Victor Demessance - Daniel Bradley

COMPTE RENDU

MI01 - TP IMAGE

Sommaire :

5. Première partie : conversion en niveaux de gris

5.1 Travail à réaliser :

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 permet de tester la nullité de l'index plutôt que de comparer avec la taille de l'image. Au niveau assembleur, cela permet d'éviter une ligne de code (cmp 0) car la décrémentation à 0 active directement les drapeaux.

b. L'adresse de l'image à traiter est contenue dans le registre *rdx*. Quant au registre *rdi*, il représente l'index décroissant du nombre de pixel à encore traiter. Rien ne nous empêche de traiter l'image en commençant par la fin. De plus, la taille d'un pixel prend en mémoire 4 octets. Le pas est donc de 4 octets. Ainsi, l'adresse du pixel en cours est dans [*rdx*+4**rdi*].

c. **mov eax, [rdx+4*rdi] //Addresse du pixel traité**
and eax, 0xff00ffff //Masque de test
mov [rcx+4*rdi], eax //On enregistre le pixel traité dans l'image tampon 1

5.1.2 Calcul de l'intensité d'un pixel :

a. On multiplie une valeur représentée sur 8 bits par un coefficient représenté sur 16 bits (avec 8 bits flottant). On peut donc effectuer une multiplication de deux valeurs sur 16 bits et ainsi obtenir le résultat dans un registre 32 bits. Le décalage sera de 8 bits flottant également.

b. **Cr= 0,2126 ~= 00000000,00110110 = 00,36**
Cv= 0,7152 ~=00000000,10110111 = 00,B7
Cb= 0,0722 ~=00000000,00010010 =00,12

c. Ce n'est pas possible que le résultat déborde de 16 bits si le programme est bien écrit (bon coefficient dont la somme est inférieure à 1 et bonne gestion des calculs et des décalages). Un débordement n'est pas voulu car il empêcherait le traitement de l'image, sachant que la valeur (même après décalage de 8 bits) serait trop grande en mémoire et occuperait la place d'une autre couleur. Ainsi, l'intensité ne serait pas seulement lu en 1 fois et son poids fort serait lu dans la partie dans le V alors que le poids faible dans le R (par exemple)

d. L'idée générale du calcul de I est de poser des masques sur des registres contenant une copie du pixel à traiter. Ainsi, l'on peut calculer les différents coefficient indépendant puis les sommer pour obtenir le I du pixel traité. (*L'algo assembleur est disponible ci-dessous)*

5.1.3 Calcul complet :

a. Programme :

```
.file "process_image_asm.S"
.intel_syntax noprefix
.text
.global process_image_asm
process_image_asm:
        push    rbp
        mov     rbp, rsp
        push rbx                    //On utilise ebx, on sauvegarde donc le registre
        push    rdi
        imul    rdi, rsi            //rdi = largeur x hauteur

    loop_gs:
        mov eax, [rdx+4*rdi]        //Adresse du pixel à traiter
        xor r11d, r11d              //Mise à 0 de r11d car il est utilisé comme registre intermédiaire
        mov ebx, eax                //On copie le pixel pour le traiter plusieurs fois
        and ebx, 0x000000ff         //Masque de rouge
        imul ebx, 0x36              //Multiplication par le coefficient rouge
        add r11b, ebx               //Stockage de la valeur dans le registre r11b
        mov ebx, eax                //On copie le pixel pour le traiter plusieurs fois
        and ebx, 0x0000ff00         //Masque de vert
        shr ebx, 8                  //Décalage pour traitement de l'octet "vert"
        imul ebx, 0xB7              //Multiplication par le coefficient vert
        add r11b, ebx               //Addition de la valeur dans le registre r11b
        mov ebx, eax                //On copie le pixel pour le traiter plusieurs fois
        and ebx, 0x00ff0000         //Masque de bleu
        shr ebx, 16                 //Décalage pour traitement de l'octet "bleu"
        imul ebx, 0x10              //Multiplication par le coefficient bleu
        add r11b, bl                //Addition de la valeur dans le registre r11b
        shr r11d, 8                 //Décalage pour suppression des bits flottants
        //shl r11d, 8               //Filtre vert
        //shl r11d, 8               //Filtre bleu
        or r11b, 0xff000000         //Masque de transparence
        mov [rcx+4*rdi], r11d       //On enregistre le pixel traité dans l'image tampon 1
        sub     rdi, 1              //Un pixel de moins à traiter
        ja      loop_gs
        pop     rdi
        jmp     epilogue

epilogue:
        pop rbx                     //Dépiler rbx
        pop     rbp                 //Dépiler le pointeur de cadre de pile sauvegardé
        ret
```

Exécution du programme assembleur

Exécution du programme C

On voit bien que le programme assembleur est bien plus efficace que le programme C, du moins en prenant en compte la vitesse d'exécution.

6 Deuxième partie : filtre de Sobel

6.1 Détection des contours d'une image

6.1.1 Principe

Les contours caractérisent les frontières des objets à l'intérieur d'une image et sont d'une importance fondamentale en termes de traitement d'images. Ils sont définis par les zones d'une image contenant de fortes variations de contraste – un saut d'intensité entre un pixel et ses voisins. Détecter les contours d'une image réduit de manière significative la quantité de données et élimine l'information inutile, tout en préservant les structures importantes de l'image. Les algorithmes de détection de contour se divisent principalement en deux catégories : les méthodes basées sur l'analyse du gradient de l'image, et les méthodes basées sur l'analyse de son laplacien. L'algorithme que vous allez utiliser dans ce TP est du premier type. Les méthodes basées sur le gradient détectent les contours en trouvant les minima et maxima de la dérivée première de l'image. En une dimension, on peut assimiler un contour à une rampe et calculer sa dérivée permet de déterminer son emplacement. Supposons un signal unidimensionnel (Figure 4) qui comprend un contour caractérisé par un saut d'intensité (on peut imaginer qu'il s'agit d'une ligne de l'image). Le gradient de ce signal qui en une dimension en est simplement sa dérivée, comporte clairement un maximum situé au centre du contour.

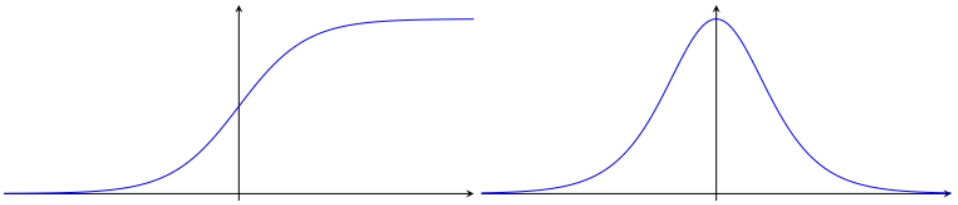


Figure 4 – Un signal (à gauche) et sa dérivée (à droite).

6.1.2 Opérateur de Sobel

Cette technique de détection peut être facilement étendue en deux dimensions pour traiter des images à condition d'avoir une approximation précise du gradient dans l'image. L'opérateur de Sobel réalise une mesure en deux dimensions du gradient de l'intensité d'une image, permettant de trouver sa norme en tout point. Cet opérateur se présente sous la forme de deux masques de convolution S_x et S_y (Équation 1) qui, appliqués à l'image comme expliqué dans le paragraphe suivant, fournissent pour chaque pixel la norme G_x du gradient dans la direction x (colonnes) et la norme G_y du gradient dans la direction y (lignes). Les masques étant bien plus petits que l'image, on les applique en les décalant d'un pixel à chaque fois pour parcourir toute l'image.

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

La norme G du gradient d'un pixel est calculée à partir de G_x et G_y comme suit :

$$G = \sqrt{{G_x}^2 + {G_y}^2}$$

qu'on peut approximer par :

$$G = |G_x| + |G_y|$$

6.1.3 Application d'un masque de convolution à une image

Le masque à appliquer glisse le long de l'image de départ pour calculer la valeur d'un pixel de l'image résultante, puis est décalé à droite d'un pixel, pour chaque pixel d'une ligne. À la fin de la ligne, le traitement recommence au premier pixel de la ligne suivante et ainsi de suite jusqu'à la fin de l'image. Le pixel pour lequel le produit de convolution est calculé est celui qui se situe au centre du masque, ses huit voisins contribuant au

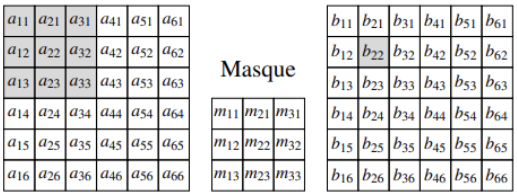


Figure 5 – Application d'un masque de convolution à un pixel.

résultat. Ainsi, dans l'exemple présenté en figure 5, on veut appliquer le masque au pixel a_{22} de l'image de départ.

La valeur du pixel b_{22} dans l'image résultante est alors :

$$b_{22} = a_{11}m_{11} + a_{12}m_{12} + a_{13}m_{13} + a_{21}m_{21} + a_{22}m_{22} + a_{23}m_{23} + a_{31}m_{31} + a_{32}m_{32} + a_{33}m_{33}$$

On constate que les pixels de la première et de la dernière ligne, ainsi que de la première colonne et de la dernière colonne ne peuvent être manipulés par un masque de taille 3x3. En effet, quand on place le centre du masque sur un pixel de la première ligne par exemple, une partie du masque se trouve en dehors des limites de l'image de départ, ce qui rend le calcul impossible. Dans le cadre de ce TP, ces pixels seront simplement ignorés.

6.2 Algorithme de traitement

L'algorithme de traitement consiste à parcourir toute l'image source et à calculer pour chaque pixel la valeur G du gradient total qui sera stockée dans l'image de destination. Cette valeur est un entier entre 0 et 255, 255 représentant les zones de gradient maximal. Cette valeur représentant aussi l'intensité maximale, l'affichage direct du gradient aboutirait à une image des contours en blanc sur fond noir. Puisqu'on veut les afficher en noir sur fond blanc, il faut inverser l'intensité avant de stocker la valeur dans l'image de destination.

```
1  source ← adresse du premier pixel de l'image source
2  dest ← adresse du second pixel de la seconde ligne de l'image de destination
3
4  pour chaque ligne i de l'image source
5      pour chaque colonne j de l'image source
6          G_x ← convolution du masque S_x avec les 9 pixels àl'adresse source
7          G_y ← convolution du masque S_y avec les 9 pixels àl'adresse source
8          G ← |G_x| + |G_y|
9          G ← 255 − G
10         si G < 0
11             G ← 0
12         fin si
13         pixel àl'adresse dest ← G
14         source ← adresse du pixel suivant dans l'image source
15         dest ← adresse du pixel suivant dans l'image dest
16     fin pour
17 fin pour
```

La figure 6 illustre le parcours de l'image. On observe que sur l'image source, le dernier masque utilisé pour calculer la valeur de la convolution du pixel $(n-2, m-2)$ est positionné au pixel de coordonnées $(n-3, m-3)$, en supposant une image de $n \times m$ pixels et que le premier pixel a pour coordonnées $(0,0)$.

6.3 Travail à réaliser

6.3.1 Construction de la double itération

Lignes :

a) Comme expliqué dans la méthode, on ne peut pas traiter les pixels en bord d'image (on traite 2 lignes en moins). Le compteur de ligne est donc rsi - 2 (rsi représentant la hauteur de l'image).

b) A la fin d'une ligne, le pixel source pointe sur l'avant avant dernier pixel de la ligne. Il doit pointer sur le premier pixel de la ligne suivante, on incrémente donc de 2 (pixels).
Le pixel de destination, de la même manière, s'incrémente de 2 afin de passer de l'avant dernier pixel au deuxième pixel de la ligne suivante.

c) Code disponible après la partie colonnes

Colonnes :

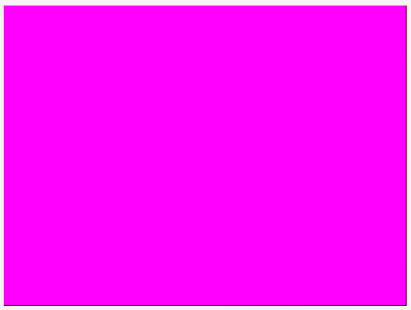
a) Comme expliqué dans la méthode, on ne peut pas traiter les pixels en bord d'image (on traite 2 colonnes en moins). Le compteur de ligne est donc rsi - 2 (rsi représentant la hauteur de l'image).

b) A la fin de chaque colonne, le pixel source doit évoluer afin de permettre le traitement du pixel de la colonne suivante, il sera donc incrémenté de 1. De la même manière, le pixel de destination sera incrémenté de 1.

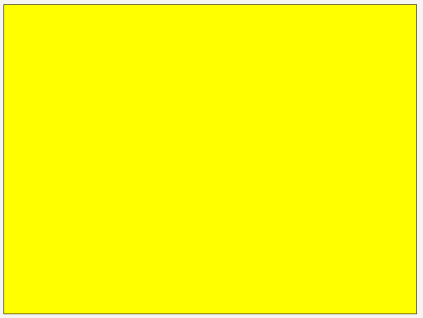
c) Code pour la partie ligne et colonne :

```
//Filtre de Sobel
        push rcx                            //Sauvegarder le pointeur de l'image tampon2
        lea r11, [rsi-2]                    //Nombre de lignes
lignes:
        lea r10, [rdi-2]                    //Nombre de colonnes
colonnes:
        //Traitement
        mov dword ptr[rcx], 0xffff00ff      //Test du parcours source
        mov dword ptr[r8+4*rdi+4], 0xff00ffff //Test du parcours destination
        //Fin traitement
        add r8, 4                           //Décalage d'un pixel du pointeur destination
        add rcx, 4                          //Décalage d'un pixel du pointeur source
        sub r10, 1                          //Une colonne de moins à traiter
        ja colonnes                         //Traitement de la colonne suivante s'il en reste
        add r8, 8                           //Décalage de 2 pixels du pointeur destination
        add rcx, 8                          //Décalage de 2 pixels du pointeur source
        sub r11, 1                          //Une ligne de moins à traiter
        ja lignes                           //Traitement de la ligne suivante s'il en reste
        pop rcx
        pop rsi
        pop rdi
```

Test des itérations :



image_source



image_destination

6.3.2 Calcul du gradient de chaque pixel

a) On utilisera rcx, le registre pointant sur le pixel a11 a chaque itération de traitement. Grâce à ce pointeur, on peut obtenir les adresses de chacune des valeurs de la matrice :

[rcx] = a11
[rcx + 4] = a12
[rcx + 8] = a13
[rcx + 4*rdi] = a21
[rcx + 4*rdi + 4] = a22
[rcx + 4*rdi + 8] = a23
[rcx + 4*rdi*2] = a31
[rcx + 4*rdi*2 + 4] = a32
[rcx + 4*rdi*2 + 8] = a33

b) Cf programme complet

c) Pour faire la valeur absolue d'un registre, on doit utiliser la valeur de celui-ci. Si elle est négative, on lui assigne le négatif de lui-même afin d'obtenir sa valeur absolue. Pour éviter d'utiliser des registres, on utilise un cmov avec une condition après une opérande de calcul sur le registre concerné.
Cf programme complet

d) On additionne |Gx|+|Gy| puis on le multiplie par -1 en lui additionnant également 255. Après cela, on vérifie que la valeur ne déborde pas en utilisant un nouveau cmov permettant de mettre à zéro toute valeur négative.
Cf programme complet

6.3.3 Finalisation du programme

Une fois G calculé, on le copie dans chacun des modules d'un pixel en mettant la transparence à 0xff. Cette duplication permettra d'avoir les contours de l'image en nuance de gris.
Cf programme complet

Programme final :

```
.file "process_image_asm.S"
.intel_syntax noprefix
.text
/*****************************************************************
        Largeur de l'image : rdi
        Hauteur de l'image : rsi
        Pointeur sur l'image source : rdx
        Pointeur sur l'image tampon 1 : rcx
        Pointeur sur l'image tampon 2 : r8
        Pointeur sur l'image finale : r9
*****************************************************************/
.global process_image_asm
process_image_asm:
        push    rbp
        mov     rbp, rsp
        push rbx        //On utilise ebx, on sauvegarde donc le registre
        push r15
        push r14
        push r12
        push r13
        # Calcul du nombre de pixels de l'image dans rdi.
        push    rdi         # Sauvegarde de la largeur de l'image
        imul    rdi, rsi     # rdi <- largeur x hauteur
loop_gs:
        mov eax, [rdx+4*rdi]                //Adresse du pixel traité
        xor r11d, r11d
        mov ebx, eax                        //On copie le pixel pour le traiter plusieurs fois
        and ebx, 0x000000ff                 //Masque de rouge
        imul ebx, 0x36                      //Multiplication par le coefficient correspondant
        add r11d, ebx                       //Stockage de la valeur dans le registre rbx
        mov ebx, eax                        //On copie le pixel pour le traiter plusieurs fois
        and ebx, 0x0000ff00                 //Masque de vert
        shr ebx, 8                          //Décalage pour traitement
        imul ebx, 0xB7                      //Multiplication par le coefficient correspondant
        add r11d, ebx                       //Addition de la valeur dans le registre rbx
        mov ebx, eax                        //On copie le pixel pour le traiter plusieurs fois
        and ebx, 0x00ff0000                  //Masque de bleu
        shr ebx, 16                         //Décalage pour traitement
        imul ebx, 0x13                       //Multiplication par le coefficient correspondant
        add r11d, ebx                        //Addition de la valeur dans le registre rbx
        shr r11d, 8                         //Décalage pour suppression des 8 bits après la virgule
        //shl r11d, 8                       //Filtre d'intensité vert
        //shl r11d, 8                        //Filtre d'intensité bleu
        or r11d, 0xff000000                  //Masque de transparence
        mov [rcx+4*rdi], r11d               //On enregistre le pixel traité dans l'image tampon 1
        sub     rdi, 1                      //Un pixel de moins à traiter
        ja      loop_gs
        pop     rdi                          //rdi <- largeur de l'image en pixels

//Filtre de Sobel
        push rcx                             //Sauvegarder le pixel pointant sur l'image tampon 2
```

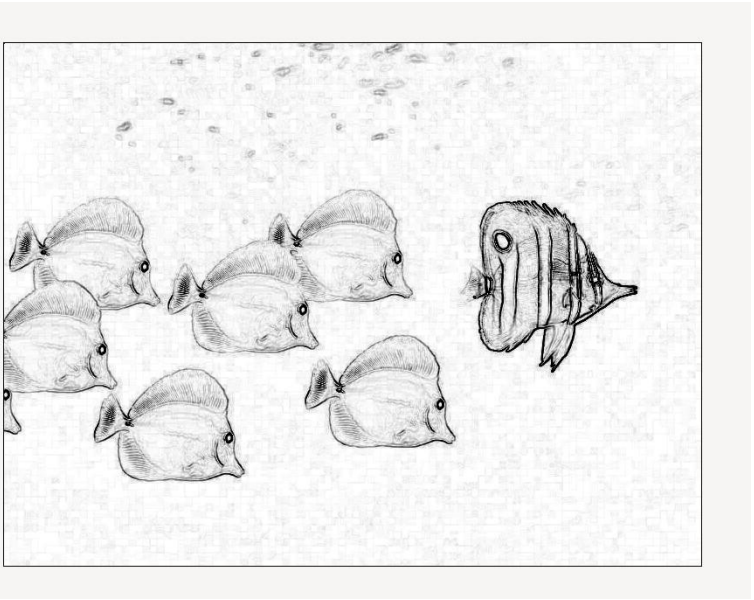
```
        lea r11, [rsi-2]                    //Nombre de lignes
lignes:
        lea r10, [rdi-2]                    //Nombre de colonnes
colonnes:
        xor r15, r15                        //Gx
        sub r15d, [rcx]                     //-a11 Gx
        add r15d, [rcx+8]                   //+a31 Gx
        sub r15d, [rcx+rdi*4]               //-a12 Gx
        sub r15d, [rcx+rdi*4]               //-a12 Gx
        add r15d, [rcx+rdi*4+8]             //+a32 Gx
        add r15d, [rcx+rdi*4+8]             //+a32 Gx
        sub r15d, [rcx+rdi*4*2]             //-a13 Gx
        add r15d, [rcx+rdi*4*2+8]           //+a33 Gx
        xor r14, r14                        //Gy
        add r14d, [rcx]                     //+a11 Gy
        add r14d, [rcx+4]                   //+a21 Gy
        add r14d, [rcx+4]                   //+a21 Gy
        add r14d, [rcx+8]                   //+a31 Gy
        sub r14d, [rcx+rdi*4*2]             //-a13 Gy
        sub r14d, [rcx+rdi*4*2+4]           //-a23 Gy
        sub r14d, [rcx+rdi*4*2+4]           //-a23 Gy
        sub r14d, [rcx+rdi*4*2+8]           //-a33 Gy

        mov r12d, r15d
        neg r12d
        cmovns r15d, r12d                   //r15d = |Gx|
        mov r12d, r14d
        neg r12d
        cmovns r14d, r12d                   //r14d = |Gy|

        add r14d, r15d                      //r14d = |Gx| + |Gy| = G
        neg r14d                            //r14d = -G
        add r14d, 255                       //r14d = 255-G
        mov r12d, 0
        cmp r14d, 0
        cmovl r14d, r12d                    //Si 255-G<0 alors r14d = 0
        mov r13d, r14d                      //r13d registre cache de G
        shl r14d, 8                         //G est appliqué sur la partie V
        add r14d, r13d                      //On rajoute G sur la partie R
        shl r14d, 8                         //G est appliqué sur la partie B
        add r14d, r13d                      //On rajoute G sur la partie R
        or r14d, 0xff000000
        mov dword ptr[r8+4*rdi+4], r14d

        add r8, 4
        add rcx, 4
        sub r10, 1                          //Une colonne de moins à traiter
        ja colonnes                         //Traitement de la colonne suivante s'il en reste
        add r8, 8
        add rcx, 8
        sub r11, 1                          //Une ligne de moins à traiter
        ja lignes                           //Traitement de la ligne suivante s'il en reste
        pop rcx
```

```
epilogue:
        pop r13                 //Dépiler r13
        pop r12                 //Dépiler r12
        pop r14                 //Dépiler r14
        pop r15                 //Dépiler r15
        pop rbx                 //Dépiler rbx
        pop    rbp              //Dépiler le pointeur de cadre de pile sauvegardé
        ret                     //Retour à l'appelant
```



Contour finaux de l'image source

6.3.4 Comparaison des performances

Programme C

Programme assembleur

Notre code assembleur exécute le programme 2 fois plus rapidement que le compilateur C. C'est un résultat satisfaisant !