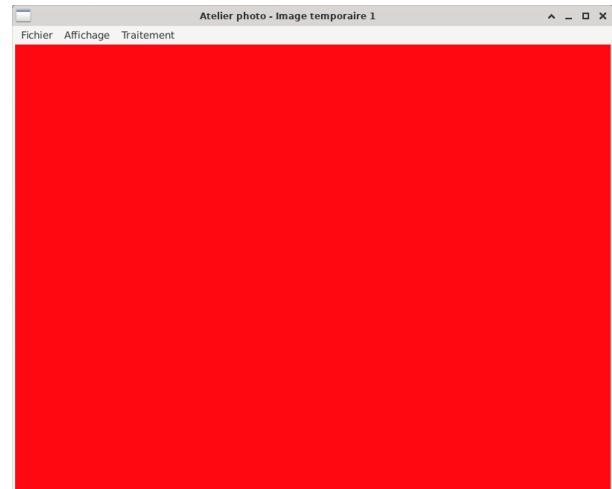


FIGURE 3 – Image tampon 1 après traitement assebleur

2.2 Calcul de l'intensité d'un pixel

2.2.1 Question a - Représentation du produit

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 composantes R,V,B, sont considérées 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 : $0, \underline{45} * 0, \underline{13} = 0, \underline{0585}$ ($2 + 2 = 4$)

Cela nous ramène à la règle de multiplication **entre un entier et un 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 $255 * 0,7152 \approx 182$, 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.

2.2.2 Question b - Représentation binaire et hexadécimale des coefficients

Voici une représentation binaire puis hexadécimale des trois coefficients constants utilisées dans le calcul sur 16 bits avec un décalage de +8 de la virgule :

- Coefficient **rouge** : $(0,2126)_{10} \iff (00000000,00110110)_2 \iff (00,36)_{16}$
- Coefficient **vert** : $(0,7152)_{10} \iff (00000000,10110111)_2 \iff (00,B7)_{16}$
- Coefficient **bleu** : $(0,0722)_{10} \iff (00000000,00010010)_2 \iff (00,12)_{16}$

Nous représentons ici la virgule dans l'écriture binaire et hexadécimale pour bien distinguer la partie entière et la partie décimale mais bien entendu ces-dernières s'écrivent sans la virgule dans la réalité.

Nous pouvons également remarquer que la représentation en base 10 est la seule exacte. Celles des bases binaire et hexadécimale sont tronquées pour pouvoir être représentées sur 16 bits dû à une trop petite taille de bits pour la partie décimale.

De plus, d'après la question c (question suivante), la somme des coefficients vaut 1. Dans notre cas :

$$00,B7 + 00,36 + 00,12 = 00,FF \neq 1,00$$

Pour conserver cette propriété, nous rajouterons 00,01 au coefficient bleu (00,12) (nous aurions pu ajouter à n'importe quel coefficient) . Ainsi nous faisons l'équivalence suivante :

$$(0,0722)_{10} \iff (00,13)_{16}$$

2.2.3 Question c - 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. On aura alors :

$$I = B * \text{coef}.b + R * \text{coef}.r + V * \text{coef}.v \leq I = 255 * (\text{coef}.b + \text{coef}.r + \text{coef}.v) = 255$$

Avec $(\text{coef}.b + \text{coef}.r + \text{coef}.v) = 1$

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.

2.2.4 Question d - Algorithme de calcul de l'intensité

Nous posons BS, VS, RS et T 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 :

$$\begin{aligned} R &\leftarrow (BS)_{16} * (00.13)_{16} \\ I &\leftarrow R \\ R &\leftarrow (VS)_{16} * (00.36)_{16} \\ I &\leftarrow I + R \\ R &\leftarrow (RS)_{16} * (00.B7)_{16} \\ I &\leftarrow I + R \end{aligned}$$

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. 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 <- c_rouge[p] * 0x0036
    R2 <- R1
    R1 <- c_vert[p] * 0x00B7
    R2 <- R2 + R1
    R1 <- c_bleu[p] * 0x0013
    R2 <- R2 + R1
    **** Utile pour la partie suivante ****
    transparence[p] <- 0xff
    c_rouge[p] <- partie_entiere(R2)
*****

```

Avec :

- $c_{\text{rouge}}[p]$, $c_{\text{vert}}[p]$ et $c_{\text{bleu}}[p]$ les composantes d'un pixel sur 16 bits avec extension de 0 sur 8 bits (même nombre de bits nécessaire pour la multiplication),
- $\text{transparence}(p)$ sur 8 bits,

- R1 et R2 des registres temporaires de 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 par exemple le registre rax en tant que R2, il suffira d'affecter l'octet du sous-registre ah à c_rouge[p] pour récupérer la valeur entière.

2.3 Calcul complet

2.3.1 Question a - Implémentation de l'algorithme en assembleur

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. L'accès aux composantes des pixel se fera alors selon les indices suivants :

Pixel n°rdi - 1	R	<code>byte ptr [rcx + rdi*4 - 4]</code>
	V	<code>byte ptr [rcx + rdi*4 - 3]</code>
	B	<code>byte ptr [rcx + rdi*4 - 2]</code>
	a	<code>byte ptr [rcx + rdi*4 - 1]</code>
Pixel n°rdi	R	<code>byte ptr [rcx + rdi*4]</code>
		...

FIGURE 4 – Structure mémoire niveaux de gris

Voici l'implémentation en assembleur selon l'algorithme précédent et cette structure mémoire :

- 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 avec extension également :

```
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 rouge :

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

- On additionne $V * coeff.v + B * coeff.b$, 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 /*opacite maximale*/
mov byte ptr [rcx + rdi*4 - 4], ah /*valeur de I dans la
composante rouge*/
```

Après ce traitement en assembleur, nous obtenons l'image suivante :

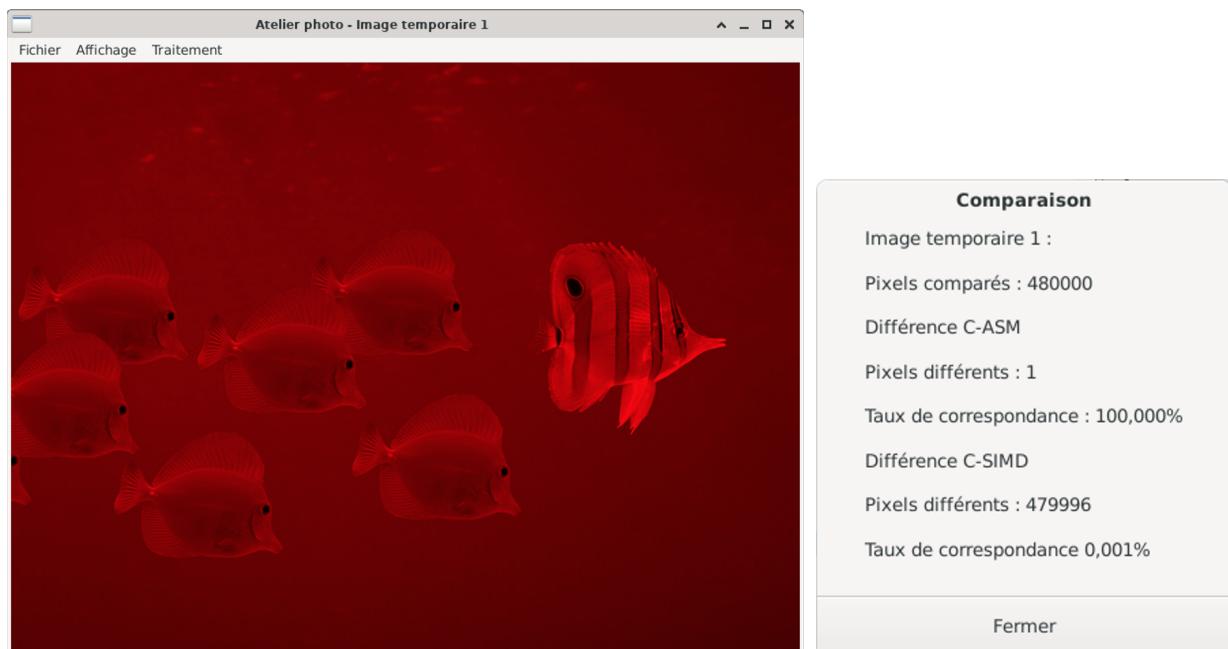


FIGURE 5 – Niveaux de rouge traité en assembleur

2.3.2 Question b - Performance et comparaison avec le C



FIGURE 6 – Implémentation ASM



FIGURE 7 – Implémentation C

Après exécution du code sur la même image, sur 1000 répétitions pour que ce soit plus significatif, nous nous apercevons que le code assembleur est nettement plus performant que le code C : environ 1.6¹ fois plus rapide ! Et ce, pour un résultat équivalent (taux de correspondance = 100%).

1. les valeurs peuvent changer en raison d'autres facteurs comme la mémoire

3 Deuxième partie : filtre de Sobel

3.1 Construction de la double itération

3.1.1 Calcul des adresses source et destination

Avant de calculer les adresses source et destination, nous faisons le rappel suivant :

- 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 r11 et r10.

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, nous obtenons ces deux commandes :

```
lea      r11, [rcx]          /*pointeur source*/
lea      r10, [r8 + rdi*4 + 4] /*pointeur destination*/
```

3.2 Itération sur les lignes

3.2.1 Question a - Compteur de lignes

Sachant que les lignes des bords ne seront pas traitées, il suffit de prendre la valeur de la hauteur (stockée dans le registre rsi) moins 2 (première ligne + dernière ligne). Ce compteur pourra donc être initialisé à valeur(rsi) - 2. Ainsi, nous obtenons ces deux commandes :

```
push    rsi      /*compteur de ligne*/
sub     rsi,2    /*elimination des bords*/
```

3.2.2 Question b - Évolution 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 pointerà vers $4 + 4 = 8$ octets plus loin. Soit :

```
lea    r11, [r11 + 8]
```

De la même manière pour le pointeur du pixel de destination :

```
lea    r11, [r11 + 8]
```

3.2.3 Question c - Implémentation de la boucle d'itération

À l'aide des informations et initialisation précédentes, nous obtenons la boucle suivante pour parcourir les lignes :

```
push    rsi      /*compteur de ligne*/
sub     rsi,2    /*elimination des bords*/

loop_ligne:
lea     r11, [r11+8]  /*evolution pointeur source*/
lea     r10, [r10 + 8] /*evolution pointeur destination*/
sub   rsi, 1 /*decrementation du nombre de lignes restantes*/
jg    loop_ligne /*nombre de lignes restantes >0 on boucle*/
pop   rsi       /*compteur de lignes */
```

3.3 Itération sur les colonnes

3.3.1 Question a - Compteur de colonnes

Sachant que les colonnes des bords ne seront pas traitées, il suffit 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(sauvegarde à effectuer dans loop_ligne pour pouvoir récupérer la hauteur à chaque saut de ligne).

Ainsi, nous obtenons ces deux commandes :

```
loop_ligne :
push    rdi      /*compteur de colonne*/
sub     rdi, 2    /*elimination des bords*/
...
```

Remarque : cette étape sera modifiée par la suite, pour les besoins du traitement de l'image (voir : 3.5.2).

3.3.2 Question b - Évolution 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     r11, [r11 + 4]
```

De la même manière pour le pointeur du pixel de destination :

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

3.3.3 Question c - Implémentation de la boucle d'itération

À l'aide des informations et initialisation précédentes, nous obtenons la boucle suivante pour parcourir les colonnes ; imbriquées dans celles des lignes :

```
push    rsi      /*compteur de ligne*/
sub     rsi,2    /*elimination des bords*/

loop_ligne:
push    rdi      /*compteur de colonne*/
sub     rdi, 2    /*elimination des bords*/
```

```

loop_colonne :
    lea    r11, [r11+4]      /*pixel suivant sur la colonne
    lea    r10, [r10 + 4]      ( incrementation) */

    sub    rdi, 1           /*decrementation du nombre de colonnes
                               restantes*/
    jg     loop_colonne    /*nombre de colonnes restantes >0
                               on boucle*/
pop    rdi                 /*compteur de colonnes */

lea    r11, [r11+8]
lea    r10, [r10 + 8]      /* partie pour parcourir
sub   rsi, 1               les lignes */
jg    loop_ligne
pop    rsi

```

3.4 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 en haut et en bas de pixels noirs et d'une colonne à droite et à gauche de pixels noirs. Pour effectuer ce test voici le code complet implémenté :

```

...
lea    r11, [rcx]          /*pointeur source*/
lea    r10, [r8 + rdi*4 + 4] /*pointeur destination*/

push   rsi
sub   rsi,2

loop_ligne:
push   rdi
sub   rdi, 2

loop_colonne:
    mov dword ptr [r11], 0xff0000ff /*teste de la boucle
    mov dword ptr [r10], 0xff0000ff affectation de couleur*/

    lea    r11, [r11+4]
    lea    r10, [r10 + 4]
    sub   rdi, 1
    jg    loop_colonne
pop    rdi

lea    r11, [r11+8]
lea    r10, [r10 + 8]
sub   rsi, 1
jg    loop_ligne
pop    rsi
jmp    epilogue
...
```

Après avoir exécuté ce code, nous obtenons les images suivantes :



FIGURE 8 – Image source après traitement

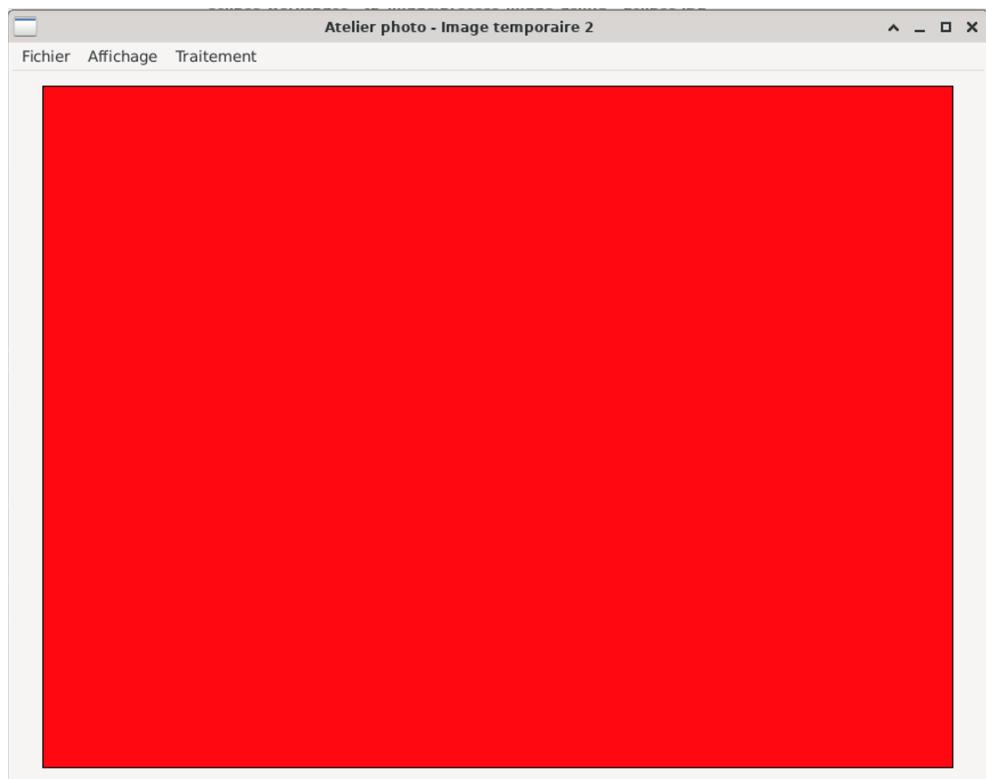


FIGURE 9 – Image destination après traitement

Les images obtenues traitées sont bien celles attendues. En effet, nous pouvons voir les bordures différentes sur les deux.

3.5 Calcul du gradient de chaque pixel

3.5.1 Question a - Calculs d'adresses

$a11$ 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 $a21$, $a12$ et $a13$. Calculer $a21$ 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 : $a21 = rcx + 4$, car on saute de 4 octets pour passer au prochain pixel.

Pour calculer $a12$, 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 : $a12 = rcx + rdi * 4$ (rappel : le facteur d'échelle vaut 4 car un pixel est stockée sur 4 octets).

Enfin, le calcul de $a13$ est identique à celui de $a12$. Il suffira simplement de sauter 2 lignes si on le calcul par rapport à $a11$: c'est-à-dire $2 * (rdi * 4)$ soit $rdi * 8$.

3.5.2 Question 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).

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

FIGURE 10 – Masques de convolutions

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 `r11` (c.f partie 3.3.2)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 `r12`, qui prendra la valeur de `rdi` avant l'entrée dans les boucles des itérations puisque `rdi` est modifiée dans celles-ci.

Ainsi, l'étape expliquée en 3.3.1 devient alors :

```

...
sub      rsi, 2
mov      r12, rdi //colonnes restantes

loop_ligne:
    //traitement des colonnes
    mov      rdi, r12
    sub      rdi, 2
...

```

Nous stockerons G_x dans dx ainsi que G_y 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 Sx)
sub ax, word ptr [r11]           // a11 * (-1)
add ax, word ptr [r11 + 8]        //+ a31 * 1
sub ax, word ptr [r11 + r12*4]    //+ a12 * (-2)
sub ax, word ptr [r11 + r12*4]
add ax, word ptr [r11 + r12*4 + 8] //+ a32 * 2
add ax, word ptr [r11 + r12*4 + 8]
sub ax, word ptr [r11 + r12*8]    //+ a13 * (-1)
add ax, word ptr [r11 + r12*8 + 8] //+ a33 * 1

//Calcul de G_y (masque Sy)
add bx, word ptr [r11]           // a11 * 1
add bx, word ptr [r11+4]          //+ a21 * 2
add bx, word ptr [r11+4]
add bx, word ptr [r11+8]          //+ a31 * 1
sub bx, word ptr [r11 + r12*8]    //+ a13 * (-1)
sub bx, word ptr [r11 + r12*8 + 4] //+ a23 * (-2)
sub bx, word ptr [r11 + r12*8 + 4]
sub bx, word ptr [r11 + r12*8 + 8] //+ a33 * (-1)
```

3.5.3 Question c - Valeur absolue

Pour calculer la valeur absolue de G_x et G_y, on pourrait procéder 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:
...
```

Mais nous cherchons 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 :

$$\begin{aligned} x \text{ xor } 1 &\rightarrow \text{not } x \\ x \text{ xor } 0 &\rightarrow x \\ \\ \text{not } x - (-1) &\rightarrow -x \\ x - (-1) &\rightarrow x \\ \\ \Rightarrow (x \text{ xor } 1) - (-1) &\rightarrow -x \\ (x \text{ xor } 0) - (-1) &\rightarrow x \end{aligned}$$

Si notre x est négatif dx sera égale à 0xFFFF qui est - 1 en complément à 2.

$xor ax, dx$ nous permet d'avoir $\text{not } ax$ et enfin $\text{not } ax - (-1) = -ax$.

Si x est positif dx sera égale à 0x0000 et donc $ax \text{ xor } 0 - (-1)$ 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 r14`. Ce sous-registre stockera la valeur absolue de G_x .

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

```
// Valeur abs(G_x)
CWD          // dx:ax <- ax avec ext. de signe
xor ax, dx
sub ax, dx

mov r14w, ax

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

3.5.4 Question d - Valeur finale du gradient G

Nous calculons la valeur finale du gradient G selon l'algorithme de traitement :

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

Premièrement, effectuons la somme des gradients G_x (stocké dans r14w) et G_y (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 lui affecte la valeur 0 :

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

Nous obtenons ainsi comme calcul final de G :

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

3.5.5 Question 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 [r10], al
```

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

```
***** Detecteur de contours de Sobel *****
*****push    rsi
*****push    rbx
*****push    r12
*****push    r14

*****lea     r11, [rcx]  /*pointeur source*/
*****lea     r10, [r8 + rdi*4 +4] /*pointeur destination*/

*****sub    rsi,2
*****mov    r12, rdi //colonnes restantes

loop_ligne:
//traitement des colonnes

*****mov    rdi,r12
*****sub    rdi, 2

loop_colonne:
*****xor    rax, rax
*****xor    rdx, rdx
*****xor    rbx, rbx
```

```

// Calcul de G_x ( masque Sx)
sub ax, word ptr [r11] // a11 * (-1)
add ax, word ptr [r11 + 8] //+ a31 * 1
sub ax, word ptr [r11 + r12*4] //+ a12 * (-2)
sub ax, word ptr [r11 + r12*4]
add ax, word ptr [r11 + r12*4 + 8] //+ a32 * 2
add ax, word ptr [r11 + r12*4 + 8]
sub ax, word ptr [r11 + r12*8] //+ a13 * (-1)
add ax, word ptr [r11 + r12*8 + 8] //+ a33 * 1

//Calcul de G_y (masque Sy)
add bx, word ptr [r11] // a11 * 1
add bx, word ptr [r11+4] //+ a21 * 2
add bx, word ptr [r11+4]
add bx, word ptr [r11+8] //+ a31 * 1
sub bx, word ptr [r11 + r12*8] //+ a13 * (-1)
sub bx, word ptr [r11 + r12*8 + 4] //+ a23 * (-2)
sub bx, word ptr [r11 + r12*8 + 4]
sub bx, word ptr [r11 + r12*8 + 8] //+ a33 * (-1)

// Valeur abs(G_x)
CWD // dx:ax <- ax avec ext. de signe
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 [r10], al
mov byte ptr [r10+3], 0xff

lea r11, [r11+4]
lea r10, [r10 + 4]
sub rdi, 1
jg loop_colonne

lea r11, [r11+8]

```

```

    lea    r10, [r10 + 8]
    sub    rsi, 1
    jg     loop_ligne

epilogue:
    pop   r14
    pop   r12
    pop   rbx
    pop   rsi
    pop   rbp    # Dépiler le pointeur de cadre de pile sauvegarde
    ret    # Retour à l'appelant

```

Nous obtenons ainsi l'image suivante :

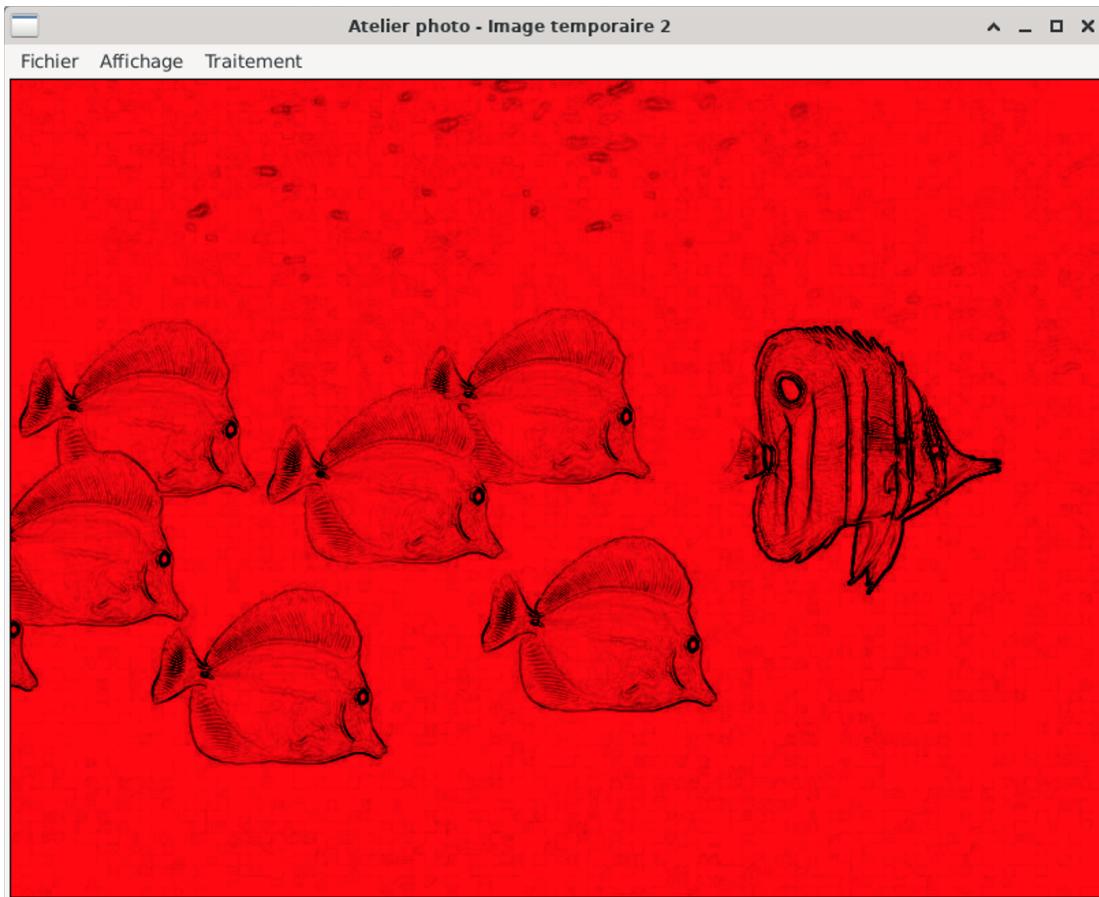


FIGURE 11 – Image obtenue en niveaux de rouge avec implémentation du calcul de gradient

3.6 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 [r10], al
    mov    byte ptr [r10 + 1], al
    mov    byte ptr [r10 + 2], al
    mov    byte ptr [r10+3], 0xff      // Opacité maximale

```

Nous obtenons comme image en niveaux de gris :

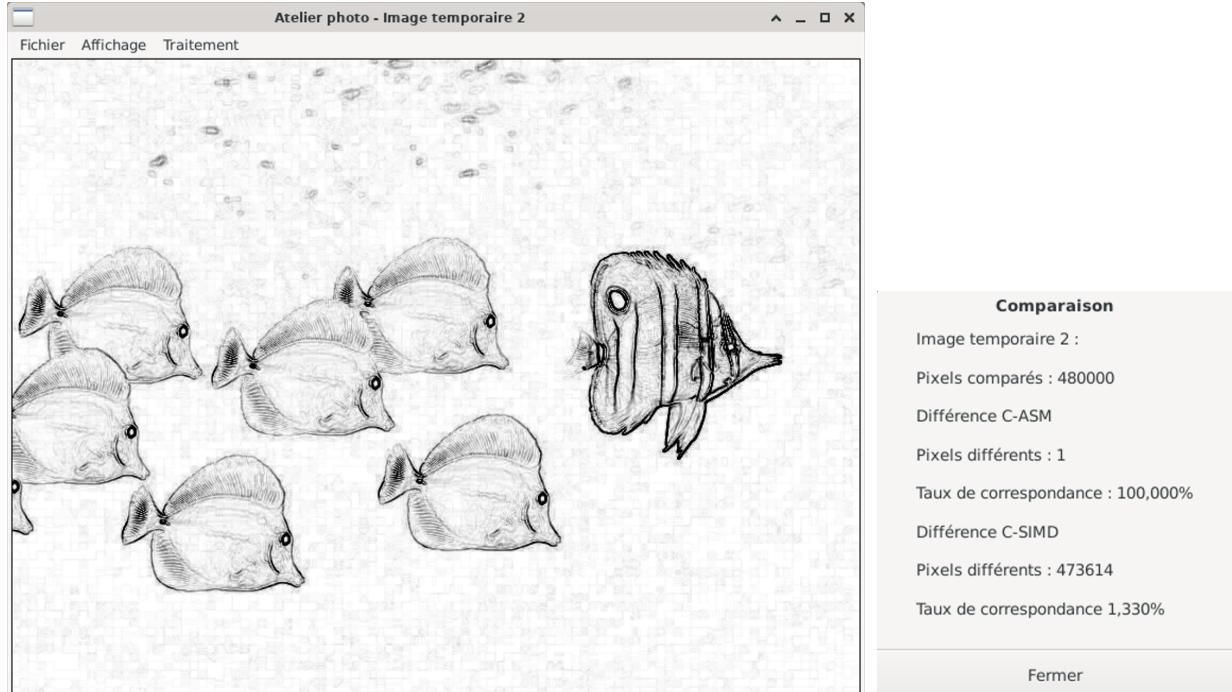


FIGURE 12 – Image obtenue en niveaux de gris avec implémentation du calcul de gradient

3.7 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 :



FIGURE 13 – Implémentation ASM



FIGURE 14 – Implémentation C

Après exécution du code sur la même image, sur 1000 répétitions pour que ce soit plus significatif, nous nous apercevons que le code assembleur est nettement plus performant que le code C : environ 2.5² fois plus rapide! Et ce, pour un résultat équivalent (taux de correspondance = 100%).

Notre objectif est donc rempli !

2. les valeurs peuvent changer en raison d'autres facteurs comme la mémoire

4 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.