MI01 TP7 – Traitement d'image – Troisième partie.

1 Travail à réaliser

Le but est de reprendre l'algorithme de conversion en niveaux de gris que vous avez réalisé dans le TP5 et de le ré-implémenter au moyen d'instructions MMX afin d'établir le gain éventuel de performances obtenu.

1.1 Mise en place

Dans votre dossier de projet « tp image », créez une copie du fichier *image.asm* que vous nommerez *image_mmx.asm*. Ajoutez le nouveau fichier à votre projet Visual C++ existant. Toujours dans votre dossier de projet, décompressez l'archive « tp_image_p3.zip » que vous trouvez sur le site de l'UV pour remplacer le fichier *appli.c*. Cette nouvelle version ajoute une option « *Lancer le traitement MMX* » dans le menu « *Traitement* ».
Modifiez le fichier source *image_mmx.asm* comme suit :

1. Ajoutez la directive **.MMX** afin d'informer l'assembleur de l'usage d'instructions MMX
2. Ajoutez l'instruction EMMS avant le retour de sous-programme afin de libérer l'unité de calcul flottant, qui partage certaines ressources avec l'unité MMX.
3. Changez le nom du sous programme en « *process_image_mmx* ».
4. Puisqu'on ne s'intéresse ici qu'aux performances de la transformation en niveaux de gris, supprimez le code lié au filtre de Sobel.

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

.686
; Autoriser les instructions MMX
.MMX
.MODEL FLAT, C

.DATA

.CODE

PUBLIC       process_image_mmx

process_image_mmx   PROC NEAR           ; Point d'entrée du programme

          push    ebp
          mov     ebp, esp

          push    ebx
          push    esi
          push    edi

; Votre code assembleur des TP précédents
          ...
          ...
          ; Libérer l'unité MMX
          emms

          pop     edi
          pop     esi
          pop     ebx

          pop     ebp

          ret                           ; Retour à la fonction MainWndProc

process_image_mmx   ENDP

          END
```

1.2 Programmes MMX

Il s'agit de réaliser la conversion en niveaux de gris en utilisant des calculs en virgule fixe sur des mots de 16 bits, de la même manière que dans le TP5, en traitant un pixel complet (d'un seul bloc de 32 bits) à chaque itération dans un premier temps, puis en traitant deux pixels par itération dans un second temps.
Afin de suivre le déroulement de votre programme, vous pouvez utiliser le débogueur intégré à Visual C++. Dans la fenêtre « Registres », un clic-droit permet d'accéder à la liste des registres affichables et d'afficher les registres MMX. Si cette fenêtre n'est pas disponible, vous pouvez l'activer par le menu « *Déboguer→Fenêtres→Registres* ».

1.2.1 Un pixel à la fois

Dans cette partie, on remplace le traitement d'un pixel dans la structure itérative existante par un programme équivalent utilisant des instructions MMX.

Questions

- Comment charger un pixel dans un registre MMX (voir les remarques ci-dessous) ?
- Les trois composantes R, V et B d'un pixel sont chacune représentées sur 8 bits. Au moyen d'une instruction de type 'unpack' (PUNPCKxxx) et d'un registre supplémentaire ne contenant que des 0, comment les transformer en un vecteur de mots de 16 bits ?
- Par quel moyen réaliser la conversion proprement dite ? Faites attention aux décalages nécessaires avant de stocker le résultat.
- Implémentez la conversion en MMX dans le sous-programme `process_image_mmx`.
- Mesurez la vitesse d'exécution du calcul MMX et du calcul assembleur IA32 (sautez le calcul du filtre de Sobel avec une instruction 'jmp' dans votre code IA32). Comparez le nombre d'instructions dans la boucle entre votre implémentation IA32 et votre implémentation MMX. Le résultat vous semble-t-il cohérent ? **Note** : pour que les résultats soient significatifs, utilisez au moins 100 répétitions (menu « *Traitement→Répétitions* »).

1.2.2 Deux pixels à la fois

L'algorithme précédent est purement séquentiel pour chaque pixel. Dans ce cas, afin d'augmenter les performances, une idée est de traiter plusieurs éléments de données à chaque itération. Ici, vous allez changer la boucle afin de traiter deux pixels à chaque itération. On suppose pour simplifier que l'image contient un nombre pair de pixels (c'est le cas pour les poissons).

Questions

- Modifiez la structure de la boucle afin de traiter deux pixels à chaque itération, en prenant soin de mettre à jour correctement le compteur et les pointeurs de pixels éventuels dans les images *source* et *temp1*.
- Chargez deux pixels à la fois depuis l'image *source* dans un registre MMX (par exemple mm0) comme indiqué dans les remarques ci-dessous. Les 32 bits de poids faible de ce registre contiennent le pixel 0, les 32 bits de poids fort le pixel 1.
- Au moyen d'un code similaire au code précédent, calculez les sommes partielles des produits des composantes R, V, et B de chacun de ces pixels avec leurs coefficients respectifs. **Les résultats doivent être dans deux registres MMX différents.**
- Au moyen de deux instructions de type 'unpack' bien choisies, formatez les deux résultats précédents pour qu'une seule instruction d'addition soit nécessaire au calcul de l'intensité finale des deux pixels. N'oubliez pas de faire les décalages nécessaires à l'ajustement de la position de la virgule dans le résultat. L'intensité finale du pixel 0 doit alors se trouver dans les 32 bits de poids faibles du résultat, celle du pixel 1 dans les 32 bits de poids fort.
- Finalement, stockez le résultat en mémoire dans l'image *temp1*.
- Travaillez le programme obtenu afin de minimiser les dépendances de données entre instructions successives, quitte à 'mélanger' le traitement du pixel 0 et du pixel 1.
- Mesurez la vitesse d'exécution de ce nouveau code MMX par rapport à l'ancien. Le nouveau code doit être plus performant. Ce résultat vous semble-t-il cohérent ? Comment l'expliquez-vous ?
- On veut maintenant pouvoir traiter des images contenant un nombre impair de pixels. Modifiez le programme pour traiter ce cas.

1.3 Remarques

- Il n'existe pas d'instruction permettant de charger une valeur immédiate dans un registre MMX. Il faut donc passer par un registre intermédiaire. Par exemple, pour charger la valeur 0FFFFFFFFh dans la partie basse du registre mm0, on utilisera le code suivant :

```
mov   eax, 0ffffffffh
movd  mm0, eax
```

- Contrairement au cas des instructions entières, l'assembleur ne détermine pas toujours automatiquement la dimension des données mémoire en fonction des autres opérandes de l'instruction. Il faut donc impérativement spécifier explicitement la taille des transferts dans les accès mémoire. Par exemple l'instruction

```
movq  mm0, qword ptr [donnee_mmx]
```

  transfère la donnée MMX sur 64 bits stockée en mémoire à l'adresse `donnee_mmx` dans le registre `mm0`.

Important : le compilateur Visual C++ s'attend à ce que les registres EBX, ESI et EDI soient préservés par les appels de sous-programmes.