list
cea tech

Programmation SIMD

Avec des diapos pompées à Sylvain Jubertie | Allan Blanchard (allan.blanchard@cea.fr)

INSTITUT CARNOT TN@UPSaclay

université PARIS-SACLAY

Quelques remarques avant de commencer

Temps d'exécution

Le temps d'exécution d'un programme dépend :

- de l'algorithme utilisé (complexité en temps/espace),
- du langage :
  - implémentation compilée : C, Rust
  - interprétation d'un bytecode : Java, Python
  - interprétation directe : Perl, Matlab
- du compilateur, de la machine virtuelle, de l'interpréteur, choisis,
- de l'architecture (spécificité du processeur, des mémoires, des IOs, ...),
- du système d'exploitation,
- d'autres paramètres plus ou moins difficiles à maitriser,
- et bien sûr du développeur ! Code bien écrit ? Exploite bien ses outils ?

Utilisation de la complexité

La complexité n'est pas nécessairement un indicateur de performance

- elle donne une indication de comment *évolue* le temps d'exécution *en fonction de la taille de l'entrée*
- à complexité équivalente (voir seulement légèrement différente), la comparaison peut être biaisée pour certaines tailles d'instance
- certains algorithmes se comportent mieux sur des grosses données, d'autres sur de petites, il peut être intéressant de choisir l'algo à l'exécution en fonction de la taille (ex : `introsort`)

Impact de l'implémentation

L'implémentation d'un langage a des fortes implications

- une implémentation compilée peut facilement être plusieurs dizaines de fois plus rapide qu'une implémentation interprétée pure
- les implémentations à bytecode peuvent se permettre de l'optimisation *just-in-time* (JIT) qui peuvent compenser ces pertes dans certaines situations
- les implémentations interprétées permettent un développement très rapide (hyper utile pour prototyper)

Néanmoins ...

- pour les bibliothèques de calcul très efficaces, on trouve souvent de l'assembleur pour les routines les plus critiques.

Attendre la prochaine version ?

Probablement pas une super idée

- si des gaps de versions de plusieurs années montrent des changements significatifs, la différence d'une version à l'autre n'est souvent que de quelques pourcents (et parfois on perd aussi)

Et puis de toute façon, c'est aujourd'hui qu'on est en train de développer.

Attendre la prochaine version ?

Probablement pas une super idée (non plus)

- une nouvelle génération de processeur ne permet pas des gains énormes, autour de 10/15% en moyenne
- pour le reste, on verra un peu plus tard

Et puis de toute façon, c'est aujourd'hui qu'on est en train de développer.

Impact de l'OS

Très dépendant du type de problème

- lorsque l'on fait du calcul pur, l'objectif est de ne *jamais* faire appel à l'OS ou presque,
- il est néanmoins possible que des changements dans la gestion mémoire ou des différences d'*Application Binary Interface* (ABI) viennent impacter les performances,
- dans les applications qui utilisent intensivement les IOs, l'OS peut avoir un impact notable (et le code doit être conçu en conséquence)

Optimisation prématurée

N'optimisez pas prématurément !

Règles de l'optimisation :

- Première règle : ne le faites pas,
- Seconde règle (expert uniquement) : ne le faites pas encore.

> *Premature optimization is the root of all evil.*
>
> — *D. Knuth*

Néanmoins ...

- Beaucoup de codes ne sont pas optimisés (sans considérer la parallélisation) et exploitent moins de 20% des performances des processeurs

> *On the other hand, we cannot ignore efficiency.*
>
> — *J. Bentley*

Mesurer pour mieux régner

La clé est d'optimiser ce qui doit l'être

Pas de secret, il faut faire des mesures :

- idéalement sur une instance représentative du problème,
- une fois que le code est **correct** et **bien testé**,
- en utilisant des outils de mesure (profilers).

If you don't need a correct result, I can do it as fast as you want.

— G. Weinberg

Processeur

Architecture générale

Processeur moderne

- Mémoire
  - registres
  - caches L1-L2-L3(-L4, rarement)
- Unités d'exécution
  - unités arithmétiques et logiques (ALU)
  - unités flottantes (FPU)
- pipeline d'exécution (comprenant notamment de la prédiction de branches)
- contrôleur mémoire (comprenant notamment de la prédiction d'accès)

Quelques observations

Mémoire et accès

- Les données et instructions sont stockées en mémoire centrale (RAM)
- Il est coûteux d'y accéder
- On accède très souvent à des données/instructions contigües/proche en mémoire
  - localité spatiale
  - données : parcours de tableaux
  - instructions : sauf lors de sauts (notamment boucles et appels)
- On réutilise souvent des données/instructions pendant certaines phases de calcul
  - données : les traitements réutilisent souvent les données tout juste produites
  - instructions : les boucles

Solution

Une mémoire locale au processeur

- Ajouter une mémoire (le cache) rapide mais de petite taille dans le processeur
- Charger les données/instructions par bloc (ligne de cache), et pas par octet
- Le premier accès est coûteux, mais accéder à la suivante profite du cache
- Limitation : les accès doivent être réguliers

Mécanismes supplémentaires

- *prefetching* : on va chercher les données avant d'en avoir besoin
- cache d'écriture : on n'envoie pas immédiatement les écritures vers la mémoire

Cache

Fonctionnement

- lors d'un accès le processeur vérifie si elle est en cache
  - si ce n'est pas le cas (*cache miss*), on va chercher la ligne de cache correspondante en mémoire,
  - sinon (*cache hit*), on prend dans le cache, pas d'accès à la mémoire
- on prend la donnée dans la ligne et on la met en registre pour le traitement
- on la replace dans sa ligne de cache
- elle y reste jusqu'à ce qu'elle soit écrasée (et remise en mémoire)

Caractéristiques

Organisation

- Mémoire directement dans le processeur :
  - très rapide : quelques cycles pour la plus rapide, jusqu'à quelques dizaines
  - taille réduite : quelques dizaines de kos pour la plus rapide, jusqu'à quelques Mos
- Hiérarchisée :
  - L1 : 32-64 kB
  - L2 : 128-256-512 kB
  - L3 : 1-12 MB
- types :
  - cache d'instructions
  - cache de données

Translation Lookaside Buffer

Accélérer l'adressage virtuel

L'adressage virtuel implique de transformer les adresses virtuelles en adresse physique, ce qui peut être coûteux. Le TLB (*Translation Lookaside Buffer*) est un cache utilisé par l'unité de gestion mémoire MMU (*Memory Management Unit*) pour garder les correspondances en question.

Performance

Le principal risque est de saturer le TLB en cas d'accès trop fréquents à de nombreuses adresses différentes.

Exemple

Exemple de mauvaise utilisation du cache

```
#define ACCESS(row, col, width) (row * width + col)

void mat_add(int* res, int* a, int* b, size_t w, size_t h){
  for(size_t col = 0 ; col < w ; col++){
    for(size_t row = 0 ; row < h ; row++){
      size_t idx = ACCESS(row, col, w) ;
      res[idx] = a[idx] + b[idx];
    }
  }
}
```

Diagnostique

```
// gcc mat-add.c -Wall -Wextra -O2
// sudo perf stat -d ./a.out
```

Explication

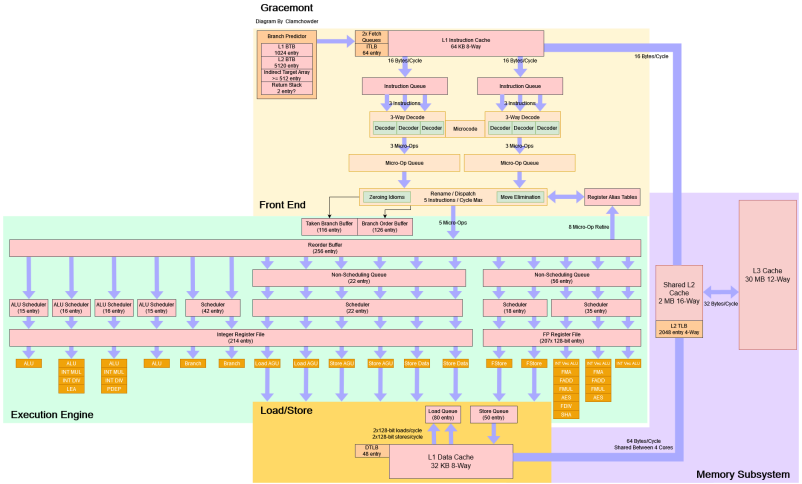
Les données ne sont pas accédées linéairement en mémoire et de nombreux *cache-miss* sont provoqués.

Étapes du traitement des instructions

Deux phases majeures

1. Front-end Fetch&Decode :
   1.1 Chargement de l'instruction à traiter
   1.2 Décodage en micro-instructions
   1.3 Placement dans le pipeline d'instructions
2. Back-end Execute : passage des instructions dans les unités de traitement

Front-end et Back-end



Optimisation Fetch&Decode

Prédiction de branchement

Lors d'un branchement conditionnel, on fait un choix

- si cela amène vers des instructions non-chargées en cache, on a un *cache-miss*
- l'unité de prédiction détermine statistiquement la branche le plus probable
- exemple :
  - dans une boucle, il est très probable que l'on continue au prochain test
  - on ne paiera l'échec que pour la dernière itération

Optimisation Execute

Exécution *out-of-order*

Des instructions indépendantes peuvent être exécutées dans le désordre.
Si une instruction attend des données pour être exécutée et qu'une autre instruction indépendante est prête à être exécutée, on peut la mettre dans le pipeline à la place de la première.
Le compilateur peut préparer ce genre d'optimisation. Mais le développeur peut favoriser les chances que cela arrive en évitant les chaînes de dépendances.

Dépendances

```
a = b+c ;
d = a+c ; // depends on a
e = d+b ; // depends on d
```

Remarques sur ces optimisations

Prédiction de branchement

- souvent couplée à l'exécution spéculative ...
- ... qui nous a amené Spectre et Meltdown

Exécution *out-of-order*

Le comportement des processeurs entraîne des surprises en multi-threading

```
*x = 1 ;      |||    *y = 1
r1 = *y ;     |||    r2 = *x
```

Comportement observables pour `r1,r2` :

- `1,1` ; `1,0` ; `0,1` ...
- ... mais aussi `0,0` !

Pipeline arithmétique

Décomposition des instructions complexes

- les instructions complexes nécessitent plusieurs cycles
- exemple : 5 pour la multiplication, décomposé en 5 étapes de 1 cycle
- si plusieurs multiplications indépendantes sont dans le pipeline :
  1. une multiplication du pipeline est placée dans le 1^{er} étage de l'unité de multiplication
  2. la multiplication passe au 2^{eme} étage et libère le 1^{er} pour une autre multiplication
  3. la multiplication 1 passe à l'étage 3, etc,
  4. ...
  5. la multiplication 1 est traitée par le dernier étage, le résultat est mis dans un registre,
  6. la multiplication 2 est traitée par le dernier étage, le résultat est mis dans un registre,

Pipeline arithmétique

Résultat

- Une fois le pipeline arithmétique plein, une multiplication sort à chaque cycle (sous réserve d'indépendance) : *throughput* de 1
- En fait, plusieurs unités de multiplication peuvent être présentes sur le processeur pour augmenter les performances, généralement 2 : *throughput* de 2
- Mais certaines instructions ne peuvent pas être complètement pipelinée, une division sur Skylake, c'est 11 cycles et un *throughput* de 0.33

Retour sur la complexité

Toutes les opérations ne sont pas faites égales

Lorsque l'on s'intéresse aux performances :

- le poids des constantes devient important
  - un accès `*p` et un accès `x` n'ont pas le même coût
- certaines constantes ne le sont plus
  - un accès `*p`, puis `*(p+1)` et `*p`, puis `*(p+100)` n'ont pas le même coût

Principe

Faire fonctionner deux threads sur le même coeur

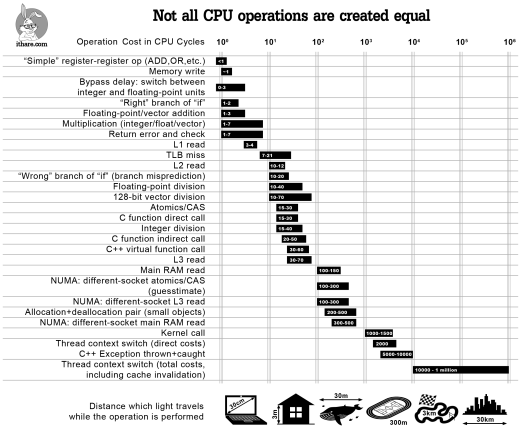
- Le Simultaneous Multi-Threading, ou Hyper-Threading chez Intel, n'est pas une technologie multi-coeurs
- Ces technologies partagent les registres et le pipeline d'exécution entre plusieurs threads (et parfois dupliquent certaines unités) pour améliorer le taux d'utilisation du cœur
- Les systèmes avec SMT font apparaître des coeurs logiques et pas physiques
- En conséquence, on ne double *pas* les performances

Gains potentiels

Le gain est très dépendant des applications

- sur du code mal optimisé, on peut gagner pas mal ...
- mais cela peut être aussi le cas sur du code qui utilise massivement les IOs
- sur du code de calcul brut, on peut atteindre du 20-30%, mais cela varie :
  - favorable : threads qui exploitent des unités distinctes et où l'ajout du second thread n'entraîne pas de saturation du cache,
  - nul : threads utilisant des unités communes et beaucoup de registres,
  - négatif : comme le cas nul, mais avec en plus saturation du cache.

Petit résumé des échelles de temps



La vitesse de la lumière devient une limitation technique aujourd'hui

Il y a pas mal de trucs intéressants sur `http://ithare.com`

`http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles`

Unités vectorielles

Calcul scalaire

Fonctionnement

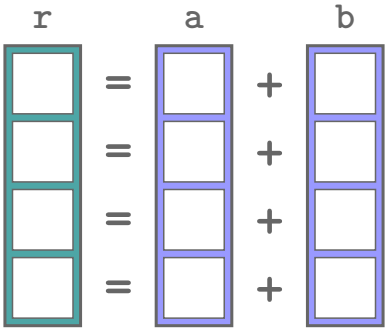
Un processeur scalaire effectue les opérations séquentiellement, chaque opération portant sur des données scalaires (un scalaire).

Exemple : addition de deux vecteurs de 4 éléments

```
r[0] = a[0] + b[0] ;
r[1] = a[1] + b[1] ;
r[2] = a[2] + b[2] ;
r[3] = a[3] + b[3] ;
```

Très commun dans le calcul numérique !

Calcul scalaire



Calcul vectoriel

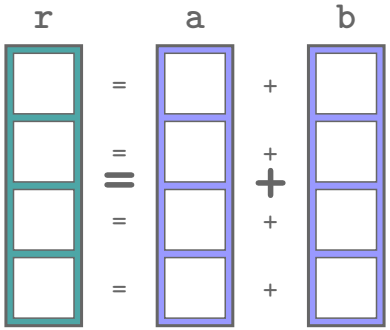
Fonctionnement

Un processeur vectoriel effectue une même instruction simultanément sur plusieurs données (un vecteur de scalaires) : *Single instruction, Multiple Data* (SIMD).

Exemple : addition de deux vecteurs de 4 éléments

```
r = a + b ; // on verra que c'est pas exactement ce que l'on ecrit
```

Calcul vectoriel



Avantages et inconvénients de l'approche vectorielle

Avantages

Calcul n fois plus rapide avec n la *largeur* (le nombre de données traitées simultanément) de l'unité vectorielle, **à condition que l'algorithme se prête au paradigme SIMD** :

- calcul vectoriel,
- traitement d'image.

Inconvénient

Gain **uniquement** si l'algorithme se prête au paradigme SIMD. Sinon on se retrouve juste dans le cas scalaire.

Premières implantations

Historique

- Années 60 : projet Solomon (Westinghouse)
  - plusieurs ALU contrôlées par 1 CPU maître
- Années 70 : ILLIAC IV, Université de l'Illinois
  - basé sur les idées du projet Solomon
  - première machine vectorielle (64 FPU à 64 bits)
- ensuite : ASC (1974) CDC (1974), Cray (1975), NEC, Hitachi, Fujitsu
- depuis 96 dans les processeurs grand public, scalaires + unités vectorielles, Pentium (MMX, SSE, AVX), PowerPC (AltiVec)

MMX

MultiMedia eXtension

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

MMX - Caractéristiques

Description

- 8 registres 64 bits
- 57 opérations *sur les entiers*

Limitation

- on aime bien les flottants dans le calcul numérique ...
- ... et registres sont partagés avec l'unité FPU

Autres jeux d'instructions SIMD

Unités SIMD

- AMD 3NNow ! : AMD K6-2, ..., Phenom II, Transmeta Crusoe et Efficeon, ...
- ARM Neon : à partir des Cortex A8-9 (et équivalents depuis)
- approches FPGA
- GPU (couplé à des approches SIMT)

Programmation - principe de base

Idée générale pour programmer avec des unités SIMD

- on charge les données depuis la mémoire vers les registres SIMD
- on fait différentes opérations sur ces registres
- on replace le résultat en mémoire

Programmation - outils

Vectorisation automatique

- Le compilateur tente de vectoriser automatiquement

Assembleur

- Programmation bas-niveau
- Manipulation directe des registres et instructions

Intrinsics

- Headers fournis par les implémentations de compilateur
  - ex : GCC `pmmintrin.h` pour SSE3

Bibliothèques haut niveau

- Abstraction de la programmation vectorielle

Performances

Astuce pour obtenir les meilleurs performances

- Organisez correctement vos données pour éviter les accès non contigus !
- Alignement des données compatibles avec les opérations vectorielles
- Rester aussi longtemps que possible dans les registres !

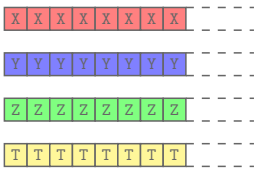
Organisation des données

Array of Structures (AoS)

```
struct v { float X, Y, Z, T ; };
struct v * vs ;
```

Structure of Arrays (SoA)

```
struct vs { float *X, *Y, *Z, *T ; };
```



Structures hybride (ex : AoSoA)

```
struct v4 {
  float X[4], Y[4], Z[4], T[4] ;
};
struct v4 * vs ;
```

Streaming SIMD Extensions

Itérations

- SSE : Pentium III, Athlon XP
- SSE2 : Pentium 4
- SSE3 : Pentium 4 Prescott
- SSSE3 - (Supplemental SSE3)
- SS4.1 - Core 2
- SS4.2 - Core i7
- chez AMD, le support peut varier. Depuis Ryzen, tout est supporté

Support disponible

Connaître la version de SSE de son processeur

```
$ cat /proc/cpuinfo
```

- sse
- sse2
- pni (Prescott New Instructions - SSE3)
- ssse3
- sse4_1
- sse4_2

Qu'est ce que SSE ?

Registres de 128 bits

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

Instructions

- transfert mémoire vers et depuis les registres SSE
- opérations arithmétiques
- opérations logiques
- tests
- permutations

Intrinsics

Définitions

Les intrinsics sont accessibles via des fichiers header qui définissent :

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

Fichiers

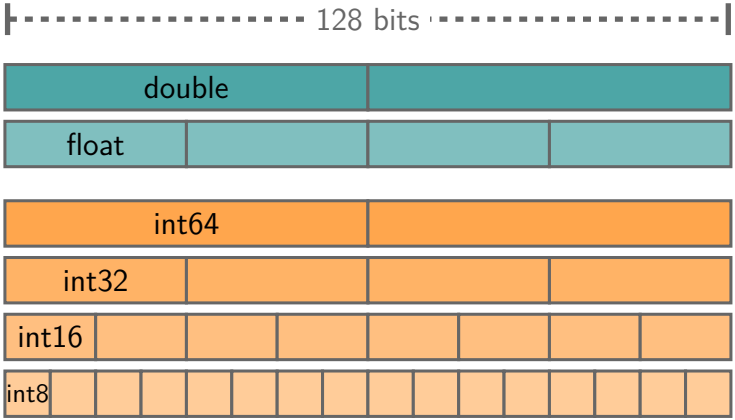
- SSE : `xmmintrin.h`
- SSE2 : `emmintrin.h`
- SSE3 : `tmmintrin.h`
- SSSE3 : `xmmintrin.h`
- SSE4.X : `smmintrin.h`

Types de données

Types

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

Types de données - Schéma



Fonctions

Nomenclature

```
_mm_{operation}{alignement}_{data-organization}{datatype}(...)
```

Exemple

```
__m128 r = _mm_add_ps
```

Nommage

Terminologie

- `s` (single) : `float`
- `d` (double) : `double`
- `i...` (integer) : `int..._t`
- `p` (packed) : contigus
- `u` (unaligned) : accès non aligné
- `l` (low) : bit de poids faible
- `h` (high) : bit de poids fort
- `r` (reversed) : dans l'ordre inverse

Fonctionnement

Idée générale

1. déclarations des variables SSE (registres)
   ```
   __m128 r1 ;
   ```
2. chargement des données de la mémoire ou création de valeur :
   ```
   r1 = _mm_load...(ptr) ;
   r1 = _mm_set...(values...) ;
   ```
3. opérations sur les registres SIMD
4. placement du cointenu des registres en mémoire :
   ```
   _mm_store...(ptr, r1);
   ```

Transferts

Différents types de transfert entre registres et mémoire

- alignés ou non
  - `_mm_load_...` ou `_mm_loadu_...`
  - `_mm_store_...` ou `_mm_storeu_...`
- par vecteur (4 `float`, 2 `double`)
  - `_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`

Alignement

Accès alignés

- Le processeur (x86_64) peut faire des transferts efficaces de 16 octets (128 bits) entre mémoire et registre SSE sous la condition que le bloc soit aligné sur 16 octets. Cette contrainte est **matérielle**.
- Certaines architectures **ne permettent pas** des accès non alignés

Attention

- L'alignement dépend de l'architecture et du type de variable. Par défaut l'alignement d'un entier de 32 bits est effectué sur 4 octets.
- Pour obtenir l'alignement, on utilise le mot-clé `__alignof__(type)`
- ex : `__alignof__(int[4]) == 4`

Alignement et ligne de cache

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.

Alignement et allocation

Allocation de données alignées

- Alignement de données statiques alignées sur 16 octets :
  - `int x __attribute__ (( aligned (16) ));`
- Alignement de données dynamiques alignées sur 16 octets :
  - C11 : `void * aligned_alloc (size_t alignment, size_t size);`
  - POSIX : `int posix_memalign (void **memptr, size_t alignment, size_t size);`
  - Windows : `void * _aligned_malloc(size_t size, size_t alignment);`
  - Intel : `void* _mm_malloc (int size, int align);`

Alignement et performances

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

```
#include <stdio.h>
#include <xmmintrin.h> // header pour SSE

int main(void){
  float a1[4] __attribute__((aligned (16))) = { 1.4f, 2.5f, 3.6f, 4.7f } ;
  float a2[4] __attribute__((aligned (16))) ;
  __m128 v1 ;
  v1 = _mm_load_ps(a1);
  _mm_store_ps(a2, v1);
  printf("%f␣%f␣%f␣%f\n", a2[0], a2[1], a2[2], a2[3]);
}
```

Opérations arithmétiques

Addition flottante

- `_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 arithmétiques

Addition entière

- `_mm_add_epi64` - Add Packed Int 64
  - Entrée : `[ A0, A1 ], [ B0, B1 ]`
  - Sortie : `[ A0+B0, A1+B1 ]`
- `_mm_add_epi32` - Add Packed Int 32
  - Entrée : `[ A0, A1, A2, A3 ], [ B0, B1, B2, B3 ]`
  - Sortie : `[ A0+B0, A1+B1, A2+B2, A3+B3 ]`

Opérations arithmétiques

Multiplication flottante

- `_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 ]`

Opérations arithmétiques

Addition & Soustraction flottante (SSE3+)

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

Opérations arithmétiques

Opérations horizontales (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 Substract Packed Double
  - Entrée : `[ A0, A1 ], [ B0, B1 ]`
  - Sortie : `[ B1+B0, A1-A0 ]`
- `_mm_hsub_ps` - Horizontal Substract Packed Single
  - Entrée : `[ A0, A1, A2, A3 ], [ B0, B1, B2, B3 ]`
  - Sortie : `[ B1-B0, B3-B2, A1+A0, A3-A2 ]`

Opérations logiques

Opérations logiques

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

Opérations de comparaison

Opérations de comparaison

- La comparaison de deux registres SSE à l'aide d'instructions `_mm_cmp...` produit un masque contenant tous les bits à 1 si la condition est vérifiée, 0 sinon.

Exemple

- `_mm_{eq|gt|lt...}_{ps|pd|epi8|epi16...}`

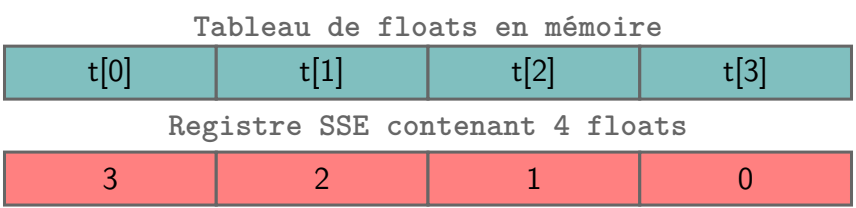
Ordre et réorganisation des données

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 inversé !



Ordre et réorganisation des données

Mélange de 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.

Ordre et réorganisation des données - Exemple

```
#include <stdio.h>
#include <xmmintrin.h> // header pour SSE

int main(void){
  float a0[4] __attribute__((aligned (16))) = { 1.f, 2.f, 3.f, 4.f } ;
  float a1[4] __attribute__((aligned (16))) = { 5.f, 6.f, 7.f, 8.f } ;
  float a2[4] __attribute__((aligned (16))) ;
  __m128 r0 = _mm_load_ps(a0);
  __m128 r1 = _mm_load_ps(a1);

  /* _MM_SHUFFLE is a macro used to create the mask
     In this example, it takes values 0 and 1 from r1
                              and values 2 and 3 from r0
  */
  r0 = _mm_shuffle_ps(r0, r1, _MM_SHUFFLE(3, 2, 1, 0));
  _mm_store_ps(a2, r0); // contains 1, 2, 7, 8

  printf("%f␣%f␣%f␣%f\n", a2[0], a2[1], a2[2], a2[3]);
}
```

Parlons de quelques instructions

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Advance Vector eXtensions

Extension de 128 à 256 bits

- Les registres passent à 256 bits et se nomment `YMM`
- 2 versions : AVX et AVX2
- En théorie 2x plus rapide

Limitations

- Physique : Fréquence du processeur diminuée lors du traitement des instructions AVX pour limiter consommation et chauffe du processeur
- Programmation : Registre AVX composé de 2 blocs de 128 bits. Le bloc bas est le registre SSE (`XMM`), le bloc haut n'a pas de nom.
  - impossible d'échanger des données avec `shuffle`, `permute`, `shift`, etc entre les sous-blocs. Il faut passer par les intrinsics `permute2f128` pour permuter les blocs.
  - Idem pour AVX-512 : 4 blocs de 128 bits.