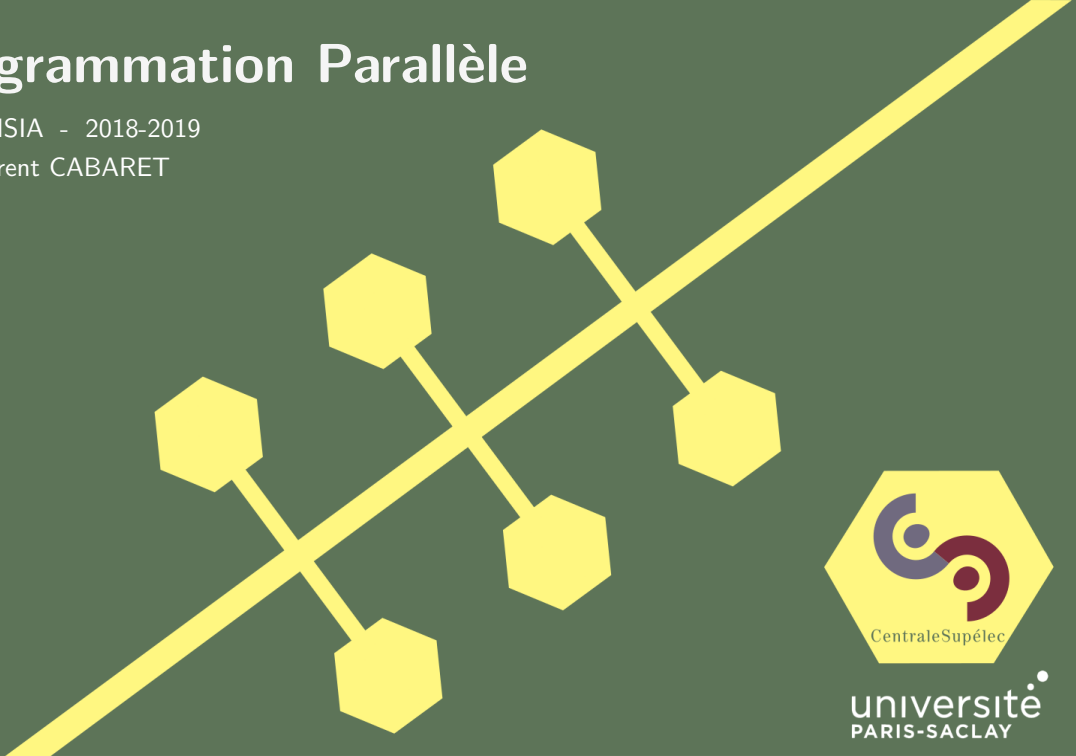
Programmation Parallèle

Option ISIA - 2018-2019

par Laurent CABARET



CentraleSupélec

université PARIS-SACLAY

Plan

- **Présentation** : Objectifs du cours, déroulement, évaluation, illustrations
- **Ab uno disce omnes** : Architecture séquentielle et mur de la chaleur
- **Paradigme parallèle**
- **Parallélisme d'instruction** : Vectorisation et SIMD
- **Parallélisme multi-cœur** : Mémoire partagée, OpenMP, Race conditions
- **Parallélisme massif** : cas du GPU, programmation CUDA

Vectorisation et SIMD

Concept de calcul vectoriel

Certaines applications nécessitent d'appliquer un très grand nombre de fois une même opération sur des données différentes.

$$\forall i \in [1; 10^6], S_i = A_i + B_i$$

Dans un cadre purement séquentiel

Il faut charger l'instruction à effectuer, la décoder, charger les données $A[i]$ et $B[i]$ et stocker le résultat dans $S[i]$.

Concept de calcul vectoriel

Certaines applications nécessitent d'appliquer un très grand nombre de fois une même opération sur des données différentes.

$$\forall i \in [1; 10^6], S_i = A_i + B_i$$

Dans un cadre purement séquentiel

Il faut charger l'instruction à effectuer, la décoder, charger les données $A[i]$ et $B[i]$ et stocker le résultat dans $S[i]$.

Dans un cadre purement vectoriel

L'opération deviendrait :

$$S = A + B$$

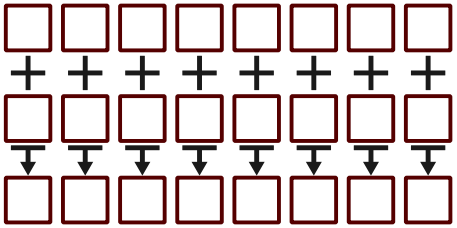
C'est le formalisme utilisé par exemple par Matlab (du point de vue utilisateur)

Dans le cas réel d'un processeur possédant des extensions vectorielles SIMD

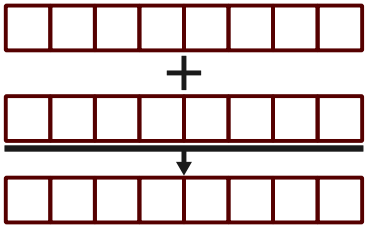
Il faut charger l'instruction à effectuer, la décoder, charger les données $A[i]...A[i+k-1]$ et $B[i]...B[i+k-1]$ et stocker le résultat dans $S[i]...S[i+k-1]$. Avec une seule instruction on traite k données.

8 additions 32 bits distinctes

En 8 opérations



En 1 opération



Concept de calcul vectoriel

Bénéfice quand ...

- les données à traiter sont contigües et ordonnées
- de nombreux calculs peuvent être fait dans les registres vectoriels.

Concept de calcul vectoriel

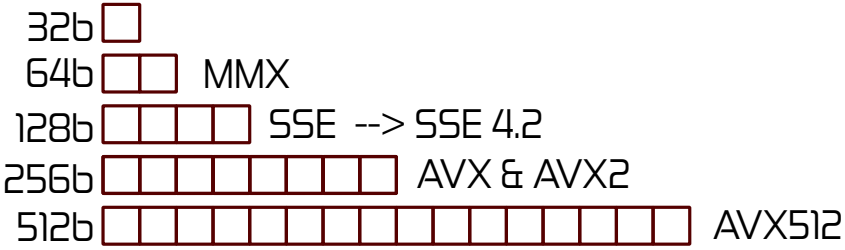
Bénéfice quand ...

- les données à traiter sont contigües et ordonnées
- de nombreux calculs peuvent être fait dans les registres vectoriels.

Perte de l'intéret quand ...

- la structure de données rend discontinu des données continues
- faible intensité algorithmique (ratio calculs/chargements mémoire). Dans ce cas on se trouve limité par la mémoire.
- les traitements ne sont pas ordonnés.
- les traitements introduisent des dépendances entre éléments proches.

Evolution des extensions SIMD «grand public»



Nombre d'opérations simultanées en AVX2 :

- double précision (64bits) : `double`$\rightarrow 4$
- simple précision (32bits) : `float` $\rightarrow 8$
- demi précision (16bits) : `F16` $\rightarrow 16$ (rare !)
- entiers 64 bits : `int64` $\rightarrow 4$
- entiers 32 bits : `int32` $\rightarrow 8$
- entiers 16 bits : `int16` $\rightarrow 16$
- entiers 8 bits : `int8` $\rightarrow 32$

IBM Power

AltiVec

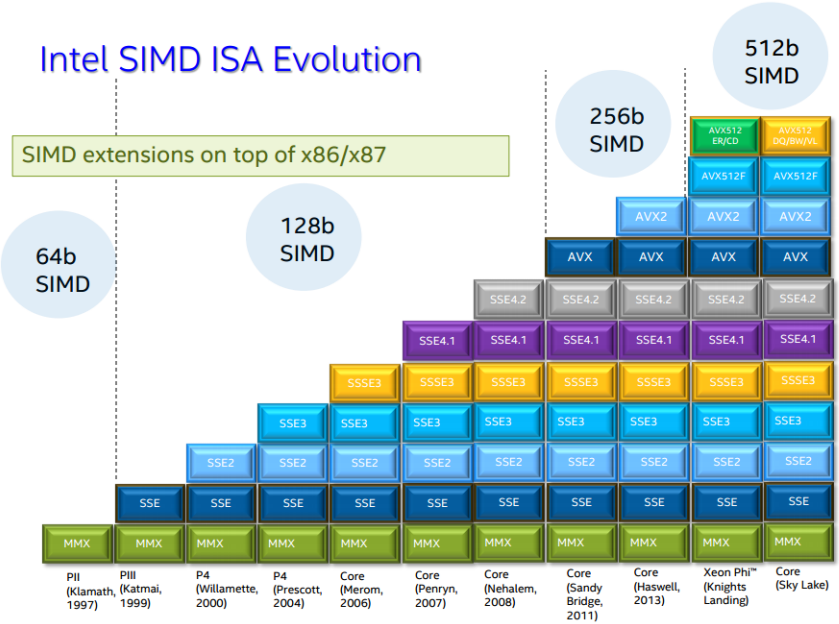
ARM

NEON

Mais aussi

Very long instruction word (VLIW)

Evolution des extensions SIMD «grand public»



Une affaire de compilateur !

Le concept VLIW était concurent du superscalaire (niveau processeur) avec pour idée principale de permettre au compilateur de rassembler des instructions similaires.

Une affaire de compilateur !

Le concept VLIW était concurent du superscalaire (niveau processeur) avec pour idée principale de permettre au compilateur de rassembler des instructions similaires.

Une affaire de compilateur ?

L'auto-vectorisation progresse mais ne fait pas tout et dans les cas les plus complexes il est plus efficace d'indiquer directement nos intentions.

SIMD et langages

Une affaire de compilateur !

Le concept VLIW était concurent du superscalaire (niveau processeur) avec pour idée principale de permettre au compilateur de rassembler des instructions similaires.

Une affaire de compilateur ?

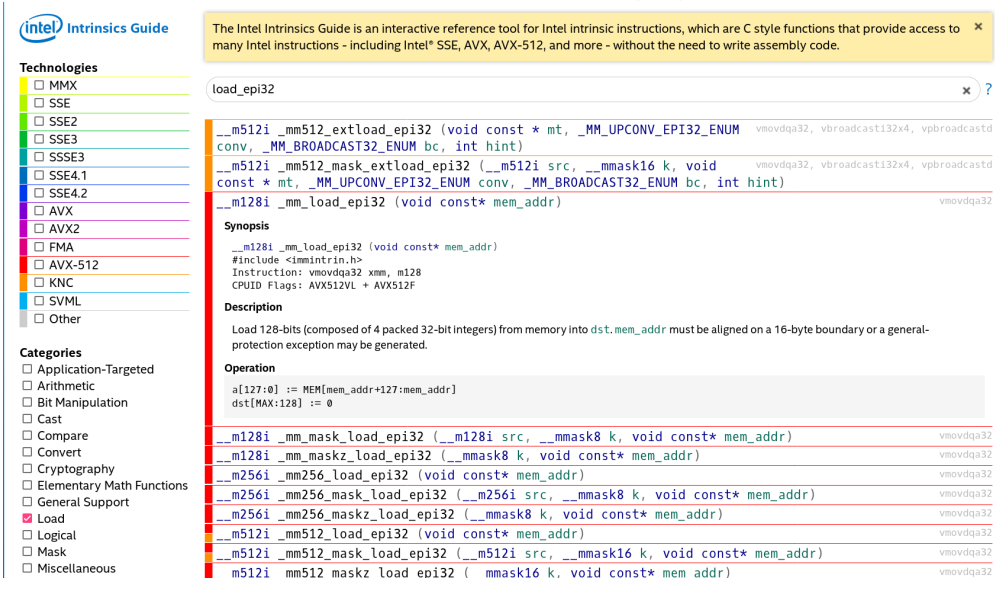
L'auto-vectorisation progresse mais ne fait pas tout et dans les cas les plus complexes il est plus efficace d'indiquer directement nos intentions.

Langages

- C
- C++
- C# - RyuJIT
- python - VecPy
- Fortran
- JAVA

Les intrinsics

https://software.intel.com/sites/landingpage/IntrinsicsGuide



Les intrinsics - Premier exemple - Somme de deux vecteurs d'entiers

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

int main() {
  __m256i a = _mm256_set_epi32(1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 );
  __m256i b = _mm256_set_epi32(11,12,13,14,15,16,17,18);
  __m256i s = _mm256_add_epi32(a, b);
  uint32_t t[8];
  _mm256_storeu_si256((__m256i *)t, s);

  for(int i=7; i>=0;i--) { std::cout << t[i] << " ";}
  std::cout << std::endl;
  return 0;
}
```

Resultat - Attention à l'ordre

$ 26 24 22 20 18 16 14 12

_mm256_set_epi32 (int e7, int e6, … int e1, int e0)

#include <immintrin.h>
CPUID Flags : AVX

Description

Set packed 32-bit integers in dst with the supplied values.

Operation

dst[31:0] := e0
dst[63:32] := e1
dst[95:64] := e2
dst[127:96] := e3
dst[159:128] := e4
dst[191:160] := e5
dst[223:192] := e6
dst[255:224] := e7
dst[MAX:256] := 0

_mm256_add_epi32 (__m256i a, __m256i b)

#include <immintrin.h>
CPUID Flags : AVX2

Description

Add packed 32-bit integers in a and b, and store the results in dst.

Operation

```
FOR j := 0 to 7
    i := j*32
    dst[i+31:i] := a[i+31:i] + b[i+31:i]
ENDFOR
dst[MAX:256] := 0
```

_mm256_storeu_si256 (__m256i * mem_addr, __m256i a)

#include <immintrin.h>
CPUID Flags : AVX

Description

Store 256-bits of integer data from a into memory. mem_addr does not need to be aligned on any particular boundary.

Operation

MEM[mem_addr+255:mem_addr] := a[255:0]

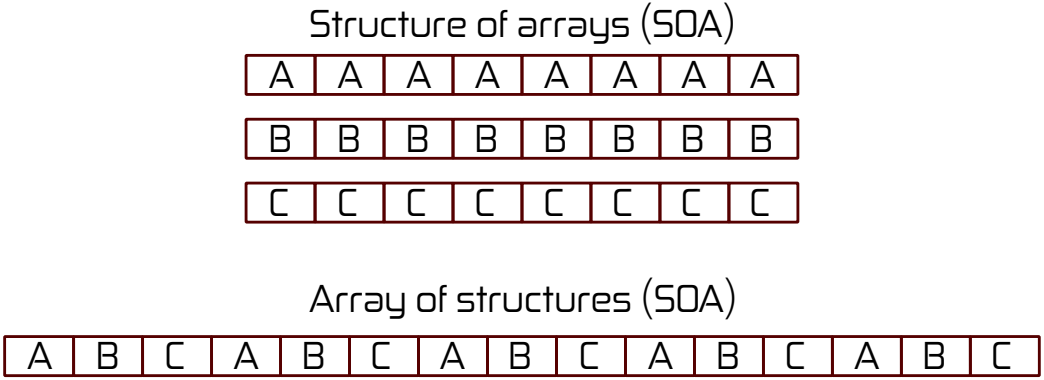
Types spécifiques au SIMD

- __m64 $\rightarrow$ pas utilisable sur les architectures X64 avec le compilateur de VisualStudio
- __m128 - contient 4 float (XMM0)
- __m128d - contient 2 double
- __m128i - contient 4 entiers 32-bits
- __m256 - contient 8 float (YMM0 - XMM0)
- __m256d - contient 4 double
- __m256i - contient 8 entiers 32-bits
- __m512 - contient 16 float (ZMM0 - YMM0 - XMM0)
- __m512d - contient 8 double
- __m512i - contient 16 entiers 32-bits

Types spécifiques au SIMD

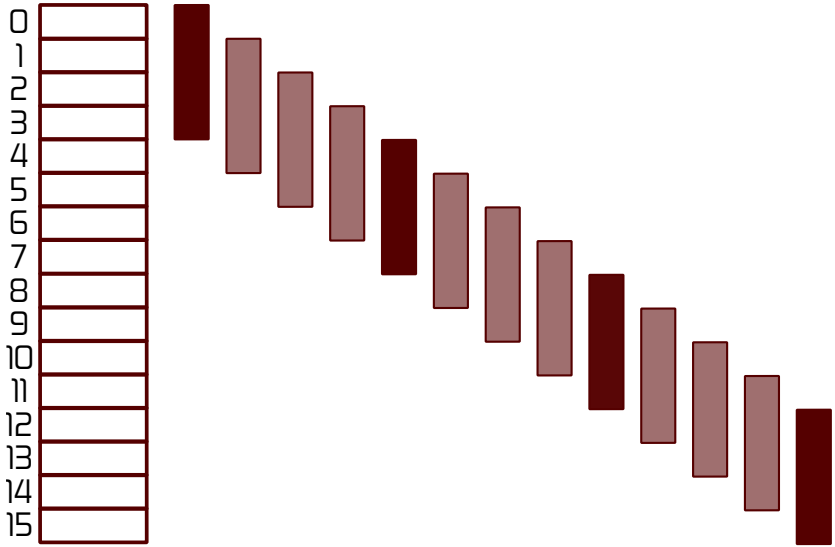
- _epi8 : les opérandes sont composés d'entiers 8 bits
- _epi16 : les opérandes sont composés d'entiers 16 bits
- _epi32 : les opérandes sont composés d'entiers 32 bits
- _epi64 : les opérandes sont composés d'entiers 64 bits
- _ps : les opérandes sont composés de float 32 bits
- _pd : les opérandes sont composés de double 64 bits

SOA vs AOS


Structure of arrays (SOA)
A A A A A A A A
B B B B B B B B
C C C C C C C C
Array of structures (SOA)
A B C A B C A B C A B C A B C

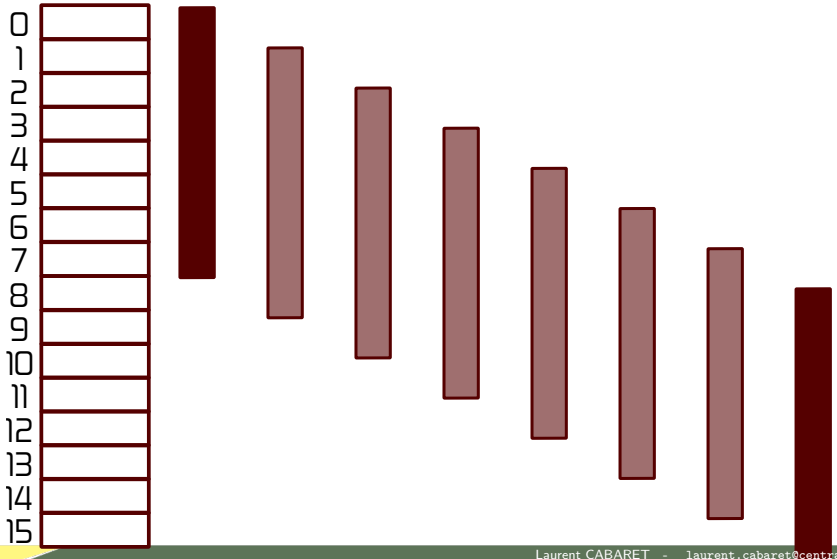
Alignement mémoire

Une donnée est alignée en mémoire si son adresse est un multiple de sa taille.



Alignement mémoire

La lecture/l'écriture d'une donnée non alignée necessite de chargée deux données et de ne sélectionner que les parties utiles. Depuis AVX les chargements alignés et non alignés sont équivalents en terme de performance.



Calcul de la moyenne de deux vecteurs de float

```
#include <immintrin.h>
#include <iostream>
int main() {
  __m256 a = _mm256_set_ps(1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 );
  __m256 b = _mm256_set_ps(11,12,13,14,15,16,17,18);
  __m256 s = _mm256_add_ps(a, b);
  a = _mm256_set1_ps(2);
  s = _mm256_div_ps(s,a);
  float t[8];
  _mm256_storeu_ps((float *)t, s);
  for(int i=0; i<8;i++) { std::cout << t[i] << " ";}
  std::cout << std::endl;
  return 0;
}
```

Resultat

```
$ 13 12 11 10 9 8 7 6
```

_mm256_set1_ps(float a)

#include <immintrin.h>
CPUID Flags : AVX

Description

Broadcast single-precision (32-bit) floating-point value a to all elements of dst.

Operation

```
FOR j := 0 to 7
    i := j*32
    dst[i+31:i] := a[31:0]
ENDFOR
dst[MAX:256] := 0
```

_mm256_div_ps (__m256 a, __m256 b)

#include <immintrin.h>
CPUID Flags : AVX

Description

Divide packed single-precision (32-bit) floating-point elements in a by packed elements in b, and store the results in dst.

Operation

```
FOR j := 0 to 7
    i := 32*j
    dst[i+31:i] := a[i+31:i] / b[i+31:i]
ENDFOR
dst[MAX:256] := 0
```

Manipulation de l'ordre

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

int main() {
  __m256i a = _mm256_setr_epi32(1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 );
  __m256i b = _mm256_setr_epi32(11,12,13,14,15,16,17,18);
  __m256i s = _mm256_add_epi32(a, b);
  uint32_t t[8];
  _mm256_storeu_si256((__m256i *)t, s);

  for(int i=0; i<8;i++) { std::cout << t[i] << " ";}
  std::cout << std::endl;
  return 0;
}
```

Resultat

$ 12 14 16 18 20 22 24 26

_mm256_setr_epi32 (int e7, int e6, … int e1, int e0)

#include <immintrin.h>
CPUID Flags : AVX

Description

Set packed 32-bit integers in dst with the supplied values in reverse order.

Operation

dst[31:0] := e7
dst[63:32] := e6
dst[95:64] := e5
dst[127:96] := e4
dst[159:128] := e3
dst[191:160] := e2
dst[223:192] := e1
dst[255:224] := e0
dst[MAX:256] := 0

Manipulation de l'ordre - permute

Somme de deux vecteurs

```
#include <immintrin.h>
#include <iostream>
int main() {
  __m256 a = _mm256_set_ps(1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 );
  a = _mm256_permute_ps(a,_MM_SHUFFLE(0, 1, 2, 3));
  a = _mm256_permute2f128_ps(a, a ,0b00000001);
  float t[8];
  _mm256_storeu_ps((float *)t, a);
for(int i=7; i>=0;i--) { std::cout << t[i] << "␣";}
  std::cout << std::endl;
  return 0;
}
```

Resultat

```
$ 8 7 6 5 4 3 2 1
```

_mm256_permute_ps (__m256 a, int imm8)

#include <immintrin.h>
CPUID Flags : AVX

Description

Shuffle single-precision (32-bit) floating-point elements in a within 128-bit lanes using the control in imm8, and store the results in dst.

Operation

```
SELECT4(src, control){
CASE(control[1:0])
0 : tmp[31:0] := src[31:0]
1 : tmp[31:0] := src[63:32]
2 : tmp[31:0] := src[95:64]
3 : tmp[31:0] := src[127:96]
ESAC
RETURN tmp[31:0]
}
```

```
dst[31:0] := SELECT4(a[127:0], imm8[1:0])
dst[63:32] := SELECT4(a[127:0], imm8[3:2])
dst[95:64] := SELECT4(a[127:0], imm8[5:4])
dst[127:96] := SELECT4(a[127:0], imm8[7:6])
dst[159:128] := SELECT4(a[255:128], imm8[1:0])
dst[191:160] := SELECT4(a[255:128], imm8[3:2])
dst[223:192] := SELECT4(a[255:128], imm8[5:4])
dst[255:224] := SELECT4(a[255:128], imm8[7:6])
dst[MAX:256] := 0
```

_mm256_permute2f128_ps (__m256 a, __m256 b, int imm8)

#include <immintrin.h>
CPUID Flags : AVX

Description

Shuffle 128-bits (composed of 4 packed single-precision (32-bit) floating-point elements) selected by imm8 from a and b, and store the results in dst.

Operation

```
SELECT4(src1, src2, control){
CASE(control[1:0])
0 : tmp[127:0] := src1[127:0]
1 : tmp[127:0] := src1[255:128]
2 : tmp[127:0] := src2[127:0]
3 : tmp[127:0] := src2[255:128]
ESAC
IF control[3]
tmp[127:0] := 0
FI
RETURN tmp[127:0]
}
```

```
dst[127:0] := SELECT4(a[255:0], b[255:0], imm8[3:0])
dst[255:128] := SELECT4(a[255:0], b[255:0], imm8[7:4])
dst[MAX:256] := 0
```

Et bien plus ...

- Shuffle
- Opérations masquées
- Additions horizontales
- Recherche de maximum
- Opérations logiques
- ...