Programmation Multi-Coeurs

M2 VIP

Sylvain Jubertie
sylvain.jubertie@univ-orleans.fr

2014/2015

Programme du module

- 10 1/2 journées cours/TD
- Evaluation
  - examen
  - projet

Prologue

Avant de paralléliser un code pour exploiter plusieurs processeurs il faut s'assurer d'exploiter correctement un seul processeur . . .
. . . donc ce cours commence par rappeler comment fonctionne un processeur !

Prologue

Sur une même machine, entre un programme séquentiel naïf et un programme séquentiel optimisé (caches, ...) le temps d'exécution peut être divisé par 100 (par exemple algo naïf de multiplication de matrice vs GotoBLAS2)

Prologue

Le temps d'exécution d'un programme dépend :

1. **de l'algorithme retenu (complexité en temps, en espace)**
2. **du langage (langage compilé : C, C++ ; langage semi-interprété : Java, Pyhton ; langage interprété : Perl, Ruby, Mathlab)**
3. **du programmeur : de la manière dont l'algorithme est écrit/optimisé**
4. **du compilateur, de la machine virtuelle ou de l'interpréteur**
5. **de l'architecture (processeur, bus, entrées/sorties, mémoires, fréquences, tailles des caches, . . .)**
6. **du système d'exploitation**
7. **. . .**

Prologue

Remarques sur la complexité

La complexité d'un algorithme indique comment évolue son temps d'exécution en fonction de la taille de l'entrée. Cela ne permet pas de comparer le temps d'exécution d'algorithmes de même complexité.
Certains algorithmes se comportent mieux sur de petites tailles de données et d'autres sur de grandes tailles de données. Il est parfois intéressant de tester la taille des données avant de brancher sur l'algorithme le plus intéressant ou de combiner les algorithmes (STL introsort).

Prologue

Remarques sur les langages

Un programme écrit dans un langage compilé peut facilement être plusieurs dizaines de fois plus performant qu'un langage interprété (C/C++ vs Mathlab)...

...par contre les langages interprétés permettent généralement un développement plus rapide et sont donc pratiques pour prototyper un programme.

Les bibliothèques les plus efficaces sont généralement écrites en assembleur pour les routines les plus critiques...

Prologue

Autres remarques

- Les nouvelles versions des compilateurs C/C++ ne permettent de gagner que quelques % de temps d'exécution sur l'ensemble d'un programme. Attention il y a parfois des régressions...
- Une nouvelle génération de processeur ne permet de gagner qu'environ 10% de temps d'exécution par rapport à la génération précédente (source Intel Haswell vs Ivy Bridge).
- Le système d'exploitation impacte généralement peu sur les performances en calcul pure. Dans des cas particuliers des différences d'ABI Application Binary Interface, ou de gestion de la mémoire peuvent impacter les performances.

Architecture générale

- Mémoire
  - registres
  - caches L1-L2-L3
- Unités d'exécution
  - unités Arithmétiques et Logiques (ALU)
  - unités Flottantes (FPU)
- pipeline d'exécution, unités de prédiction de branchements, contrôleur mémoire, TLB, ...

Principe

Placer les données et instructions dans le processeur pour y accéder plus rapidement.

1. Lors de l'accès à une donnée, le processeur vérifie si elle est disponible en cache.
   - Si ce n'est pas le cas (cache miss), la donnée est rapatriée en cache depuis la mémoire (très lent)
   - Si la donnée est présente dans le cache (cache hit), il n'y a pas d'accès mémoire !
2. La donnée est placée dans un registre et traitée
3. La donnée est ensuite replacée en cache
4. La donnée reste en cache tant qu'une autre donnée ne l'écrase pas, sinon elle est remise en mémoire

Caractéristiques

- mémoires directement dans le processeur
  - très rapides : accès en quelques cycles
  - taille réduite : quelques dizaines d'octets à quelques MB
- hiérarchie :
  - L1 : 32-64kB
  - L2 : 128-256-512kB
  - L3 : 1-12MB
- types :
  - instruction cache
  - data cache

Translation Lookaside Buffer

Le TLB (**T**ranslation **L**ookaside **B**uffer) est un cache utilisé par l'unité de gestion mémoire MMU (**M**emory **M**anagment **U**nit) pour stocker les correspondances entre adresses virtuelles utilisées dans les programmes et adresses physiques.

Performance

Le TLB peut parfois se retrouver saturé lors d'accès mémoire fréquents.
Diagnostique à l'aide d'outils de profiling tels que `perf`.

Constat

Les données et instructions sont généralement accédées linéairement (tableaux, matrices, images, blocs d'instructions sans sauts, ...).

Ligne de cache

Partant de ce constat, lorsqu'une donnée est demandée par le processeur, les données voisines sont également chargées. Ces blocs de données forment une ligne de cache. Cela permet de ne pas engendrer de cache miss lorsqu'une donnée voisine est ensuite accédée.

Prefetch des lignes de cache

Le processeur dispose également d'un mécanisme de préchargement des données en mémoire cache appelé prefetch. Lorsqu'une ligne de cache est demandée, la ligne suivante commence à être préchargée en cache de manière à être disponible et éviter des cache miss.

Exemple de mauvaise utilisation du cache

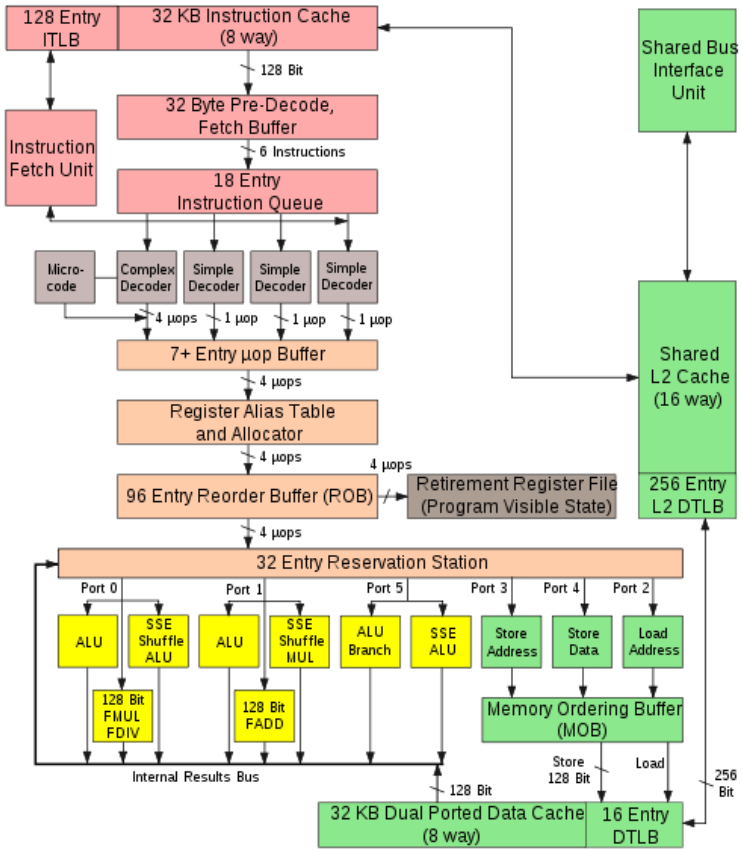
```
1  // somme de 2 matrices de taille NxN : C = A x B
2  for(j = 0 ; j < N; ++j)
3    for(i = 0 ; i < N ; ++i)
4      C[i][j] = A[i][j] + B[i][j]
```

Explications

Les données ne sont pas accédées linéairement en mémoire et de nombreux cache miss sont provoqués.

Pipeline

Architecture super-scalaire



Intel Core 2 Architecture

Out-Of-Order execution

Les instructions indépendantes (ne travaillant pas sur les mêmes données) peuvent être interverties. Si une instruction doit attendre une donnée et monopolise une unité de traitement, une autre instruction indépendante dans le pipeline pouvant être placée sur une unité disponible peut être exécutée.

Out-Of-Order execution

Le compilateur peut optimiser le code en amont lorsqu'il connaît l'architecture cible. Cependant le développeur peut également éviter les longues chaînes de dépendances afin de simplifier la tâche au compilateur.

Branch prediction

Lors d'un branchement conditionnel (if, for, while, switch), un cache miss peut être provoqué si les instructions et données nécessaires pour continuer l'exécution ne sont pas disponibles en cache. Afin d'optimiser l'exécution des branchements conditionnels, l'unité de prédiction de branchement du processeur peut essayer de déterminer la branche qui va être empruntée à partir de méthodes statistiques. Par exemple une boucle for répétée 50 fois a de fortes chances d'être répétée une nouvelle fois. La pénalité ne se produit alors qu'après la dernière itération de la boucle. Voir aussi exécution spéculative.

Simultaneous Multi-Threading (SMT)

- Attention le Simultaneous Multi-Threading, ou Hyper-Threading (HT) chez Intel, ne sont pas des technologies multi-coeurs.
- Ces technologies visent à améliorer le taux d'utilisation du processeur en permettant simultanément le partage des registres et du pipeline d'exécution entre plusieurs threads, optimise le Thread Level Parallelism.
- Les systèmes avec SMT font apparaître des processeurs logiques (`lscpu`) et non pas physiques.
- Les performances ne sont donc pas doublées !
- Dans le cas de l'Hyper-Threading les processeurs peuvent être partagés par 2 threads, on parle de SMT à 2 voies.

Gains

Le gain apporté dépend des codes exécutés par les threads et peut être de 20 à 30% dans certains cas mais varie très fortement selon les applications et peut même être négatif :

- plus favorable : threads exploitant des unités de calcul distinctes et dont les besoins en cache n'engendrent pas des défauts de cache supplémentaires.
- nul : threads utilisant des unités communes et beaucoup de registres.
- négatif : threads utilisant les mêmes unités de calcul et dont l'exécution simultanée sature le cache ce qui engendre des défauts de page supplémentaires coûteux.

Calcul scalaire

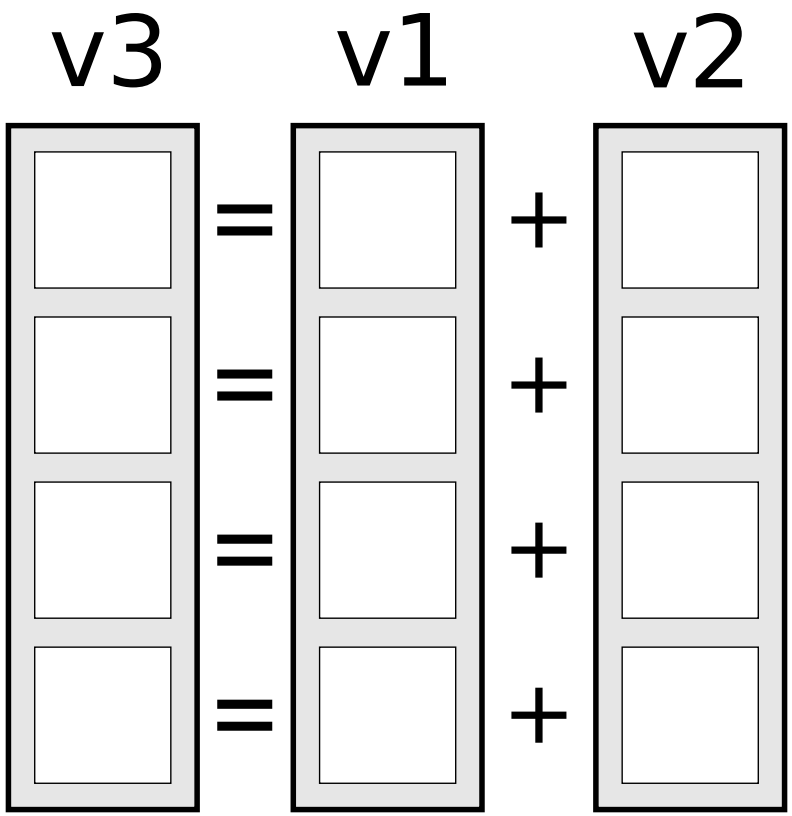
Principe

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];
```

Calcul scalaire



Calcul vectoriel

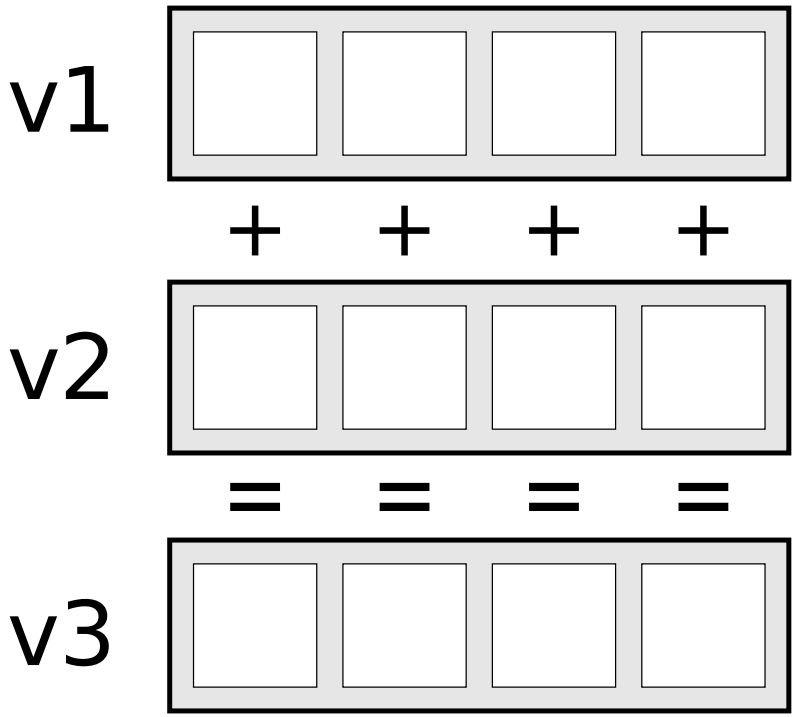
Principe

Un processeur vectoriel applique une même instruction simultanément sur plusieurs données (un vecteur de scalaires).

Exemple : addition de vecteurs de 4 éléments

```
v3 = v1 + v2;
```

Calcul vectoriel



Avantages/inconvénients de l'approche vectorielle

Avantage

Calcul *n* fois plus rapide, n largeur de l'unité vectorielle si l'algorithme se prête au paradigme SIMD : une même instruction simultanément sur plusieurs données.

- calcul vectoriel
- traitement d'images

Inconvénient

Gain uniquement si l'algorithme se prête au paradigme SIMD. Sinon on se retrouve dans le cas scalaire, perte d'efficacité.

Premières implantations

Historique

- 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)

MMX

MultiMédia eXtension ?

Première unité vectorielle “grand public” introduite par Intel sur certains Pentium de première génération.

MMX

Description

- 8 registres 64bits
- 57 opérations sur les entiers

Limitations

- uniquement sur les entiers
- registres partagés avec l'unité FPU

Autres jeux d'instructions SIMD

Unités SIMD

- AMD 3DNow! : AMD K6-2, ..., Phenom II, certains VIA C5, Transmeta Crusoe et Efficeon
- ARM Neon : Cortex-A8&9 (Archos 5, Pandora)
- approches FPGA
- GPU

Programmation

Fonctionnement général des unités SIMD

1. chargement des données de la mémoire vers les registres SIMD
2. opération sur les registres SIMD
3. placement du résultat en mémoire

Programmation

Vectorisation automatique

Le compilateur tente de vectoriser le code automatiquement...

Assembleur

- programmation bas-niveau
- manipulation directe des registres et instructions

Intrinsics

fichiers .h inclus dans GCC : `pmmintrin.h` pour SSE3

Bibliothèques de haut-niveau

masquage de la programmation vectorielle

Performance

Tips pour obtenir les meilleurs performances

- organiser correctement les données pour éviter les accès non contigus et également optimiser le cache
- aligner les données
- rester le plus longtemps possible dans les registres !

Organisation des données

Tableau de structures (Array of structures - AoS)

xyzwxyzwxyzw...

Structure de tableaux (Structure of Arrays - SoA)

xxxxxxxxx...

yyyyyyyyy...

zzzzzzzzz...

wwwwwwwww...

Structure hybride

xxxxyyyyzzzzwwwwxxxxyyyyzzzzwwww...

Streaming SIMD Extensions

- SSE - Pentium III, Athlon XP
- SSE2 - Pentium 4
- SSE3 - Pentium 4 Prescott
- SSSE3 - (Supplemental SSE3)
- SSE4.1 - Core2
- SSE4.2 - Core i7

Connaître la version de SSE de son processeur

`cat /proc/cpuinfo`

- sse
- sse2
- pni (Prescott New Instructions - SSE3)
- ssse3
- sse4_1

Composition

Chaque nouvelle version de SSE ajoute des instructions à la précédente.
SSE(N) = nouvelles instructions + SSE(N-1)

Registres de 128 bits

- 8 registres 128bits sur x86 : XMM0 -> XMM7
- 16 registres 128bits sur x86_64 : XMM8 -> XMM15

Instructions

- transferts mémoire <->registres SSE
- opérations arithmétiques
- opérations logiques
- tests
- permutations

Définitions

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

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

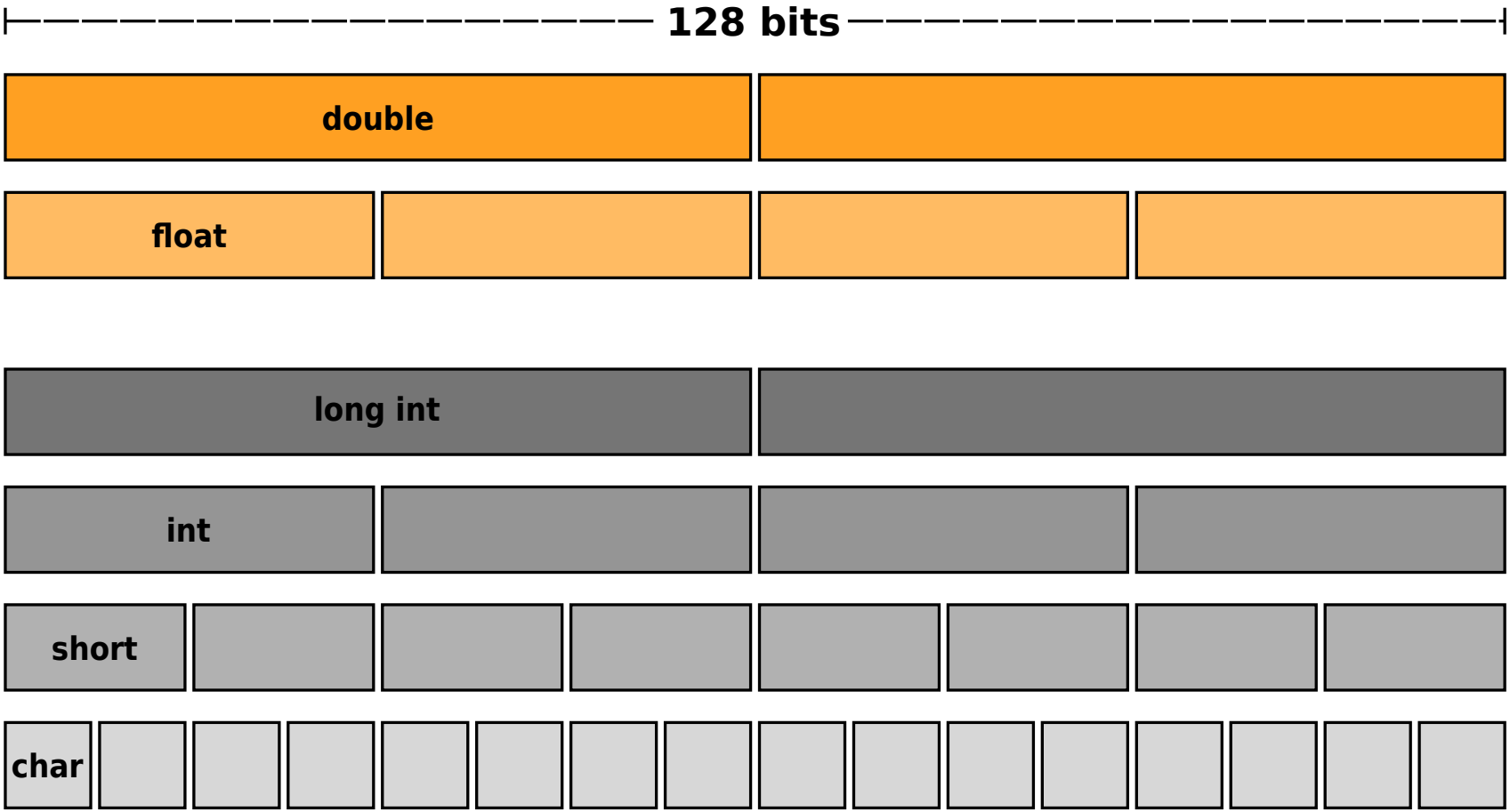
Fichiers

- SSE : xmmintrin.h
- SSE2 : emmintrin.h
- SSE3 : pmmintrin.h
- SSSE3 : tmmintrin.h
- SSE4.1 : smmintrin.h

Types de données

Types

- __m128 : 4 floats
- __m128d : 2 doubles
- __m128i : entiers


128 bits
double
float
long int
int
short
char

Fonctions

Nomenclature générale

_mm_{operation}{alignement}_{dataorganization}{datatype}(...)

Exemple : addition de 2 vecteurs de 4 floats

__m128 C = _mm_add_ps(__m128 A, __m128 B)

Terminologie

- `s` (single) : flottant simple précision (32bits)
- `d` (double) : flottant double précision (64bits)
- `i...` (integer) : entier
- `p` (packed) : contigus
- `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 SSE (registres)
   `__m128 r1`
2. chargement des données de la mémoire vers les registres SIMD
   `r1 = _mm_load...(type* p)`
3. opérations sur les registres SIMD
4. placement du contenu des registres en mémoire
   `_mm_store...(type* p, r1)`

Transferts mémoire <->registres SSE

- alignés ou non alignés
  `_mm_load_...` ou `_mm_loadu_`
  `_mm_store_...` ou `_mm_storeu_`
- par vecteurs : 4xSP, 2xDP, ... `_mm_load{u}_ps`, `_mm_load{u}_pd`, ...
  `_mm_store{u}_ps`, `_mm_store{u}_pd`, ...
- par élément scalaire
  `_mm_load_ss`, `_mm_load_sd`, ...
  `_mm_store_ss`, `_mm_store_sd`, ...

Accès alignés

Le processeur peut effectuer des transferts efficaces de 16 octets (128 bits) entre la mémoire et un registre SSE sous la condition que le bloc soit aligné sur 16 octets. Cette contrainte est matérielle.

Attention

L'alignement dépend de l'architecture et du type de variable. Par défaut l'alignement d'un entier 32 bits est effectué sur 4 octets.
Pour obtenir l'alignement : `__alignof__ (type)`
`__alignof__(int[4])` $\rightarrow$ `4`

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 16 octets :

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

Alignement de données dynamiques sur 16 octets :

```
int posix_memalign(void **memptr, 16, sizeof(type))
```

Accès non aligné

Nécessite plusieurs accès mémoire car les données sont réparties sur 2 blocs de 16 octets.

Impact sur les performances

SSE fournit des opérations d'accès sur des données alignées et non alignées. Les accès non alignés sont cependant beaucoup plus lents !

Exemple : Copie de 4 floats

```
1  #include <stdio.h>
2  #include <xmmintrin.h> // Header pour SSE
3
4  int main() {
5    float a1[4] __attribute__ ((aligned (16)))
6                              = {1.4, 2.5, 3.6, 4.8};
7    float a2[4] __attribute__ ((aligned (16)));
8    __m128 v1, v2;
9    v1 = _mm_load_ps(a1);
10   _mm_store_ps(a2, v1);
11   printf("%f %f %f %f\n", a2[0], a2[1], a2[2], a2[3]);
12   return 0;
13 }
```

Opérations arithmétiques

- _mm_add_pd - (Add-Packed-Double)
  - Entrée : [A0, A1], [B0, B1]
  - Sortie : [A0 + B0, A1 + B1]
- _mm_add_ps - (Add-Packed-Single)
  - Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Sortie : [A0 + B0, A1 + B1, A2 + B2, A3 + B3]

Opérations aritmétiques

- _mm_add_epi64 - (Add-Packed-LongInt)
  - Entrée : [A0, A1], [B0, B1]
  - Sortie : [A0 + B0, A1 + B1]
- _mm_add_epi32 - (Add-Packed-Int)
  - Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Sortie : [A0 + B0, A1 + B1, A2 + B2, A3 + B3]

Opérations aritmétiques

- _mm_mul_pd - (Multiply-Packed-Double)
  - Entrée : [A0, A1], [B0, B1]
  - Sortie : [A0 * B0, A1 * B1]
- _mm_mul_ps - (Multiply-Packed-Single)
  - Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Sortie : [A0 * B0, A1 * B1, A2 * B2, A3 * B3]

Addition&Soustraction (disponibles à partir de SSE3)

- _mm_addsub_pd - (Add-Subtract-Packed-Double)
  - Entrée : [A0, A1], [B0, B1]
  - Sortie : [A0 - B0, A1 + B1]
- _mm_addsub_ps - (Add-Subtract-Packed-Single)
  - Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Sortie : [A0 - B0, A1 + B1, A2 - B2, A3 + B3]

Opérations horizontales (disponibles à partir de SSE3)

- _mm_hadd_pd - (Horizontal-Add-Packed-Double)
  - Entrée : [A0, A1], [B0, B1]
  - Sortie : [B0 + B1, A0 + A1]
- _mm_hadd_ps (Horizontal-Add-Packed-Single)
  - Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Sortie : [B0 + B1, B2 + B3, A0 + A1, A2 + A3]
- _mm_hsub_pd - (Horizontal-Subtract-Packed-Double)
  - Entrée : [A0, A1], [B0, B1]
  - Sortie : [A0 - A1, B0 - B1]
- _mm_hsub_ps - (Horizontal-Subtract-Packed-Single)
  - Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]
  - Sortie : [A0 - A1, A2 - A3, B0 - B1, B2 - B3]

Opérations logiques

_mm_{and | or | xor | ...}_{ps | pd | si128}

Comparaisons

La comparaison de 2 registres SSE à l'aide des instructions _mm_cmp... produit un masque contenant un champs de 1 lorsque la condition est vérifiée, et un champs de 1 dans le cas contraire.

Exemple de comparaison

_mm_cmp{eq | gt | lt | . . . }_{ps | pd | epi8 | epi 16 | ...}

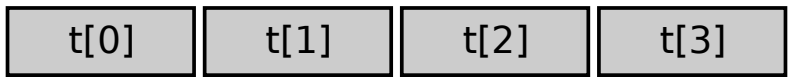
Réorganisation des données

Les données peuvent être réorganisées dans les registres, par exemple pour inverser leur ordre ou effectuer une transposition.

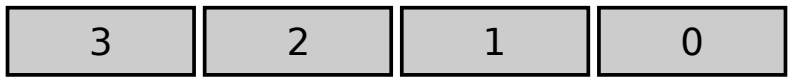
Ordre des données

Attention à l'ordre des données en mémoire et dans les registres qui est inversée !

tableau de float en mémoire



registre SSE contenant 4 floats



Mélange des données : Shuffle

L'instruction shuffle permet de combiner des données de 2 registres donnés en argument suivant un masque indiquant les positions à récupérer.
Il est possible de passer le même registre en argument.

Limitation

L'instruction shuffle impose de récupérer autant de données dans les 2 registres. Par exemple, il est possible de récupérer 2 flottants dans chacun des 2 registres mais pas 1 flottant de l'un et 3 flottants de l'autre.

Exemple

```
1  float a0[4] __attribute__((aligned(16))) = {1, 2, 3, 4};
2  float a1[4] __attribute__((aligned(16))) = {5, 6, 7, 8};
3  float a2[4] __attribute__((aligned(16)));
4
5  __m128 r0 = _mm_load_ps(a0);
6  __m128 r1 = _mm_load_ps(a1);
7
8  // _MM_SHUFFLE is a macro used to create the mask.
9  // In this example, it takes values 0 and 1 from r1
10 //                          and values 2 and 3 from r0
11 r0 = _mm_shuffle_ps(r0, r1, _MM_SHUFFLE(3, 2, 1, 0));
12
13 _mm_store_ps(a2, r0); // contains {1, 2, 7, 8}
```

Quelques instructions

_mm_cvt...