SEOC-3A Travaux Pratiques
Utilisation des extensions SSE du x86_64 pour le calcul de la conversion YCbCr vers RGB en Motion Jpeg

Frédéric Pétrot/Arthur Perais
Durée : 3 heures

1 Organisation

Le travail se fait en binôme cette fois-ci (l'expérience forme la jeunesse). Le sujet est découpé en parties qui peuvent au départ être faites plus ou moins indépendamment, aussi il est raisonnable que les 2 membres du binôme travaillent en parallèle (de degré 2 ici).

2 Introduction

Ce TP a pour objectif de mettre en pratique l'utilisation des opérations SIMD du processeur x86_64. Ces opérations ont été ajoutées au fil du temps via de nombreuses extensions aux jeux d'instruction.

SIMD signifie *Single Instruction, Multiple Data*, ce qui veut dire que la même opération va être exécutée sur n données différentes (et indépendantes) simultanément. Un exemple simple est l'exécution d'une addition sur les 4 octets d'un mots de 32 bits considérés indépendamment.

Nous utiliserons les extensions dites « SSE » (SSE, SSE2,SSE3, SSSE3, SSE4.1 et SSE4.2), sachant qu'il en existe de plus modernes (AVX, AVX2, AVX512) et de plus anciennes (MMX), ... Il y a 396 instructions SSEx (rien que ça), mais vous serez – un peu – guidés pour sélectionner celles dont vous aurez l'usage. Pour les courageux, tout ce qui est dit ci-dessous peut se faire avec les extensions AVX (AVX2 pour ce qui concerne les entiers), et vous pouvez donc partir là-dessus pour avoir un facteur d'accélération qui devrait, sur la partie du programme qui utilise ces instructions, avoir un gain double de celui du SSE.

Les instructions SSE d'Intel travaillent sur des registres de 128 bits, qui peuvent contenir 2 doubles (`double`), 4 flottants (`float`), 4 entiers de 32 bits (`int32_t`), 8 entiers de 16 bits (`int16_t`), ou 16 entiers de 8 bits (`int8_t`). Dans le cas des entiers, certaines instructions interprètent les entiers comme des entiers signés, alors que d'autres les interprètent comme dans entiers non-signés. Idem pour l'AVX, en remplacant 128 par 256, et en ajustant les facteurs multiplicatifs.

Afin de simplifier l'écriture du code dans du C, vous utiliserez les fonctions dites « intrinsèques » qui permettent d'utiliser des instructions assembleur plus simplement que l'`inline asm`. `Gcc` donne accès à ces fonctions en incluant le fichier `x86intrin.h` (c'est fait dans les fichiers à trous que je vous fournis), et elles commencent toutes par `_mm_`. `Gcc` fournit également trois types : `__m128d` pour 2 doubles 64 bits, `__m128` pour 4 flottants 32 bits, `__m128i` pour tous les types entiers, donc c'est au programmeur de savoir ce qu'il contient en réalité. De manière générale, on peut s'attarder sur le suffixe de chaque intrinsèque pour déterminer sur quelle type de donnée on opère :

Type	INT	FP
SISD	Scalaire entier = instructions classiques	"Scalar Single Precision (1x32b)" (**_ss**) "Scalar Double Precision (1x64b)" **(_sd**)
SIMD	"Packed Integer" (**_epi8/16/32/64** ou **_si**) "Packed Unsigned Integer" **(_epu8/16/32/64)**	"Packed Single Precision (4x32b)" (**_ps**) "Packed Double Precision (2x64b)" (**_pd**)

En pratique, j'ai utilisé, pour l'implantation que j'ai réalisée en SSE, uniquement les types `__m128` et `__m128i` et les 9 fonctions intrinsèques suivantes :

```
_mm_set_ps1        _mm_store_si128   _mm_unpacklo_epi16
_mm_unpacklo_epi8  _mm_packus_epi16  _mm_packus_epi32
_mm_cvtps_epi32    _mm_set1_epi32    _mm_setr_ps
```

Il est cependant possible d'implémenter l'algorithme en utilisant d'autres instructions, la liste n'est donc ni exhaustive ni contraignante. Le détail du comportement de ces instructions est disponible sur le site d'Intel à l'url suivante (indispensable de l'avoir ouverte pendant le TP !) : `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#expand&techs=SSE,SSE2,SSE3,SSSE3,SSE4_1,SSE4_2`. On se limitera cependant à SSE/SSSE, donc au SIMD 128-bit (pas d'AVX, sauf si vous voulez faire une version 256-bit après avoir fait la version 128-bit).

Le TP utilise un décodeur vidéo Motion-JPEG (celui du projet C de 1ère année pour être précis), dans lequel nous allons tenter d'optimiser une fonction particulière, celle qui assure la conversion des composantes de luminance et chrominance en rouge, vert, bleu pour l'affichage d'une image dans un *frame buffer*.

3 Travail demandé

3.1 Préliminaires

Si vous comptez travailler sur votre machine, il conviendra probablement d'installer la librarie SDL afin d'afficher ce que l'on décode à l'écran. Sur Ubuntu :

```
sudo apt install libsdl-sound1.2 libsdl-net1.2 libsdl-gfx1.2-5 libsdl-image1.2 libsdl-mixer1
.2 libsdl-ttf1.2 libsdl1.2-dev
```

On travaillera dans le répertoire `3a-archi/TP/tp-simd/tp2_src_etd`, qui contient une vidéo `Ice_age_256x144_444.mjpeg` qui sera notre benchmark, un `Makefile`, des fichiers objets contenant les différentes phases du décodage hormis la conversion YUV vers RGB, et des fichiers sources contenant diverses implantations (dont certaines partielles) de la conversion.

`conv-float.c`
: la version initiale en virgule flottante ;

`conv-int.c`
: la version en entiers ;

`conv-unrolled4-a-trou.c`
: la version qui est le sujet de la question 1 ;

`conv-sse-a-trou.c`
: la version qui est le sujet de la question 2.

L'exécutable crée prend 1, 2 ou 3 arguments. Le premier est le nom de la vidéo, le second est un entier donnant le nombre de frames à décoder sauf s'il vaut -1 auquel cas la totalité de la vidéo est décodée, et le troisième n'importe quoi. En présence de ce troisième argument, le résultat du décodage n'est pas affiché (à 25 images par secondes), c'est ainsi que la vitesse brute du décodage peut être mesurée. Afin de voir combien de temps prend le décodage (sur le film complet), on peut lancer :

```
petrot@tilleul% time ./mjpeg-float ice_age_256x144_444.mjpeg -1 zyva
./mjpeg ../src/ice_age_256x144_444.mjpeg -1 zyva  3,88s user 0,02s system 99% cpu 3,905 total
```

Le SSE permettant d'exploiter le parallélisme des données, on va chercher à le mettre en évidence dans le code existant. Si la première version des instructions SIMD d'Intel travaillait uniquement en entiers, les instructions SSE visent essentiellement le flottant, aussi vous travaillerez pour le calcul de la conversion proprement dite avec des flottants, et plus précisément 4 flottants 32 bits dans un `__m128`.

Question 1

Proposez une nouvelle version de cette fonction, *toujours écrite en C*, qui mette en évidence un parallélisme de degré 4 sur la boucle interne. Il n'est pas possible d'exprimer le parallélisme dans le C standard, aussi on utilisera des tableaux de taille 4. On déroulera explicitement les boucles de 0 à 3, ce qui nécessite de la recopie de code. Il est clair que ce n'est pas une pratique de programmation recommandée, mais on fera une exception cette fois, c'est pour la bonne cause.

Ceci est à faire dans le fichier `conv-unrolled4-float-a-trou.c` dans lequel il y a quelques commentaires pour vous indiquer quoi faire. Il s'agit ici de recopier brutalement ce qui est dans `[language=bash]conv-float.c` en répliquant les lignes qui vont bien et en ajustant les indices de boucles. Notez bien que si le type de `R`, `G` et `B` est `int32_t`, la « promotion » des types du C fait que les calculs sont exécutés en flottant avant d'être converti en entier lors de l'affectation dans ces variables.

Relancez l'exécution pour vérifier que le décodage de la vidéo est bien celui attendu.

Quelques informations supplémentaires

Comme nous travaillons avec un parallélisme de degré 4 et que les registres sont de 128 bits, la taille des données manipulées sera de $\frac{128}{4} = 32$ bits pour les calculs. Les instructions SSE supportent le flottant, donc nous utiliserons le type `__m128` pour la conversion proprement dite, mais comme le pixel de l'image est constitué de 8 bits de transparence (α), 8 bits de rouge, 8 bits de vert et 8 bits de bleu, il faudra se ramener à des octets avant de mettre à jour le macro-bloc de sortie. On utilisera donc pour cette seconde phase des fonctions permettant de faire la conversion entre flottants et entiers, et entre entiers de tailles différentes. Plus fort, il existe des instructions qui font ces conversions en effectuant des saturations, ce qui permet d'éviter les tests d'appartenance de la composante c du pixel à l'intervalle $0 \leq c \leq 255$.

Question 2

Ouvrez le fichier `conv-sse-a-trou.c` et suivez les consignes qui s'y trouvent. La mauvaise nouvelle est qu'il n'est pas possible de changer par étapes successives le code, car les données sont dans des registres spéciaux avec des formats spéciaux, et que la conversion vers les formats du C n'est pas immédiate (*c.f.* l'implantation de la fonction `p128_x`). Donc soit gdb est votre ami (qui affiche les

`__mm128` selon toutes les configurations possibles), soit vous utilisez la fonction `p128_x` qui permet d'afficher le contenu d'un registre (que vous pouvez comparer à ce que vous obtenez lorsque vous exécutez la version initiale, c'est ainsi que j'ai débuggé). A noter : les variables sont données à titre indicatif et vous avez le droit de ne pas toute les utiliser ou d'en utiliser plus.

Une dernière chose, la commande `objdump -d executable | less` vous affichera les instructions du binaire (il suffit alors de chercher la bonne fonction). Utile si vous voulez voir ce que donne la traduction C vers assembleur avec instructions SSE.

Question 3

Compilez votre version et les différentes versions fournies, décodez le film full patate, et faites un graphe du temps de décodage des différentes implantations. Conclusion ?