Leonardo D'ALESSIO - GI01
Soulaymane KEBLI - GI01

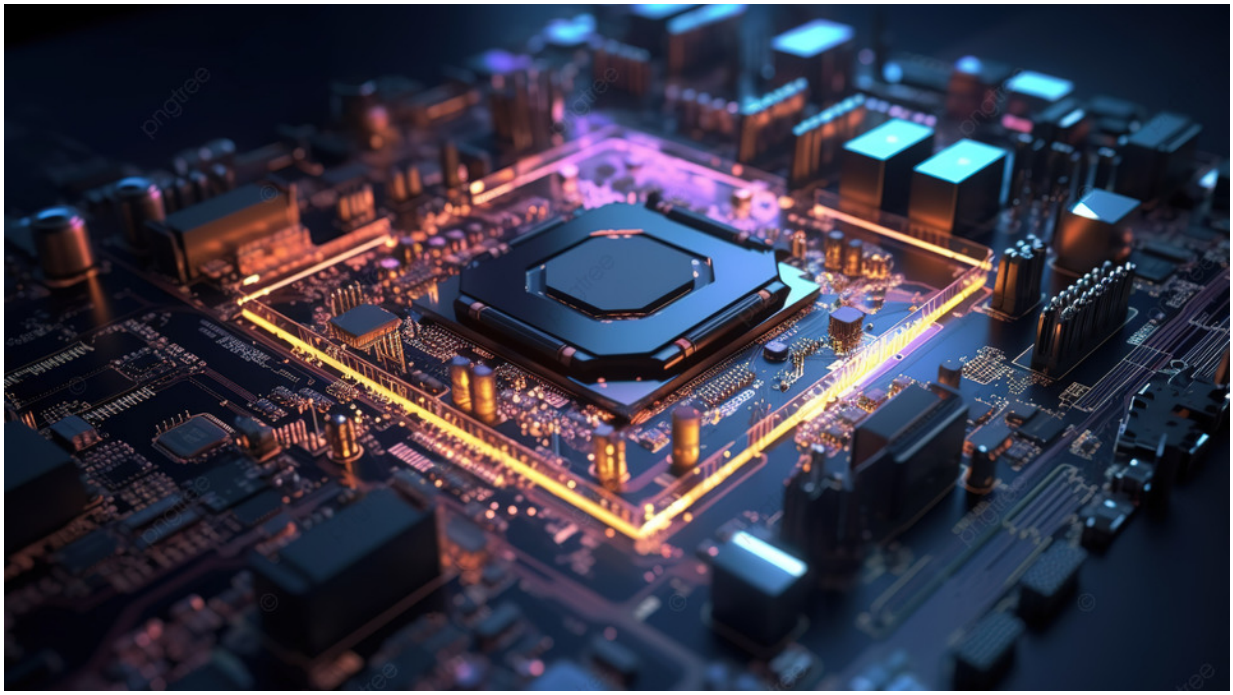
Université de Technologie de Compiègne

MI01 - Structure d'un calculateur

Responsable
Mr. Shawky Marc

TP8 – Traitement d'image - Simd

Responsable TP
Mr. Sanahuja Guillaume



TP1 - Groupe n°6

Sommaire

Table des figures

1 Introduction

Au cours de ce rapport nous allons présenter les résultats du TP 8 : Traitement d'images - SIMD. Ce TP a pour objectif de reprendre le calcul de l'intensité en niveau de gris du précédent TP. Cette fois-ci, nous utiliserons les instructions des SSE [1]. Il s'agit d'un jeu supplémentaire d'instructions permettant de simuler une architecture SIMD [2] sur l'architecture SISD [3]. Ce calcul parallèle de différentes données par une même instruction permet d'effectuer plus rapidement un traitement sur une image. Nous allons mettre en place un tel calcul et vérifier si l'efficacité supposée est bien effective.

1. Streaming SIMD Extensions
2. Single Instruction stream Multiple Data stream
3. Single Instruction stream Single Data stream

2 Calcul de l'intensité des pixels avec SSE

2.1 Structure itérative

Dans la boucle qui itère sur les pixels de l'image source (img_src) à partir de process_image_asm, nous prenions en compte un seul pixel par itération : il suffisait de commencer par le dernier pixel [rdx + rdi*4 - 4] et de décrémenter de 1 le nombre de pixels restant (stocké dans rdi) à chaque itération.
Cette fois-ci, nous souhaitons prendre en compte des blocs de quatre pixels par itération car les vecteurs SSE, nous permettant donc de traiter 4*32 bits. Ainsi pour avoir le dernier bloc de 4 pixels il suffit de prendre l'adresse du 4ème pixel en partant de la fin de l'image. Donc on commence par l'adresse du dernier bloc de 4 pixels [rdx + rdi*4 - 16] et on décrémente de 4 le nombre de pixel restants (rdi) à chaque itération.

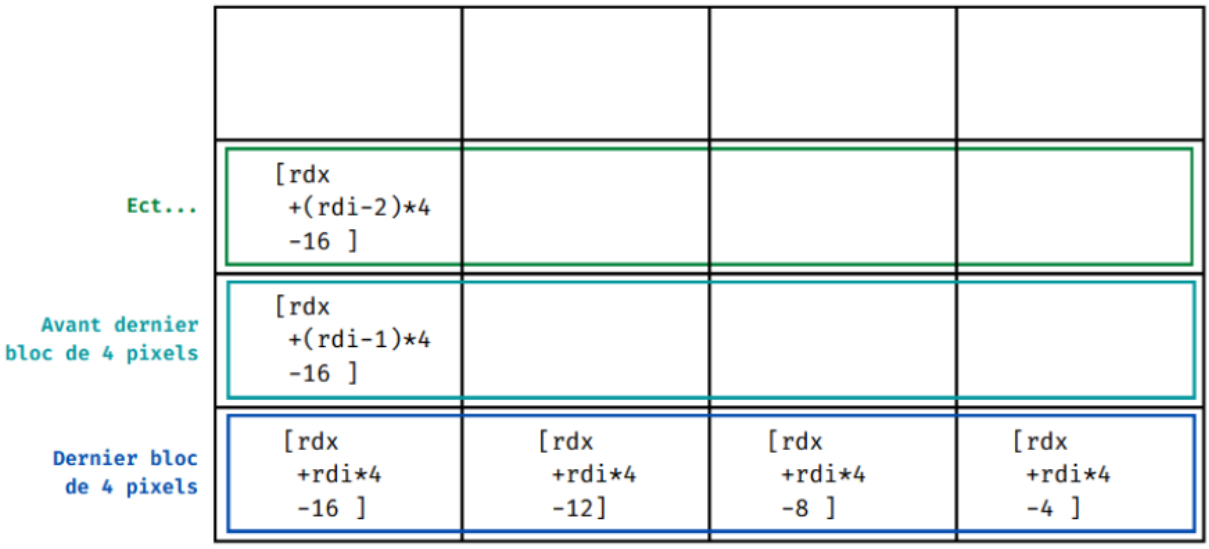


FIGURE 1 – Parcours des pixels - SIMD

On obtient ainsi le code suivant :

```
loop_gs:
    movdqa   xmm0, [rdx + rdi*4 - 16] /*deplacement du groupe de
                                       pixels, vers un registre SSE*/
    sub      rdi, 4                    /*Un vecteur de 4 pixels
                                       de moins a traiter*/
    ja loop_gs
```

3 Implémentation

3.1 Première étape : Construction d'un vecteur d'octets contenant les constantes associées à chaque composante de chaque pixel

Cette partie s'effectue en dehors de la boucle loop_gs. En effet, il n'est pas nécessaire de répéter à chaque groupe de pixel à traiter, la construction d'un vecteur constant. Les étapes suivantes seront toutes réalisées dans la boucle.

Lors de cette première étape, nous voulons reconstruire le vecteur suivant :

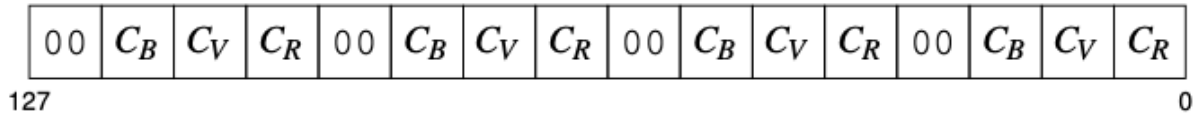


FIGURE 2 – Vecteurs des coefficients

Pour rappel, les coefficients équivalents en hexadécimal de chaque composantes sont les suivants :

— Coefficient **rouge** : $(0,2126)_{10} \iff (00,36)_{16}$
— Coefficient **vert** : $(0,7152)_{10} \iff (00,B7)_{16}$
— Coefficient **bleu** : $(0,0722)_{10} \iff (00,13)_{16}$

Il existe plusieurs façons de construire un tel vecteur. Pour notre part, nous allons construire un premier bloc de 32 bits :

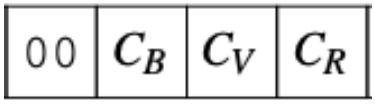


FIGURE 3 – Premier bloc de 32bits

```
mov eax, 0x0013B736
```

Et le placer dans un registre SSE (nous choisissons xmm1) :

```
movd xmm1, eax
```

Puis, nous le dupliquons sur les 96 bits restants :

```
punpckldq xmm1,xmm1 // copie des 32 bits de poids faibles
                       sur le second bloc de 32  bits
punpcklqdq xmm1,xmm1 // copie des 64 bits de poids faibles sur
                       les 64 bits restants
```

A ce stade, nous venons de construire notre vecteur constant.

3.2 Deuxième étape : Chargement de 4 pixels dans un registre SSE

Pour ce faire, il suffit simplement d'effectuer un déplacement d'une donnée de taille 128 bits vers un registre/vecteur SSE. Nous choisissons xmm1 et utilisons l'adresse du bloc de pixels calculé lors de la partie sur la structure itérative.
Ce déplacement s'effectue à l'aide de l'instruction SSE **movdqa** (aligné selon l'énoncé) :

```
movdqa xmm0, [rdx + rdi*4 - 16]
```

Nous obtenons donc le vecteur suivant :

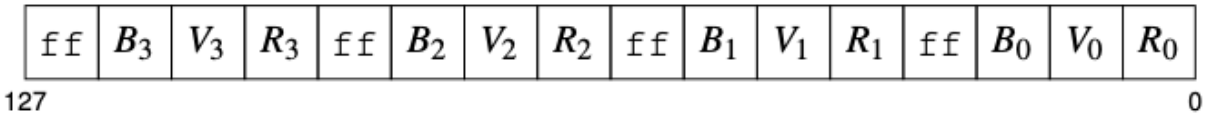


FIGURE 4 – Vecteur des pixels

3.3 Troisième étape : Ajustement du vecteur constant et premier calcul

Le fait que l'opérande source de l'instruction **pmaddubsw** soit signée implique que chaque octet est considéré comme étant encodé en complément à 2. Ainsi, sa valeur ne sera pas correctement lue si elle dépasse 127 puisqu'elle ne peut être comprise qu'entre [-128 ;127].
En revanche, si l'on suppose que chaque coefficient est divisible par 2, il suffit de les diviser par 2, de faire le calcul, puis de re-multiplier par 2 pour contourner ce problème. Les nouveaux coefficients sont donc :

$$0x13/2 = 0x09 \;\mid\; 0xB7/2 = 0x5b \;\mid\; 0x36/2 = 0x1B$$

Remarque : En passant par la division, une perte d'information est inévitable. Cependant, les valeurs étant minimes, cela ne devrait pas se voir à l'œil nu.

Le vecteur coefficient prendra donc comme premier bloc :

```
// mov eax,0x0013B736
mov eax, 0x00095b1B
.................
```

Maintenant que notre vecteur des coefficients est ajusté, nous pouvons effectuer un premier produit scalaire entre chaque composante (en octet / byte) et son coefficient correspondant (en octet / byte). Le résultat sera un mot (word) :

```
pmaddubsw xmm0,xmm1
```

Le vecteur résultat sera donc le suivant : [4]

$B_3 * C_B$	$R_3 * C_B + V_3 * C_V$	$B_2 * C_B$	$R_2 * C_B + V_2 * C_V$	$B_1 * C_B$	$R_1 * C_B + V_1 * C_V$	$B_0 * C_B$	$R_0 * C_B + V_0 * C_V$

FIGURE 5 – xmm0 - Vecteur résultat

Remarque : Nous n'avons pas besoin de nous préoccuper de la multiplication avec le canal de transparence puisqu'elle sera toujours nulle ; par construction du vecteur des coefficients !

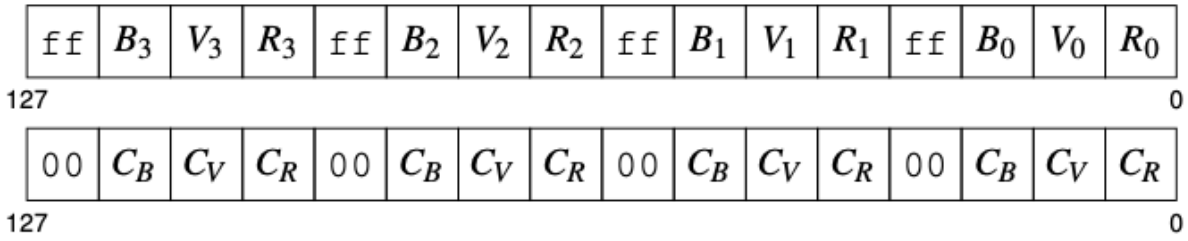


FIGURE 6 – Vecteurs des pixels et des coefficients

3.4 Quatrième étape : Finalisation du calcul d'intensité

L'objectif de cette étape est d'obtenir le vecteur suivant :

00	00	00	I_3	00	00	00	I_2	00	00	00	I_1	00	00	00	I_0

FIGURE 7 – xmm0 - Calcul d'intensité

Pour faire cela, nous pouvons premièrement effectuer une addition horizontale de mots deux à deux. En utilisant l'instruction **phaddw** avec un vecteur nul (nous choisissons xmm2), nous avons :

0	0	0	0	$B_3 * C_B + R_3 * C_R + V_3 * C_V$	$B_2 * C_B + R_2 * C_R + V_2 * C_V$	$B_1 * C_B + R_1 * C_R + V_1 * C_V$	$B_0 * C_B + R_0 * C_R + V_0 * C_V$

FIGURE 8 – xmm0 - Calcul d'intensité 2

Suite d'instruction associé :

```
pxor    xmm2,xmm2
phaddw  xmm0,xmm2
```

4. C_B, C_R, C_V : *coefficients divisés par* 2

Il ne faut pas oublier de démultiplier ces résultats par deux ! Il suffit d'additionner les mots du vecteur xmm0 par eux même :

```
paddw xmm0,xmm0 // multiplication par 2
```

Maintenant, nous devons reporter ce résultat sur 32 bits. Pour ce faire, il suffit simplement d'intercaler des mots nuls entre chaque intensité à l'aide de :

```
punpcklwd xmm0,xmm2
```

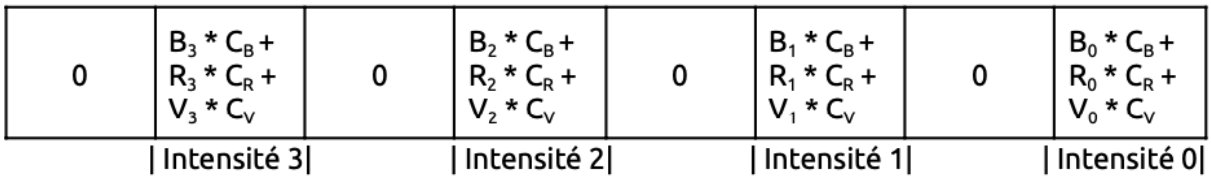


FIGURE 9 – xmm0 - Calcule d'intensité 3

L'intensité n'étant pas supérieure à 16 bits (voir rapport précédent) il n'est pas nécessaire de placer les mots de 0 avant les intensités. Cela nous permettra d'effectuer un décalage moins important par la suite.
Puisque seules la partie entière de l'intensité nous intéresse, il suffit simplement de faire un décalage indépendant de chaque mot de 8 bits, soit 1 octet (virgule avec un décalage de 8 bits).
On obtient ainsi :

```
psrlw xmm0, 8
// decalage de 1 octets, placement de l'intensite
```

Soit le vecteur suivant que l'on voulait obtenir :

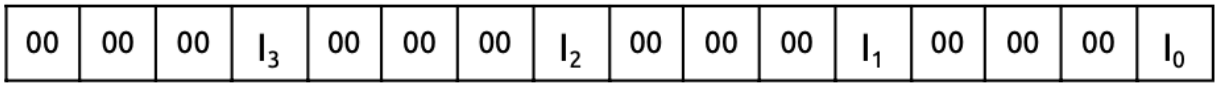


FIGURE 10 – xmm0 - Calcul d'intensité

3.5 Cinquième étape : Ajout du canal de transparence

Il s'agit de la dernière étape de construction de notre vecteur résultant. Celui-ci ressemblera à :

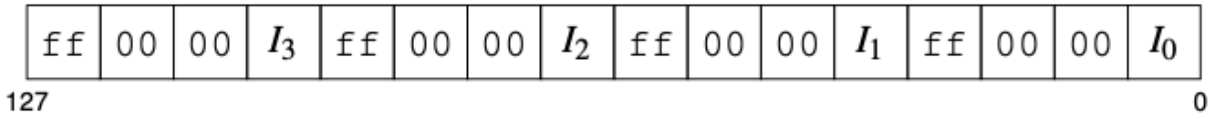


FIGURE 11 – Forme du vecteur résultant à stocker dans l'image

Pour commencer, nous allons construire le masque : A l'aide de l'instruction suivante,

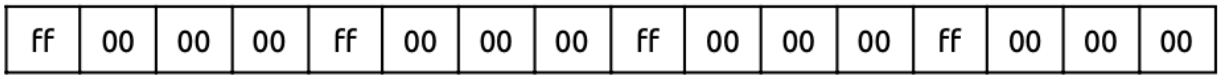


FIGURE 12 – xmm2 - Masque

nous obtenons un vecteur dont tous les bits sont à 1 :

```
pcmpeqq xmm2,xmm2
```

Nous obtenons le vecteur xmm2 suivant : Pour retrouver la forme du masque souhaité, un

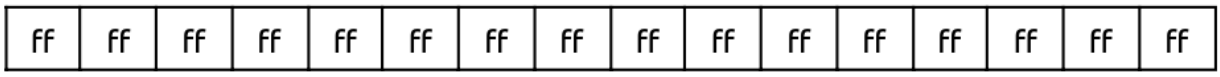


FIGURE 13 – xmm2 à 1

simple décalage indépendant de double mot de 3 octets (24 bits) vers la gauche suffit :

```
pslld xmm2,24
```

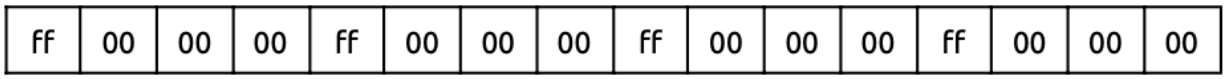


FIGURE 14 – xmm2 après application du masque

Enfin, il reste à additionner par octet ce masque avec notre vecteur d'intensités :

```
paddb xmm0,xmm2
```

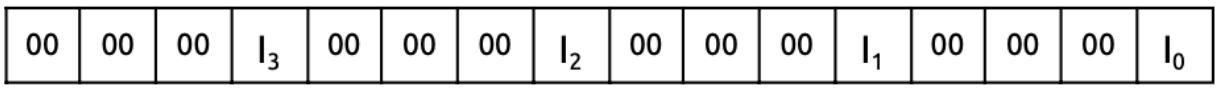


FIGURE 15 – xmm0

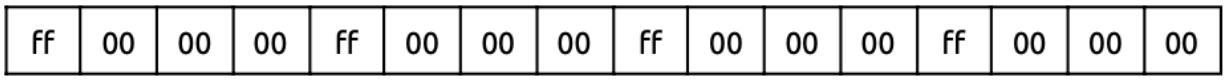


FIGURE 16 – xmm2

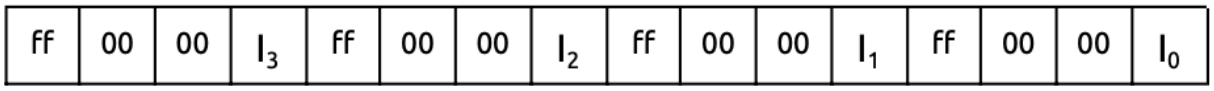


FIGURE 17 – xmm0 Final

3.6 Sixième étape : Stockage du vecteur dans l'image de destination

L'image temporaire 1 est à l'adresse présente dans le registre rcx. Pour reconstruire l'image, il suffit de stocker ce vecteur au même emplacement que le groupe de pixels de départ, mais cette fois-ci par rapport au registre d'arrivée qui est donc rcx.
L'instruction SSE **movdqa** le permet :

```
movdqa [rcx + rdi*4 - 16], xmm0
```

4 Performances et comparaison

Après 1000 répétitions de ce calcul en niveau de rouge, voici les temps par itération pour chaque implémentation :

FIGURE 18 – Implémentation C

FIGURE 19 – Implémentation Assembleur x86-64

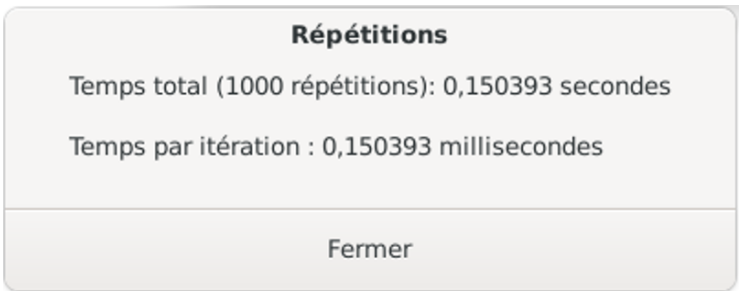


FIGURE 20 – Implémentation SSE

Nous remarquons un gain de rapidité fulgurant ! L'implémentation SSE est environ 6,6 fois plus rapide que l'implémentation en assembleur x86-64. Cette différence est encore plus grande avec l'implémentation C où elle est environ 10,5 fois plus rapide. Ce gain de temps n'est pas négligeable et montre clairement l'efficacité du SSE.

Étant donné que les images soient techniquement différentes suite à la perte de précision et d'information sur le calcul d'intensité, nous avons un taux de correspondance faible et différent de 100% avec les autres images des autres implémentations :

FIGURE 21 – Comparaison de img_temp1 des différentes implémentations

Cependant, cette différence est minime puisque nous obtenons bien un résultat identique à l'oeil nu :

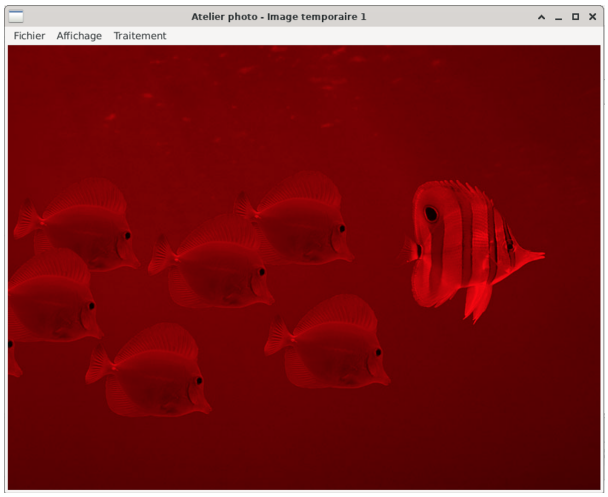


FIGURE 22 – Implémentation C

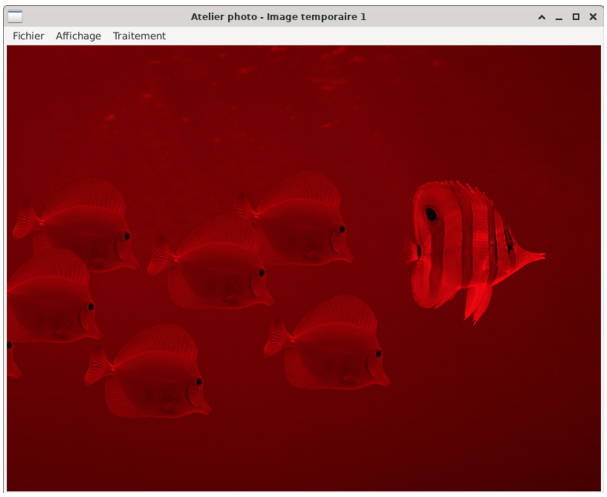


FIGURE 23 – Implémentation SSE

5 Conclusion

Pour conclure ce rapport, nous pouvons confirmer que les instructions du SSE sont nettement plus efficaces que celles du jeu “classique” d’instructions. Sur les machines, cette performance s’est démarquée par une rapidité grandement supérieure à celle de notre programme en langage assembleur ou encore du langage C. Au-delà de la rapidité, le programme est également beaucoup plus court que celui précédent. Le gain est donc double : à la fois en spatialité et en temporalité.

Ce TP nous aura permis de comprendre l’intérêt réel de cette extension, de mieux comprendre ses instructions, mais également de la manipuler dans un véritable programme.