Université de Technologie de Compiègne

MI01

Rapport de TP

7 - Traitement d'image

Automne 2014

Romain PELLERIN - Kyâne PICHOU
Groupe 1
31 décembre 2014

Table des matières

1 Introduction

L'objectif de ce TP est de reprendre l'algorithme de conversion en niveaux de gris que nous avons réalisé dans le TP 5 et de le ré-implémenter au moyen d'instructions MMX afin d'établir le gain éventuel de performances obtenu. Nous commencerons par traiter chaque pixel de l'image un par un, puis nous traiterons deux pixels par itération (lorsque c'est possible) pour améliorer les performances.

2 Développement

2.1 Préambule

À nouveau dans ce TP, nous allons utiliser le mini-logiciel fourni lors du TP 5 pour traiter des images. Sa structure est la même (cf. section 2.1.1 de notre rapport sur les TP 5 et 6). La réprésentation d'une image est également la même (cf. section 2.1.2 de notre rapport sur les TP 5 et 6). Enfin, la conversion en niveaux de gris s'effectue exactement de la même façon (cf. section 2.2.1 de notre rapport sur les TP 5 et 6).

La différence notable par rapport au TP 5 est qu'ici nous avons la possibilité d'utiliser des « registres » MMX de 64bits. Nous avons également accès à de nouvelles instructions très puissantes, qui nous permettent de paralléliser des traitements, tels que la multiplication de plusieurs mots de 16 bits ainsi que leur addition (par exemple).

2.2 Réalisation

Pour l'écriture du code, nous sommes repartis de ce que nous avions écrit pour le TP 5, en supprimant l'algorithme de conversion et en modifiant quelques lignes pour pouvoir utiliser les instructions MMX.

Puis nous avons pu commencer à réfléchir à nos deux algorithmes :

— Traitement d'un pixel par itération (chap. 3)

— Traitement de deux pixels par itération (chap. 4)

3 Traitement d'un pixel par itération

3.1 Principe

MMX propose une instruction très puissante : `PMADDWD`. Elle permet de multiplier les mots de 16 bits contenus sur deux registres de 64 bits, puis d'additionner les résultats deux à deux de manière à obtenir deux sommes de 32 bits.

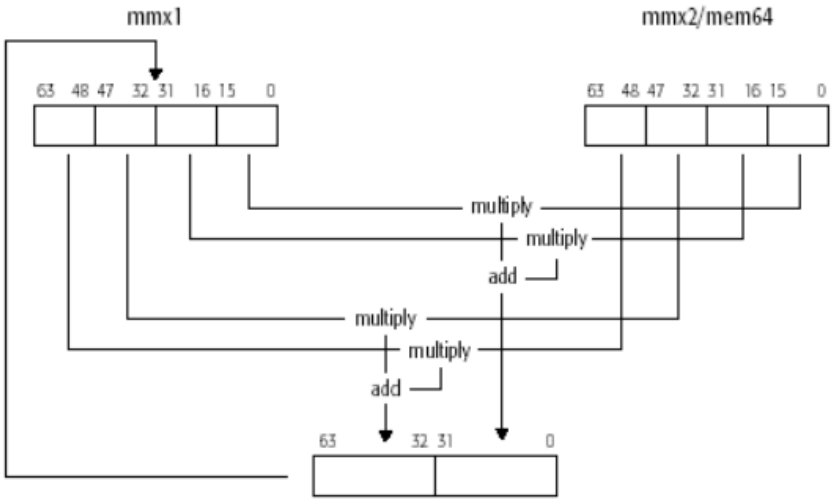

FIGURE 3.1 – Fonctionnement de PMADDWD

Sachant que l'on doit multiplier chaque composante d'un pixel par un cœfficient (selon la même méthode que dans le TP 5), on peut imaginer stocker sur deux registres MMX différents les cœfficients et les valeurs des composantes d'un pixel. Il faut donc stocker sur les bits de poids faible de chacun des 4 mots de 16 bits d'un registre les différentes composantes ainsi que les cœfficients, comme ceci

64 bits	0x0	0x0	0x0	R	0x0	G	0x0	B

Composantes d'un pixel

64 bits	0x0	0x0	0x0	Coef. R	0x0	Coef. G	0x0	Coef. B

Coefficients

Cependant, pour obtenir une telle « configuration » des registres, nous devons utiliser l'instruction `PUNPCKLBW` qui permet « d'éclater » les octets de deux registres en les espaçant. Nous devrons au préalable placer les cœfficients dans un registre MMX et les composantes d'un pixel dans un autre registre.

Une fois les mutliplications et additions faites grâce à `PMADDWD`, nous obtenons quelque chose comme ceci :

64 bits	0x0	0x0	R*Coef	0x0	0x0	G*Coef+B*Coef

Composantes d'un pixel

Il ne nous reste plus qu'à additionner les 32 bits de poids fort aux 32 bits de poids faible et à décaler de 8 bits vers la droite pour obtenir notre valeur entière recherchée (pour le gris). Cela se fait aisément grâce aux instructions `paddd` et `psrlq`. Nous pouvons finalement sauvegarder la nouvelle valeur de notre pixel en mémoire centrale.

3.2 Code

Voici le code final permettant de traiter un pixel par itération :

Assembleur

```
; ...
; CODE fourni

    dec ecx
    imul ecx, 4

    ; Copie des constantes multipliées par 256 dans mm1
    mov eax, 150d ; GREEN
    mov ah, 77d ; RED
    shl eax, 8
    mov al, 29d ; BLUE
    movd mm1, eax

    pxor mm3, mm3 ; RESET
    punpcklbw mm1, mm3 ; "Eclatement" des coefficients


boucle:
    movd mm0, dword ptr [esi+ecx] ; Copie du pixel source (32 bits)

    pxor mm2, mm2 ; RESET
    punpcklbw mm0, mm2 ; "Eclatement" des composantes R, G et B
    pmaddwd mm0, mm1 ; Multiplication par les coefficients et addition
    movq mm2, mm0 ; Copie de mm0 dans mm2
    psrlq mm2, 32 ; On ne conserve que les 32 bits de poids fort (donc R*Coef)
    paddd mm0, mm2 ; Addition de R*Coef sur mm2 avec (G*Coef + B*Coef) sur mm0
    psrld mm0, 8 ; Décalage d'un octet vers la droite
    movd eax, mm0 ; On sauvegarde les 32 bits de poids faible de mm0 dans eax (
        donc la valeur du gris)

    mov [edi+ecx], eax ; On sauvegarde la valeur du pixel que l'on copie dans l'
        image de destination

    sub ecx, 4 ; Pixel suivant
    jne boucle ; On continue tant qu'on n'a pas traité tous les pixels


; CODE fourni
; ...
```

Attention : Ce code ne fonctionne pas pour une image contenant un seul pixel. Pour corriger cela, il faudrait modifier le saut conditionnel de la fin. Néanmoins, à la fin de ce rapport, un code est proposé : il fonctionne pour une image ayant 2n+1 pixels (avec n $\geq$ 0) et traite deux pixels à la fois.

3.3 Comparaison des performances par rapport au TP 5

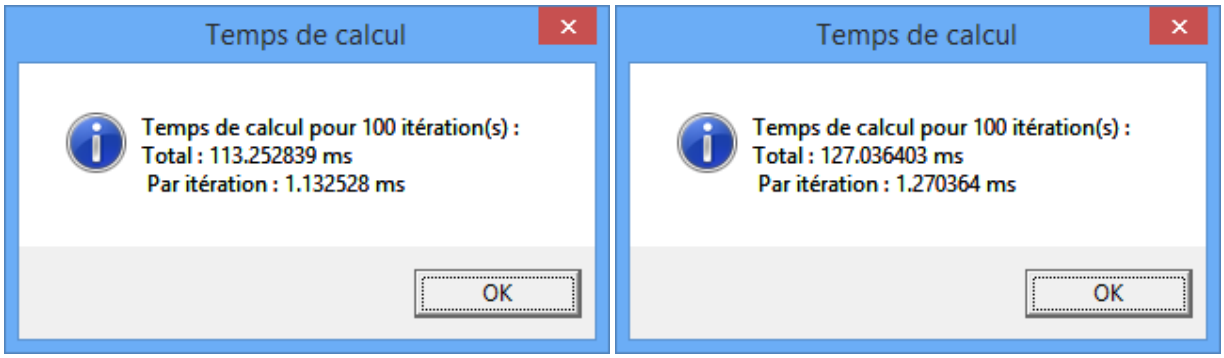

FIGURE 3.2 – À gauche x86, à droite x86 avec MMX

On constate des performances légèrement inférieures à ce que l'on avait obtenu lors du TP 5. Cela peut s'expliquer par un nombre d'instructions un peu plus élevé dans notre algorithme du TP 6 par rapport à celui du TP 5. La puissance des instructions MMX n'est pas exploitée convenablement. Il convient donc de traiter maintenant deux pixels par itération de manière à augmenter significativement les performances.

4 Traitement de deux pixels par itération

4.1 En traitant une image avec un nombre pair de pixels

4.1.1 Principe

Le principe est le même que pour un pixel, sauf que l'on va traiter en même temps deux pixels.

Pour cela, à chaque itération, il suffit de copier deux pixels sur un registre MMX (un pixel dans les 32 bits de poids fort, un autre dans les 32 bits de poids faible). Les composantes R, G et B de chaque pixel sont ensuites « éclatées » (espacées) sur deux registres différents à l'aide d'instructions de type **unpack**.

Ensuite, les composantes de chaque pixel sont multipliées par les bons cœfficients à l'aide de l'instruction `pmaddwd`. Puis, R*Coef est additionné à (G*Coef + B*Coef) pour chaque pixel, comme dans l'algorithme qui traite un pixel un par un. **Ce traitement est le même que celui du chapitre 3**.

Finalement, les deux sommes sont remises dans un registre MMX (une somme dans les 32 bits de poids fort, l'autre dans les 32 bits de poids faible, en faisant attention à ne pas inverser les deux pixels). Et le registre est copié en mémoire centrale, dans l'image de destination. Le code détaillé ci-dessous explique tout cela grâce à des commentaires.

4.1.2 Code

Assembleur

```
; ...
; CODE fourni

    dec ecx
    imul ecx,4

    ; Copie des constantes dans mm1
    mov eax, 150d ; GREEN
    mov ah, 77d ; RED
    shl eax, 8
    mov al, 29d ; BLUE
    movd mm1, eax

    pxor mm3, mm3
    punpcklbw mm1, mm3

    sub ecx, 4 ; On veut traiter le pixel 0 ET le pixel 1, donc on avance pour
        ne pas lire le pixel 0 et le "pixel -1" (qui n'existe pas)

bouclee:
```

```
    movq mm0, qword ptr [esi+ecx] ; On charge deux pixels d'un coup (64 bits)

    pxor mm2, mm2 ; RESET
    punpckhbw mm2, mm0 ; Espacement des composantes R, G et B du pixel contenu
        dans les bits de poids fort (high)
    psrlw mm2, 8 ; On met les composantes dans les bits de poids faible des mots
         de 16 bits

    punpcklbw mm0, mm3 ; Espacement des composantes R, G et B du pixel contenu
        dans les bits de poids faible (low)

    pmaddwd mm0, mm1 ; Multiplication par les coefficients et addition
    pmaddwd mm2, mm1 ; Multiplication par les coefficients et addition

    movq mm4, mm0 ; Copie de mm0 dans mm4

    punpckldq mm4, mm2 ; On met les composantes G*Coef + B*Coef de chaque pixel
        cote a cote dans mm4
    punpckhdq mm0, mm2 ; On met la composante R*Coef de chaque pixel côte à côte
         dans mm0

    paddd mm4, mm0 ; Pour chaque pixel, addition de R*Coef avec (G*Coef + B*Coef
        )
    psrld mm4, 8 ; On ne garde que la partie entière de la valeur du gris
        obtenue (décalage de 8 bits)

    movq qword ptr [edi+ecx], mm4 ; On sauvegarde la valeur du pixel que l'on
        copie dans l'image de destination

    sub ecx,8 ; On saute 2 pixels à chaque fois
    jne bouclee ; On continue tant qu'on n'a pas tout traité

; CODE fourni
; ...
```

4.1.3 Comparaison des performances par rapport au traitement d'un pixel par itération

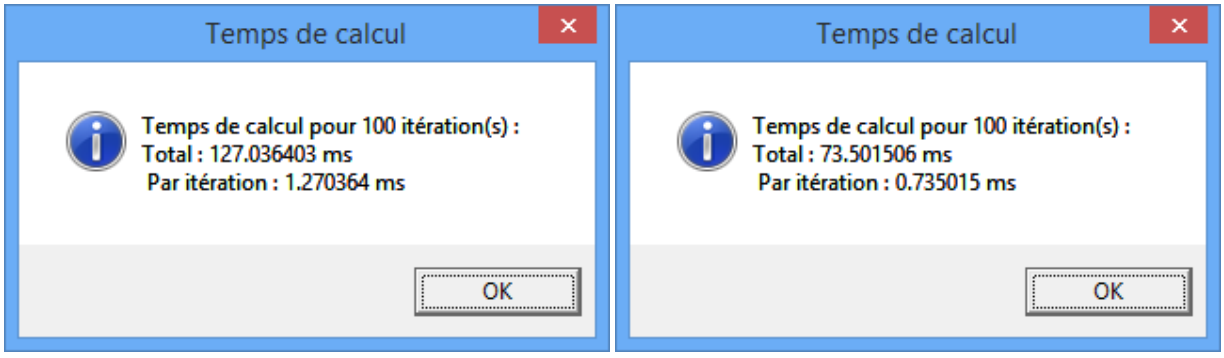

FIGURE 4.1 – À gauche un pixel par itération, à droite deux pixels par itération

On constante un gain très important de performance : il faut presque deux fois moins de temps pour traiter une image. Cela s'explique par le fait que l'on exploite pleinement la puissance des instructions MMX ici, en parallélisant les traitements.

4.2 En traitant une image avec n'importe quel nombre de pixels

Le code précédent souffre d'un problème : si l'image contient un nombre de pixels impair, à un moment le programme accèdera à de la mémoire qu'il n'est pas sensé utiliser.

Il faut donc prendre en considération deux cas :

— L'image possède un seul pixel

— L'image possède 2n+1 pixels (avec n > 0)

Cela se fait aisément en ajoutant des sauts conditionnels au début du programme et à la fin des boucles, comme ceci :

Assembleur

```
; ...

    dec ecx
    imul ecx,4

    ; Copie des constantes dans mm1
    ; ...

    pxor mm3, mm3
    punpcklbw mm1, mm3

    ; Si l'image fait AU MOINS 2 pixels, ecx sera supérieur à 0 car ecx = (
        nombre de pixels - 1) * 4
    cmp ecx, 0
    ja twopixels
    ; Sinon il n'y a qu'un seul pixel

unpixel:
    movd mm0, dword ptr [esi+ecx] ; 32 bits

    ; ...

    mov [edi+ecx], eax ; On sauvegarde la valeur du pixel que l'on copie dans l'
        image de destination

    jmp fin ; C'était le dernier pixel à traiter

twopixels:
    sub ecx, 4

bouclee:
    movq mm0, qword ptr [esi+ecx] ; On charge deux pixels d'un coup (64 bits)

    ; ...
```

```
    movq qword ptr [edi+ecx], mm4 ; On sauvegarde la valeur du pixel que l'on
        copie dans l'image de destination

    sub ecx, 4
    cmp ecx, 0
    je unpixel ; S'il reste QU'UN SEUL pixel à traiter, on appelle l'algo à l'é
        tiquette 'unpixel'
    jl fin ; Si c'est inférieur à 0, on a tout traité
    sub ecx, 4 ; Sinon il reste au minimum 2 pixels à traiter
    jmp bouclee ; Et donc on traite les 2 pixels suivants

fin:

; ...
```

Le code complet de cette partie est trouvable en annexes.

5 Conclusion

Lors du TP 5, nous avions remarqué que pour certains types de traitement de données (ici d'une image) l'optimisation du code en assembleur offrait un réel gain en performance par rapport au code écrit en C. Ici, nous avons pu constater que **le jeu d'instructions MMX** conçu pour les microprocesseurs de type x86 **offre un impressionnant gain de performances par rapport au jeu d'instructions assembleur classique, dès lors que certains traitements sont parallélisés** (comme le traitement de deux pixels « en même temps »). C'est d'ailleurs la raison pour laquelle MMX a été créé : accélérer certaines opérations répétitives dans des domaines tels que le traitement de l'image 2D, du son et des communications (source : Wikipédia).

A Code complet du traitement de deux pixels par itération fonctionnant avec un nombre impair de pixels

Assembleur

```
; IMAGE.ASM
;
; MI01 - TP Assembleur 2 à 5
;
; Réalise le traitement d'une image 32 bits.

.686
.MODEL FLAT, C
.MMX

.DATA

.CODE

; ************************************************************************
; Sous-programme _process_image_asm
;
; Réalise le traitement d'une image 32 bits.
;
; Entrées sur la pile : Largeur de l'image (entier 32 bits)
;          Hauteur de l'image (entier 32 bits)
;          Pointeur sur l'image source (dépl. 32 bits)
;          Pointeur sur l'image tampon 1 (dépl. 32 bits)
;          Pointeur sur l'image tampon 2 (dépl. 32 bits)
;          Pointeur sur l'image finale (dépl. 32 bits)
; ************************************************************************
PUBLIC     process_image_mmx
process_image_mmx PROC NEAR      ; Point d'entrée du sous programme

       push   ebp        ; construction du cadre de pile (sauvegarde de ebp)
       mov    ebp, esp   ; sauvegarde du pointeur de pile

       ; SAUVEGARDE DES REGISTRES
       push   ebx
       push   esi ; image source
       push   edi ; image tmp1

       ; multiplication de la largeur de l'image par la hauteur
       mov    ecx, [ebp + 8]
```

```
        imul    ecx, [ebp + 12]

        mov     esi, [ebp + 16] ; esi = image source (adresse du premier pixel de
            l'image)
        mov     edi, [ebp + 20] ; edi = image tmp1

        ;*********************************************************************
        ;*********************************************************************
        ; Ajoutez votre code ici
        ;*********************************************************************
        ;*********************************************************************

    dec ecx
    imul ecx,4

    ; Copie des constantes dans mm1
    mov eax, 150d ; GREEN
    mov ah, 77d ; RED
    shl eax, 8
    mov al, 29d ; BLUE
    movd mm1, eax

    pxor mm3, mm3
    punpcklbw mm1, mm3

    ; si l'image fait AU MOINS 2 pixels, ecx sera supérieur à 0 car ecx = (
        nombre de pixels - 1) * 4 (cf lignes 51 et 52)
    cmp ecx, 0
    ja twopixels

unpixel:
    movd mm0, dword ptr [esi+ecx] ; 32 bits

    pxor mm2, mm2
    punpcklbw mm0, mm2
    pmaddwd mm0, mm1
    movq mm2, mm0
    psrlq mm2, 32
    paddd mm0, mm2
    psrld mm0, 8
    movd eax, mm0

    mov [edi+ecx], eax ; on sauvegarde la valeur du pixel que l'on copie dans l'
        image de destination

    jmp fin

twopixels:
    sub ecx, 4

bouclee:
```

```
    movq mm0, qword ptr [esi+ecx] ; on charge deux pixels d'un coup (64 bits)

    pxor mm2, mm2
    punpckhbw mm2, mm0
    psrlw mm2, 8

    punpcklbw mm0, mm3

    pmaddwd mm0, mm1
    pmaddwd mm2, mm1

    movq mm4, mm0

    punpckldq mm4, mm2
    punpckhdq mm0, mm2

    paddd mm4, mm0
    psrld mm4, 8

    movq qword ptr [edi+ecx], mm4 ; on sauvegarde la valeur du pixel que l'on
        copie dans l'image de destination

    sub ecx, 4
    cmp ecx, 0 ; si il reste QU'UN SEUL pixel à traiter, on appelle l'aglo à l'
        étiquette 'unpixel'
    je unpixel
    jl fin ; si c'est inférieur à 0, on a tout traité
    sub ecx, 4 ; sinon il reste au minimum 2 pixels à traiter
    jmp bouclee

fin:
        ; Libérer l'unité MMX
        emms
        pop    edi
        pop    esi
        pop    ebx

        pop    ebp

        ret                     ; Retour à la fonction MainWndProc

process_image_mmx ENDP
END
```