Théo Germain
Thomas Bouix

EISE-5

Calcul Haute Performance

-

Rapport final

Détection de mouvement sur processeurs SIMD et sur GPU

M. Lacassagne

2020-2021

M. Meunier

Sommaire

Introduction

Ce projet, réalisé par Théo Germain et Thomas Bouix, a lieu dans le cadre du cours de “Calcul Haute Performance” de la classe EISE-5 à Polytech Sorbonne. Ce cours est encadré par M.Lacassagne et M.Meunier, chercheurs au Laboratoire d’Informatique de Paris 6 (LIP6).

Le but de ce projet est triple : implémenter une chaîne de traitement représentative d’applications en traitement d’images pour les systèmes embarqués, réaliser un ensemble d’optimisations afin d’accélérer l’exécution de cette chaîne de traitement, et enfin développer une méthodologie d’analyse des résultats d’un point de vue qualitatif et quantitatif.

Nous allons donc dans un premier temps discuter de la chaîne de traitement en elle-même, puis des optimisations réalisées, des méthodes de tests, et enfin des résultats de ces optimisations.

1. Code scalaire mono-thread

On présente ici brièvement les différentes parties de la chaîne de traitement, implémentées dans un premier temps en code scalaire mono-thread.

A) Détection de mouvement

La première partie de la chaîne de traitement consiste en la détection de mouvement. Le but est “d’allumer” les pixels d’une image si on pense qu’ils sont en mouvement par rapport à l’image précédente. On teste pour cela deux algorithmes de détection différents, l’algorithme “Frame-Difference” et l'algorithme “Sigma-Delta”.

1. Algorithme Frame-Difference

L’algorithme “Frame-Difference” est une détection naïve du mouvement. On se contente ici de faire la différence des niveaux de gris entre deux images consécutives, et si cette différence est au-delà d’un certain seuil theta, le pixel “résultat” sera allumé. On utilise ici l’encodage 0-255 afin de pouvoir visualiser les images résultats avec des logiciels visionneurs d’image.

```
uint8** routine_FrameDifference(uint8** I_t, uint8** I_t_moins_1,
int nrl, int nrh, int ncl, int nch, int theta){

  uint8** E_t = ui8matrix(nrl, nrh, ncl, nch);

  for(int i = nrl; i <= nrh; i++){
    for(int j = ncl; j <= nch; j++){
      if(abs(I_t[i][j] - I_t_moins_1[i][j]) < theta)
        E_t[i][j] = (uint8)0;
      else
        E_t[i][j] = (uint8)255;
    }
  }
  return E_t;
}
```

2. Algorithme Sigma-Delta

L'algorithme "Sigma-Delta" fait lui l'hypothèse qu'il peut y avoir du bruit de l'image, et essaie donc d'en faire abstraction en utilisant la moyenne et la variance de chaque pixel sur l'ensemble des images fournies. Un pixel sera alors considéré comme étant en mouvement si la différence qu'il y a entre sa valeur et "l'image de fond" (donc la moyenne) est N fois supérieure à l'écart-type.

On implémente ce traitement en deux temps : une initialisation **SigmaDelta_step0** et une fonction courante **SigmaDelta_1step** (voir code source).

B) Morphologie mathématiques

La seconde partie de la chaîne de traitement consiste en l'implémentation de filtres non-linéaires. Ces filtres vont permettre de "lisser" les images résultats afin d'éliminer le bruit de fond.

1. Dilatation / Érosion

Les opérations de dilatation et d'érosion réalisent respectivement des OR et AND logiques dans un voisinage 3x3 ou 5x5 autour d'un pixel. On décide d'ajouter du padding aux images avant de les passer dans ces fonctions, c'est-à-dire de recopier les bords sur de nouvelles lignes et colonnes autour des images, cela afin de calculer les pixels proches des bordures.

2. Ouverture / Fermeture

Les opérations d'ouverture et de fermeture consiste en la composition des opérations élémentaires d'érosion et de dilatation vues plus haut. Plus précisément, l'ouverture est une dilatation d'une érosion, et la fermeture est une érosion d'une dilatation. Nous verrons plus tard qu'il est possible de fusionner ces opérations élémentaire plutôt que de les enchaîner naïvement, afin d'accélérer le traitement des images. Ces opérations complexes permettent alors de filtrer nos images afin d'en supprimer le bruit.

3. Chaîne de traitement complète

Enfin, la chaîne de traitement complète consiste en la séquence : érosion - dilatation - dilatation - érosion. On l'implémentera ici avec plusieurs optimisations différentes pour voir lesquelles sont les plus efficaces.

C) Exemple de traitement

On montre sur la page suivante un exemple des traitements de mouvement et de morphologie :

Image originale (car_3110.pgm)

Image après algorithme Sigma-Delta

Image après traitement morphologique complet

2. Optimisations SIMD

La première série d'optimisations que nous avons ajouté se base sur la technologie SIMD. Cette technologie repose sur l'utilisation de registres CPU spéciaux possédant des tailles supérieures aux registres classiques (souvent 2 à 4 fois supérieures). Ces registres SIMD permettent alors de réaliser des instructions parallèles sur leurs données, ce qui augmente considérablement la vitesse d'exécution.

Possédant des machines Intel, nous utiliserons ici les jeux d'instructions SSE. Ces instructions assembleurs sont accessibles par un code source C via les API d'Intel, documentées au lien suivant : https://software.intel.com/sites/landingpage/IntrinsicsGuide/

A) Routines SIMD

On utilise dans ce projet plusieurs routines permettant de réaliser des opérations élémentaires sur des variables SIMD. On détail donc ici l'implémentation de ces différentes routines.

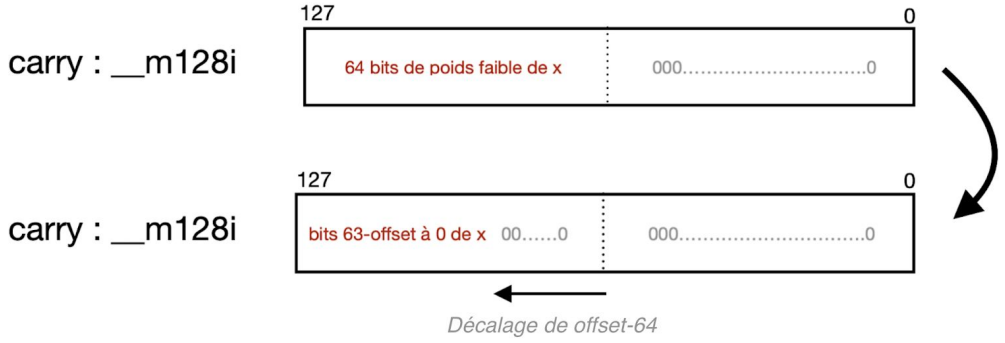
```
__m128i _mm_bitshift_left(__m128i x, unsigned char offset) {
	__m128i carry = _mm_bslli_si128(x, 8);		// sll de 8 octets
	if(offset >= 64)
		return _mm_slli_epi64(carry, offset-64);
	carry = _mm_srli_epi64(carry, 64-offset);
	x = _mm_slli_epi64(x, offset);
	return _mm_or_si128(x, carry);
}
```

Cette fonction sert au décalage bit à bit vers la gauche (on a le même principe pour les décalages à droite). On ne peut en effet pas utiliser les opérateurs "<<" et ">>", car ces opérateurs effectuent un décalage bit à bit sur les 64 bits de poids faible et sur les 64 bits de poids fort du __m128i de manière indépendante. C'est-à-dire que le bit 63 ne sera pas propagé au bit 64.
Le fonctionnement de cette routine s'inspire d'une solution décrite sur Stack Overflow. Elle prend en paramètre une donnée de type __m128i ainsi qu'un offset correspondant au nombre de bits à décaler.

Pour effectuer ce décalage, la première étape consiste à effectuer un décalage de 8 octets vers la gauche afin que les 64 bits de poids faible de *'x'* deviennent les 64 bits de poid fort de *carry* :


```
__m128i carry = _mm_bslli_si128(x, 8)
```

Ensuite, dans le cas où le décalage est supérieur ou égal à 64 bits, il ne reste qu'à effectuer un décalage de **offset - 64** sur les bits de poids fort de *carry* pour obtenir le résultat final attendu. C'est ce que fait *_mm_slli_epi64(carry, offset-64)* dans le code ci-dessus.

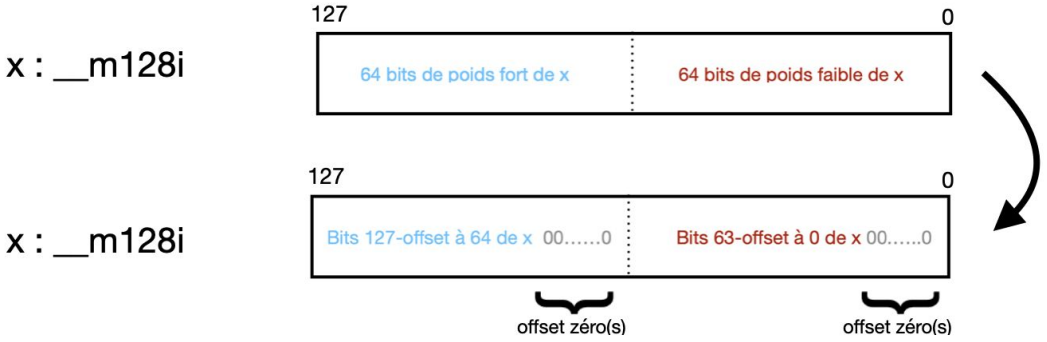



```
return _mm_slli_epi64(carry, offset-64);
```

Les fonctions *_mm_slli_epi64* et *_mm_srli_epi64* effectuent un décalege d'un nombre donné de bits à la fois sur les 64 bits de poids fort et de poids faible du __m128i passé en paramètre. Cependant ces fonctions ne propagent pas de bit entre les 64 premiers bits et les 64 suivants.

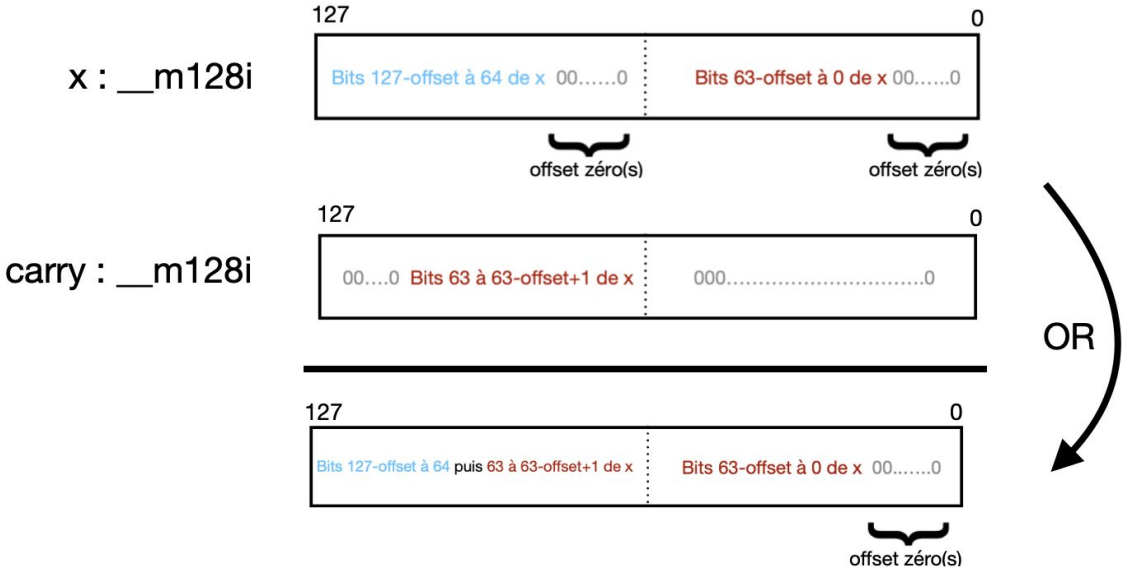
Dans le cas où le décalage est inférieur à 64 bits, on effectue d'abord un décalage à droite de **64 - offset** bits sur *carry* ainsi qu'un décalage à gauche de offset sur *x* :


```
carry = _mm_srli_epi64(carry, 64-offset);
```




```
x = _mm_slli_epi64(x, offset);
```

Enfin, pour obtenir le résultat attendu, il faut effectuer un OR logique bit à bit entre *x* et *carry*. On obtiendra alors le résultat suivant :




```
return _mm_or_si128(x, carry);
```

On obtient donc bien le décalage de *offset* bit(s) souhaité.
Un version analogue de cette routine (avec des décalages inversés) permet de réaliser un décalage à droite de *offset* bit(s).

```
vbits** convert_to_binary(uint8** img, size_t height, size_t width);
```

Cette fonction permet de convertir une image dont les pixels sont codés sur 8 bits en une matrice de vbits (__m128i) où chaque pixel sera codé sur 1 seul bit. Cette conversion n'est pertinente que pour les images dont les pixels n'ont que deux états possibles (typiquement les images en sortie des algorithmes FD et SD). Cette conversion permet de représenter 128 pixels dans un seul __m128i au lieu de 16 pixels si les valeurs de ces derniers sont codées sur 8 bits.

Le fonctionnement de cette fonction est le suivant : dans un premier temps, on calcul le nombre de vbits nécessaires pour représenter une ligne de l'image passée en paramètre. Dans le cas où le nombre de colonnes de l'image n'est pas un multiple de 128, nous aurons des bits non utilisés dans la dernière colonne de vbits de notre matrice (vbits**) finale.
On alloue ensuite un tableau de uint64_t qui servira à construire les __m128i. Ce tableau contient un nombre de cases égal à 2 fois le nombre de vbits nécessaire pour représenter une ligne de l'image calculé précédemment. Nous parcourons ensuite l'image d'entrée ligne par ligne et remplissons le tableau de uint64_t de la manière suivante. Pour chaque pixel de l'image, on effectue un décalage vers la gauche (<<) sur le uint64_t à la case *num_colonne/64*, puis, si le pixel est à son état haut (=255) on incrément de 1 la valeur du uint64_t. De cette manière, les bits de poids fort des uint64_t représenteront les pixels les plus à gauche de l'image, et les bits de poids faibles ceux les plus à droite. A la fin de chaque ligne, on utilise les uint64_t deux à deux pour générer des __m128i et nous stockons les valeurs obtenues dans la matrice de vbits.
La matrice de vbits retournée par cette fonction représente donc une image, où chaque case de la matrice code 128 pixels, et les pixels les plus à gauche sont codés par les bits de poids fort.

```
uint8** convert_from_binary(vbits** binary_img, size_t height, size_t width);
```

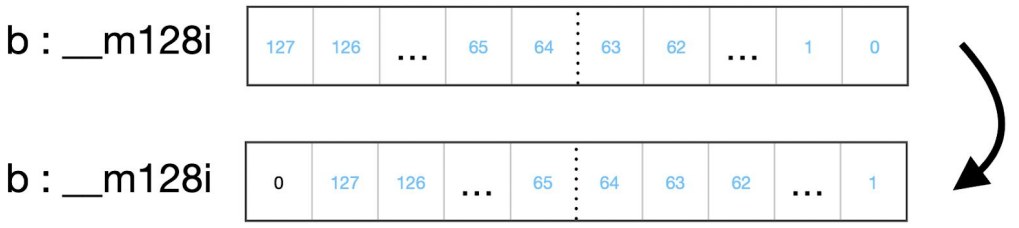
Cette fonction permet d'effectuer la conversion inverse. Cette fonction va allouer une matrice de uint8 (uint8**), et remplir chaque case avec les valeurs des pixels (0 et 255). Pour cela, on parcourt la matrice de vbits (vbits**) et pour chaque case on convertit le __m128i en deux uint64_t (dans un tableau uint64_t[2]). On va ensuite appliquer un masque sur le bit de poids faible de chacune des uint64_t (uint64_t & 1). En fonction de la valeur, on va remplir les cases des pixels correspondants dans la matrice de uint8. On effectue ensuite un décalage à droite sur les uint64_t (>>), puis on recommence pour les 64 pixels codés par chaque uint64_t.

B) Macros SIMD

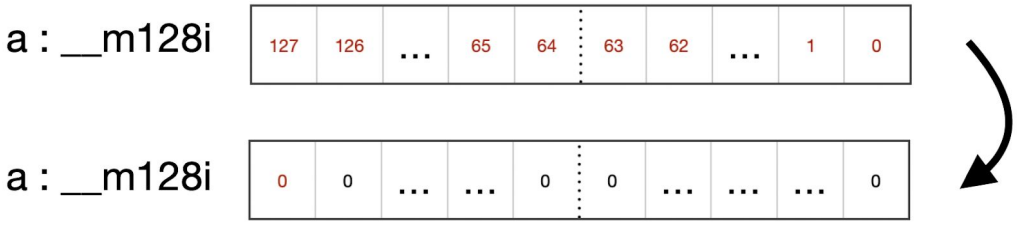
Nous utilisons dans notre projet de nombreuses macros qui permettent d'augmenter la lisibilité du code. Ces macros servent à la fois de wrapper pour les fonctions intrinsèques d'Intel, qui sont souvent lourdes à lire et écrire, mais aussi pour les routines SIMD décrites plus haut. On détail donc ici l'implémentation de ces différentes macros, définies dans le fichier header "simd_macro.h".

```
#define vec_left1_bin(a,b)
(_mm_or_si128(_mm_bitshift_left(a, 127), _mm_bitshift_right(b, 1)))
```

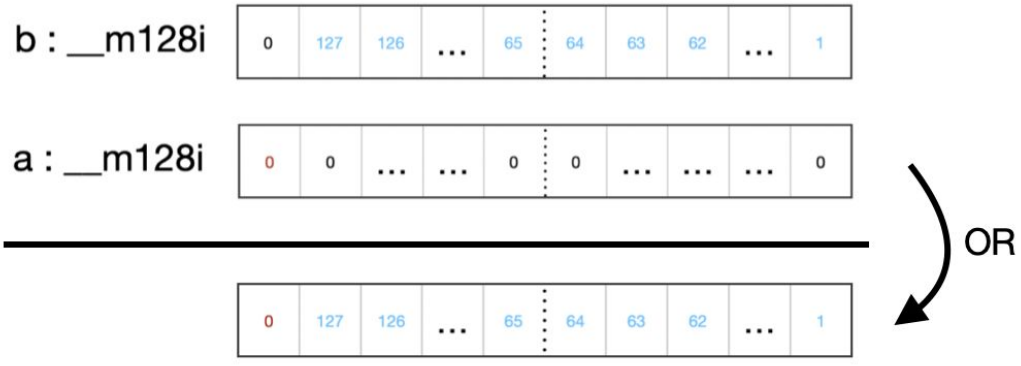
Cette macro effectue une OR logique entre le __m128i b décalé de un bit vers la droite à l'aide de la fonction *_mm_bitshift_right* détaillée ci-dessus, et le __m128i a ayant subi un décalage de 127 bits vers la gauche.



Décalage à droite de 1 bit sur b



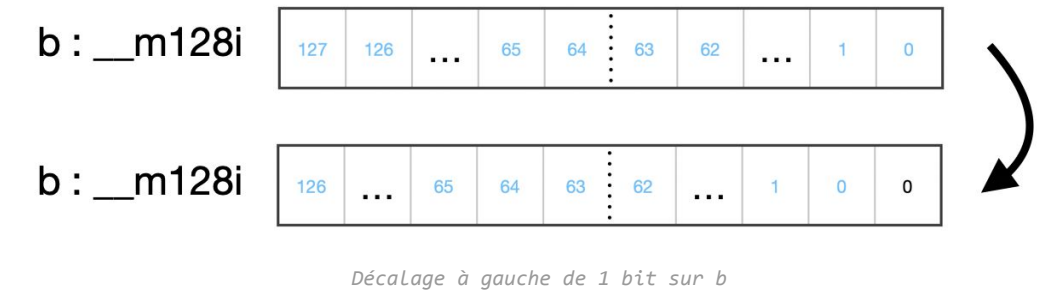
Décalage à gauche de 127 bits sur a



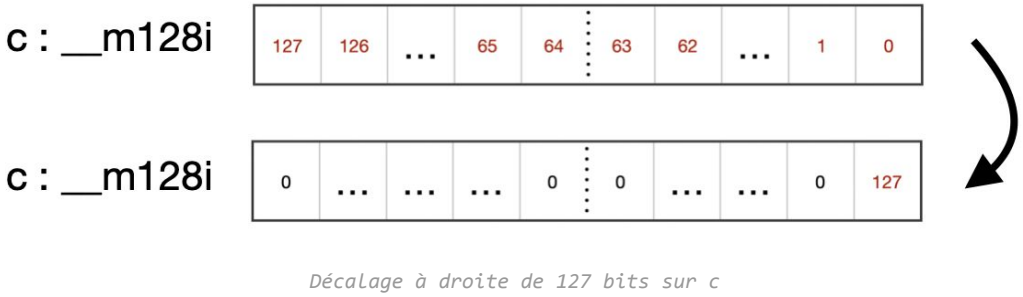
Résultat de vec_left1_bin

```
#define vec_right1_bin(b,c)
(_mm_or_si128(_mm_bitshift_right(c, 127), _mm_bitshift_left(b, 1)))
```

Le principe de cette macro est le même que pour la précédente, sauf que b sera décalé de 1 bit vers la gauche et c de 127 bits vers la droite.



Décalage à gauche de 1 bit sur b



Décalage à droite de 127 bits sur c

Résultat de vec_right1_bin

```
#define vec_right1_bin_unused_col(b,c,nb_unused_col)
(_mm_or_si128(_mm_bitshift_right(c, 127-nb_unused_col),
_mm_bitshift_left(b, 1)))
```

De la même manière, une macro permet de gérer le dernier paquet de colonnes des images, dans le cas où le nombre de total de colonne n'est pas un multiple de 128. Dans ce cas, on souhaite que le bit 0 du vbits c se retrouve au niveau de la dernière colonne utilisée dans b.

Pour cela on effectue un décalage de 127 - nb_unused_col plutôt que le décalage de 127 bits précédent.

```
#define vABS_DIFF(v0,v1) (_mm_or_si128(_mm_subs_epu8(v0,v1),
_mm_subs_epu8(v1,v0)))
```

Cette macro est utilisée pour le calcul de la valeur absolue de la différence. Chaque pixel à une valeur non signée comprise entre 0 et 255. On utilise une fonction SIMD qui va faire la différence entre des entiers 8 bits non signés. Le vecteur résultat de cette opération contiendra 0 si le résultat est négatif, et la valeur de la différence sinon. Le principe de la macro est donc d'effectuer **v0-v1** et **v1-v0**, et de faire un OR logique sur les deux vecteurs résultats pour "réunir" les résultats dans un seul vecteur. De cette manière toutes les différences sont positives ou nulles et le vecteurs résultant correspond bien à **|v0-v1|**.

```
#define vCMP_THRESHOLD(vec, threshold)
(_mm_cmpgt_epi8(_mm_sub_epi8(vec, init_vuint8(128)),
init_vsint8(threshold-128)))
```

Cette macro permet de comparer les 16 valeurs d'un vuint8 avec une seuil (threshold). Comme nos pixels sont des entiers non signés et que les fonctions SIMD de comparaison opèrent sur des entiers signés, on procède comme suit. Dans un premier temps, on soustrait 128 à notre vuint8 pour changer l'intervalle [0:255] → [-128:127]. On initialise ensuite un vuint8 contenant la valeur du seuil à laquelle on retranche 128 également. Enfin on utilise une fonction SSE de comparaison pour déterminer si les entiers 8 bits du vecteur sont supérieurs ou non au seuil.

```
#define vec_cmpgt_vui8(x0,x1) (_mm_cmpgt_epi8(_mm_sub_epi8(x0,
init_vuint8(128)), _mm_sub_epi8(x1, init_vuint8(128))))
```

Cette macro permet d'effectuer une comparaison (greater than) entre deux vecteurs d'entiers non signés sur 8 bits. Pour cela, on soustrait 128 aux uint8 et on utilise une fonction SSE qui permet d'effectuer la comparaison entre des entiers signés sur 8 bits.

```
#define vec_cmpeq_vui8(x0,x1) (_mm_cmpeq_epi8(x0, x1))
```

Cette macro effectue simplement une comparaison d'égalité entre les uint8 des deux vecteurs de uint8.

```
#define vec_cmp_pixels_SD(t_moins_1,t)
(_mm_blendv_epi8(_mm_blendv_epi8(vec_add2(t_moins_1,init_vuint8(1
)), vec_sub2(t_moins_1,init_vuint8(1)), vec_cmpgt_vui8(t_moins_1,
t)), t_moins_1,vec_cmpeq_vui8(t_moins_1,t)))
```

Cette macro est définie pour les comparaisons nécessaires aux calculs de V(t) et M(t) dans l'algorithme Sigma-Delta. Elle permet d'effectuer en SIMD le pseudo code suivant :

```
if( X(t-1) < Y(t) )            X(t) = X(t-1) + 1;
else if( X(t-1) > Y(t) )       X(t) = X(t-1) - 1;
else                           X(t) = X(t-1)
```

Dans un premier temps, on utilise les macros *vec_add2* et *vec_sub2* pour ajouter et soustraire 1 à chaque uint8 contenu dans le vuint8 *t_moins_1 (*argument de la macro). On compare ensuite les valeurs non signées des uint8 contenus dans *t_moins_1* et *t* en utilisant la macro *vec_cmpgt_vui8* décrite ci-dessus. La fonction SSE4.1 *_mm_blendv_epi8* nous permet de faire un *vec_sel* SIMD. C'est-à-dire qu'en fonction de la comparaison de *t_moins_1* et *t,* le vecteur résultat va prendre *t_moins_1* +/- 1. En effet, si le uint8 de *t_moins_1* est strictement supérieur à l'élément uint8 de *t,* la macro de comparaison sortira 0xFF sur les bits correspondant dans le vecteur de résultat et 0 sinon. Notre *vec_sel* va ensuite lire cette valeur et attribuer *t_moins_1* + 1 au vecteur résultat si la comparaison a "retourné" 0 et *t_moins_1* - 1 sinon (0xFF).
À ce stade nous ne traitons que les deux premières lignes du pseudo code ci-dessus. Pour traiter le cas d'égalité, nous réutilisons la fonction *_mm_blendv_epi8* en utilisant cette fois-ci comme masque, la comparaison d'égalité entre *t_moins_1* et *t* grâce à la macro *vec_cmpeq_vui8*. Dans le cas d'une égalité entre les deux uint8, nous sélectionnons la valeur de *t_moins_1* et dans le cas contraire, on récupère la valeur du *vec_sel* précédent.

Pour des raisons de compatibilité, il est possible de remplacer le vec_sel effectué ci-dessus avec une fonction SSE4.1 en le réalisant avec des fonctions SSE2 de la manière suivante :

```
#define vec_sel(a,b,c) (_mm_or_si128(_mm_and_si128(c,b),
_mm_andnot_si128(c,a))
```

```
#define vec_MAX(x0,x1) (_mm_blendv_epi8(x1, x0,
vec_cmpgt_vui8(x0,x1)))
```

Cette macro permet de comparer les uint8 des vecteurs de uint8 qui lui sont passés, et de construire un vecteur contenant les valeurs maximales. Pour cela, elle compare dans un premier temps les valeurs de vuint8 à l'aide de la macro *vec_cmpgt_vui8* décrite ci-dessus. Cette-dernière construit un vecteur contenant la valeur 0xFF si **x0 > x1** et 0 sinon pour chaque uint8 du vecteur. Ce masque est ensuite utilisé par le *vec_sel* (*_mm_blendv_epi8)* qui va sélectionner la valeur de x1 si le masque vaut 0 (donc si x0 <= x1), et la valeur de x0 sinon.
On aura donc bien un vecteur de uint8 contenant les valeurs max des uint8 présents dans x0 et x1.
Sur le même principe, nous avons écrit une macro *vec_MIN*.

C) Principes des algorithmes SIMD

1. Frame Difference

L'optimisation SIMD pour cet algorithme consiste à effectuer plusieurs différences sur les pixels de deux images successives en même temps.

```
vuint8** routine_FrameDifference_SIMD(vuint8** I_t, vuint8**
I_t_moins_1, int vi0, int vi1, int vj0, int vj1, int theta);
```

Pour cet algorithme on charge des vecteurs de 128 bits qui représentent 16 pixels chacun. On parcourt les images I(t) et I(t-1). Pour chaque paquet de 16 pixels on calcule la valeur absolue de la différence grâce à la macro *vABS_DIFF* décrite ci-dessus. On va ensuite comparer les valeurs obtenues avec le seuil grâce à la macro *vCMP_THRESHOLD*. Enfin on utilise la macro *vec_store* qui va permettre de stocker le résultat à l'adresse de l'image qui va être retournée.
De cette manière on à **2* [(NB_COL / 16) * NB_ROW]** accès mémoire en lecture, et moitié moins en écriture.
Contre **[2 * NB_COL * NB_ROW]** accès mémoire en lecture et deux fois moins en écriture pour la version scalaire.

2. Sigma Delta

Dans cette fonction, on souhaite diminuer les accès mémoires et le nombre d'opérations à effectuer en traitant les pixels 16 par 16.

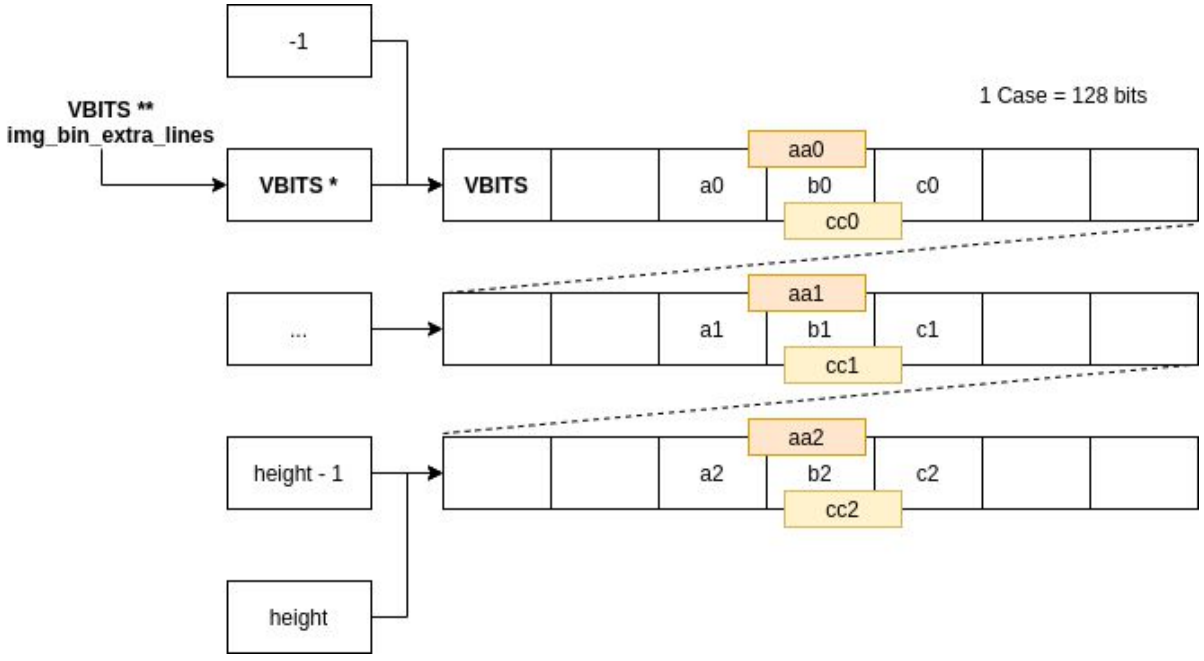
```
vuint8** SigmaDelta_1step_SIMD(vuint8** M_t_moins_1, vuint8***
M_t_save, vuint8** V_t_moins_1, vuint8*** V_t_save, vuint8**
I_t, int vi0, int vi1, int vj0, int vj1)
```

Dans un premier temps nous avons observé que les comparaison à effectuer entre les pixels pour calculer les valeurs de M(t) et V(t) sont similaires. Nous avons donc écrit une macro *vec_cmp_pixels_SD* que nous avons détaillé dans la section précédente.
Nous utilisons donc cette macro pour déterminer M(t) en fonction de M(t-1) et I(t). On détermine ensuite la O(t) en utilisant la macro *vABS_DIFF* qui calcule la valeur absolue de la différence entre M(t) et I(t). On multiplie ensuite cette valeur avec une constante N à l'aide de la macro *vec_muls_epi8_const* qui permet de plafonner la valeur de la multiplication à 255 en cas de dépassement. On calcul la valeur de V(t) en fonction de V(t-1) et de O(t)*N à l'aide une nouvelle fois de la macro *vec_cmp_pixels_SD* et des macros *MIN* et *MAX*. On range ensuite en mémoire les valeurs M(t) et V(t) pour pouvoir les réutiliser à l'itération suivante. Enfin, on compare la valeur de O(t) avec celle de V(t) à l'aide de la macro *vec_cmpgte_vui8* qui nous permet de faire des comparaisons sur des entiers non signés plus facilement. Le résultat de cette comparaison est stocké en mémoire pour pouvoir retourner une image dont les pixels sont soit à 0 soit à 255.
Grâce à ces optimisations, nous avons **3* [(NB_COL / 16) * NB_ROW]** accès mémoire en lecture et autant en écriture.
Précédemment, avec la version scalaire, nous avions **[3 * NB_COL * NB_ROW]** accès mémoire en lecture et autant en écriture.

3. Morphologie Mathématiques

Pour les fonctions de morphologie mathématiques, nous avons fait le choix d'utiliser des images représentées par des matrices de vbits (__m128i). Ces fonctions manipulent des images dont les pixels ne peuvent être que dans deux états différents, nous pouvons donc représenter chacun de ces pixels par un seul bit. De cette manière, nous sommes en capacité d'effectuer des opérations sur 128 pixels en même temps, en manipulant des registres de 128 bits. Nous avons décrit les fonctions de conversion d'image de uint8** vers vbits** (et inversement) ci-dessus.

Le fonctionnement général de nos fonctions de morphologie mathématiques peut être représenté de la manière suivante :



Cas général

img_bin_extra_lines = implémentation de bords horizontaux virtuels (nous allouons un vbits** dont la dimension est **NB_LIGNE + 2** et faisons pointer les lignes ajoutées vers la première et dernière ligne de l'image)
Lignes de droites = un unique malloc

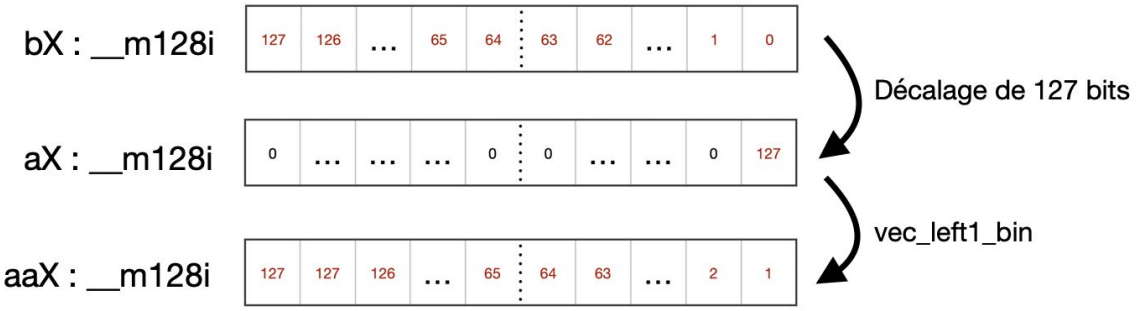
Le principe général est donc de charger les _m128i trois par trois sur 3 lignes. On utilise ensuite les macros *vec_left1_bin* et *vec_right1_bin* pour chacune des lignes :

- un vecteurs de 128 bits aaX correspondant à un décalage *vec_left1_bin(aX,bX)* dont le fonctionnement est décrit ci-dessus.
- un vecteur de 128 bits bX
- un vecteur de 128 bits ccX correspondant au décalage *vec_right1_bin(bX,cX)* également décrit ci-dessus

On peut ensuite effectuer l'opération (OR et AND logique) sur les 9 vecteurs obtenus. Le vecteur résultant sera alors un OR ou AND logique sur un voisinage de 3x3 des 128 pixels contenus dans le vecteur b1 initial.

Gestion des bords :

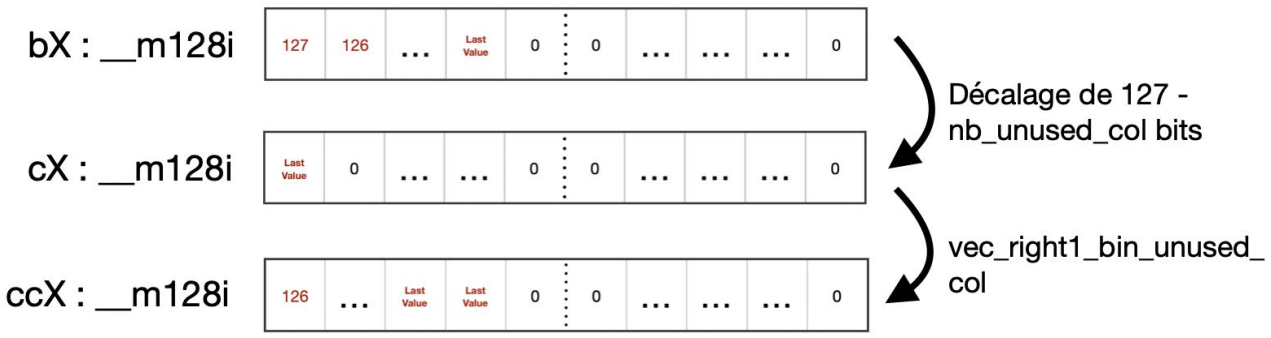
Pour les images avec un nombre de colonne supérieur à 128, on traite séparément les 128 premières colonnes, les colonnes intermédiaires, et les colonnes de la fin (pour les dernières, il peut y en avoir moins de 128 si le nombre total n'est pas un multiple de 128) afin de gérer les bords.
Pour les 128 premières colonnes, on ne peut pas utiliser de colonne d'indice -1 de notre matrice de vbits car elle n'existe pas. Pour le *vec_left1_bin*, on utilise alors le vbits correspondant à bX sur lequel on effectue un décalage de 127 bits vers la gauche afin de recopier le premier pixel de la ligne.



Gestion des bords gauches de l'image

Pour les vbits de notre matrice de vbits qui ne sont ni le premier vecteur de chaque ligne ni le dernier, aucun problème particulier. On a en effet toujours au moins un vecteur vbits à l'indice i-1 et i+1 par rapport à l'indice actuel.

L'autre attention particulière à avoir est pour le dernier vecteur vbits de chacune des lignes de la matrice. On ne peut avoir accès à un vecteur d'indice i+1 car il n'existe pas. De manière similaire à la gestion des bords droits de l'image, nous recopions le vecteur bX décalé cette fois de **127-nb_unused_col** vers la gauche avant d'utiliser la macro *vec_right1_bin_unused_col* décrite précédemment pour obtenir ccX (*nb_unused_col* est la taille en bit d'un vbits à laquelle on retranche le reste de la division euclidienne du nombre de colonne de l'image par 128).



Gestion des bords droits de l'image

Image de moins de 128 colonnes :

Nous avons également séparé le cas particulier des images ayant moins de 128 colonnes. Une fois convertis en matrice vbits, ces images ont un seul vecteur vbits par ligne. Nous avons donc fusionné les gestions des bords droits et gauches précédents pour calculer aaX et ccX à chacune des lignes.

3. Optimisations Algorithmiques

A) Déroulage de boucle et re-use

La première optimisation que nous avons mise en place est le déroulage de boucle. En ajoutant en prologue et un épilogue, nous effectuons les opérations AND/OR sur 3 lignes consécutives à chaque itération de boucle. On peut de cette manière réduire le nombre d'accès mémoire en conservant les valeurs des deux lignes précédentes (comme chaque calcul nécessite des valeurs de 3 lignes différentes), et éviter la rotation de variable qui consiste à recopier les valeurs les plus récentes dans des variables temporaires.
Rotation de variable possible :

```
aa0 = aa1; aa1 = aa2;
b0 = b1; b1 = b2;
cc0 = cc1; cc1 = cc2;
```

Le déroulage de boucle permet de calculer les nouvelles valeurs et de remplacer les plus anciennes à chaque fois. L'ordre de déroulage de 3 permet de retomber sur un remplacement de la variables la plus ancienne à chaque boucle.

Sur le schéma ci-dessous, chaque case correspond à un vecteur de 128 bits, et chaque cadre pointillé correspond à un AND/OR entre aa0, b0, cc0, aa1, b1, cc1, aa2, b2 et cc2.

	0	1	2	3	4	5	6	7	8	9
AND i	10	11	12	13	14	15	16	17	18	19
AND i+1	20	21	22	23	24	25	26	27	28	29
AND i+2	30	31	32	33	34	35	36	37	38	39
...	40	41	42	43	44	45	46	47	48	49
	50	51	52	53	54	55	56	57	58	59

Déroulage de boucle

B) Séparation d’opérateurs

Dans un second temps, pour réduire le nombre de calculs, nous avons séparé les opérateurs AND et OR qui effectuaient l’opération sur 9 vecteurs à chaque ligne. Maintenant, à chaque nouvelle ligne, nous calculons un AND/OR sur les 3 vecteurs aa2, b2, et cc2. Puis on utilise une macro pour effectuer un AND/OR sur les valeurs des deux lignes précédentes et de la nouvelle valeur calculée. On remplace le calcul le plus ancien par le nouveau calcul. De cette manière, on divise par 3 le nombre d'opérations AND/OR par ligne par rapport à la version naïve (avec le déroulage de boucle on effectue 9 opérations OR/AND au lieu de 27 précédemment par itération).

Sur le schéma ci-dessous, chaque case correspond à un vecteur de 128 bits, et chaque cadre bleu correspond à un AND/OR entre aaX, bX et ccX.

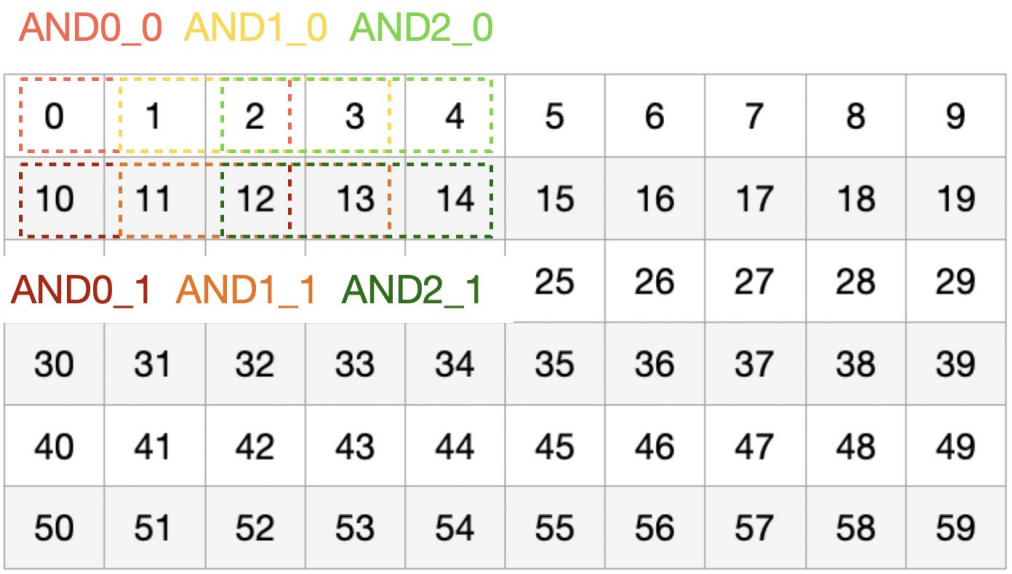
Séparation d’opérateur

4. Fusion d'opérateurs

La fusion d'opérateurs est une modification logicielle consistant à regrouper les instructions de plusieurs fonctions qui s'enchaînent en une unique fonction. Le fait de réaliser plusieurs fonctions en une seule permet alors de faciliter les optimisations logicielles comme les rotations de registres ou les déroulages de boucle.

Ici, nous utiliserons cette méthode afin de fusionner les fonctions "érosion" et "dilatation" en les fonctions "ouverture" et "fermeture".

Voyons dans un premier temps le cas général, ne nécessitant pas de traitement particulier dû aux bords. Dans le prologue de chaque colonne de la matrice de vbits, nous commençons par calculer aa0, aa1, cc0 et cc1 avec les macros *vec_left1_bin* et *vec_right1_bin* détaillées précédemment. Puis on effectue un premier AND/OR (ouverture/fermeture) entre aa0, b0 et cc0 que l'on stock dans une variable appelée *and0_0 (or0_0)*. On effectue la même opération entre aa1, b1 et cc1 que l'on conserve dans une variable appelée *and0_1 (or0_1)*. On va ensuite changer la valeur des deux *aX* afin d'y stocker le vbits qui suit *cX* (colonne de cX + 1), et calculer les nouvelles valeurs de *aaX* et *ccX (aaX = vec_left1_bin(bX, cX) et ccX = vec_right1_bin(cX, aX))* on calculera par la suite un AND/OR logique entre aaX, cX et ccX que l'on stockera dans des variables and1_0 (*or1_0*) et and1_1 (*or1_1*). On répète ensuite cette opération une troisième fois pour calculer *and2_0* (*or2_0*) et *and2_1* (*or2_1*).



Dans le prologue, on traite la ligne '-1', pour cela on utilise deux fois les calculs effectués sur la première ligne. Cela revient à recopier la première ligne dans une éventuelle ligne -1 rajoutée.

On va alors réaliser un AND/OR logique entre :

AND0_0, AND0_1 et AND0_2 → stocké dans AND0
AND1_0, AND1_1 et AND1_2 → stocké dans AND1
AND2_0, AND2_1 et AND2_2 → stocké dans AND2

Dans le cas du prologue ANDX_2 sont remplacés par ANDX_0 pour la recopie de la première ligne.

AND0 AND1 AND2

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59

On va ensuite utiliser ces 3 valeurs pour effectuer l'autre opération OR/AND et conserver la valeur dans une variable OR0 (AND0). Pour cela on réutilise les macro *vec_left1_bin* et *vec_right1_bin* avec les valeurs AND0, AND1 et AND2 (OR0, OR1 et OR2).

OR0

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59

Pour la gestion de la première ligne, la variable OR1 sera identique à OR0.

À la fin du prologue, on entre dans une boucle. À chaque nouvelle itération, on calcul des valeurs de ANDX_2 de la ligne suivante, qui nous permettent de calculer une variable ORX (ANDX) de manière analogue. Puis, cette nouvelle valeur ORX (ANDX) ainsi que les deux précédentes permettent de calculer une valeur finale à stocker dans la matrice à retourner.
Nos fonctions d'ouverture et de fermeture utilisent également le déroulage de boucle. À chaque itération on va donc successivement calculer toutes les valeurs précédentes sur 3 lignes.

Concernant la gestion des bords au niveau des colonnes, on utilise les mêmes méthodes que pour l'érosion et la dilatation en utilisant la première/dernière colonne existante et en effectuant un décalage de 127 bits du côté qui nous intéresse pour pouvoir l'utiliser dans nos calculs de aaX et ccX.

Pour gérer les bords efficacement, on traite différemment les images ayant moins de 4 vbits par ligne, c'est à dire les images de moins de 4*128 = 512 colonnes. On parcourt ces matrices en une seule fois, de la première ligne à la dernière en effectuant des décalages sur les valeurs en périphérie de l'image de la même manière que pour l'érosion et la dilatation. Il n'y a donc qu'une seule boucle pour parcourir les lignes ce qui rend l'algorithme particulièrement performant pour ces tailles d'image.

Pour les images de plus de 512 colonnes, on traite séparément les deux premiers et les deux derniers vbits de chaque ligne (les deux premières et dernières colonnes). Les vbits centraux seront traités dans la boucle principale ne nécessitant pas de gestion de bords.

On résume le principe utilisé dans la boucle principale à la page suivante.

Résumé en schémas du principe utilisé

AND0_0 AND1_0 AND2_0
AND0_1 AND1_1 AND2_1
AND0_2 AND1_2 AND2_2

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59

AND0 AND1 AND2

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59

OR0

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59

On déroule ensuite les calcul ligne par ligne, en utilisant à chaque fois les valeurs calculées aux lignes précédentes pour réduire le nombre d'opérations et d'accès mémoire. Pour calculer une "valeur finale" qui sera rangée dans la matrice retournée par la fonction, il faut effectuer des opérations sur un voisinage 5x5 de la valeur à l'indice considéré.

5. Optimisations matérielles

On utilise l'interface de programmation OpenMP afin de paralléliser les traitements de notre chaîne. Le but est ici de se servir de tous les cœurs de CPU d'une machine en même temps, car un programme ne s'exécute par défaut que sur un cœur à la fois. On peut donc par exemple partager le traitement d'une boucle "for" entre plusieurs threads (ce sera d'ailleurs l'utilisation principale que nous ferons de OpenMP). Il faut néanmoins être prudent avec ce genre d'optimisations. En effet, la création de threads a un coût en temps et en ressources matérielles. Pour de petites boucles, il sera donc parfois plus avantageux de garder une exécution mono-thread.

A) Mouvements

Les algorithmes de mouvements possèdes des boucles for dont les itérations ne sont pas dépendantes les unes des autres. Elles sont donc parallélisables.

Pour la fonction *routine_FrameDifference_SIMD*, on parallélise l'unique boucle for, déjà optimisée. On fait de même pour la fonction *SigmaDelta_1step_SIMD*.

B) Morphologies

Pour ce qui est des fonctions de morphologies, on ne tente pas de paralléliser les boucles "for" correspondant aux cas particuliers, même si elles sont parallélisables. On observe en effet que dans ces fonctions, le surcoût correspondant à l'ouverture des threads dépasse le temps qu'ils nous font gagner.

En plus de ça, les boucles "for" principales des fonctions optimisées ne sont pas parallélisables, car une itération (i) dépend de l'itération précédente (i-1), dû aux rotations de registres. On les parallélise donc uniquement pour les tests, afin d'avoir une idée des gains potentiels.

6. Tests

Une étape importante du projet fut la validation des fonctions écrites sous forme de tests unitaires à valider. Nous nous sommes surtout concentrés sur la validation des fonctions morphologiques, car leurs optimisations ont été difficiles, et il fallait donc être certain de leur véracité. Ces tests sont disponibles dans les fichiers *test _<nom>.c*

Pour les codes scalaires et SIMD des fonctions d'érosion et de dilatation, nous avons donc développé une vingtaine de tests unitaires permettant de valider leurs comportements sur des motifs élémentaires 3x3.

On a également développé des tests pour des images possédant différentes tailles, comme par exemple un test de bordure, où l'on vérifie qu'une bordure comportant des 1 est bien dilatée correctement par nos fonctions de dilatations.

Pour tester nos fonctions de fusion morphologiques, nous avons développé des tests universels, comparant les résultats de nos fonctions ouverture et fermeture basées sur des compositions des fonctions erosion_SIMD et dilatation_SIMD, validés par des tests. Ces tests universels prennent une taille de matrice en entrée, instancient la matrice de zéros correspondant, et changent les valeurs de ses bits 1 par 1, en réalisant une nouvelle comparaison des résultats à chaque fois.

Ces tests nous ont alors révélé de nombreuses erreurs qui ont donc pu être corrigées. Par exemple, c'est ici que nous nous sommes rendu compte que les codes OpenMP pouvaient être source d'erreurs quand les itérations d'une boucle sont dépendantes les unes des autres.

7. Résultats

Pour mesurer le temps d'exécution de nos fonctions, on utilise la macro suivante :

```
#define CHRONO(X,t)                          \
    tmin = 1e38;                             \
    for(run=0; run<nrun; run++) {            \
        t0 = (double) _rdtsc();              \
        for(iter=0; iter<niter; iter++) {    \
            X;                               \
        }                                    \
        t1 = (double) _rdtsc();              \
        dt=t1-t0; if(dt<tmin) tmin = dt;     \
    }                                        \
    t = tmin / (double) niter
```

Cette macro lance plusieurs fois la fonction X, pour récupérer une "moyenne minimale" du temps d'exécution. Ce temps est mesuré en cycles machines, donc le nombre de cycle du CPU qu'on a compté entre le début et la fin de la fonction. Le résultat est stocké dans la variable "t". On produit ici des fuites mémoires lorsqu'on appelle une fonction qui renvoie un pointeur sur une zone nouvellement allouée. On définit donc une macro "BENCH" pour libérer la mémoire à l'intérieur des fonctions appelées lorsqu'on réalise des mesures.

Cette méthode d'évaluation possède des limites dans l'optique où nos mesures sont lancées sur un OS classique : le scheduler peut donc interrompre l'exécution du programme au profit de routines ou de démons de l'OS. Afin d'affiner nos mesures, on pourrait les lancer sur un OS temps réel, afin de s'assurer que notre programme a la priorité sur tout autre processus. Malheureusement, nous n'avons pas de machines avec RTOS sous la main. On se contente donc de simplifier au maximum notre environnement d'exécution en tuant tous les programmes parasites, et en se débarrassant de l'interface graphique (mode console). Cependant, et ce malgré ces précautions, on observe parfois des résultats allant du simple au quadruple pour une même mesure, ce qui nous oblige à prendre avec des pincettes les résultats qui vont suivre.

On effectue tous nos tests de parallélisation sur une machine possédant 4 coeurs. Si on ne spécifie pas le nombre de threads à utiliser avec omp_set_num_threads(), le compilateur l'adapte au nombre de coeurs disponibles. On peut alors choisir avec notre makefile si l'on souhaite activer les optimisations de parallélisation ou non. Elles sont par défaut désactivées, mais la commande suivante compile en les activant :
$ make C_OPENMP_FLAGS="-fopenmp"

On présente dans les tableaux et graphiques ci-dessous les mesures obtenues pour nos fonctions de mouvements et de morphologies.

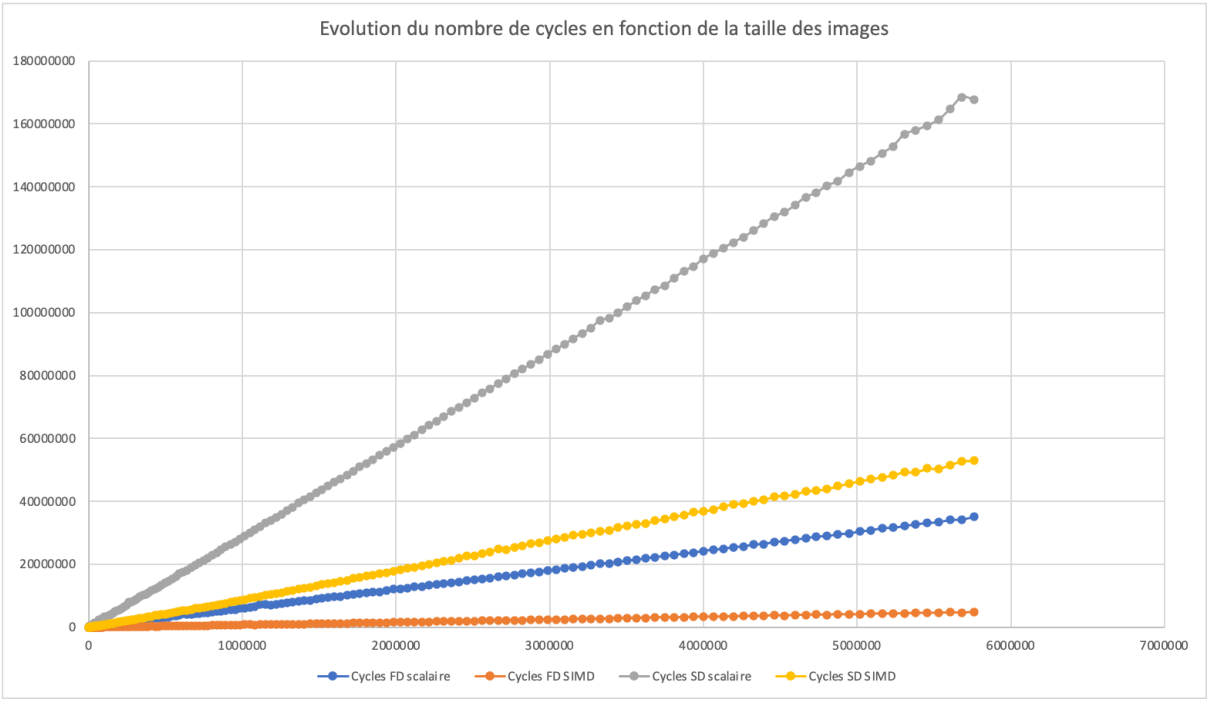
A) Mouvements

	Frame-Difference	Sigma-Delta_1step
Scalaire mono-thread	420k	3 600k
SIMD	120k	700k
SIMD + OpenMP	130k	1 200k

Temps d'exécutions des opérations de mouvement sur des images standard (en nombre de cycles machines)

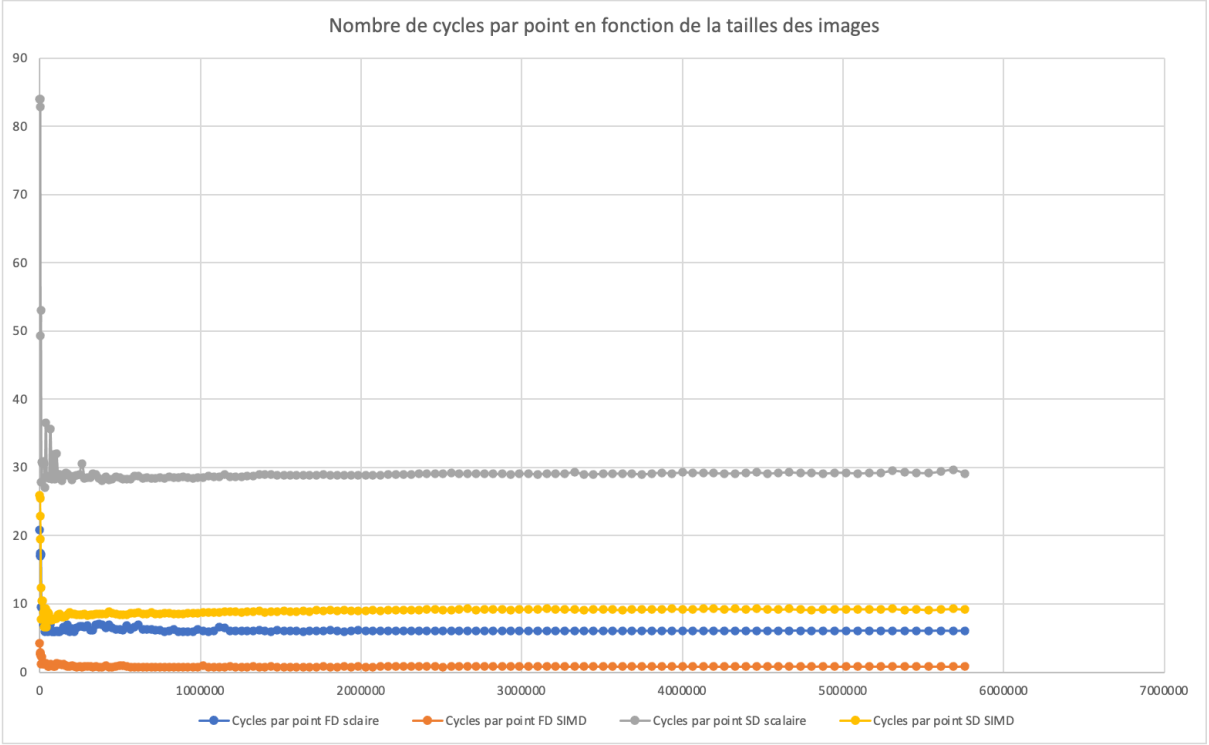
On remarque ici que l'étape d'optimisation SIMD est très rentable niveau performances. On constate en effet un ratio de 5 entre la version scalaire et son équivalent SIMD. En revanche, les parallélisations OpenMP ne semblent rien apporter, elles ont même tendances à dégrader les performances.
Nous avons également écrit une fonction qui permet de tester les algorithmes de Frame-Difference et Sigma-Delta (scalaire et SIMD) sur des images carrés de tailles différentes (dont la valeur des pixels est générée aléatoirement), et qui consigne les données dans un fichier au format CSV.

On a ensuite pu tracer les courbes suivantes :



Evolution du nombre de cycles en fonction de la taille de l'image en entrée (échelle linéaire)

Ces premières courbes représentent le nombre de cycles mesurés pour les versions scalaire et SIMD des deux algorithmes de mouvement. On observe une augmentation linéaire du nombre de cycles. On voit également que les version SIMD apportent une diminution significative du nombre de cycles par rapport à aux versions scalaires.



Evolution du nombre de cycles par point en fonction de la taille de l'image (échelle linéaire)

Ces autres courbes représentent le nombre de cycles par point (pixel) en fonction de la taille des images en entrée. On remarque alors que le nombre de cycles par point est assez élevé pour des images de petite taille, et se stabilise pour des images dont la taille est supérieure à 350x350 environ. Pour l'algorithme de Frame-Difference, la version scalaire se stabilise autour de 6 cycles par point, et la version SIMD autour de 0,8 cycle par point. On a donc une version SIMD qui utilise 6,5 fois moins de cycle machine que la version scalaire pour des images suffisamment grandes.
Pour l'algorithme de Sigma-Delta, la version scalaire se stabilise autour de 29 cycles par point, et la version SIMD autour de 9 cycles par point. La version SIMD utilise donc environ 3 fois moins de cycles machine que la version scalaire.

B) Morphologies

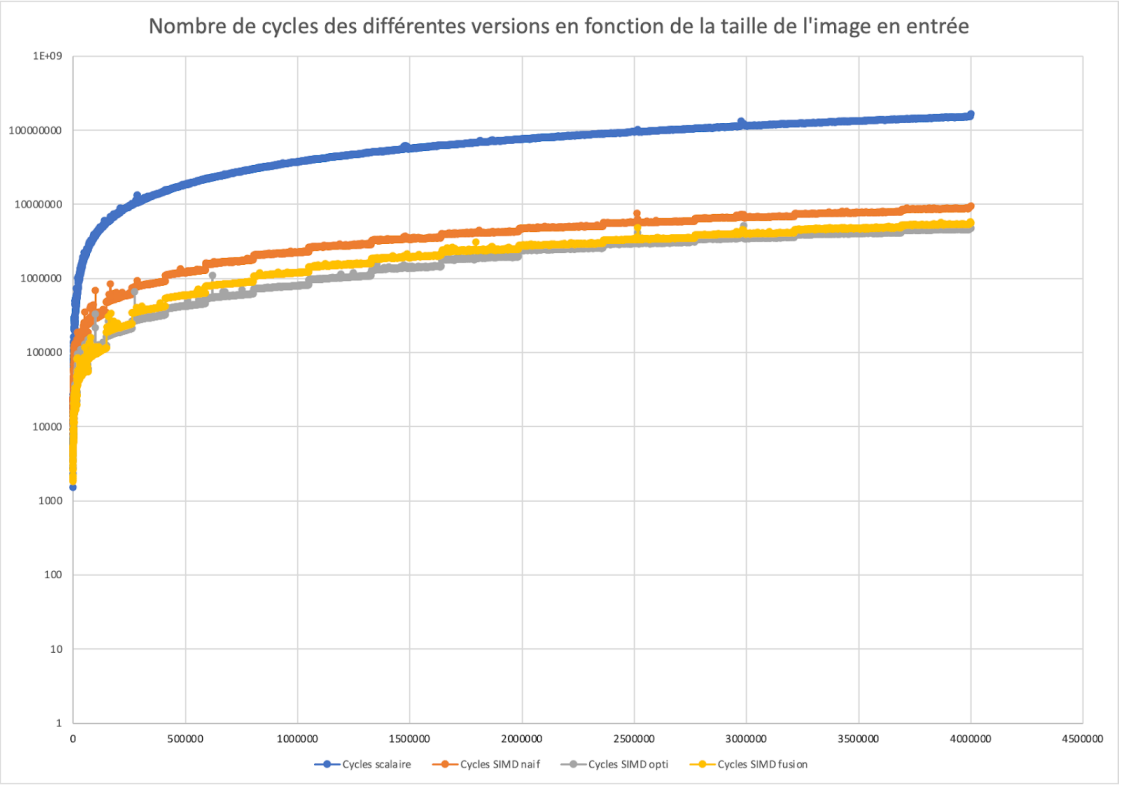
	Erosion / Dilatation	Ouverture / Fermeture	Chaîne complète
Scalaire mono-thread	600k	1 100k	2 300k
SIMD naïf	78k	186k	355k
SIMD naïf + OpenMP	130k	250k	520k
SIMD déroulage de boucle	41k	83k	152k
SIMD opti + fusion		65k	110k
SIMD + OpenMP	48k	65k (fusion)	130k

Temps d'exécutions des opérations de morphologies sur des images standard (en nombre de cycles machines)

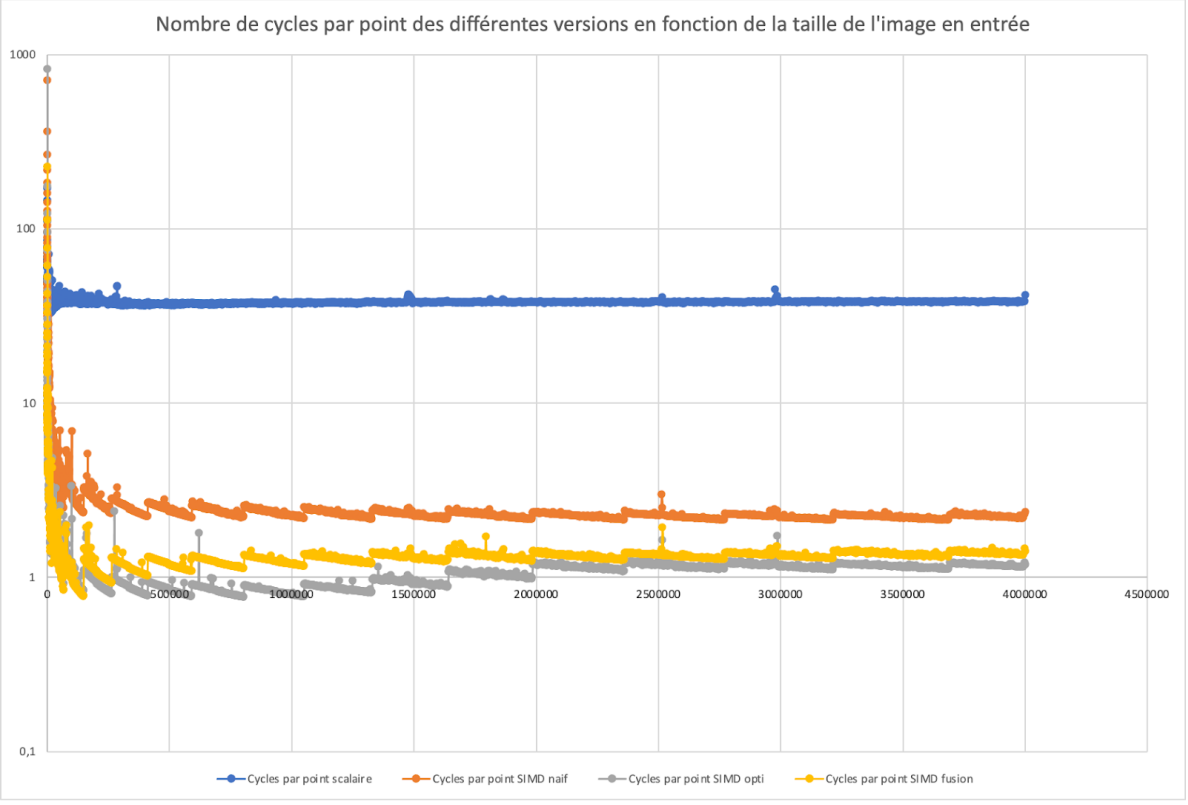
Ici encore, c'est le passage d'un code scalaire à un code SIMD qui présente le plus de gain en temps (ratio d'environ 6). Les optimisations algorithmiques viennent en second avec un ratio d'environ 2. Les optimisations de parallélisations sont en revanche une nouvelle fois inefficaces, sachant que la dernière ligne (en rouge) ne passe plus les tests de vérification (résultats corrompus).

Comme précédemment, nous avons écrit un banc de test permettant de mesurer le nombre de cycles machine nécessaires à réaliser pour la séquence érosion - dilatation - dilatation - érosion, avec différentes optimisations. Ces mesures sont réalisées pour des images carrées de taille grandissante, et consignées dans un fichier au format CSV.

Nous avons tracé les courbes suivantes (pour des images allant d'une taille 3x3 à 2000x2000 par pas de 1) :



Evolution du nombre de cycles en fonction de la taille de l'image en entrée (échelle semi log)



Evolution du nombre de cycles par point en fonction de la taille de l'image (échelle semi log)

Toutes ces courbes nous apprennent plusieurs choses. Dans un premier temps, comme pour les algorithmes de mouvement, les versions SIMD diminuent significativement le nombre de cycles machine par rapport à la version scalaire. On remarque aussi un motif de dent de scie sur les versions SIMD, qui se répète à chaque fois que les dimensions de l'image augmentent de 128 pixels. En effet, nos versions SIMD utilisent une représentation binaire de l'image où chaque bit d'un vecteur vbits représente 1 pixel. Notre matrice de vbits gagnera une colonne (1 vecteur vbits par ligne) à chaque fois que la taille modulo 128 sera égale à 1. On peut ensuite ajouter 127 autres colonnes à l'image d'entrée sans modifier les nombre de vbits par ligne de la matrice représentant l'image. Le nombre de cycles reste donc constant sur 128 pixels. Cependant comme le nombre de points augmente, on a une diminution du nombre de cycle par point. Le fonctionnement le plus optimisé de nos fonctions est donc pour des images dont le nombre de colonne est un multiple de 128.

On peut aussi observer que la version SIMD fusion utilise moins de cycles machine que la version SIMD opti pour des images de dimensions inférieures à 380x380 environ, mais est moins performante pour des images de plus grande taille. Cela peut s'expliquer par la différence de traitement entre les images ayant moins de 4 vbits par ligne (< 4*128 colonnes) et celle en ayant plus. En effet, pour gérer les bords de ces "petites" images, on parcourt l'image une seule fois ligne par ligne, en effectuant tous les calculs pour toutes les colonnes à chaque ligne. Nous n'avons donc qu'une seule boucle permettant de parcourir les lignes, ce qui rend cette version particulièrement efficace pour les images de petite taille.