Unités vectorielles

Oguz Kaya

Université Paris-Saclay

Orsay, France

Outline

Outline

université PARIS-SACLAY

Outline

université PARIS-SACLAY

Calcul Scalaire

Principe

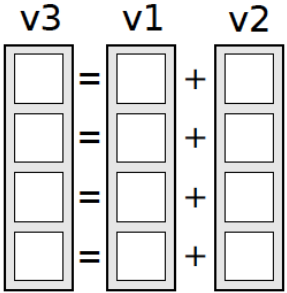
Un processeur scalaire effectue les opérations séquentiellement, chaque opération portant sur des données scalaires (un scalaire)

Exemple: addition de vecteurs de 4 éléments

```
v3[0] = v1[0] + v2[0];
v3[1] = v1[1] + v2[1];
v3[2] = v1[2] + v2[2];
v3[3] = v1[3] + v2[3];
```

université PARIS-SACLAY

Calcul Scalaire



Calcul Vectoriel

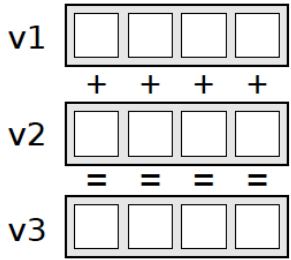
Principe

Un processeur vectoriel effectue les opérations en parallèle sur un ensemble de scalaires, chaque opération portant sur des vecteurs

Exemple: addition de vecteurs de 4 éléments

v3 = v1 + v2;

Calcul Vectoriel



Avantages/inconvénients de l'approche vectorielle

1. Avantage:
   - Instructions n fois plus rapide, n la largeur de l'unité vectorielle si l'algorithme se prête au paradigme SIMD: une même instruction simultanément sur plusieurs données
     - calcul vectoriel
     - simulations scientifiques
     - traitement d'images
     - apprentissage automatique (réseaux neurons)
2. Inconvénient:
   - Gain uniquement si l'algorithme se prête au paradigme SIMD.
   - Sinon c'est une perte de circuit qui pourrait être utilisé pour d'autres fins (cœurs, cache, registre, . . .).
   - Parfois difficile (et sale) à utiliser à la main (automatisations possible).

université PARIS-SACLAY

Premières implantations

- Années 60: premier projet Solomon
- Années 70: ILLIAC IV, université de l'Illinois
- ensuite: CDC, Cray, NEC, Hitachi, Fujitsu
- depuis 1996: processeurs "grand public" scalaires + unités vectorielles, Pentium (MMX, SSE, AVX), PowerPC (AltiVec)
- maintenant: dans quasiment toutes les architectures (x86, ARM, RISC-V)

université PARIS-SACLAY

Fonctionnement général des unitées SIMD

Pareil que le fonctionnement des registres scalaires, sauf chaque opération se fait sur un ensemble de données.

1. Chargement des données de la mémoire vers les registres SIMD
2. Opération sur les registres SIMD
3. Placement du résultat en mémoire

université PARIS-SACLAY

Programmation

- **Vectorisation automatique:** Le compilateur tente de vectoriser le code automatiquement.
- **Programmation assembleur:** Programmation bas-niveau, manipulation directe des registres et des instructions.
- **Intrinsèques:** Manipulation des variables par des fonctions intrinsèquements définies dans un fichier ?mmintrin.h .
- **Bibliothèques de haut-niveau (OpenMP):** Auto-véctorisation à l'aide des astuces données à la bibliothèque.

université PARIS-SACLAY

Outline

université PARIS-SACLAY

Streaming SIMD Extensions (SSE)

- SSE - Pentium III, Athlon XP
- SSE2 - Pentium 4 Willamette
- SSE3 - Pentium 4 Prescott
- SSSE3 - (Supplemental SSE3) Core Woodcrest
- SSE4.1 - Core 2 Penryn
- SSE4.2 - Core i7 Nehalem
- AVX - Sandy Bridge
- **AVX2** - Haswell
- AVX512 - Skylake et la suite

Composition

Chaque nouvelle version de SSE/AVX contient la précédente
SSE(N) = nouvelles instructions + SSE(N-1)

Intrinsics

Les intrinsics sont accessibles à travers des fichiers .h qui déffinissent:

- des types de variables
- des fonctions opérant sur ces variables

Fichiers

- SSE: xmmintrin.h
- SSE2: emmintrin.h
- SSE3: pmmintrin.h
- SSSE3: tmmintrin.h
- SSE4.1: smmintrin.h
- SSE4.2: nmmintrin.h
- **AVX/AVX2: immintrin.h**

Lancer **lscpu** sous Linux pour afficher la version supportée.

université PARIS-SACLAY

Advanced Vector Extensions (AVX)

Registres de 256 bits

16 registres 256bits sur x86-64: YMM0 $->$ YMM15

Instructions

- transferts mémoire $<->$ registres AVX
- opérations arithmétiques
- opérations logiques
- tests
- permutations

Compilation avec AVX

Compilation du programme.c

gcc programme.c -o programme **-mavx2**

Exécution

./programme

université PARIS-SACLAY

Types de variables

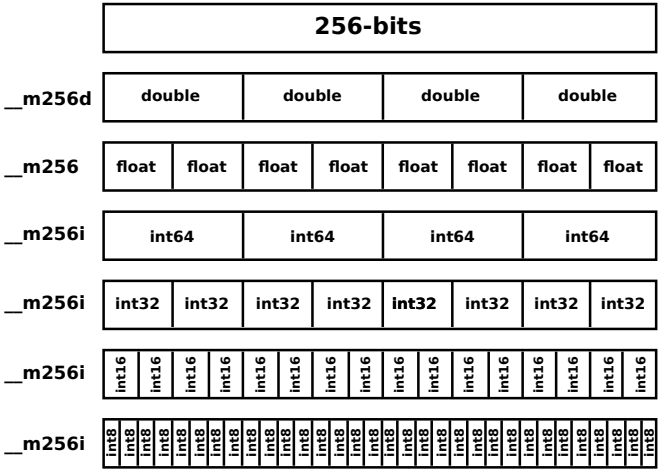
Types

- __m256: 8 floats
- __m256d: 4 doubles
- __m256i: $256/\textit{largeur}$ entiers ($\textit{largeur} = 8, 16, 32, 64$)

Il y a un jeu d'instructions définies pour chaque type.

université PARIS-SACLAY

Vecteurs AVX



Fonctions

Nomenclature générale

_mm256_{operation}{non-alignement}_{dataorganization}{datatype}...

Exemple: addition de 2 vecteurs de 8 flottants

__m256 _mm256_add_ps(__m256 A, __m256 B)

Terminologie

_mm256_loadu_ps256

- s (single): flottant simple précision (32bits)
- d (double): flottant double précision (64bits)
- i... (integer): entier
- p (packed): contigus, opérer sur le vecteur entier
- s (scalar): n'opérer que sur un élément (obsolète!)
- u (unaligned): données non alignées en mémoire
- l (low): bits de poids faible
- h (high): bits de poids fort
- r (reversed): dans l'ordre inverse

Fonctionnement général

1. déclaration des variables AVX (registres):
   __m256 r1;

2. chargement des données de la mémoire vers les registres AVX:
   float A[8] __attribute__ ((aligned(16));
   ...
   r1 = _mm256_load_ps(&A[0]);

3. opérations sur les registres SIMD
   r1 = _mm256_add_ps(r1, r1);

4. placement du contenu des registres en mémoire:
   _mm256_store_ps(&A[0], r1)

université PARIS-SACLAY

Transferts entre la mémoire et les registres AVX

- **Flottants:**
  __m256 _mm256_load_ps(float *addr)
  void _mm256_store_ps(float *addr, __m256 a)
- **Doubles:**
  __m256d _mm256_load_pd(double *addr)
  void _mm256_store_pd(double*addr, __m256 a)
- **Entièrs:**
  __m256i _mm256_load_si256(__m256i const * addr)
  void _mm256_store_si256(__m256i const * addr, __m256i a)

université PARIS-SACLAY

Transferts depuis la mémoire vers les registres AVX

- alignés ou non alignés:
  _mm256_load... ou _mm256_loadu...
  _mm256_store... ou _mm256_storeu...

- par vecteurs: $8 \times SP$, $4 \times DP$, ...
  _mm256_load_ps, _mm256_load_pd, ...
  _mm256_store_ps, _mm256_store_pd, ...

université PARIS-SACLAY

Accès alignés

- L'accès à une variable est **alignée** si l'adresse de cette variable est un multiple de sa taille. Cette contrainte est matérielle.
- Pour les variables AVX, le processeur peut effectuer des transferts efficaces des variables AVX 32 octets (256 bits) si elle est aligné sur 32 octets.
- Pour obtenir l'alignement d'un type: __alignof__(type)

université PARIS-SACLAY

Accès alignés

Cacheline splits

Une ligne de cache a généralement une taille de 32 à 64 octets.

Si la donnée chargée dans un registre SSE provient d'une zone mémoire à "cheval" sur 2 lignes de cache, alors il faut lire les 2 lignes de cache pour pouvoir remplir le registre SSE, ce qui implique une baisse de performance

Allocation de données alignées

- Alignement de données statiques sur 32 octets:
  `int x __attribute__ ((aligned (32)))`
  `float y[8] __attribute__ ((aligned (32)))`

- Alignement de données dynamiques sur 32 octets:
  `type* x = (type*) _mm_malloc(size*sizeof(type), 32);`
- Les accès non alignés peuvent être beaucoup plus lents (mais pas autant dans les nouvelles générations).

Opérations arithmétiques

- _mm256_add_pd(__m256d A, __m256d B) - (Add-Packed-Double)
  - Entrée: [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Sortie: [A0 + B0, A1 + B1, A2 + B2, A3 + B3]
- _mm256_add_ps(__m256 A, __m256 B) - (Add-Packed-Single)
  - Entrée: [A0, . . . , A7], [B0, . . . , B7]
  - Sortie: [A0 + B0, . . . , A7 + B7]
- _mm256_add_epi64(__m256i A, __m256i B) - (Add-Packed-Int64)
  - Entrée: [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Sortie: [A0 + B0, A1 + B1, A2 + B2, A3 + B3]
- _mm256_add_epi32(__m256i A, __m256i B) - (Add-Packed-Int32)
- _mm256_add_epi16(__m256i A, __m256i B) - (Add-Packed-Int16)
- _mm256_add_epi8(__m256i A, __m256i B) - (Add-Packed-Int8)

Idem pour _mm256_sub_. . .

université PARIS-SACLAY

Opérations arithmétiques

- _mm256_mul_pd(__m256d A, __m256d B) - (Multiply-Packed-Double)
  - Entrée: [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Sortie: [A0 * B0, A1 * B1, A2 * B2, A3 * B3]
- _mm256_mul_ps(__m256 A, __m256 B) - (Multiply-Packed-Single)
  - Entrée: [A0, . . . , A7], [B0, . . . , B7]
  - Sortie: [A0 * B0, . . . , A7 * B7]
- _mm256_mul_epi64(__m256i A, __m256i B) - (Multiply-Packed-Int64)
  - Entrée: [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Sortie: [A0 * B0, A1 * B1, A2 * B2, A3 * B3]
- _mm256_mul_epi32(__m256i A, __m256i B) - (Multiply-Packed-Int32)
- _mm256_mul_epi16(__m256i A, __m256i B) - (Multiply-Packed-Int16)
- _mm256_mul_epi8(__m256i A, __m256i B) - (Multiply-Packed-Int8)

Idem pour _mm256_div_. . . ; attention, la division est **très lente**!

université PARIS-SACLAY

Opérations arithmétiques

- _mm256_fmadd_pd(__m256 A, __m256 B, __m256 C) - (Fused-Multiply-Add-Packed-Double)
  - Entrée: [A0, A1, A2, A3], [B0, B1, B2, B3], [C0, C1, C2, C3]
  - Sortie: [A0 * B0 + C0, A1 * B1 + C1, A2 * B2 + C2, A3 * B3 + C3]
- _mm256_fmadd_ps(__m256 A, __m256 B, __m256 C) - (Fused-Multiply-Add-Packed-Single)
  - Entrée: [A0, . . . , A7], [B0, . . . , B7], [C0, . . . , C7]
  - Sortie: [A0 * B0 + C0, . . . , A7 * B7 + C7]

Idem pour _mm256_fmsub_. . .

université PARIS-SACLAY

Opérations horizontales

- _mm256_hadd_pd(A, B) - (Horizontal-Add-Packed-Double)
  - Entrée: [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Sortie: [A0 + A1, B0 + B1, A2 + A3, B2 + B3]
- _mm256_hadd_ps(A, B) (Horizontal-Add-Packed-Single)
  - Entrée: [A0, . . . , A7], [B0, . . . , B7]
  - Sortie: [A0 + A1, A2 + A3, B0 + B1, B2 + B3, A2 + A3, . . . , B6 + B7]
- _mm256_hadd_epi32(__m256i A, __m256i B, __m256i C) - (Horizontal-Add-Packed-Int32)
- _mm256_hadd_epi16(__m256i A, __m256i B, __m256i C) - (Horizontal-Add-Packed-Int16)
- Idem pour _mm256_hsub_. . .

université PARIS-SACLAY

Opérations logiques et de comparaison

- _mm256_{and , or , xor, . . . }_{ps, pd, si256}(A, B)
- _mm256_cmp_{ps, pd}(A, B, op)
  - op = _CMP_LT_OS: Comparaison A $<$ B
  - op = _CMP_LE_OS: Comparaison A $\leqslant$ B
  - op = _CMP_EQ_OS: Comparaison A $==$ B
  - op = _CMP_GT_OS: Comparaison A $>$ B
  - op = _CMP_GE_OS: Comparaison A $\geqslant$ B
- _mm256_cmpeq_epi{64, 32, 16, 8}(__m256i A, __m256i B)
- _mm256_cmpgt_epi{64, 32, 16, 8}(__m256i A, __m256i B)

Comparaisons

Pour C = _mm256_cmp. . . (A, B), si A[i] op B[i] est correcte, C[i] = 0b11. . . 1. Si c'est faux, C[i] = 0b00. . . 0.

Mélange des données: Permutation

L'instruction permet d'obtenir n'importe quel parmutation des données dans un registre AVX. La permutation est precisé par un entièr immediat dont bits servent des indices.

- __m256d B = _mm256_permute4x64_pd(__m256d A, const int imm8)
  B[0] = A[imm8[1:0]]
  B[1] = A[imm8[3:2]]
  B[2] = A[imm8[5:4]]
  B[3] = A[imm8[7:6]]
- __m256i B = _mm256_permute4x64_epi64(__m256i A, const int imm8)
  B[0] = A[imm8[1:0]]; B[1] = A[imm8[3:2]]
  B[2] = A[imm8[5:4]]; B[3] = A[imm8[7:6]]
- __m256 B = _mm256_permutevar8x32_ps(__m256 A, __m256i idx)
  B[i] = A[idx[(32 * i) + 2 : (32 * i)]]
- __m256i B = _mm256_permutevar8x32_epi32(__m256i A, __m256i idx)
  B[i] = A[idx[(32 * i) + 2 : (32 * i)]]

université PARIS-SACLAY

Affectation des constantes

- _mm256_set_ps(float f0, . . . , float f7)
- _mm256_set_pd(double d0, double d1, double d2, double d3)
- _mm256_set_epi64x(int64_t i0, . . . , int64_t i3)
- _mm256_set_epi32(int i0, . . . , int i7)
- _mm256_set_epi16(short i0, . . . , short i15)
- _mm256_set_epi8(char i0, . . . , char i31)
- Il y a aussi la variante _mm256_set1_ps(float f) qui affecte la même valeur à tous les éléments du vecteur (idem pour double, int$\{64, 32, 16, 8\}$)

université PARIS-SACLAY

Astuces de bonnes performances

- Organiser correctement les données pour éviter les accès non contigus.
- Rester dans le cache autant que possible.
- Aligner les données (moins importants dans les versions récentes).
- Rester le plus longtemps possible dans les registres (c'est à dire, éviter les lectures/écritures sur la mémoire)

université PARIS-SACLAY

Organisation des données pour la véctorisation

La structure initiale ne pourrait pas être adaptée à la vectorisation. Dans ce cas là il faudrait y repenser.

- Tableau de structures (Array of structures - AoS):
  xyzwxyzwxyzw
- Structure de tableaux (Structure of Arrays - SoA):
  xxxxxxxxx . . . yyyyyyyyy . . . zzzzzzzzz . . . wwwwwwwww . . .
- Structure hybride:
  xxxxyyyyzzzzwwwwxxxxyyyyzzzzwwww . . .

Conclusion

- Unités vectorielles permettent le mode d'opération SIMD sur un vecteur de données.
- L'état du jeux d'instructions AVX est assez chaotique (tout comme x86-64)
- Instructions pas compatibles avec tous les types de donnée
- Potentiel de gain substantiel (16-32x avec AVX2)

université PARIS-SACLAY