

# Opérations Vectorisées

Rémi Forax

# Vocabulaire

## Concurrence

- Exécuter plusieurs calculs en même temps

## Parallèle

- Un calcul est décomposé et exécuté sur plusieurs threads

## **Vectorisé**

- Une opération simple (+, min, etc) est exécutée par un même thread sur plusieurs données adjacentes en mémoire (sur 128 bits, 256 bits, etc)

## Distribué

- Un calcul est décomposé et exécuté sur plusieurs machines

# SIMD

## Single Instruction, Multiple Data

Catégorie de CPUs capable d'exécuter une même opération sur des données multiples

Apparition pour les CPUs grand public en 1996 avec l'extension MMX sur Intel Pentium

3DNow (AMD 1998) et SSE (Intel 1999)

16 registres de 128 bits

AVX (AMD et Intel 2011)

32 registres sur 256 bits

puis AVX2 (2013), puis AVX-512 (2016), 32 registres de 512 bits

Neon (ARM / Apple / Qualcomm / Samsung 2011)

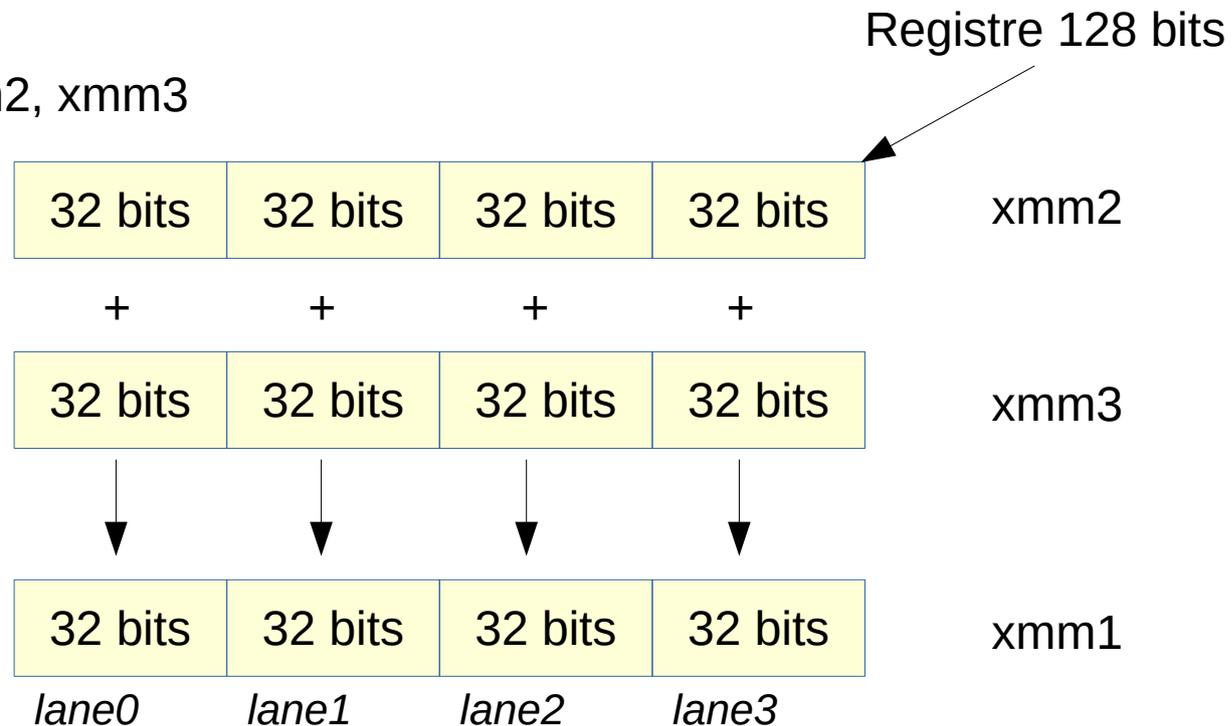
16 registres 128 bits

puis Helium/MVE (2019), support des flottants 16 bits par lignes

# En pratique

Des registres spécialisées (xmm0, ymm0, zmm0 sur Intel) effectue une même opération sur plusieurs colonnes (*lanes*)

PADDW xmm1, xmm2, xmm3



# En pratique (2)

Une instruction indique

- La taille des registres (64 bits, 128 bits, 256 bits)
- L'opération (ADD, SUB, etc)
- La taille d'une ligne (8 bits, 32 bits, 64 bits, 128 bits)

**P**ADD**W** xmm1, xmm2, xmm3

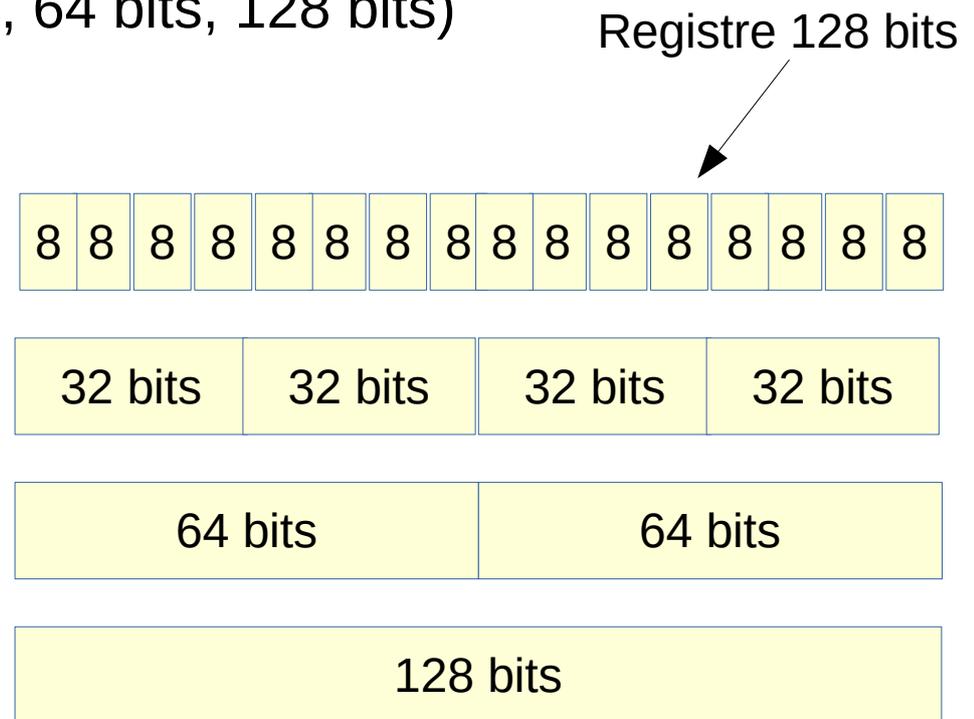
## Préfixe

P pour 128 bits  
VEX VP pour 256 bits  
EVEX VP pour 512 bits

ADD pour l'opération

## Taille d'une ligne

B (byte) = 8 bits  
W (word) = 32bits  
D (double) = 64bits  
Q (quad) = 128bits



# Opération vectorisée et mémoire

Les registres vectorisés ont une taille supérieur aux registres normaux

Plus efficace pour les lectures/écritures

Par ex: 128 bits = 16 octets en 1 lecture/écriture

Rappel: La lecture/écriture en mémoire a une granularité en multiple de 8 bits (1 octet)

# Vecteur et mémoire

Au lieu d'accéder à un tableau octet par octet

```
for(var i = 0; i < 16; i++) { // avec 16 lanes
  a[i] = b[i] + c;
}
```

On passe par des vecteurs intermédiaires

```
var vb = *Vector.fromArray(b, 0); // lit 16 valeurs
var vc = *Vector.broadcast(c);    // on met c dans chaque lane
var va = vb.add(vc);
va.intoArray(a, 0);                // écrit 16 valeurs
```

# Vector API en Java

(jdk.incubator.vector)

# Accès aux Instructions

Même idée que pour CAS, un appel a une méthode Java est transformé en une instruction spécifique au CPU par le JIT

```
IntVector v1 = ...  
IntVector v2 = ...  
v1.add(v2);
```

Les types (IntVector par ex) n'indique pas le nombre de *lanes*

IntVector.add(IntVector) peut être VPADD[B,W,D,Q]

# API `jdk.incubator.vector`

Un vecteur correspond à un registre (immutable) ayant un type et un nombre de lanes

- `Vector`
  - Interface de base
- `[Byte|Short|Int|Long|Float|Double]Vector`
  - Version spécifique
- `VectorSpecies`
  - Un type + shape (taille en bits)  
permet de dériver un type de vecteur + nombre de *lanes*
- `VectorMask`
  - Vecteur de boolean, bit sets
- `VectorShuffle`
  - Vecteur d'index d'indirection

# VectorSpecies<\*>

Représente un type (byte|short|int|long|float|double) et une shape (64|128|256|512 bits)

Création par méthodes statiques

- of(Class<?> elementType, VectorShape shape)  
VectorShape.S\_64\_BIT, S\_128\_BIT, S\_256\_BIT, S\_512\_BIT)
- ofPreferredShape(Class<?> elementType)

Constantes dans [Byte|Short|Int|Long|Float|Double]Vector

- pour IntVector, on a les constantes  
IntVector.SPECIES\_[64|128|256|512] de type VectorSpecies<Integer>

Il a une notion de taille préférée (SPECIES\_PREFERRED)

par ex, certains CPUs supportent les opérations 256 bits en découpant en deux 128 bits

la taille préférée est alors 128 bits

# Exemple

Créer deux vecteurs d'entiers, les additionner et afficher les valeurs de chaque *lane*

```
VectorSpecies<Integer> species =  
    IntVector.SPECIES_PREFERRED;
```

```
IntVector v1 = IntVector.zero(species);  
IntVector v2 = IntVector.broadcast(species, 42);  
IntVector v3 = v1.add(v2);
```

```
for(var i = 0; i < v3.length(); i++) { // length == lane count  
    System.out.println(v3.lane(i));  
}
```

```
// affiche plusieurs 42 (128 bits = 4, 256 bits = 8, etc)
```

# Vector vs Objet

Un vecteur n'**est pas** un objet avec une adresse en mémoire (c'est un *value based object*)

- On le manipule comme un objet, mais il marche comme un int
  - Il correspond à **un registre** du processeur
  - Un vecteur est une valeur non-mutable
    - `v1.add(v2)` ne change pas `v1` !
- Les opérations nécessitant l'adresse en mémoire ne marche pas ou renvoie des résultats fantaisistes
  - Pas de `synchronized` / `wait()` / `notify()`
  - Pas de `System.identityHashCode()`
  - Pas de `v1 == v2`

# VectorMask<\*>

Masque booléen indiquant si une lane doit être prise en compte (true) ou non (false)

## Exemple

```
var species = IntVector.SPECIES_256;  
var v1 = IntVector.zero(species);  
var v2 = IntVector.broadcast(species, 42);
```

```
VectorMask<Integer> mask =  
    VectorMask.fromLong(species, 0b11010); // F, T, F, T, T, F, F, F
```

```
var v3 = v1.blend(v2, mask); // v1 ou v2 suivant le masque
```

```
System.out.println(v3); // affiche [0, 42, 0, 42, 42, 0, 0, 0]
```

# VectorMask<\*> et if

Faire un if dans une boucle qui ne fait que des calculs est super lent

Au lieu de

```
for(var i = 0; i < 8; i++) {  
  if (a[i] < 3) {  
    b[i] = a[i] + c;  
  }  
}
```

On utilise un masque

```
IntVector va = IntVector.fromArray(species, a, 0);  
IntVector vc = IntVector.broadcast(3);  
VectorMask<Integer> mask = va.lt(vt); // '<' renvoie un mask  
IntVector vb = va.add(vc, mask);      // on fait le '+' suivant le mask  
vb.intoArray(b, 0);
```

# Vector<\*>

Interface commune de [Byte|Short|Int|Long|Float|Double]Vector

- Opérations de base
  - length(), lane(int index), withLane(int index, value), addIndex(int scale)
- Opérations spécifiques
  - blend, rearrange, etc
- Opérations par *lane* (*lanewise*)
  - Tests unaire et binaire
  - Applications unaire, binaire et ternaire
  - Conversions

VectorOperators définit chaque opération

# Vector<\*> - Opérations spécifiques

Voir un vecteur comme un autre, par ex 32 bits → 8 bits

`reinterpretShape(Species, int part)`

Intervertir les lanes d'un vecteur

`rearrange(VectorShuffle, VectorMask?)`

`rearrange(VectorShuffle, Vector, VectorMask?)`

Mixer deux vecteurs selon un masque

`blend(Vector, VectorMask)`

Découper les lanes + compléter avec les valeurs d'un vecteur

`slice(int origin, Vector, VectorMask?)`

`unslice(int origin, Vector, int part, VectorMark?)`

# Exemple

## Exemple de rearrange

```
var species = IntVector.SPECIES_128;  
var v1 = IntVector.zero(species);  
for(var i = 0; i < v1.length; i++) {  
    v1 = v1.withLane(i, i * 10);  
}  
System.out.println(v1); // [0, 10, 20, 30]
```

```
VectorShuffle<Integer> shuffle =  
    VectorShuffle.fromValues(species, 0, 2, 1, 3);  
var v2 = v1.rearrange(shuffle);  
System.out.println(v2); // [0, 20, 10, 30]
```

# Exemple (2)

Exemple de rearrange + addIndex(scale)

```
var species = IntVector.SPECIES_128;  
var v1 = IntVector.zero(species).addIndex(10);
```

```
System.out.println(v1); // [0, 10, 20, 30]
```

```
VectorShuffle<Integer> shuffle =  
    VectorShuffle.fromValues(species, 0, 2, 1, 3);  
var v2 = v1.rearrange(shuffle);  
System.out.println(v2); // [0, 20, 10, 30]
```

# VectorOperators

Enum qui définit les opérations que l'on peut faire sur un Vector

- Test unaire et binaire (renvoie un VectorMask)
  - IS\_NAN, IS\_NEGATIVE, etc
  - EQ(==), NE(!=), LE(<=), LT(<), GE(>=), GT(>)
- Application unaire, binaire et ternaire
  - ABS, COS, SIN, etc
  - ADD, SUB, MUL, DIV, AND, OR, etc
  - FMA, BITWISE\_BLEND
- Conversion
  - I2L, I2D, ZERO\_EXTENDED\_B2I, etc

# Vector<\*> - Opérations *Lanewise*

## Opérations par *lane* utilisant les VectorOperators

- Tests unaire et binaire
  - test(Test op, VectorMask?) → VectorMask
  - compare(Comparison op, Vector, VectorMask?) → VectorMask
- Applications unaire, binaire et ternaire
  - lanewise(Unary op, VectorMask?)
  - lanewise(Binary op, Vector, VectorMask?)
  - lanewise(Ternary op, Vector, Vector, VectorMask?)
- Conversions
  - convert(Conversion op, int part)
  - convertShape(Conversion op, Species, int part)

# Exemple

Exemple d'application avec `lanewise()`

```
var species = IntVector.SPECIES_128;
```

```
var v1 = IntVector.fromArray(species,  
    new int[] { 7, 3, 16, 5 },  
    0);
```

```
var v2 = IntVector.broadcast(species, 42);
```

```
var v3 = v1.lanewise(VectorOperators.ADD, v2);
```

```
System.out.println(v3); // [49, 45, 58, 47]
```

# Vector - Reduce en ligne

On peut demander d'effectuer une réduction sur toutes les *lanes* avec une opération associative

- Cette opération est pas définie sur Vector mais sur chaque [Byte|Short|Int|Long|Float|Double]Vector
  - car le type de retour est différent
- Les opérations associatives sont définies dans VectorOperators
  - ADD, MUL, AND, OR, etc

Par exemple sur IntVector,

`IntVector.reduceLane(Associative op, VectorMask?)` → int

# Exemple

## Exemple de reduceLanes()

```
var species = IntVector.SPECIES_128;  
var array = new int[] { 1, 2, 3, 4, 11, 12, 13, 14 };  
  
var v1 = IntVector.fromArray(species, array, 0)  
var v2 = IntVector.fromArray(species, array, 4);  
var v3 = v1.add(v2);  
  
int sum = v3.reduceLanes(VectorOperators.ADD);  
  
System.out.println(sum); // 60
```

# Lire/Ecrire dans un tableau

[Byte|Short|Int|Long|Float|Double]Vector

Méthodes statique de lecture

- `fromArray(Species, array, offset, VectorMask?)`
- `fromByteArray(Species, byte[], offset, ByteOrder)`

Méthodes d'instance d'écriture

- `intoArray(array, offset, VectorMask?)`
- `intoByteArray(byte[], offset, ByteOrder)`

Le masque optionnel permet de lire moins de données (et éviter les IAOOB)

# Exemple

Dupliquer un tableau d'entier

```
private static final VectorSpecies<Integer> =  
    IntVector.SPECIES_PREFERRED;
```

```
static int[] copy(int[] array) {  
    var length = array.length;  
    var newArray = new int[length];  
    for(var i = 0; i < length; i += SPECIES.length()) {  
        var v = IntVector.fromArray(SPECIES, array, i);  
        v.intoArray(newArray, i);  
    }  
    return newArray;  
}
```

Ce code est **FAUX** !

- marche que si array.length est un multiple de SPECIES.length()

# Exemple avec une postLoop

On fini avec une boucle classique

```
static int[] copy(int[] array) {  
    var length = array.length;  
    var newArray = new int[length];  
    var loopBound = length - length % SPECIES.length();  
    var i = 0;  
    for(; i < loopBound; i += SPECIES.length()) {  
        var v = IntVector.fromArray(SPECIES, array, i);  
        v.intoArray(newArray, i);  
    }  
    for(; i < length; i++) { // post loop  
        newArray[i] = array[i];  
    }  
    return newArray;  
}
```

# Exemple avec un masque

On utilise un masque

```
static int[] copy(int[] array) {  
    var length = array.length;  
    var newArray = new int[length];  
    var loopBound = length – length % SPECIES.length();  
    var i = 0;  
    for(; i < loopBound; i += SPECIES.length()) {  
        var v = IntVector.fromArray(SPECIES, array, i);  
        v.intoArray(newArray, i);  
    }  
    var mask = maskAllBefore(length % SPECIES.length());  
    var v = IntVector.fromArray(SPECIES, array, i, mask);  
    v.intoArray(newArray, i, mask);  
    return newArray;  
}
```

# Calculer le masque

Le masque:  $iota < broadcast(index)$

```
static VectorMask maskAllBefore(int index) {  
  var iota = IntVector.zero(SPECIES).addIndex(1);  
  var value = IntVector.broadcast(SPECIES, index);  
  return iota.compare(VectorOperators.LT, value);  
}
```

Par exemple,  
si l'index vaut 3

iota	0	1	2	3
	<	<	<	<
broadcast(3)	3	3	3	3
	=	=	=	=
mask	true	true	true	false

# Exemple avec un masque

En fait, il existe déjà `loopBound()` et `indexInRange()` sur `VectorSpecies`

```
static int[] copy(int[] array) {
    var length = array.length;
    var newArray = new int[length];
    var loopBound = SPECIES.loopBound(length);
    var i = 0;
    for(; i < loopBound; i += SPECIES.length()) {
        var v = IntVector.fromArray(SPECIES, array, I);
        v.intoArray(newArray, i);
    }
    var mask = SPECIES.indexInRange(i, length);
    var v = IntVector.fromArray(SPECIES, array, i, mask);
    v.intoArray(newArray, i, mask);
    return newArray;
}
```

# Performance

L'API permet uniquement de typer le vecteur  
pas d'indiquer sa longueur/shape

IntVector pas IntVector128

donc l'information de longueur/shape doit toujours  
être accessible au JIT

=> mettre SPECIES en constante (static final)

L'implantation actuelle est pas finie

Stocker un vecteur dans un champ ou une case de  
tableau est pas efficace