# Instructions SIMD

Christophe Mouilleron

ensiie
école nationale supérieure d'informatique
pour l'industrie et l'entreprise

# Motivation

Beaucoup d'algo. manipulent des tableaux :

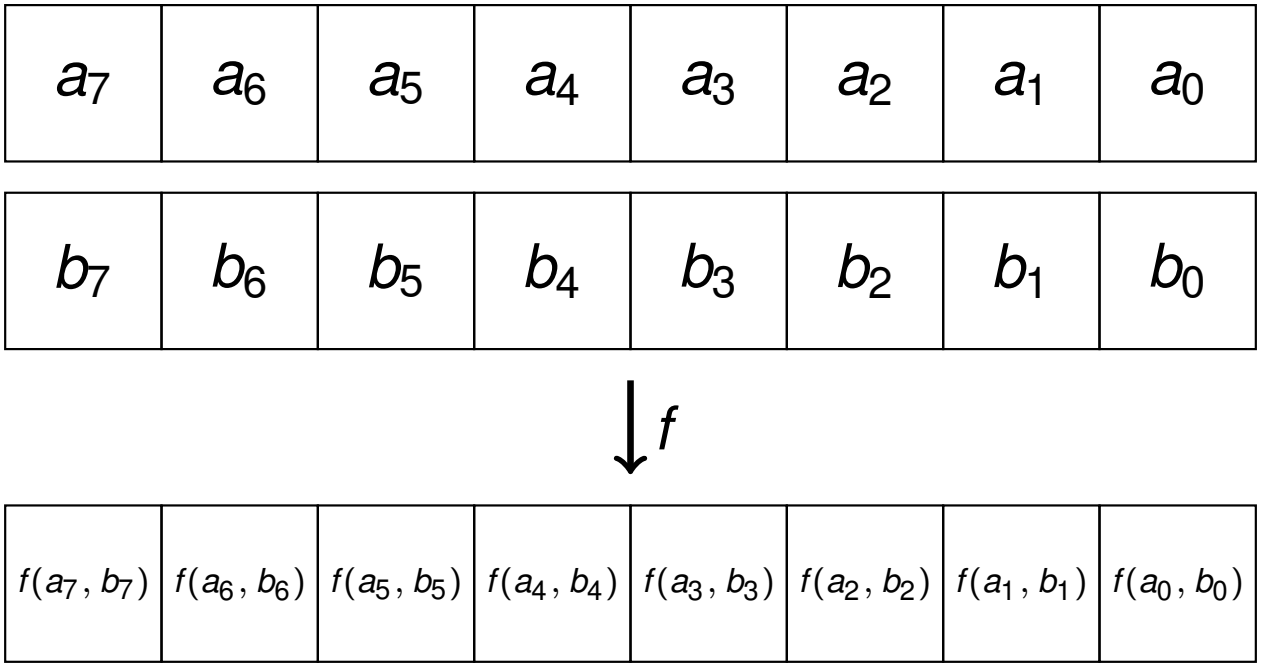
- optimisation (algèbre linéaire)
- statistiques
- traitements d'images
- traitements audio/vidéo

Idée = support matériel pour :

- manipuler directement des petits tableaux (2 à 16 cases)
- faire des opérations point à point dessus

# SIMD : parallélisme de données

SIMD = Single Instruction, Multiple Data




$\rightsquigarrow$ calcul en parallèle sur les données

# Extensions du jeu d'instructions x86

Succession de plusieurs extensions au fil des années :

| | | |
|---|---|---|
| MMX | registres 64 bits, vecteurs de 2 à 8 entiers | 1996 |
| 3DNow ! | + vecteur de 2 `float` | AMD,1998 |
| SSE | reg. 128 bits, vecteurs de 4 `float` | Intel,1999 |
| SSE2 | + vecteurs de `char`, `short`, `long`, `double` | 2001 |
| SSE3/4 | plus d'instructions | 200x |
| AVX | reg. 256 bits | 2011 |
| AVX-512 | reg. 512 bits | 2016 |

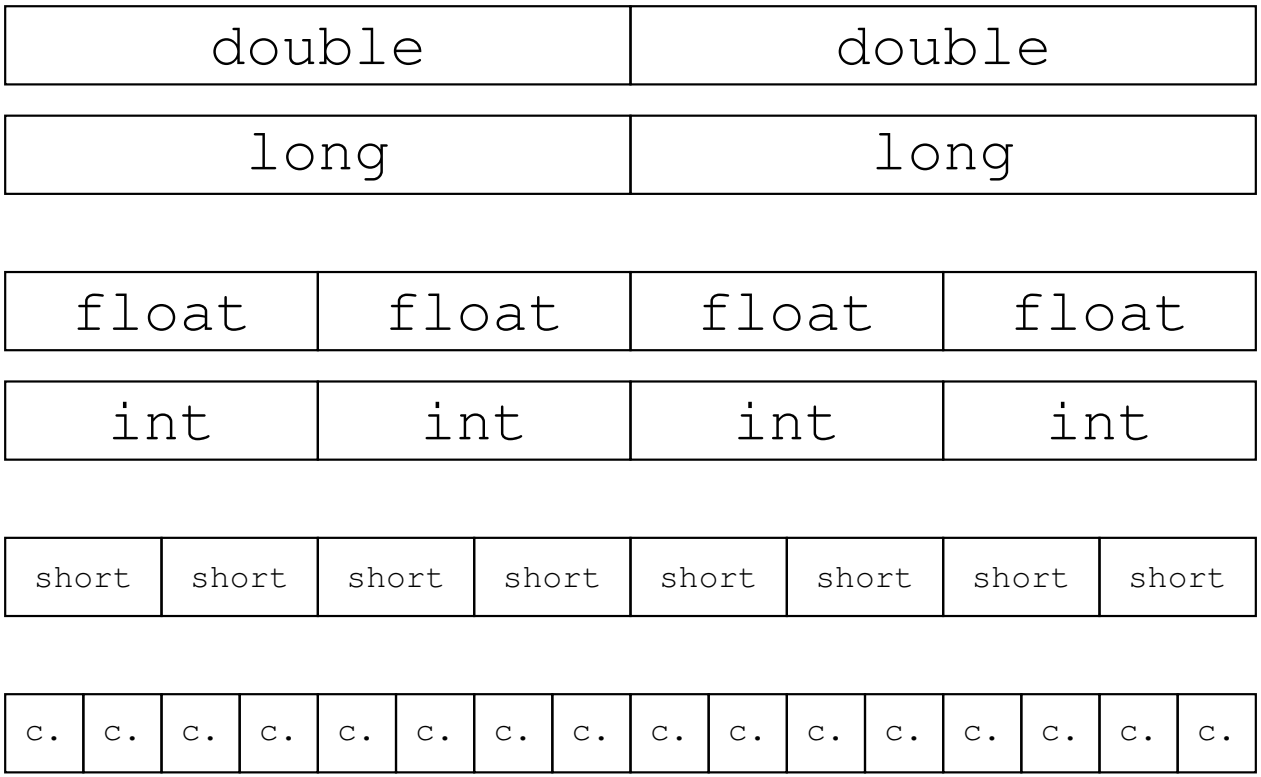
Extensions supportées par une machine ? `/proc/cpuinfo`

# Registres xmm

Registres utilisés pour les instructions SSE/SSE2 :

- 8 registres de 128 bits

xmm0 - xmm7




Attention à l'endianness !

# Opérations sur les registres `xmm`

Transfert mémoire :

- load
- store (avec ou sans masque)

Arithmétique / logique :

- arithmétique entière (avec ou sans saturation)
- arithmétique flottante
- comparaisons
- logique

Instructions spécifiques :

- racine carrée, moyenne, entrelacement, ...

# Utilisation de SSE en pratique

Coder directement en assembleur :

- ✗ trop fastidieux
- ✗ bloque d'autres optimisations du compilateur

Utiliser la macro `asm` de `gcc` :

- ✗ fastidieux
- ✓ garantie d'avoir le code assembleur voulu

# Utilisation de SSE en pratique

Coder directement en assembleur :

- ✗ trop fastidieux
- ✗ bloque d'autres optimisations du compilateur

Utiliser la macro `asm` de `gcc` :

- ✗ fastidieux
- ✓ garantie d'avoir le code assembleur voulu

Utiliser les fonctions intrinsèques fournies dans `x86intrin.h` :

- ✓ usage nettement plus simple
- ✓ bon support par les divers compilateurs

# Utilisation de SSE en pratique (suite)

Doc. sur les fonctions intrinsèques : `x86intrin.h`

`http://software.intel.com/sites/landingpage/IntrinsicsGuide/`

Types `C` disponibles :

- `__m64` 64 bits, tableau d'entiers
- `__m128i` 128 bits, tableau d'entiers
- `__m128` tableau de 4 `float`
- `__m128d` tableau de 2 `double`

Fonctions `_mm_<op>_<type>` avec :

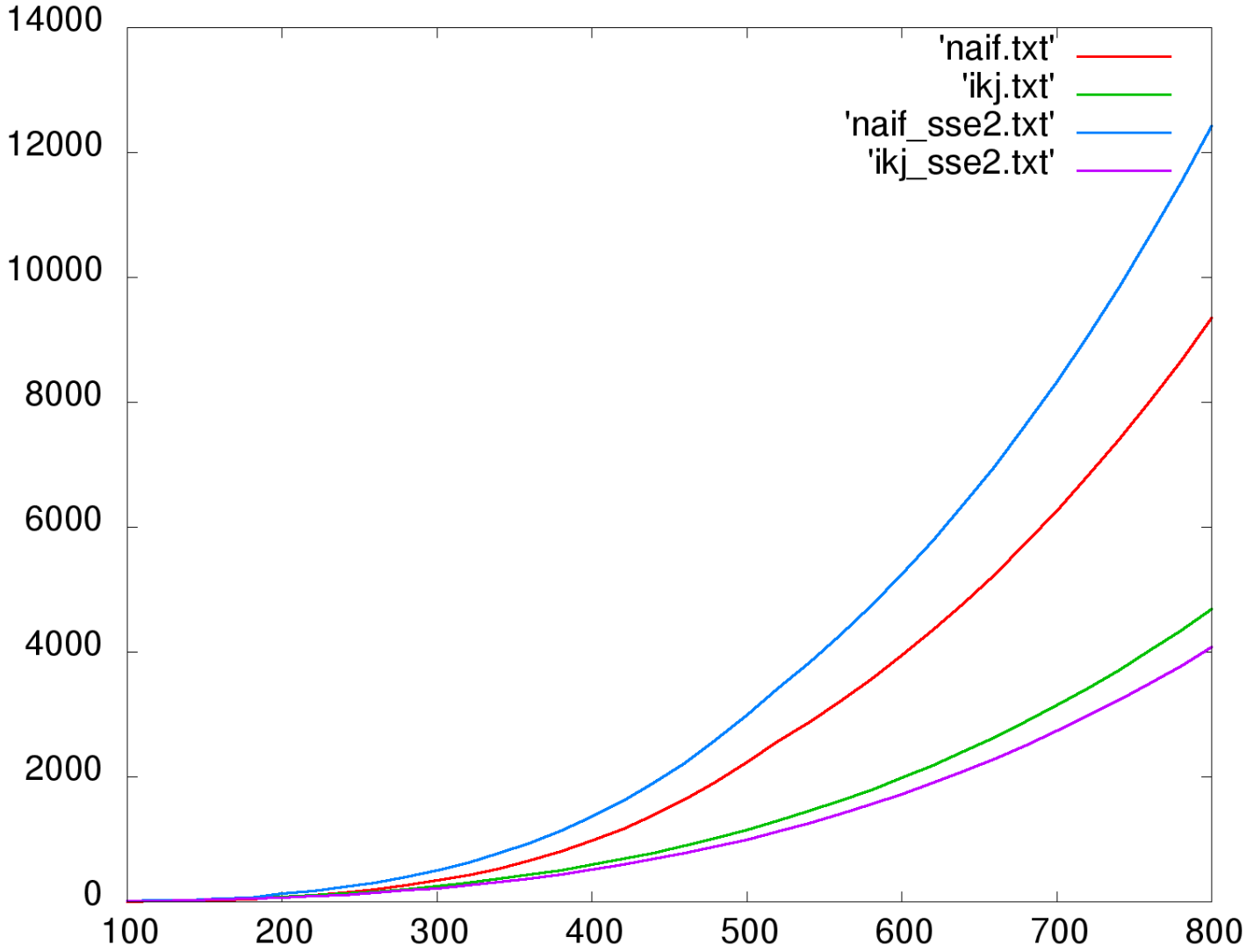
- `<op>` = opération à effectuer — add
- `<type>` = type des données — `pi8` = packed 8-bit integer

# Retour sur le produit de matrices (1)

$\rightsquigarrow$ `naif_sse2.c` et `ikj_sse2.c`

# Retour sur le produit de matrices (1)

⇝ `naif_sse2.c` et `ikj_sse2.c`



# Retour sur le produit de matrices (2)

Résultats pour `naif_sse2.c` :

- résultat décevant
- mauvais accès aux données

# Retour sur le produit de matrices (2)

Résultats pour `naif_sse2.c` :

- résultat décevant
- mauvais accès aux données

Résultats pour `ikj_sse2.c` :

- léger gain
- manipuler des `double[2]` = pas si rentable

⇝ passage à `float` pour meilleur parallélisme de données

# AoS contre SoA

**Problème :** comment représenter $N$ points en $2D$ ?

# AoS contre SoA

**Problème :** comment représenter $N$ points en $2D$ ?

AoS Array of Structures

```
struct point { float x; float y; };
struct point coords[N];
```

SoA Structure of Arrays

```
struct points { float x[N]; float y[N]; };
```

# AoS contre SoA

**Problème :** comment représenter $N$ points en $2D$ ?

AoS Array of Structures

```
struct point { float x; float y; };
struct point coords[N];
```

SoA Structure of Arrays

```
struct points { float x[N]; float y[N]; };
```

Organisation des données cruciale pour utiliser le SIMD :

- travail sur chaque point en parallèle $\Rightarrow$ AoS  nb. à dist. $\leq d$ de 0
- travail en parallèle sur une coordonnée $\Rightarrow$ SoA  réflexion $x \mapsto -x$

# Vectorisation automatique

`gcc` peut produire du code utilisant SSE quand :

- compilation avec `-O3` — `-march=native` aide aussi
- manipulation de tableaux — pas de pointeurs
- accès à pas constants — boucle `for`
- pas de dépendance

$\rightsquigarrow$ transformer les nids de boucles à la main pour aider `gcc`

Autre approche :

- parallélisation des boucles via des pragma — OpenMP, OpenCL