Introduction aux instructions vectorielles SIMD

Avec le SSE x86

M. Malek Bengougam
malek.bengougam@gmail.com

Quel intérêt ?

- Instruction Level Parallelism : pouvoir exécuter des opérations sur un groupe de données (contigües)

- SIMD – Single Instruction Multiple Data

- Il s'agit d'appliquer la même opération (single instruction) sur le groupe de donnée (multiple data).

Opérations vectorielles

- Unité de calcul souvent dénommé **VFPU** pour '*Vectorial Floating Point Unit*', car initialement prévu pour le calcul vectoriel, mais en pratique s'utilise pour ce qui peut être parallélisé.

- **SSE** et **AVX** = Intel / AMD (aussi PS4 et Xbox One)
- **NEON** = architectures ARM (mobiles, PS Vita…)
- **Altivec/VMX** = PowerPC Motorola/IBM (PS3/360)
- Les **SPU**s de la PS3 fonctionnent sur ce principe.

Processeurs SIMD x86/x64 (1)

- **MMX** apparu avec le **Pentium MMX**. Registres de **64** bits partagés avec le FPU, au nombre de 8. Ils peuvent contenir des groupes (pack) de 8/16/32/64 bits (entiers seulement).

- **3DNow!** extension du **MMX** développée par **AMD** (Intel ne les a jamais intégré), registres float séparés de 64/32 bits.

- **SSE** (Streaming SIMD Extension) successeur du **MMX**, et réponse au **3DNow!** Registres de **128** bits, groupes de 32 bits (float).

- Le jeu d'instruction du **SSE1** est essentiellement axé sur les *float* 32 bits. Le **SSE2** ajoute particulièrement des instructions pour la manipulation de groupes d'*entiers*.

Processeurs SIMD x86/x64 (2)

- Le **SSE3** ajoute des instructions plus puissantes (additions horizontales, dot product etc…)

- Les dernières versions du SSE sont **SSE4.2** (seulement pour Intel) et **SSE4a** / **SSE5** (pour le moment spécifique à AMD).

- **AVX** dernière variante en date (architecture Core i3/i5/i7 chez Intel, et certains processeurs AMD récents) propose des registres de 256 bits.

Types du SSE

- Pour faciliter la programmation, la plupart des compilateurs définissent des types vectoriels.

- **__m64** registre MMX / 3DNow! (MM0-7).
- **__m128** registre SSE (XMM0-7).
- **__m256** registre AVX et AVX2 (YMM0-15).

- Ce type fonctionne comme le type vec4 en GLSL mais seul l'opérateur = est défini. Jeu d'instructions séparé.

- Les architectures x86-64 et AMD64 ajoutent 8 registres supplémentaires (XMM8-15) au SSE et retirent les registres MMX.

Note: La future extension AVX-512 disposera de 32 registres 512 bits

Fonctions intrinsèques (1)

- Le compilateur met à disposition des fonctions bas niveaux dites ‘intrinsèques’ (intrinsics) qui sont directement converties en leur équivalent machine au moment de la compilation.

- Parfois il faut activer une option dans les propriétés du projet pour que le compilateur les prenne en compte.

- Sous Visual Studio cela se trouve dans le panneau « C/C++ », sous-panneau « Optimisation » : « Activer les fonctions intrinsèques ».

  Note: ceci n’est pas nécessaire pour les instructions SSE

Fonctions intrinsèques SSE (2)

- Fichier d'en-tête requis

  #include <xmmintrin.h> // SSE (**mmintrin.h** pour le MMX)

  #include <emmintrin.h> // SSE2

  #include <immintrin.h> // SSE3 et SSE4.x, AVX

  #include <ammintrin.h> // SSE4a AMD

  Cf. doc du compilateur et manuel d'optimisation du processeur pour plus de détail

Instructions SSE

Nomenclature intrinsics SSE

- Les instructions sont toujours préfixées par **_mm_**.

- Il existe deux types d'instructions:
- Des instructions vectorielles (dites '*packed*') agissant sur plusieurs éléments en parallèle.
- Des instructions scalaires (*scalar*) similaire au FPU.

- En SSE ces instructions se distingue par leur suffixe **_ps**() pour les instructions vectorielles (ps = packed scalars) et **_ss**() pour les instructions scalaire (ss = single scalar).

Instructions simples

- Opérations arithmétiques standard:

_mm_add_ps(), **_mm_sub_ps**(), **_mm_mul_ps**(), **_mm_div_ps**()

*Ces instructions existent aussi en **_ss()***

- Opérations logiques:

- **_mm_and_ps**(), **_mm_or_ps**(), **_mm_xor_ps**(), **_mm_andnot_ps**()

Exemple: addition

__m128 A = {…}, B = {…};
__m128 C = **_mm_add_ps**(A, B);

	|A.x | A.y | A.z | A.w |
+	|B.x | B.y | B.z | B.w |
=	|C.x | C.y | C.z | C.w |

- L'opérateur ici est l'addition (+), Les opérandes sont A et B, le résultat étant C.

  L'instruction produit une somme **verticale** de chaque colonnes séparément (float 32 bits).

Instructions avancées

- **_mm_rcp_ps**() : fonction réciproque (y=1/x)
- **_mm_sqrt_ps**() : racine carrée
- **_mm_rsqrt_ps**() : réciproque de la racine carrée

- **_mm_max_ps**() et **_mm_min_ps**() : retourne un vecteur composé respectivement du maximum ou du minimum de chaque composants des opérandes.

- *Toutes ces instructions existent en **_ss()***

Instructions complexes (1.a)

- L'instruction **_mm_shuffle_ps**(*a,b,mask*) est une instruction puissante qui permet de mélanger ou permuter (*shuffle*) le contenu des deux opérandes selon un mode dicté par le masque.

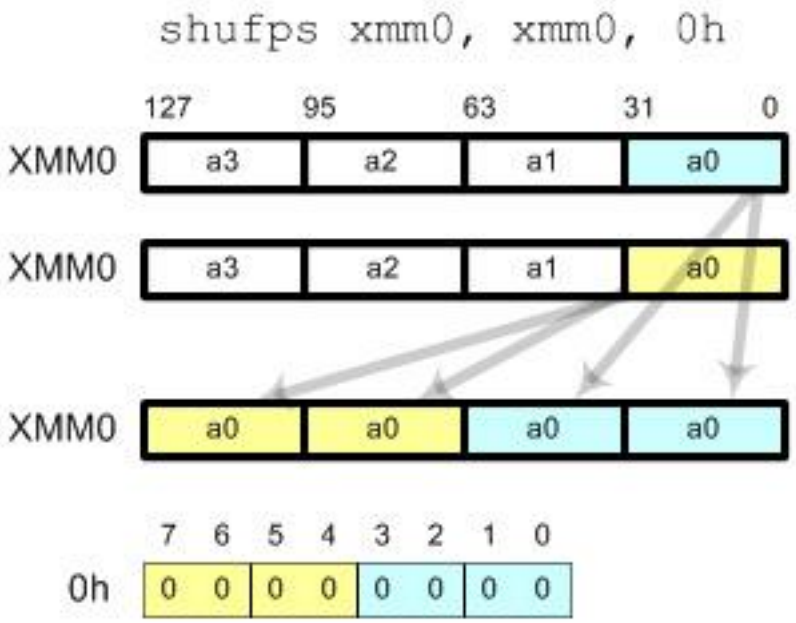
- Cette instruction met toujours les éléments de la 1ère opérande (ici '***a***') dans les deux premiers éléments de la destination, et les éléments de la 2nde opérandes (ici '***b***') dans les deux derniers éléments de la destination.

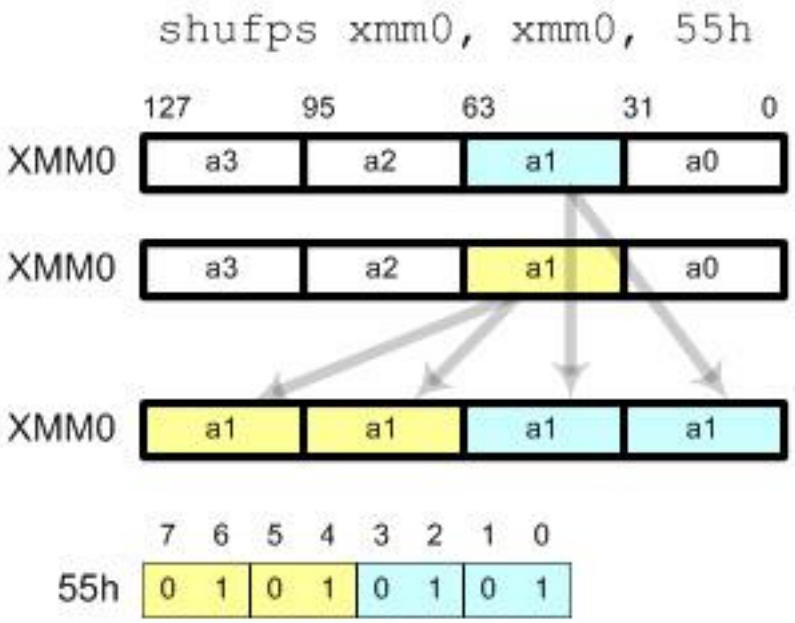
- Le masque est composé de 4x2 bits, chaque sous groupe de 2 bit représente l'indice de l'élément à traiter (0 à 3 donc).

- Peut aussi servir à répliquer une valeur scalaire sur tout un registre (aussi appelé *splatting*, *to splat*=étaler, ou encore *broadcasting*)

Instructions complexes (1.b)

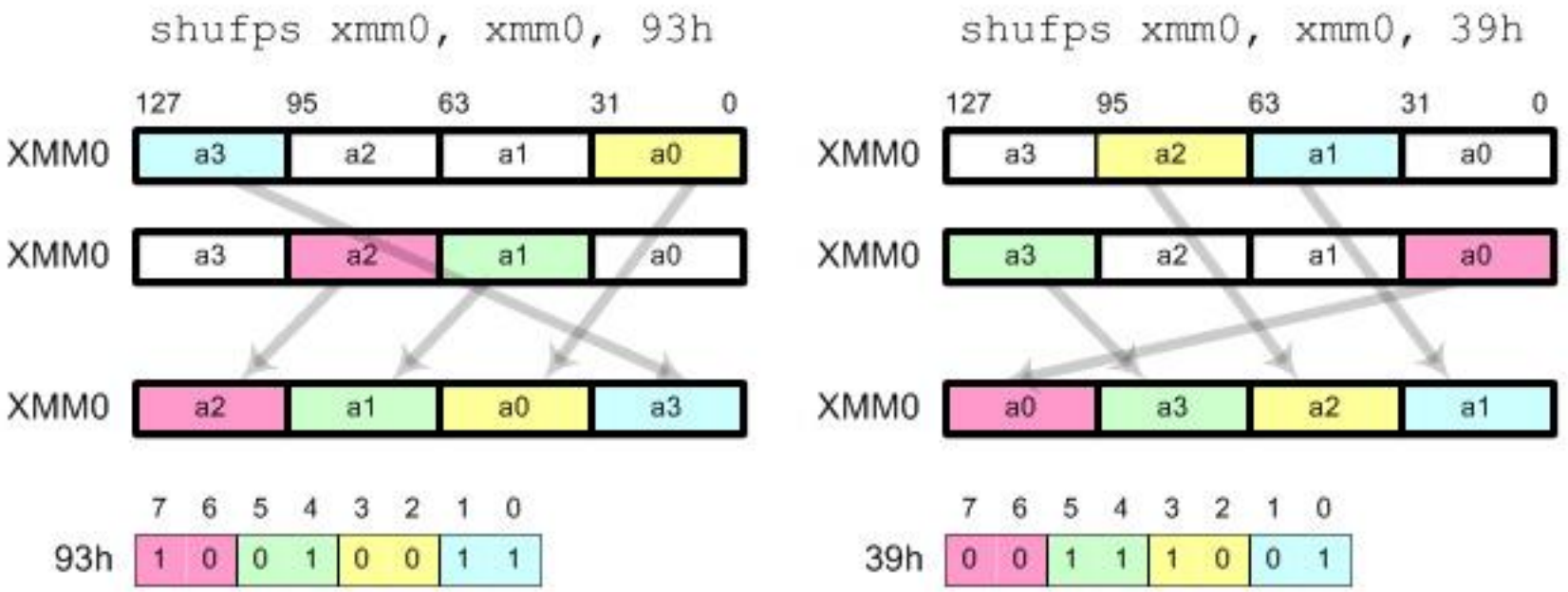
- Copie





Instructions complexes (1.c)

- Rotation



Instructions complexes (2.a)

- dépaquetage (*Unpack*)
- Consiste à prendre certains éléments des opérandes sources pour les stocker dans le registre de destination.

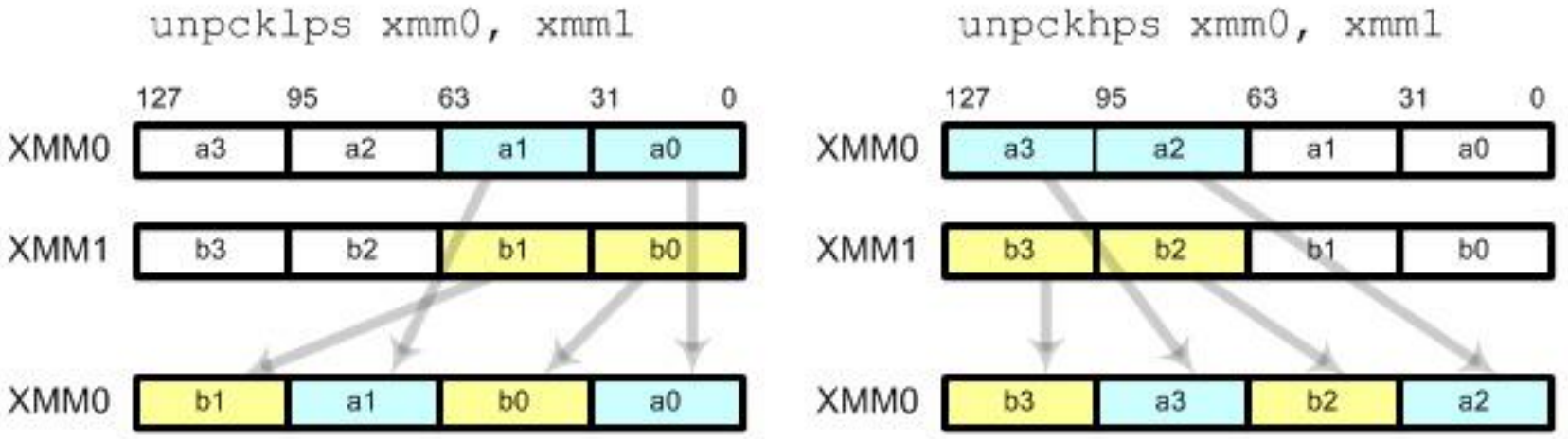
- **_mm_unpckl_ps**() : ‘l’ pour ‘low’; agit sur les deux premiers éléments des opérandes (poids faibles).

- **_mm_unpackh_ps**() : ‘h’ pour ‘high’; agit sur les deux derniers éléments des deux opérandes (poids forts).

- Le paquetage (*pack*) se fait généralement à l’aide de la fonction _mm_shuffle_ps() en SSE, ou des fonctions spécialisées sur les entiers en SSE2.

Instructions complexes (2.b)

- Ces instructions permettent de copier et entrelacer le contenu des opérandes selon le schéma suivant:



Load et Store (1)

- Pour affecter un élément scalaire d'un registre:
- **_mm_set_ss**()

- Pour affecter une valeur 32 bits à tout le registre:
- **_mm_set1_ps**() ou **_mm_set_ps1**()
- Ou **_mm_setzero_ps**() pour initialiser à zéro.

- Pour affecter les quatres composantes 32 bits:
- **_mm_set_ps**()
- **_mm_setr_ps**() idem mais 'r' pour 'reverse'

Load et Store (2)

- Pour lire et stocker des variables en mémoire il est nécessaire d'utiliser des fonctions spécifiques.
- **_mm_load_ss**() / **_mm_store_ss**()
- **_mm_load_ps**() / **_mm_store_ps**()

Précision

- La plupart des instructions complexes (types sqrt, rsqrt etc…) utilisent en pratique des méthodes approximatives. Ce qui fait que le résultat peut être différent de l'équivalent en float.

- On peut augmenter la précision en effectuant une ou plusieurs itération suivant la méthode de Newton-Raphson. Ici pour rsqrt:

```
__m128 three = _mm_set1_ps(3.0f), half = _mm_set1_ps(0.5f);
__m128 res, muls;
res = _mm_rsqrt_ps(x);
// x' = x*r*r
// res0 = r/2.0 * 3.0 – x'
muls = _mm_mul_ps(_mm_mul_ps(x, res), res);
res = _mm_mul_ps(_mm_mul_ps(half, res), _mm_sub_ps(three, muls));
```

Appendice A

Alignement des données pour le traitement SIMD

Alignement

- Consiste à s'assurer que l'adresse mémoire, ou relative, d'un élément ou d'une structure soit multiple du fonctionnement naturel du processeur.

- Le 'padding' (bourrage, remplissage) automatique du compilateur participe de cela pour le code c++ mais il s'agit là d'alignement interne.

- Le compilateur ne peut aligner automatiquement les adresses mémoires, seulement les adresses relatives (offsets).

Pourquoi recourir à l'alignement ?

- Si les données sont correctement alignées en mémoire le chargement est **optimal**. C'est le fonctionnement idéal.

- Si les données ne sont pas correctement alignées en mémoire mais que le processeur dispose d'un jeu d'instructions pouvant lire les données non alignées (***unaligned***), il y'a une latence plus importante.

- Si les données ne sont pas correctement alignées en mémoire avec utilisation du jeu d'instruction standard (***aligned***) ou si le processeur ne supporte que les adresses alignées le programme plantera lors de l'accès à la première adresse incorrecte !

Alignement des données (1)

- Les registres du SSE étant de 128 bits, l'alignement naturel des données est de 16 octets (4*32 bits = 16*8 bits).

- Ce qui signifie d'une part que le compilateur alignera toujours (si nécessaire avec bourrage) une structure contenant un type vectoriel sur 16 octets.

- **Note**: Le SSE peut lire des données mal alignées en utilisant les fonctions adéquats et sous peine d'une latence plus importante en lecture et écriture.

Alignement des données (2)

- Cependant cet alignement automatique ne peut être fait lorsque la structure est allouée dynamiquement.

- C'est donc le rôle du programmeur de s'assurer que les données en mémoire sont alignées correctement.

- L'adresse de la structure allouée doit être multiple de 16 (0x10 en hexadécimal). En pratique, les 4 derniers bits de l'adresse doivent être à zéro. Exemples :

  - *0xBAAD1234* n'est aligné sur 16 octets (mais sur 4)
  - *0x900D1230* est bien aligné sur 16 octets

Alignement des données (3)

- La méthode usuelle pour aligner des données allouées dynamiquement est:

- a. allouer plus de place que nécessaire - au minimum (alignement-1) octets
- b. stocker l'adresse originale de l'allocation (afin de pouvoir libérer la mémoire correctement)
- c. aligner l'adresse vers le haut, câd arrondir au premier multiple supérieur de l'alignement
- d. effectuer toutes les opérations sur cette nouvelle adresse.

références

- http://msdn.microsoft.com/en-us/library/t467de55(v=vs.100).aspx
- http://msdn.microsoft.com/fr-fr/library/y0dh78ez(v=vs.110).aspx
- http://msdn.microsoft.com/fr-fr/library/7t5yh4fd.aspx

- http://www.songho.ca/misc/sse/sse.html

- http://www.cortstratton.org/articles/OptimizingForSSE.php