

## Un point sur le codage efficace en SIMD

Lionel Lacassagne

### 1 Introduction

L'écriture d'un algorithme en SIMD que ce soit en Altivec, en SSE, SSE2 ou SPE (pour le processeur Cell) est bien plus complexe que l'équivalent en scalaire. Les types sont différents, les *intrinsics* (l'appel d'une routine assembleur depuis le C) sont différents et il y a bien plus d'écueils lors du codage. Il est donc nécessaire de développer ou d'utiliser des outils pour simplifier les phases de codage et de *debug*. Ces outils, bibliothèques de fonctions, ces conventions et ces types sont présentés dans les paragraphes suivants.

### 2 bibliothèque, conventions, notations et *debug*

#### 2.1 Les bibliothèques NRC et vNRC

Deux bibliothèques sont fournies :

- NRC qui est une extension de la librairie *Numerical Recipes in C* (voir site web [www.nr.com](http://www.nr.com)), et qui contient 2 fichiers :
  - `def.h` pour la définitions des types scalaires
  - `vnrcutil` pour les allocations et desallocations mémoire, pour les entrées/sorties (affichage/lecture/écriture de vecteurs -tableaux 1D- et de matrices -tableaux 2D) ainsi que pour quelques calculs arithmétiques et initialisations des structures mémoires.
- vNRC qui est la version SIMD SSE de la librairie vNRC. On y trouve donc les fichiers `vdef.h` et `vnrcutil`.

#### 2.2 Les types utilisateurs

Ces bibliothèques spécifient types utilisateurs, afin d'être indépendant de la plateforme utilisée. Si en Altivec il y a un équivalent SIMD à chaque type scalaire, il n'existe en SSE2+ que deux types de registres 128 bits :

- le type entier : `_m128i` qui peut être découpé en blocs de 8, 16 ou 32 bits entiers,
- le type flottant : `_m128` qui peut être découpé en blocs 32 bits flottants.

Afin d'avoir des variables plus fortement typées, les types usuels scalaires sont redéfinis en SIMD, comme c'est le cas en Altivec ou en SPE sur le Cell. La table 1 liste l'ensemble des types utilisateurs et la table 2 indique les préfixes à utiliser dans les fonctions. Ainsi pour

type	scalaire		SIMD	
entier 8 bits	non signé	signé	non signé	signé
entier 8 bits	<code>uint8</code>	<code>sint8</code>	<code>vuint8</code>	<code>vsint8</code>
entier 16 bits	<code>uint16</code>	<code>sint16</code>	<code>vuint16</code>	<code>vsint16</code>
entier 32 bits	<code>uint32</code>	<code>sint32</code>	<code>vuint32</code>	<code>vsint32</code>
flottants 32 bits	<code>float32</code>		<code>vfloat32</code>	

TAB. 1 – type utilisateurs, scalaires et vectoriels

#### 2.3 Fonction de *debug* et de mise au point

Pour le *debug*, il existe une fonction d'affichage par type T et par dimension d'objets.

- tableau 2D : `display_Tmatrix` par exemple `display_vui8matrix` pour le type `vuint8`,

type	scalaire		SIMD	
entier 8 bits	non signé	signé	non signé	signé
entier 8 bits	<code>ui8</code>	<code>si8</code>	<code>vui8</code>	<code>vsi8</code>
entier 16 bits	<code>ui16</code>	<code>si16</code>	<code>vu16</code>	<code>vsi16</code>
entier 32 bits	<code>ui32</code>	<code>si32</code>	<code>vui32</code>	<code>vsi32</code>
flottants 32 bits	<code>f32</code>		<code>vf32</code>	

TAB. 2 – prefixes utilisés pour typer les fonctions

- tableau 1D : `display_Tvector` par exemple `display_vf32vector` pour le type `vuint8`
- élément : le nom complet du type est utilisé par exemple pour afficher `display_vsint16` pour afficher un `vsint16`

De même il existe des fonctions pour initialiser les éléments les tableaux 1D et les tableaux 2D. Le principe est de partir d'une valeur de départ `x0` et pour chaque colonne, d'ajouter un pas `xstep`. Dans le cas des tableaux 2D, il y a en plus un pas vertical lorsqu'on change de ligne : `ystep`.

## 2.4 Fonction de *wrapping*

Afin d'aider à la mise au point de code, le modèle mémoire suivant a été adopté :

- Allocation d'une matrice en vectoriel, avec pointeurs 2D,
- wrapping scalaire de la matrice avec pointeurs 2D

Ainsi une zone allouée en vectoriel peut être lue et vérifiée en scalaire, ce qui améliore encore le *debug*. De plus d'un point de vue efficacité, c'est optimal puisque c'est la même zone qui est utilisée. Enfin cela permet de combiner des traitements scalaires et vectoriels sur une même zone mémoire : un algorithme est codé en vectoriel s'il se prête aux contraintes du codage SIMD, ou en scalaire dans le cas contraire. Avec ce double modèle d'accès mémoire, inutile de faire des copies lorsqu'alternent opérateurs scalaires et opérateurs SIMD.

## 2.5 Gestion des bords

Un dernier point rendu possible via le format NRC est une gestion simple des bords : il est non seulement possible d'allouer de manière transparentes des bords à l'image, mais en plus, il est possible de gérer des indices négatifs. L'arithmétique des pointeurs est caché au sein des fonctions d'allocation et de wrapping et les conversions d'indices se font via deux fonctions :

- `s2v` qui converti les indices scalaires (éventuellement) négatif en indices vectoriel, en fonction du cardinal du type scalaire utilisé,
- `v2m` qui calcule les indices scalaires à partir des indices vectoriels, pour que les wrappings scalaires soient compatibles avec la tailles des types vectoriels et avec les contraintes d'alignement.

Soient

- .  $h$  et  $w$  la hauteur et la largeur d'une image
- .  $si_0, si_1, sj_0, sj_1$  les indices scalaires
- .  $vi_0, vi_1, vj_0, vj_1$  les indices vectoriels
- .  $mi_0, mi_1, mj_0, mj_1$  les indices scalaires mémoires :  $mi_0 \leq si_0, mi_1 \geq si_1$
- .  $card$  le cardinal de l'élément vectoriel ( $card = 16$  en 8 bits, 8 en 16 bits et 4 en 32 bits)

Les indices  $m$  correspondent à une image vectorielle  $v$  avec (ou sans) bord(s) dans laquelle on plonge une image scalaire  $s$  non alignée et sans bord.

# 3 Architecture mémoire architecture logicielle

## 3.1 Architecture mémoire utilisée

Les processeurs Intel et AMD sont de type *little endian* par opposition aux processeurs PowerPC qui sont de type *big endian*. Il en résulte que lors d'un chargement mémoire (figures 1, 2 et 3), les blocs de données composant le registre vectoriel (appelé par abus de langage *vecteur*) sont inversé. Ainsi les macros utilisées

pour calculer des vecteurs *non alignés* réalisent des décalages de bits dans le sens opposé de celui indiqué par le nom de la macro.

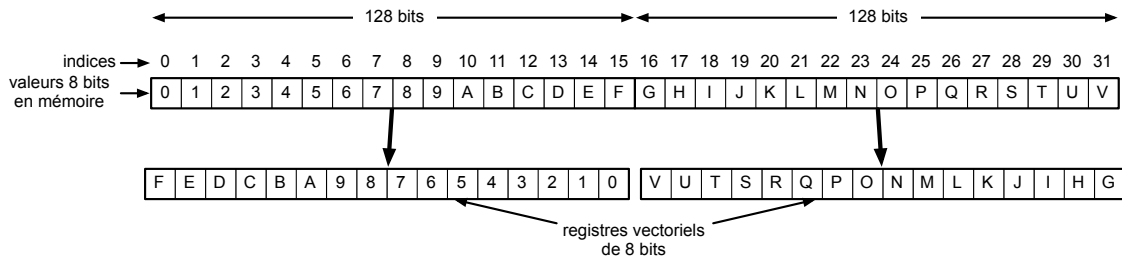


FIG. 1 – chargement de deux vecteurs 128 bits contenant des 8 bits

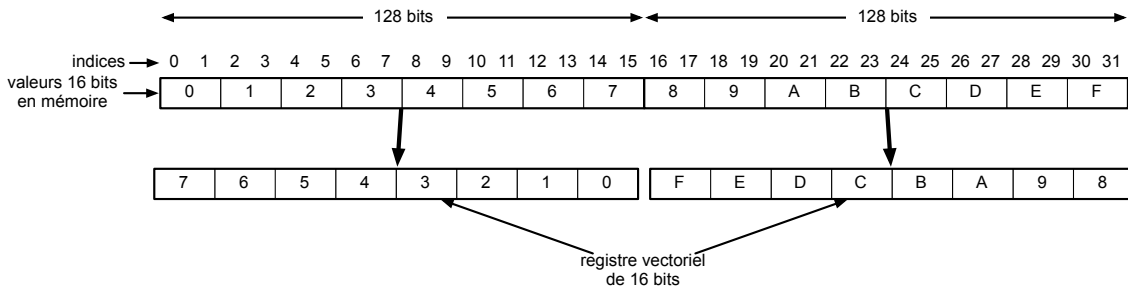


FIG. 2 – chargement de deux vecteurs 128 bits contenant des 16 bits

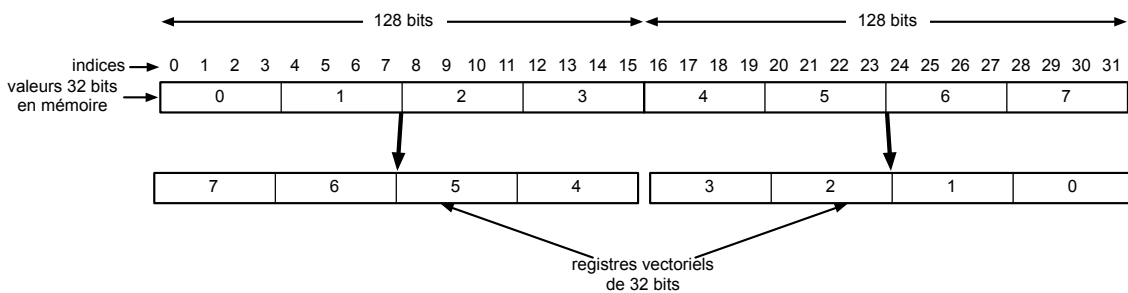


FIG. 3 – chargement de deux vecteurs 128 bits contenant des 32 bits

## 3.2 Architecture logicielle utilisée

L'architecture logicielle mise en oeuvre se base sur les formats de données et sur les types de la librairie *NRC* et son extension *SIMD vNRC*.

Afin de réaliser plus simplement un certain nombre d'opérations courantes, des macros sont définies pour calculer des vecteurs non alignés et réaliser des conversions de types

### 3.2.1 Macros `vec_left` et `vec_right`

La première série de macros manipulent des entiers et servent à calculer, à partir de deux vecteurs `v0` et `v1` soit un vecteur non aligné à gauche (macro `vec_left`) soit un vecteur non aligné à droite (macro

`vec_right`). Le chiffre terminant le nom de la macro indique le nombre d'octets de décalage et non le nombre de blocs de décalage car le décalage en SSE tout comme en Altivec est indiqué en octets et non en blocs.

Ainsi `vec_left1` sert à calculer un vecteur non aligné à gauche d'un octet, alors que la macro `vec_left2` décale un vecteur de 2 octets, c'est à dire soit un décalage d'un élément 16 bits, soit de deux éléments 8 bits. Ces macros se trouvent dans le fichier `vdef.h`

La seconde série de macros manipulent des flottants (`float32`). Dans le cas de SSE, le chiffre terminant le nom de la macro indique le nombre de blocs de `float32` à décaler, car cette macro utilise l'instruction SSE `shuffle` qui code les mélanges en terme de blocs et non d'octets. Ces macros se trouvent dans le fichier `vdef.h`. Ces macros ont été "instanciées" pour tous les types vectoriels entiers et flottants. On a aussi :

type	décalage à gauche		décalage à droite	
	1 bloc	2 blocs	1 bloc	2 blocs
entier 8 bits	<code>vec_i8left1</code>	<code>vec_i8left2</code>	<code>vec_i8right1</code>	<code>vec_i8right2</code>
entier 16 bits	<code>vec_i16left1</code>	<code>vec_i16left2</code>	<code>vec_i16right1</code>	<code>vec_i16right2</code>
entier 32 bits	<code>vec_i32left1</code>	<code>vec_i32left2</code>	<code>vec_i32right1</code>	<code>vec_i32right2</code>
flottants 32 bits	<code>vec_f32left1</code>	<code>vec_f32left2</code>	<code>vec_f32right1</code>	<code>vec_f32right2</code>

TAB. 3 – macros typées pour décalage à gauche et à droite

### 3.2.2 Macros de conversion

Afin d'avoir des macros au nom plus explicites que les intrinsics fournis en Altivec et SSE, les macros suivantes sont définies :

src → dst	macro
<code>vuint8→vuint16</code>	<code>CONVERT_8_16(x,y0,y1)</code>
<code>vuint16→v?int32</code>	<code>CONVERT_16_32(x,y0,y1)</code>
<code>32→16</code>	<code>y=vec_i32pack(x0,x1)</code>
<code>16→8</code>	<code>y=vec_i16pack(x0,x1)</code>

TAB. 4 – macros de conversion de type

Attention, ces macros utilisent un argument caché afin d'être plus courtes : il est nécessaire que `zero8` et `zero16` soient définis et initialisés à zéro.

### 3.3 One more thing...

Les notations pointeurs `*ptr++` ou encore `*(ptr+i*N+j)` et `ptr[i*N+j]` pour adresser la case  $(i, j)$  d'un tableau sont obsolètes. Non seulement ces notations sont moins lisibles, mais en plus elles empêchent les compilateurs optimisants de générer du code efficace. Au 21<sup>ème</sup> siècle, il n'y a plus qu'une seule notation : la notation tableau. Toutes les autres datent du siècle dernier !