

# Collections Concurrentes

## LAB 5 - Single instruction, Multiple Data (SIMD)

[https://github.com/MelissaDaCosta/structures\\_concurrentes](https://github.com/MelissaDaCosta/structures_concurrentes)

Mélissa DA COSTA

# Contexte

Nous avons vu la parallélisation avec la méthode **Fork/Join** qui décompose un problème en sous-problème et les résout dans différents threads.

Dans ce lab, nous étudions les **Vector** de l'API `jdk.incubator.vector` du JDK15. Les Vector sont sur une architecture SIMD.

**SIMD** (Single instruction, Multiple Data) :  
« Instruction unique, données multiple »  
Désigne un mode de fonctionnement utilisant le parallélisme. La même instruction est appliquée simultanément à plusieurs données.

# Vector

Un vecteur regroupe un ensemble de valeur d'un même type.

Le nombre de valeurs (**lanes**) dépend du processeur.

La constante **IntVector.SPECIES\_PREFERRED** permet d'obtenir un nombre de lanes préféré.

Généralement, c'est le nombre de lanes max supportable par le processeur.

# Fork/Join VS Vector

Les APIs Fork/Join et Vector ont pour but de faire des calculs en **parallèles**.

La méthode **Fork/Join** répartie le calcul des sous-problèmes dans des cœurs différents.

Les **vector** font les calculs sur un même cœur.  
Le CPU s'occupe de transformer le calcul pour améliorer les opérations.

Avec un **IntVector** et un **SPECIES** de taille 8, on manipule 8 int, donc on a besoin de 8 fois moins d'instructions pour réaliser les calculs.

# Parcours de lane en lane

Si la taille du tableau n'est pas un multiple de SPECIES. Il faut reparcourir le tableau pour veillez à ce qu'il ne reste pas d'éléments non parcouru. C'est la **postLoop**.

```
var i = 0;
var limit = array.length - (array.length % SPECIES.length());
// main loop
// parcours le tableau SPECIES par SPECIES
for (; i < limit; i += SPECIES.length()) {
    ...
}
// post loop
for (; i < array.length; i++) {
    ...
}
```

# Lanewise

Le mode **Lanewise** équivaut à demander à chaque **lane** du vecteur de travailler afin de réaliser un calcul en **parallèle**.

Les opérations **add, sub, mul, etc** sur un vector appliquent la même opération sur toutes les valeurs du vector. Ces opérations appellent la méthode **lanewise**.

```
resultVector = resultVector.lanewise(VectorOperators.ADD, vector2);  
==  
resultVector = resultVector.add(vector2);
```

# reduceLanes

La méthode **reduceLane** permet de prendre les valeurs de toutes les **lanes** d'un vecteur et d'appliquer la même opération.

*Exemple pour calculer la somme d'un tableau :*

```
public static int sumLanewise(int array[]) {
    var i = 0;
    var limit = array.length - (array.length % SPECIES.length());
    // main loop
    var resultVector = IntVector.zero(SPECIES);
    for (; i < limit; i += SPECIES.length()) {
        IntVector vector2 = IntVector.fromArray(SPECIES, array, i);
        // opération add en lanewise;
        resultVector = resultVector.lanewise(VectorOperators.ADD, vector2);
    }
    var sum = resultVector.reduceLanes(VectorOperators.ADD);
    // post loop
    for (; i < array.length; i++) {
        sum += array[i];
    }
    return sum;
}
```

# differenceLaneWise

Comme l'opération de soustraction n'est pas associative  $(x-y)-z \neq x-(y-z)$ , ex :  $(1-2)-3 = 4 \neq 1-(2-3) = 2$

On doit réaliser la soustraction à la main :

```
var i = 0;
var limit = array.length - (array.length % SPECIES.length());
// main loop
for (; i < limit; i += SPECIES.length()) {
    IntVector vector2 = IntVector.fromArray(SPECIES, array, i);
    resultVector = resultVector.sub(vector2);
}

var sub = 0;
for (; i < array.length; i++) { // post loop
    sub += array[i];
}
sub -= resultVector.reduceLanes(VectorOperators.ADD);
return -sub;
```



# Performances

Voici les performances des méthodes codées (exécutées avec Travis CI) :

Benchmark	Mode	Cnt	Score	Error	Units
VectorizedBenchMark.minmax	avgt	5	1733.265 ±	46.316	us/op
VectorizedBenchMark.sum_lane_wise	avgt	5	913.838 ±	9.483	us/op
VectorizedBenchMark.sum_loop	avgt	5	300.254 ±	1.445	us/op
VectorizedBenchMark.sum_reduce_lane	avgt	5	252.224 ±	37.672	us/op

The command `./travis_script.sh` exited with 0.

En théorie, la méthode *sum\_lane\_wise* est censé être plus efficace que *sum\_loop* et *sum\_reduce\_lane*. La méthode *minmax* est la moins performante. Peut-être car les opération MIN et MAX sur les vecteurs demande le plus de ressources.