Calcul haute performance : notions de base (HPC)

Cours 6 : programmation SIMD

P. Fortin – Sorbonne Université

M1 Info SFPN / MAIN4

Bibliographie et remerciements

- Intel Advanced Vector Extensions Programming Reference : http://software.intel.com/en-us/avx
- Intel Intrinsics Guide : https://software.intel.com/sites/landingpage/IntrinsicsGuide
- *Programmation vectorielle/SIMD*, S. Jubertie
- *Introduction to Intel Advanced Vector Extensions*, C. Lomont
- *OpenMP 4.0 for HPC in a Nutshell*, M. Klemm, Intel
- Standard OpenMP 4.0, http://openmp.org/wp/

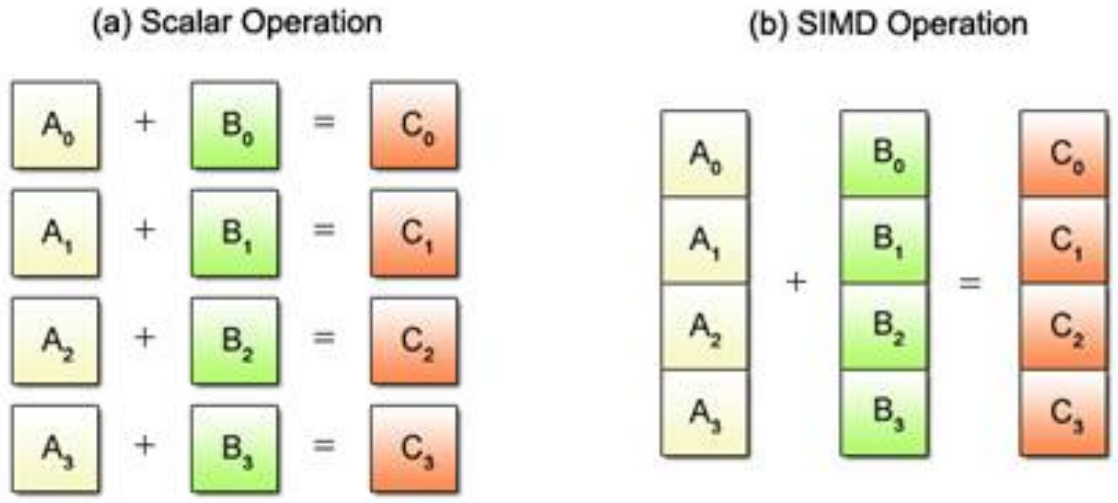
Le calcul vectoriel / SIMD

- SIMD : *Single Instruction – Multiple Data*
  - même instruction appliquée simultanément à tous les éléments (scalaires) d'un (ou plusieurs) « vecteurs »
  - au niveau matériel : unité SIMD composée de *lanes* (« voies »)
- Exemple : addition terme à terme de 2 vecteurs de 4 éléments
  - En scalaire :

```
C[0] = A[0] + B[0];
C[1] = A[1] + B[1];
C[2] = A[2] + B[2];
C[3] = A[3] + B[3];
```

  - En vectoriel :

```
C[0:3] = A[0:3] + B[0:3];
```



(*Cell Programming Primer*, Sony)

Historique

- Années 80 et 90 : machines vectorielles (Cray, NEC, Hitachi, Fujitsu ...)
- Depuis les années 90, processeurs scalaires dotés de « petites » unités vectorielles :
  - Architecture x86 (Intel et AMD) :
    - MMX
    - 3DNow!
    - SSE, SSE2, SSE3, SSSE3, SSE4.1 et SSE4.2 (Intel), SSE4a (AMD)
    - AVX, AVX2
  - PowerPC (IBM, Motorola) : AltiVec
  - ARM Neon : Cortex-A8 et A9
- Et aussi :
  - Intel Xeon Phi
  - GPU

Importance croissante du calcul SIMD

- Taille des registres des unités SIMD :
  - MMX (1997) : registres 64 bits (entiers)
  - 3DNow! (1998) : registres 64 bits (entiers et flottants)
  - SSE (1999 → 2007) et AltiVec (1999) : registres 128 bits (entiers et flottants)
  - AVX 1 (2011) : 256 bits (flottants)
  - AVX 2 (2013): 256 bits (entiers et flottants)
  - Intel Xeon Phi (première génération, 2013) : 512 bits (entiers et flottants)
  - AVX-512 (2016 pour 2ième génération de Xeon Phi, et 2017 pour CPU Intel) : 512 bits (entiers et flottants)
  - Et aussi GPU : exécution SIMD sur 32 ou 64 éléments de 32 bits chacun
- Unités SIMD : permettent d'augmenter la puissance de calcul à moindre coût (architectural)
- Mais nécessitent :
  - Efforts de développement
  - Algorithmes adaptés

Programmation SIMD

- Plusieurs paradigmes de programmation possibles :
  - Programmation assembleur :
    - Le plus « bas-niveau »
    - Manipulation explicite des registres et des instructions
  - Les *intrinsèques (intrinsics)* :
    - Programmation C/C++ avec fonctions C « intrinsèques » remplacées directement à la compilation par les instructions assembleur correspondantes
  - Vectorisation automatique par le compilateur
    - Le compilateur doit pouvoir déterminer par lui même (analyse des dépendances) que la boucle visée est parallèle
    - Exemples de contraintes :
      - pas de «chevauchement des zones mémoire pointées» (*pointer aliasing*),
      - alignement mémoire adapté,
      - seule la boucle la plus interne est visée dans un nid de boucles,
      - pas d'appel de fonctions dans le corps de boucle, …

Programmation SIMD (2)

- Plusieurs paradigmes de programmation possibles (suite) :
  - Directives de compilation
    - Permet d'indiquer au compilateur que la boucle visée est parallèle → permet de surmonter les limites de l'analyse des dépendances par le compilateur
    - Permet de définir ses propres fonctions vectorielles
    - Divers jeux de directives :
      - compilateurs Intel C/C++,
      - Intel Cilk Plus,
      - OpenMP 4.0
  - Méta-programmation : par exemple via des templates C++
  - ...
- Dans ce cours : intrinsèques AVX2 et directives OpenMP 4.0

SIMD et performances

- Principe général en programmation assembleur / intrinsèques (pour les parties les plus calculatoires du programme) :
  - Chargement des données de la mémoire vers les vecteurs SIMD
  - Opérations SIMD sur les vecteurs SIMD (mis dans les registres SIMD)
  - Écriture du résultat depuis les vecteurs SIMD vers la mémoire

  → minimiser le nombre de chargements et d'écritures vers/depuis les vecteurs SIMD

SIMD et performances (2)

- Alignement des données
  - Rappel : une adresse en mémoire désigne un octet.
  - Donnée de N octets considérée conme « alignée en mémoire » si stockée à une adresse multiple de N.
    - Vecteur de 128 bits : à aligner sur 16 octets
    - Vecteur de 256 bits : à aligner sur 32 octets
      → Le matériel peut effectuer efficacement un accès vectoriel en mémoire lorsque les données sont alignées
      → Données non alignées : nécessitent plusieurs accès mémoire
      → Privilégier les données alignées en mémoire
  - Accès mémoire en SSE/AVX : besoin d'expliciter les accès mémoire alignés et ceux non alignés

SIMD et performances (3)

- Connaître l'alignement par défaut d'un type en mémoire :

```
__alignof__ (type)
```

- Alignement de données statiques :

```
int x __attribute__ ((aligned (16))) ;
```

→ alignement sur 16 octets

- Alignement de donnés dynamiques :

```
int posix_memalign(void **memptr, 32,
                   NbElts*sizeof(type));
```

→ allocation mémoire et alignement sur 32 octets

→ Voir aussi : `aligned_alloc()` (C11)

SIMD et performances (4)

- « Remplissage » (*padding*) pour la boucle parallélisée en SIMD
  - si nombre d'itérations ≠ multiples de la taille du vecteur SIMD

    → généralement besoin de compléter par des zéros le dernier vecteur SIMD et/ou les tableaux en mémoire
- Gestion de la divergence
  - Au niveau des accès mémoire :
    - SSE, AVX : uniquement chargement et écritures vectoriels pour des données contigues
      → Si non contiguité : successions d'opérations scalaires
    - AVX2 : chargement vectoriel non contigu (*gather*)
    - Xeon Phi : *gather* et *scatter*

SIMD et performances (5)

- Gestion de la divergence
  - Au niveau des calculs, divergence possible entre les lanes :
    - pour les branchements conditionnels :
      `if (…) then … else …`
      → certaines lanes devront effectuer les instructions de la branche `then` et les autres lanes celles de la branche `else`
    - pour les boucles `while`, internes au code SIMD, avec un nombre d'itérations différent par lane → certaines lanes ont quitté la boucle alors que les autres lanes sont encore en train d'exécuter la boucle
    - A chaque fois : impact négatif sur les performances dû à la diminution du degré de parallélisme
  - Gestion en fonction de l'architecture :
    - SSE, AVX :
      - Toutes les lanes font toutes les instructions
      - Les effets de bord non voulus sont évités grâce à des *vecteurs de masquage* → surcoût supplémentaire
    - Xeon Phi : vecteurs de masquage intégrés aux instructions SIMD

SIMD et performances (6)

- Organisation des données en mémoire
  - Tableau de structures (*Array of structures – AoS*) :

    `xyzxyzxyz…`
  - Structure de tableaux (*Structure of arrays – SoA*) :

    `xxxxxxxxx...`

    `yyyyyyyyy...`

    `zzzzzzzzz...`
- Si accès mémoire coûteux : favoriser SoA
- Autre possibilité : structure «hybride SoA» adaptée à la longueur du vecteur SIMD

  `xxxxyyyyzzzzxxxxyyyyzzzz...`

AVX2

- CPU avec AVX2 :
  - Intel : Haswell (2013) et suivants
  - AMD : Excavator (2015) puis Ryzen (2017)
  - Pour vérifier le support SIMD de son CPU (sous linux) :

    ```
    cat /proc/cpuinfo
    ```

    Champ `flags` : sse, sse2, pni (Prescott New Instructions – SSE3), ssse3, sse4_1, sse4_2, avx, avx2

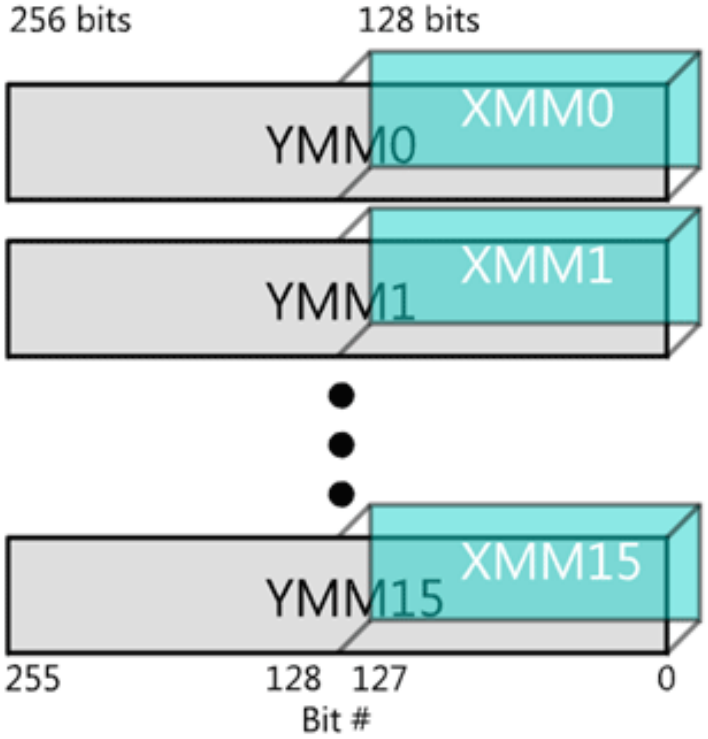
- Evolution incrémentale du jeu d'instruction :
  - AVX(N) = nouvelles instructions + AVX(N-1)

AVX2 : registres et instructions

- On considère une architecture x86 en mode 64 bits (x86_64)
- Registres :
  - 16 registres de 256 bits : YMM0 → YMM15
  - 1 registre de 32 bits d'état/contrôle : MXCSR
- Types d'instructions :
  - accès mémoire ↔ vecteurs AVX
  - opérations arithmétiques
  - opérations logiques
  - tests
  - ...
- Même jeu de fonctions intrinsèques pour compilateurs : gcc/g++, Intel C/C++ et Visual Studio C/C++

AVX2 : registres et instructions (2)

- Les registres SSE de 128 bits (XMM0 → XMM15) sont toujours accessibles via la moitié «basse» des registres YMM



(C. Lomont)

- Possible de programmer en SSE sur unités AVX[2]

Définitions et compilation

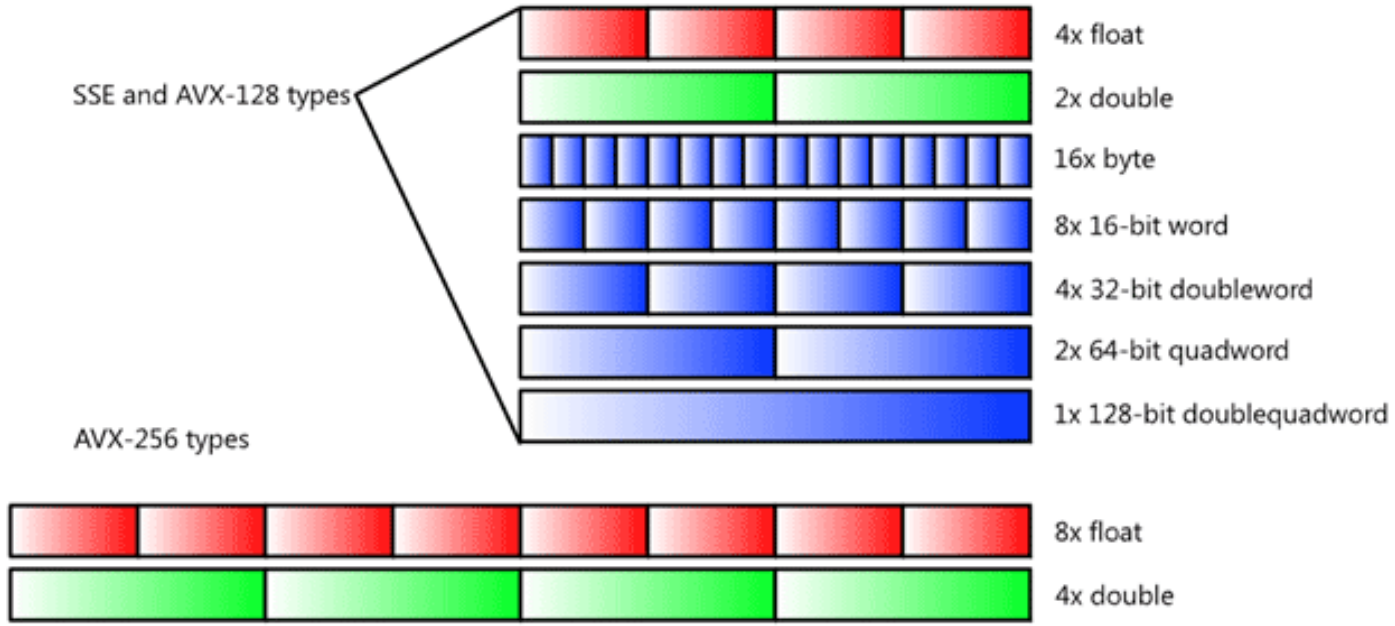
- AVX2 avec GCC (depuis version 4.7) :
  - Option de compilation : -mavx2
  - Types de données et fonctions intrinsèques opérant sur ces types de données déclarés dans : <immintrin.h>
- Autres jeux d'instructions :
  - SSE : <xmmintrin.h>
  - SSE2 : <emmintrin.h>
  - SSE3 : <pmmintrin.h>
  - SSSE3 : <tmmintrin.h>
  - SSE4.1 : <smmintrin.h>
  - SSE4.2 : <nmmintrin.h>
  - AVX : <immintrin.h>
  - AVX-512 : <zmmintrin.h>

Types de données

- Chaque type correspond à un emplacement en mémoire dont la valeur peut être mise dans un registre YMM
  - `__m256` : 256 bits composés de 8 nombres à virgule flottante simple précision (`float`)
  - `__m256d` : 256 bits composés de 4 nombres à virgule flottante double précision (`double`)
  - `__m256i` : 256 bits composés d'entiers
  - `__m128` : 128 bits composés de 4 nombres à virgule flottante simple précision
  - `__m128d` : 128 bits composés de 2 nombres à virgule flottante double précision
  - `__m128i` : 128 bits composés d'entiers

Types de données (2)

- AVX :



(C. Lomont)

- En plus avec AVX 2 : 32x bytes, 16x 16-bit shorts, 8x 32-bit integers, 4x 64-bit integers, 2x 128-bit integers

Fonctions intrinsèques

- Nomenclature générale :

  `data_type _mm256_{operation}_{suffix}(data_type param1, data_type param2, data_type param3)`

  - `suffix` en 2 parties :
    - 1ère partie :
      - `p` (*packed*) et `ep` (*extended packed*) : calcul SIMD
      - `s` (*scalar*) : vecteur vu comme 1 seul élément
    - 2ième partie : indique les types suivants
      - `[s/d]` : nombre flottant en simple ou double précision
      - `[i/u]nnn` : entier signé ou non-signé (*unsigned*) de `nnn` bits (`nnn` vaut 256, 128, 64, 32, 16, ou 8)
  - Paramètres :
    - Deux premiers : registres source
    - Troisième (si présent) : vecteur d'entiers pour masquage, sélection ou décalage

Opérations mémoire

- Fonctionnement général
  - déclaration des variables AVX :
    `__m256 r1`
  - chargement des données de la mémoire vers les vecteurs SIMD :
    `r1 = _mm256_load...(type* p)`
  - opérations sur les registres SIMD
  - écriture du contenu des vecteurs SIMD en mémoire :
    `_mm256_store...(type* p, r1)`
- Accès mémoire vectoriels (8*`float`, 4*`double`, entiers ...) :
  - Pour données alignées sur 32 octets :
    - `_mm256_load_ps`, `_mm256_load_pd`, `_mm256_load_si256` …
    - `_mm256_store_ps`, `_mm256_store_pd`, `_mm256_store_si256` …
  - Pour données non alignées :
    - `_mm256_loadu_ps`, `_mm256_loadu_pd`, `_mm256_loadu_si256` …
    - `_mm256_storeu_ps`, `_mm256_storeu_pd`, `_mm256_storeu_si256` …

Opérations mémoire (2)

- Exemple : copies de 8 float

```
#include <stdio.h>
#include <immintrin.h>

int main(){
  float a1[8] __attribute__((aligned(32))) =
      { 0.1, 0.2 , 0.3, 0.4, 0.5, 0.6, 0.7, 0.8};
  float a2[8] __attribute__((aligned(32)));
  __m256 v1;
  v1 = _mm256_load_ps(a1);
  _mm256_store_ps(a2,v1);

  printf("%f %f %f %f %f %f %f %f\n",
        a2[0], a2[1] ,a2[2], a2[3], a2[4], a2[5], a2[6], a2[7]);
  return 0;
}
```

Opérations arithmétiques

- Addition pour nombres flottants :
  - `_mm256_add_pd` (Add-Packed-Double)
    - Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]
    - Sortie : [A0 + B0, A1 + B1, A2 + B2, A3 + B3]
  - `_mm256_add_ps` (Add-Packed-Single)
    - Entrée : [A0, A1, A2, A3, A4, A5, A6, A7], [B0, B1, B2, B3, B4, B5, B6, B7]
    - Sortie : [A0 + B0, A1 + B1, A2 + B2, A3 + B3, A4 + B4, A5 + B5, A6 + B6, A7 + B7]
- Soustraction : `add` → `sub` (A-B)
- Et aussi :
  - additions / soustractions entrelacées : `_mm256_addsub_pd` et `_mm256_addsub_ps`

Opérations arithmétiques (2)

- Addition pour nombres entiers :
  - `_mm256_add_epi64` (Add-Packed-Int64)
    - Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]
    - Sortie : [A0 + B0, A1 + B1, A2 + B2, A3 + B3]
  - `_mm256_add_epi32` (Add-Packed-Int32)
    - Entrée : [A0, A1, A2, A3, A4, A5, A6, A7], [B0, B1, B2, B3, B4, B5, B6, B7]
    - Sortie : [A0 + B0, A1 + B1, A2 + B2, A3 + B3, A4 + B4, A5 + B5, A6 + B6, A7 + B7]
- Soustraction : `add` → `sub` (A-B)

Opérations arithmétiques (3)

- Additions horizontales :
  - `_mm256_hadd_ps` (Horizontal-Add-Packed-Single)
    - Entrée : [A0, A1, A2, A3, A4, A5, A6, A7], [B0, B1, B2, B3, B4, B5, B6, B7]
    - Sortie : [A0+A1, A2+A3, B0+B1, B2+B3, A4+A5, A6+A7, B4+B5, B6+B7]
  - Et aussi : `_mm256_hadd_pd`, `_mm256_hadd_epi32` ...
  - Versions soustractives

Opérations arithmétiques (4)

- Multiplication pour nombres flottants :
  - `_mm256_mul_pd` (Multiply-Packed-Double)
    - Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]
    - Sortie : [A0 * B0, A1 * B1, A2 * B2, A3 * B3]
  - `_mm256_mul_ps` (Multiply-Packed-Single)
    - Entrée : [A0, A1, A2, A3, A4, A5, A6, A7], [B0, B1, B2, B3, B4, B5, B6, B7]
    - Sortie : [A0 * B0, A1 * B1, A2 * B2, A3 * B3, A4 * B4, A5 * B5, A6 * B6, A7 * B7]
- Soustraction : `mul` → `div` (A / B)
  - Attention : division flottante très coûteuse

Opérations arithmétiques (5)

- Multiplication pour nombres entiers :
  - `_mm256_mullo_epi64` (*Multiply packed int64 (intermediate 128int results) and returns the low int64 part of the results)*
    - Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]
    - Sortie : [(A0*B0)(0:63), (A1*B1)(0:63), (A2*B2)(0:63), (A3*B3)(0:63)]
  - `_mm256_mul_epi32` (*Multiply low int32 from each packed int64, and returns the int64 results*)
    - Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]
    - Sortie : [A0(0:31) * B0(0:31), A1(0:31) * B1(0:31), A2(0:31) * B2(0:31), A3(0:31) * B3(0:31)]

Opérations arithmétiques (6)

- Multiplication pour nombres entiers (suite) :
  - `_mm256_mullo_epi32` (*Multiply packed int32 (intermediate int64 results), and returns the low int32 part of the results*)
    - Entrée : [A0, A1, A2, A3, A4, A5, A6, A7], [B0, B1, B2, B3, B4, B5, B6, B7]
    - Sortie : [(A0*B0)(0:31), (A1*B1)(0:31), (A2*B2)(0:31), (A3*B3)(0:31), (A4*B4)(0:31), (A5*B5)(0:31), (A6*B6)(0:31), (A7*B7)(0:31)]
  - `_mm256_mul[hi/lo]_epi16` *(Multiply packed int16 (intermediate int32 results), and returns the [high/low] int16 part of the results)*
    - Entrée : [A0, A1, … , A15], [B0, B1, … , B15]
    - Sortie : [(A0*B0)(hi/lo), (A1*B1)(hi/lo), … , (A15*B15)(hi/lo)]
- Pas de division entière en AVX (ni en SSE, ni en AVX-512)

FMA en AVX2

- Spécifique à l'AVX2
- *Fused Multiply-Add* pour nombres flottants :
  - `_mm256_fmadd_ps` (Packed-Single)
    - Entrée : [A0, A1, … , A7], [B0, B1, … , B7], [C0, C1, … , C7]
    - Sortie : [A0 * B0 + C0, A1 * B1 + C1, … , A7 * B7 + C7]
  - `_mm256_fmadd_pd` (Packed-Double)
  - Versions soustractives et versions entrelacées addition/soustraction
- Intérêts :
  - 2 flops en une opération
  - Meilleure précision : résultats intermédiaires traités en précision infinie, et arrondi effectué uniquement sur le résultat final

Autres opérations

- Opérations logiques :
  `_mm256_[and/or/xor/...]_[ps/pd/si256]`

- Tests
  - Nombres flottants :
    ```
    _mm256_cmp_[pd/ps](__m256[d] a, __m256[d] b,
                       const int imm8)
    ```
    avec `imm8` qui indique l'opérateur de comparaison voulu
  - Nombres entiers :
    `_mm256_cmp[eq/gt]_epi[8/16/32/64]`

Autres opérations (2)

- Déplacements des éléments au sein des vecteurs SIMD : `_mm256_shuffle...`, `_mm256_permute` ...
- Conversions entre vecteurs de types différents : `_mm256_cvt...`
- Et aussi :
  - `_mm256_broadcast...`
  - `_mm256_set...`
  - Extraction d'un élément d'un vecteur SIMD :
    - `_mm256_extract_epi[8/16/32/64]`
    - `_mm256_extractf128_[pd/ps]` et `_mm_extract_ps`
    - Autre possibilité :

```
__m256 v;
…
float a = ((float*)&v)[4]
```

Directives OpenMP 4.0 pour la programmation SIMD

- Vectorisation d'une boucle ou d'un nid de boucles :

```
#pragma omp simd [clause[[,] clause],…]
for-loop[s]
```

  - Regroupe N itérations d'une boucle (*chunk*) pour effectuer le calcul dans une unité SIMD à N éléments
  - C'est l'utilisateur qui indique / garantit que la boucle est « vectorisable »
  - Pas de parallélisation multi-thread de la boucle avec cette directive

Clauses de la directive `simd`

- Clauses de partage de données : `private(var-list)`, `lastprivate(var-list)`, `reduction(op:var-list)`

  → fonctionnement similaire au multi-threading OpenMP
- Autres clauses :
  - `simdlen (length)`
    - nombre souhaité d'itérations exécutées concurremment en SIMD
  - `safelen (length)`
    - nombre maximal d'itérations qui peuvent être exécutées concurremment (sans problème de dépendance)
    - en pratique : taille maximum pour le vecteur SIMD
  - `linear (list[:linear-step])`
    - La valeur de la variable est liée au numéro de l'itération (« i ») : `x_i = x_initial + i * linear-step`
  - `aligned (list[:alignment])`
    - Précise que la variable est alignée sur « `alignement` » octets
  - `collapse (n)`
    - Fusionne les « `n` » boucles imbriquées qui suivent la directive `simd` en un seul espace d'itération qui est ensuite « vectorisé »

Exemple pour la directive `simd`

```
void produit_scalaire(float *a, float *b, int n) {
   float res = 0.0f;

#pragma omp simd reduction(+:res)
   for (int i=0; i<n; i++){
      res += a[i] * b[i];
   }
   return res;
}
```

Directive `for` avec `simd`

- Paralléliser et vectoriser un nid de boucles

```
#pragma omp for simd [clause[[,] clause],...]
for-loop[s]
```

  - Les itérations sont réparties entre les threads
  - Les itérations attribuées à un thread sont exécutées en SIMD

Déclaration de fonctions SIMD

- Déclarer des fonctions qui devront être compilées en version vectorielle pour pouvoir être appelées depuis des boucles exécutées en SIMD

```
#pragma omp declare simd [clause[[,] clause],...]
function-definition-or-declaration
```

- Permet de vectoriser des boucles avec des appels de fonctions

Clauses de la directive `declare simd`

- Mêmes clauses que pour la directive `simd`
  - `simdlen, linear, aligned, reduction`
- Clauses spécifiques :
  - `uniform (argument-list)`
    - L'argument de la fonction a une valeur constante pour tous les appels concurrents à la fonction depuis la boucle SIMD
  - `inbranch`
    - La fonction est *toujours* appelée depuis un branchement conditionnel (`if`) dans la boucle SIMD
  - `notinbranch`
    - La fonction n'est *jamais* appelée depuis un branchement conditionnel dans la boucle SIMD

Exemple pour la directive `declare simd`

```
#pragma omp declare simd
float distance(float x1, float x2, float y1, float y2){
   return sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
}

void calcul_distances(){
#pragma omp simd
   for (i=0; i<N; i++){
      d[i] = distance(Ax[i], Bx[i], Ay[i], By[i]));
   }
}
```

- Avec OpenMP, le compilateur génère du code SIMD pour la fonction `distance` → les 4 arguments x1, x2, y1, y2 et la variable de retour correspondent alors à des vecteurs SIMD

Directives OpenMP 4.0 SIMD dans `gcc`

- Disponible depuis la version 4.9 de `gcc`
- Option `-fopenmp-simd` : permet de ne prendre en compte que les directives SIMD d'OpenMP
- Implémentation actuellement limitée :
  - boucle la plus interne sans branchements conditionnels
  - double boucle imbriquée avec une boucle interne très simple