TD 1 - Architectures Spécialisées

Mercredi 13-01-2021, 13h45

1. Vectorisez les programmes ci-dessous avec des instructions `sse` *intrinsics*. On considère que les données sont alignées en mémoire.

   (a)
   ```
   double a[N], b[N], c[N];
   ...
   for (i = 0; i < N; i++)
     a[i] = b[i] + c[i];
   ```

   (b)
   ```
   char a[N];
   ...
   for (i = 0; i < N; i++)
     a[i] = i;
   ```

   (c)
   ```
   int a[N], x = 0;
   ...
   for (i = 0; i < N; i++)
     x = x + a[i];
   ```

   (d)
   ```
   float a[N], x = 0.0;
   ...
   for (i = 0; i < N/3; i++)
     x = x + a[3*i];
   ```

   (e)
   ```
   float a[N], b[N], c[N];
   ...
   for (i = 0; i < N; i++)
     if (a[i] > 0)
       a[i] = b[i] / c[i];
   ```

2. écrivez un programme qui calcule les valeurs `min` et `max` du tableau `float Tab[N]` en exploitant des instructions `sse` *intrinsics*

3. écrivez un programme qui calcule le produit scalaire entre deux vecteurs de 4 `float` en exploitant des instructions `sse` *intrinsics*

ANNEXE Rappel du formalisme *intrinsics* des opérations *SSE*

Type de données `__m128{|d|i}` registre : `__m128` *single precision*, `__m128d` *double*, `__m128i` *integer*

Format des fonctions `_mm_op_suffix()` où `op` est une opération et `suffix` le type des données :

- pour les flottants `{s|p}{s|d}` : scalaire (s) ou parallèle (p) suivi de simple (s) ou double (d) précision
- pour les entiers `si128|ep{i|u}{8|16|32|64}` : `si128` pour tout le registre ou *extented paqued* (ep) suivi de i# entier ou u# entier non signé de # bits

Opérations sur tout le vecteur `load, store, and, andnot, or, xor`

`__m128{|d|i} _mm_load_{ps|pd|si128} ({float|double|__m128i}* mem_addr)`
charge 128bits de l'adresse mémoire `mem_addr` (`loadu` si pas alignée) dans un registre

`void _mm_store_{ps|pd|si128} ({float|double|__m128i}* mem_addr, __m128{|d|i} a)`
enregistre 128bits du registre `a` vers l'adresse `mem_addr` (`storeu` si pas alignée)

`__m128{|d|i} _mm_{and|or|xor}_{ps|pd|si128} (__m128{|d|i} a, __m128{|d|i} b)`
opération logique ET, OU ou OU exclusif bit à bit entre les deux registres `a` et `b`

`__m128{|d|i} _mm_andnot_{ps|pd|si128} (__m128{|d|i} a, __m128{|d|i} b)`
ET logique bit à bit entre le complément (NON) du registre `a` et le registre `b`

Entiers , opérations sur des parties du vecteur : `load, store, move`

`__m128i _mm_loadl_epi64 (__m128i* mem_addr)`
charge 64bits de l'adresse `mem_addr` dans la partie basse du registre et zéro partie haute

`void _mm_storel_epi64 (__m128i* mem_addr, __m128i a)`
enregistre 64bits de la partie basse du registre `a` vers l'adresse `mem_addr`

`__m128i _mm_move_epi64 (__m128i a)`
copie 64bits de la partie basse du registre `a` dans la partie basse du registre et zéro partie haute

Opérations de comparaisons : `cmp{eq|ge|gt|le|lt|neq}`

`__m128{|d|i} _mm_cmpeq_{{p|s}{s|d}|epi{8|16|32|64}} (__m128{|d|i} a, __m128{|d|i} b)`
compare les données des registres `a` et `b`, renvoie le résultat sous forme de masque binaire

Opérations arithmétiques sur les flottants : `add|sub|mul|div|max|min|sqrt|hadd|hsub`

`__m128{|d} _mm_{add|sub|mul|div|max|min|sqrt}_{p|s}{s|d} (__m128{|d} a, __m128{|d} b)`
opération ($+$, $-$, *, /, max, min, $\surd$) entre les registres `a` et `b`.

`__m128{|d} _mm_{hadd|hsub}_p{s|d} (__m128{|d} a, __m128{|d} b)`
addition ou soustraction horizontale entre données adjacentes deux-à-deux dans `a` et `b`.

Opérations arithmétiques sur les entiers : `add|sub|abs|sign|adds|subs|avg|mul`

`__m128i _mm_{add|sub}_epi{8|16|32|64} (__m128i a, __m128i b)`
addition ou soustraction signée entre les registres `a` et `b`.

`__m128i _mm_{abs|sign}_epi{8|16|32} (__m128i a, __m128i b)`
absolue ou négation des valeurs dans les registres `a` et `b`.

`__m128i _mm_{adds|subs}_ep{i|u}{8|16} (__m128i a, __m128i b)`
addition ou soustraction saturée, signée ou non entre les registres `a` et `b`.

`__m128i _mm_avg_epu{8|16} (__m128i a, __m128i b)`
moyenne non signée entre les registres `a` et `b`.

`__m128i _mm_mul_epu32 (__m128i a, __m128i b)`
multiplication non signée 32 bits entre parties basses des registres `a` et `b`, résultat sur 64 bits.

`__m128i _mm_mulhi_ep{i|u}16 (__m128i a, __m128i b)`
multiplication sur 16 bits entre `a` et `b`, renvoie la partie haute sur 16 bits du résultat.

`__m128i _mm_mullo_epi16 (__m128i a, __m128i b)`
multiplication sur 16 bits entre `a` et `b`, renvoie la partie basse sur 16 bits du résultat.

`__m128i _mm_sad_epu8 (__m128i a, __m128i b)`
calcule la valeur absolue des différences entre `a` et `b`, puis la somme horizontale entre données adjacentes deux-à-deux , résultat sur 16 bits.

`__m128i _mm_{max|min}_ep{u8|i16} (__m128i a, __m128i b)`
max ou min entre les registres `a` et `b`.

Opérations de conversion de type : `cvtepi32|cvtpd|cvtps|cvttt`

`__m128{|d} _mm_cvtepi32_{ps|pd} (__m128i a)`
conversion entier 32 bits vers flottant simple ou double précision.

`__m128{|i} _mm_cvtpd_{ps|epi32} (__m128d a)`
conversion du registre `a` vers flottant simple précision ou un entier sur 32 bits avec arrondi.

`__m128{d|i} _mm_cvtps_{pd|epi32} (__m128 a)`
conversion du registre `a` vers flottant double précision ou un entier sur 32 bits avec arrondi.

`__m128i _mm_cvttt{ps|pd}_epi32 (__m128{|d} a)`
conversion du registre `a` vers entier sur 32 bits avec troncature.

Opérations d'affectation de valeurs constantes : `set|setr|set1|setzero`

`__m128{|d|i} _mm_set_{{p|s}{s|d}|epi{8|16|32|64}} ({char|short|int|float|double} x[,...])`
renvoie un vecteur composé des valeurs passées en paramètre.

`__m128{|d|i} _mm_setr_{{p|s}{s|d}|epi{8|16|32|64}} ({char|short|int|float|double} x[,...])`
renvoie un vecteur composé des valeurs passées en paramètre lues en sens inverse.

`__m128{|d|i} _mm_set1_{p{s|d}|epi{8|16|32|64}} ({char|short|int|float|double} x)`
renvoie un vecteur composé de copies de la valeur passée en paramètre.

`__m128{|d|i} _mm_setzero_{p{s|d}|si128}} ()`
renvoie un vecteur composé de zéros.

exemples : Initialise chacun des 4 entiers signés de 32 bits du vecteur `ones4` avec la valeur 1

```
__m128i ones4 = _mm_set1_epi32(1);
```

Chargement aligné de 4 *floats* de l'adresse mémoire `in` vers le vecteur `a` :

```
__m128 a =  _mm_load_ps(in);
```

Copie alignée du vecteur `a` à l'adresse mémoire `out` :

```
_mm_store_si128(out,a);
```

Multiplie 2 vecteurs de 4 *floats* :

```
__m128 a, b;
__m128 x = _mm_mul_ps(a, b);
```

Additionne 2 vecteurs de 8 entiers signés sur 16 bits avec saturation

```
__m128i a, b;
__m128i x = _mm_adds_epi16(a, b);
```

Calcule le min de 2 vecteurs de 4 *floats* :

```
__m128 a, b;
__m128 x = _mm_min_ps(a,b);
```

Compare 2 vecteurs de 4 *floats* : `a<b`

```
__m128 a, b;
__m128 x = _mm_cmplt_ps(a,b);
```

Réalise un ET logique entre 2 vecteurs de 4 *floats* en inversant le premier : `NON a ET b`

```
__m128 a, b;
__m128 x = _mm_andnot_ps(a,b);
```

Réalise une addition horizontale de deux vecteurs de 4 *floats* :

```
__m128 a, b;
__m128 x = _mm_hadd_ps(a,b); /* x = {b0+b1, b2+b3, a0+a1, a2+a3} */
```