SORBONNE UNIVERSITÉ

UPMC LICENCE INFORMATIQUE

RAPPORT L3 PROJET VISION

Par:
THURAIRAJAH Shaithan
ABITBOL Ethan

Optimisations d'opérations morphologiques pour un rendu en temps réel.

Mai 2021

Sommaire

1 Introduction

Ce projet de vision par ordinateur est divisé en 3 parties.
La première partie **algorithmique** demande l'implémentation d'une chaîne de traitement de type système embarqué que nous allons voir avec **sigma delta**.
La seconde consiste à réaliser un ensemble **d'optimisations sur des morphologies mathématiques** à la fois algorithmiques, logicielles et architecturales afin d'avoir un rendu en temps réel et aussi de comprendre leur gain de performance séparé et combiné.
Enfin, le dernier objectif consiste a développer une **méthodologie d'implantation et validation pour améliorer ses capacités de débogage**.

Notre travail ici consistera tout d'abord à implémenter **sigma delta** suivi d'un algorithme de morphologie "naïf" pour obtenir une **version de référence**, qu'on utilisera dans un second temps afin de tester et d'optimiser cette chaîne de traitement pour qu'elle soit la plus rapide possible en appliquant des transformations de bas niveaux tel que les **déroulements de boucles** et des transformations de haut niveaux : **pipeline d'opérateurs, fusions d'opérateurs**. Pour finir, nous implémenterons du **subword parallelism (SWP)** qui semble être l'optimisation la plus prometteuse. Ainsi, nous testerons et nous comparerons toutes les fonctions pour en déduire la meilleur.

2 Sigma Delta (ΣΔ ou SD)

Nous allons implémenter sigma delta un algorithme se basant sur la différence de l'image à un point entre t et $t+1$. Cet algorithme prends une séquence d'image source I et retourne une image d'étiquettes binaires E. Sigma Delta prend aussi en considération que le niveau de bruit peut varier en tout point ainsi il calcule la moyenne M et l'écart-type V pour produire E ce qui le différencie par exemple de Frame-Difference (FD).

2.1 Initialisation

Tout d'abord, on initialise l'algorithme à $t = 0$, ci-dessous la version ligne de l'initialisation de algorithmique.

```
// ---------------------------------------------------------
void SigmaDelta_Step0_line(uint8 *I, uint8 *M, uint8 *O, uint8 *V, uint8
    *E, int j0, int j1)
// ---------------------------------------------------------
{
    /*
        Initialisation de SD pour t = 0 version ligne
    */
    setborder1(j0, j1);// debug
    int j;
    uint8 m, i, v;
    for (j = j0; j < j1; j++)
    {
        i = load1(I, j);// Load
        m = i;
        v = SD_VMIN;
        // Store
        store1(M, j, m);store1(V, j, v);
    }
}
```

M(0) est initialisé à **I(0)** et **V(0)** à **SD_VMIN** correspondant à une valeur minimale cette fonction sera appelé pour toutes les lignes de I(0).

2.2 Calcul de E

On peut ensuite exécuter l'algorithme pour $t > 0$.
L'algorithme est constitué de plusieurs étapes :

Estimation de M(t) : On compare M(t-1) et I(t) pour le pixel courant pour déterminer M(t)

Calcul de O(t) : Ici, on fait la différence entre le pixel courant source et la valeur moyennée des pixels antérieurs. Comme dans FD, on fait la différence entre 2 pixels subséquents dans le temps mais, ici, on prends en compte le bruit en utilisant M(t) au lieu de I(t-1)

Mis à jour de V(t) : Dans cet étape, nous allons mettre à jour l'écart type en le comparant avec la valeur de O(t) multiplié par une constante k. Nous allons ensuite restreindre le domaine de sa valeur entre Vmin et Vmax[1].

Étiquetage binaire E(t) : Et enfin, en comparant O(t) et V(t), on pourra déterminer si le pixel courant est en mouvement ou non.

```
// ----------------------------------------------------------------
void SigmaDelta_1Step_line(uint8 *I, uint8 *M, uint8 *O, uint8 *V, uint8
    *E, int k, int j0, int j1)
// ----------------------------------------------------------------
{
    /*
        debut de SD a partir de t =1
    */
    setborder1(j0, j1);// debug
    int j;
    uint8 i, m, o, v, e;
    for (j = j0; j < j1; j++)
    {
        // load
        i = load1(I, j);m = load1(M, j);o = load1(O, j);
        v = load1(V, j);e = load1(E, j);

        // STEP 1 : Estimation Mt
        if (m < i)
        {
            m = m + 1;
        }
        else if (m > i)
        {
            m = m - 1;
        }

        // STEP 2 : Ot computation
        o = abs(m - i);

        // STEP 3 : Vt update and clamping
        if (v < (k * o))
        {
            v = v + 1;
        }
        else if (v > (k * o))
        {
            v = v - 1;
        }

        v = ((v < SD_VMAX)? v: SD_VMAX);
        v = ((v > SD_VMIN)? v: SD_VMIN);

        // STEP 4 : Et estimation
        if (o < v)
```

[1]Dans notre cas, Vmin=2 et Vmax=253

```
45          {
46              e = 0;
47          }
48          else
49          {
50              e = 1;
51          }
52
53          // store
54          store1(M, j, m);store1(O, j, o);
55          store1(V, j, v);store1(E, j, e);
56      }
57
58  }
```

Malgré le fait de prendre en compte le bruit dans cet algorithme, il n'est pas parfait. Ainsi, on va enchaîner SD + convolution d'opérations morphologiques.

3 Optimisations

Voyons les préparations effectués pour assurer la robustesse et un débogage efficace du code des opérations morphologiques.

3.1 MACROS

Toutes les macros ci-dessous ont été implémentées dans ”morpho.h” :

3.1.1 Macros de débogage

En commençant par des macros “classiques“ de débogage :

Affichage (stdout) : plusieurs macros pour le débogage par affichage dans la console principalement idisp (affichage d'un entier), idisp9 (affichage de 9 entiers) et des fonctions[2].

Contrôle d'accès : des macros permettants le contrôle d'accès sur un tableau (load et store) lorsque les macros **DEBUG** et **CONTROL_ENABLE** sont définis

```
#define setborder1(i0, i1) int dbg_i0 = i0-1; int dbg_i1 = i1+1
#define load1(X, i) X[i];\
        if((i < dbg_i0) || (i > dbg_i1)){\
            printf("\n\e[31mFATAL ERROR\e[0m line %d in file %s :\nLoad %s[%s] =>
    interval is [%d, %d] but accessing index %d\n", __LINE__, __FILE__, #X, #i,
    dbg_i0, dbg_i1, i);\
            exit(EXIT_FAILURE);\
        }
#define store1(Y, i, y) Y[i]=y;\
        if((i < dbg_i0) || (i > dbg_i1)){\
        printf("\n\e[31mFATAL ERROR\e[0m line %d in file %s :\nStore %s[%s] =>
    interval is [%d, %d] but accessing index %d\n", __LINE__, __FILE__, #Y, #i,
    dbg_i0, dbg_i1, i);\
        exit(EXIT_FAILURE);\
        }

```

L'utilisation des macros **store1** et **load1** impliquent en mode debug que **setborder1** soit appelé au préalable.

3.1.2 Macros opérations

Ensuite, nous allons créer des macros pour les opérations mathématiques que nous utiliserons pour améliorer la sémantique des programmes et rendre le débogage plus facile. Les opérations que nous devons implémenter sont le max et le min qui correspondent en binaire à OR et AND[3] respectivement :

Les opérations logiques bit à bit sont très difficile à déboguer. En effet, nous travaillons d'abord sur le bit de poids faible d'un $uint8$[4], soit avec des 0 et des 1.

[2] pour SWP, nous utiliserons les fonctions d'affichage (format bits) fournis dans un VERBOSE
[3] Il s'agit ici du AND & et OR || logique bit à bit
[4] correspondant à un unsigned char en C

On va donc, en mode debug, utiliser un autre opérateur plus simple à déboguer l'addition :

```
#ifdef DEBUG
    #define AND +
    #define OR +
#else
    #define AND &
    #define OR |
#endif
#define OP2MAX(x, y) ((x) OR (y))
#define OP2MIN(x, y) ((x) AND (y))
```

Cependant, nous travaillons toujours sur des $uint8$ et l'enchaînement d'addition va générer des dépassements (Overflow) qui rendent le débogage et les tests plus compliqués. Une autre solution serait l'utilisation des 8 bits et d'un opérateur max et min qui effectue le maximum et minimum respectant l'ordre naturel N :

```
#define OP1MAX(x, y) ((x) > (y) ? (x) : (y))
#define OP1MIN(x, y) ((x) < (y) ? (x) : (y))
```

On peut alors utiliser ces macros opérateurs pour définir nos macros MAX et MIN que nous utiliserons dans le code.

MAX : ces macros vont nous permettre de faire un maximum entre n éléments, principalement, utilisés dans Dilatation.

```
#define MAX(x, y) OP2MAX(x, y)
#define MAX3(x, y, z) MAX(MAX(x, y), z)
#define MAX4(x0, x1, x2, x3) MAX(MAX(x0, x1),MAX(x2, x3))
#define MAX6(x0, x1, x2, x3, x4, x5) MAX(MAX3(x0, x1, x2),MAX3(x3, x4,x5))
#define MAX9(x0, y0, z0, x1, y1, z1, x2, y2, z2) MAX3(MAX3(x0, y0, z0),MAX3(x1, y1,
    z1),MAX3(x2, y2, z2))
```

MIN : ces macros vont nous permettre de faire un minimum entre n éléments, principalement, utilisés dans Érosion.

```
#define MIN(x, y) OP2MIN(x, y)
#define MIN3(x, y, z) MIN(MIN(x, y), z)
#define MIN4(x0, x1, x2, x3) MIN(MIN(x0, x1),MIN(x2, x3))
#define MIN6(x0, x1, x2, x3, x4, x5) MIN(MIN3(x0, x1, x2),MIN3(x3, x4,x5))
#define MIN9(x0, y0, z0, x1, y1, z1, x2, y2, z2) MIN3(MIN3(x0, y0, z0),MIN3(x1, y1,
    z1),MIN3(x2, y2, z2))
```

SHIFT : ces macros vont nous permettre de faire un décalage à gauche et à droite pour des paquetages sur 8, 32, et 64 bits

```
#define LEFT_8(a,b)   (((a) >> (7))  | ((b) << 1     ))
#define RIGHT_8(b,c)  (((b) >> 1     )  | ((c) << (7)))

#define LEFT_32(a,b)  (((a) >> (31))  | ((b) << (1)     ))
#define RIGHT_32(b,c) (((b) >> (1)     )  | ((c) << (31)))

#define LEFT_64(a,b)  (((a) >> 63)  | ((b) << 1     ))
#define RIGHT_64(b,c) (((b) >> 1     )  | ((c) << 63))
```

En mode DEBUG, il est donc possible de contrôler l'accès à une case, afficher les différents opérations effectués et utiliser l'addition comme opération de base.
En mode RELEASE, tous les affichages et contrôles d'accès qui ont un coût en performance conséquent disparaissent et les opérateurs sont OR et AND.
Cette structure permet aussi l'implémentation de test unitaire grâce à l'utilisation des 8 bits et les opérations de min et de max ternaire.

Le nommage des macros a été pensé pour pouvoir les différencier d'une fonction (nom en majuscule) et pour pouvoir passer d'une opération à l'autre très facilement. En effet, la structure du projet nous permet de faire Erosion et de copier le même code dans Dilatation mais en changeant les MIN par des MAX ce qui est très simple dans la plupart des éditeurs de code modern. De même, il est très facile de passer d'un SWP à un autre, il suffit de changer le suffixe des macros LEFT et RIGHT.

3.2 Érosion et Dilatation

3.2.1 Érosion

L'érosion va permettre de réduire le bruit dans les images en appliquant un élément structurant de taille 3x3, cela fera disparaître tout groupe de pixels de rayon inférieurs à sa taille.

Pour procéder a cette réduction on va d'abord rajouter un bord égale a 0 et pour chaque ligne on va faire case par case le minimum de la case avec les 8 cases qui l'entourent.

1	1	1
1	1	0
1	1	1

en ajoutant les bords cela devient :

0	**0**	**0**	**0**	**0**
0	1	1	1	**0**
0	1	1	0	**0**
0	1	1	1	**0**
0	**0**	**0**	**0**	**0**

ensuite on fait le min des 9 cases pour chaque case de chaque ligne :

0	**0**	**0**	0	0
0	1	**1**	1	0
0	**1**	**1**	0	0
0	1	1	1	0
0	0	0	0	0

-

0	**0**	**0**	**0**	0
0	**1**	1	**1**	0
0	**1**	**1**	**0**	0
0	1	1	1	0
0	0	0	0	0

-

0	0	**0**	**0**	**0**
0	1	**1**	1	**0**
0	1	**1**	**0**	**0**
0	1	1	1	0
0	0	0	0	0

On obtiens finalement comme matrice, le résultat suivant :

0	0	0
0	0	0
0	0	0

Figure. Explication de érosion.

Toutes les fonctions suivantes ont été implémentées dans **morpho_erosion.c**

3.2.2 Version basique

La version basique de line_érosion est tout simplement le MIN9 de chaque case de la ligne. Toutes les versions basiques sont testées avec tests unitaires.

```
// ------------------------------------------------------------
void line_erosion3_ui8matrix_basic(uint8 **X, int i, int j0, int j1, uint8 **Y)
// ------------------------------------------------------------
{
    /*
        Version basique
    */
    int j;
    for (j = j0; j <= j1; j++)
    {
        Y[i][j] = MIN9(
            X[i - 1][j - 1], X[i - 1][j + 0], X[i - 1][j + 1],
            X[i + 0][j - 1], X[i + 0][j + 0], X[i + 0][j + 1],
            X[i + 1][j - 1], X[i + 1][j + 0], X[i + 1][j + 1]);
    }
}
```

Cette version, va être notre version de référence, elle va nous permettre de la comparer avec les différentes optimisations que nous avons implémentées. Évidemment, elle n'est pas optimale. Par exemple en assembleur[5], le corps de boucle effectue des Load, Add et And qui se chevauchent :

```
// dans la boucle de line basique
Y[i][j] = MIN9(
    X[i - 1][j - 1], X[i - 1][j + 0], X[i - 1][j + 1],
    X[i + 0][j - 1], X[i + 0][j + 0], X[i + 0][j + 1],
    X[i + 1][j - 1], X[i + 1][j + 0], X[i + 1][j + 1]
);
```

```
ld      a3,0(t4)
addi    a1,a2,-1
ld      a6,0(t5)
addi    t1,a2,1
add     a7,a3,a2
add     a5,a3,a1
lbu     t6,0(a7)
lbu     a5,0(a5)
add     a7,a3,t1
add     t0,a6,a1
ld      a3,0(a0)
lbu     a7,0(a7)
lbu     t2,0(t0)
add     t0,a6,a2
lbu     t0,0(t0)
and     a5,a5,t6
add     a6,a6,t1
lbu     t6,0(a6)
add     a1,a3,a1
and     a5,a5,a7
and     a5,a5,t2
lbu     a7,0(a1)
add     a1,a3,a2
lbu     a6,0(a1)
and     a5,a5,t0
add     a3,a3,t1
ld      a1,0(a4)
lbu     a3,0(a3)
and     a5,a5,t6
and     a5,a5,a7
and     a5,a5,a6
add     a2,a1,a2
and     a5,a5,a3
sb      a5,0(a2)
mv      a2,t1
```

Le compilateur n'arrive pas à "bien" optimiser ce code. Nous voulons donc améliorer cette implémentation naïve de **Erosion**.

3.2.3 Scalarisation, Mise en registres (reg)

Comme vu précédemment, des opérations sur des Load en mémoire est très coûteux, ici, à chaque itération, nous faisons 9 Load. Un moyen de palier à ce problème serait la mise en registre des cases chargées. Ainsi, le compilateur pourra effectué des calculs directement avec des registres. Pour cela, on va séparer le code en 3 parties:

Load: on fait une lecture mémoire des 9 cases que l'on met en variable locale,
Calcul: On calcule le minimum de ses 9 cases,
Store : On écrit en mémoire le résultat.

Dans la phase de calcul, en admettant que les 9 variables locales sont mis en registre, le processeur n'a besoin de faire que le AND entre 9 registres:

[5] décompilé depuis RISC-V rv64gc gcc 8.2.0 en -O3

```
// calc
        y = MIN9(
            a00, a01, a02,
            a10, a11, a12,
            a20, a21, a22
            );
```

```
lbu     t1,-1(a1)
lbu     a5,-1(a4)
lbu     a7,-1(a3)
lbu     a6,0(a4)
lbu     a0,0(a1)
and     a5,a5,t1
and     a5,a5,a7
lbu     t1,0(a3)
lbu     a7,1(a4)
and     a5,a5,a6
and     a5,a5,a0
lbu     a6,1(a1)
lbu     a0,1(a3)
and     a5,a5,t1
and     a5,a5,a7
and     a5,a5,a6
and     a5,a5,a0
sb      a5,0(a2)
addi    a4,a4,1
addi    a1,a1,1
addi    a3,a3,1
addi    a2,a2,1
```

La mise en registre nous permet aussi de repérer plus facilement les optimisations et factorisations possibles.

3.2.4 Rotation de registre (rot)

On remarque dans la version par mise en registre que d'une itération à une autre, on conserve 2 colonnes or le chargement depuis la mémoire (ou le cache) à un coût très élevé ; on veut donc réduire leur nombre.
Ainsi en utilisant la scalarisation, on va effectuer une rotation de registres, cela signifie que nous allons Load les 2 premières cases de chaque ligne avant de rentrer dans la boucle pour ensuite faire une rotation avec la 3e colonne Load dans la boucle.
On passe, ainsi dans la boucle intérieure, de 9 Load à seulement 3.

(1)

x	x	x
x	x	x
x	x	x

(2)

x	x	x
x	x	x
x	x	x

(1) *les colonnes, en rouge et bleu, est Load avant l'entrée de boucle et en orange au début du premier tour de boucle.*
(2) *A la fin du premier tour de boucle, on fait un rotation,ce qui était rouge devient bleu, ce qui était bleu deviens orange et la nouvelle colonne qu'on a Load dans cet nouvelle itération est en rouge.*

3.2.5 Réduction de colonnes (red)

La rotation de registre nous a permis de réduire le nombre de Load, on veut maintenant réduire le nombre d'opérations et pour cela, on va pré-calculer le minimum entre chaque cases des 2 première colonnes avant de rentrer dans la boucle pour ensuite faire une rotation de colonnes avec la 3e colonne calculée dans la boucle.

a01	a11	a21
a02	a12	a22
a03	a13	a23

c1 = MIN3(a01,a02,a03)
c2 = MIN3(a11,a12,a13)
c3 = MIN3(a21,a22,a23)

On calcule c1 et c2 avant l'entrée de boucle et c3 en entrée de boucle et en fin de chaque itération, on fait une rotation de colonnes, c1 = c2 et c2 = c3. Cependant, ces assignations restent coûteuses.
En assembleur[6] :

```
86  // load des nouvelles cases
87  a02 = load1(a0, j + 1);
88  a12 = load1(a1, j + 1);
89  a22 = load1(a2, j + 1);
90
91  // calc
92  y = MIN9(
93      a00, a01, a02,
94      a10, a11, a12,
95      a20, a21, a22
96      );
97
98  // store
99  store1(Y[i], j, y);
100
101 // rotation de registre
102 a00 = a01;
103 a10 = a11;
104 a20 = a21;
105
106 a01 = a02;
107 a11 = a12;
108 a21 = a22;
```

```
109
110 add     a5,t2,a2
111 lbu     t1,1(a5)
112 add     a5,t0,a2
113 lbu     a7,1(a5)
114 add     a6,t6,a2
115 and     a5,a0,a1
116 lbu     a6,1(a6)
117
118 # Enchainement de 7 AND
119 # grace a la mise en registre
120 and     a5,a4,a5
121 and     a5,t1,a5
122 and     a5,a7,a5
123 and     a5,a6,a5
124 and     a5,t5,a5
125 and     a5,t4,a5
126 and     a5,t3,a5
127 add     t4,s0,a2
128 addi    a2,a2,1
129 sb      a5,0(t4)
130 sext.w  a5,a2
131
132 # l'effet des rotations
133 mv      t5,a0
134 mv      t4,a1
135 mv      t3,a4
136 mv      a0,t1
137 mv      a1,a7
138 mv      a4,a6
```

Cette optimisation permet de faire baisser à la fois la complexité et le nombre d'accès mémoire mais demande une rotation de colonne à chaque itération dont on peut voir la répercussion en assembleur.

3.2.6 Déroulage de boucles interne de dégrée 3 (ilu3)

On va maintenant faire un déroulage de boucle d'ordre 3 dans la boucle interne. Ce déroulage ne permet pas forcément un gain de performance significatif mais ouvre la possibilité à des optimisations plus intéressantes. Le déroulage de boucle permet aussi de réduire le nombre de saut et d'évaluation de la condition de branchement ($j <= j1$). Ci-dessous : la décompilation d'un for "classique".

[6] décompilé depuis RISC-V rv64gc gcc 8.2.0 en -O3

```
139 for(int j=j0; j<=j1; j++);
```

```
1  .L1
2  # cette partie n'est appelee
3  # qu'une fois au debut du for
4  lw       a5,-48(s0)
5  sw       a5,-20(s0) # j = j0
6  j        .L2
7
8  .L3
9  # corps du boucle
10 # incr mentation du compteur j
11 lw       a5,-20(s0)
12 addi     a5,a5,1
13 sw       a5,-20(s0) # j++
14
15 .L2
16 # condition de branchement
17 lw       a4,-20(s0) # j
18 lw       a5,-52(s0) # j1
19 ble      a4,a5,.L3  # j<=j1
```

Le déroulage de boucle en lui même est effectué en itérant de 3 à 3 et recopiant 3 fois l'itération dans cette nouvelle boucle. Si l'intervalle parcourut par j n'est pas un multiple de 3 alors on ajoute un prologue permettant de calculer les cases restantes.

3.2.7 Déroulage de boucle interne de dégrée 3 et réduction de colonnes (ilu3_red)

Comme expliqué précédemment, on va combiner déroulage de boucle interne de degré 3 et réduction de colonnes. Pour ce faire, on va tout d'abord avant de rentrer dans la boucle pré-calculée les 2 premières colonnes et déterminer l'intervalle de j qui est incrémenté de 3 en 3. Ensuite, à chaque tour de boucle, on calcul 3 cases par 3 cases de la ligne en faisant une rotation manuelle des colonnes sans assignations.

Exemple d'un tour de boucle :

(1)

a00	a01	a02
a10	a11	a12
a20	a21	a22

(2)

a00	a01	a02
a10	a11	a12
a20	a21	a22

(3)

a00	a01	a02
a10	a11	a12
a20	a21	a22

(1) *On calcul* ***avant la boucle*** *:*

c1=MIN3(a00,a10,a20) *c2=MIN3(a01,a11,a21)*

Une fois j déterminé, on rentre dans la boucle et en 1 tour de boucle on fait toutes les étapes suivantes :

c3= MIN3(a02,a12,a22) ***y = MIN3 (c1,c2,c3)***

Avant de calculer (2), on fait la rotation de colonnes : ***c1=c2 et c2=c3***

(2) *Grâce à cet rotation, on a déjà les colonnes c1 et c2, il suffit alors seulement de calculer la nouvelle colonne c3, on refait ensuite une rotation de colonnes avant de passer à* ***(3)****.*

Grâce à ilu3_red, on a pu supprimer complètement les rotations de variables qu'on avait dans red. De plus, le déroulage de boucle divise par trois le nombre de saut et de calcul de condition.

3.2.8 Déroulage de boucle externe de dégrée 2 et réduction de colonnes (elu2_red)

On va s'intéresser maintenant au déroulage de boucle non plus interne mais externe de degré 2 avec une réduction de colonnes. Pour réaliser cela, nous partons de la remarque qu'à chaque itération ; nous recalculons 2 colonnes déjà charger dans l'itération précédente. Ainsi, on va rajouter une ligne pour avoir une matrice 4x3, et faire 2 écritures mémoire par tour de boucle pour 2 lignes. On recopie l'indice de boucle à l'extérieur de la boucle et on déplace les chargements et les calculs des 2 premières colonnes des 2 lignes de l'intérieur à l'extérieur de la boucle, c'est le prologue. Une fois rentré dans la boucle, a chaque itération, on Load la 3e colonne des 2 lignes, on calcule et on store les **2 cases dans Y** correspondant aux 2 lignes. A la fin de chaque itération, on fait la rotation de colonnes.

(1)

a00	a01	a02
a10	a11	a12
a20	a21	a22
a30	a31	a32

(2.1)

a00	a01	a02
a10	a11	a12
a20	a21	a22
a30	a31	a32

(2.2)

a00	a01	a02
a10	a11	a12
a20	a21	a22
a30	a31	a32

(1) *en rouge, la nouvelle ligne ajoutés en elu2*
(2.1) *en rouge, les 9 cases nécessaires pour calculer la ligne i*
(2.2) *en bleu, les 9 cases nécessaires pour calculer la ligne i+1*

3.2.9 Déroulage de boucle externe de dégrée 2 et réduction de colonnes en version factorisée (elu2_red_factor)

On remarque que dans les schémas (2.1) et (2.2) de l'optimisation en **3.2.8** on aura toujours 6 cases en commun, $[a10, a20, a11, a21, a12, a22]$. Pour cela, on va créer une variable temporaire **f**, qui va nous permettre de réduire le nombre de calcul en lui donnant pour valeur le MIN des 2 cases de chaque colonnes.

```
140     // factorisation du prologue
141     f   = MIN(a10, a20);
142     c00 = MIN(a00, f ); c01 = MIN(f , a30);
143     f   = MIN(a11, a21);
144     c10 = MIN(a01, f ); c11 = MIN(f , a31);
145
146     // factorisation dans la boucle
147     f = MIN(a12, a22);
148     c20 = MIN(a02, f); c21 = MIN(f, a32);
```

Voyons maintenant l'impact de la factorisation. Ci-dessous 2 codes assembleurs correspondant au calcul de c20 et c21 :
Sans factorisation :

```
and     a1,a4,a1
and     a1,a5,a1
and     a5,a4,a5
and     a5,a5,a3
```

Avec factorisation :

```
and     a5,a5,a3
and     a4,a5,a4
and     a5,a5,a1
```

3.2.10 Déroulage de boucle interne de dégrée 3, déroulage de boucle externe de dégrée 2 et réduction de colonnes (ilu3_elu2_red)

L'optimisation précédente nous a permis de réduire le nombre d'opération par point mais à faire resurgir le problème qu'on avait avec red à savoir les rotations de variable à chaque itération dont on avait trouvé la solution en effectuant un ilu3. Ainsi, nous obtenons cette optimisation qui regroupe 2 optimisations que nous avons déjà implémenté séparément **ilu3_red** qui permet de calculer 3 cases de la ligne par tour de boucle et **elu2_red** qui permet de calculer à chaque itération 1 case de 2 lignes.
En regroupant les 2 méthodes, on va dans le **prologue** calculer les 2 premières colonnes de chaque ligne pour satisfaire elu2, et déterminer l'intervalle de j pour ilu3. On suppose ici qu'on rentre dans la boucle. A chaque itération, on va donc calculer les 3 cases des 2 lignes, sans oublier la rotation des colonnes entre chaque case calculée.

Exemple d'un tour de boucle :

(1)

a00	a01	a02
a10	**a11**	**a12**
a20	**a21**	**a22**
a30	**a31**	a32

(2)

a00	a01	a02
a10	**a11**	**a12**
a20	**a21**	**a22**
a30	a31	**a32**

(3)

a00	a01	a02
a10	**a11**	**a12**
a20	**a21**	**a22**
a30	**a31**	**a32**

(1) *Ce qui est en rouge et bleu correspond à la matrice 3x3 de la 1ère ligne et ce qui est en* ***gras*** *de la 2e ligne. On calcule les 2 premières colonnes dans le prologue et la 3e au début de la boucle.*
(2) *Toujours dans la boucle, on fait la rotation de colonnes, c01 = c02, c11 = c12 et c02 = c03, c12 = c13 manuellement et on calcule la 3e colonne.*
(3) *On refait la même chose avant de sortir de la boucle.*

3.2.11 Déroulage de boucle interne de dégrée 3,déroulage de boucle externe de dégrée 2 et réduction de colonnes en version factorisée (il3_elu2_red_factor)

Pour factorisé cette dernière optimisation, le principe est le même que pour le **3.2.9**. Cette factorisation

3.2.12 Dilatation

La Dilatation, avec un élément structurant de taille 3x3, est l'inverse de l'érosion, au lieu de faire disparaître des pixels, elle en rajoute.

Pour procéder a cette ajout on va d'abord rajouter un bord égale a 0 et pour chaque ligne on va faire case par case le maximum de la case avec les 8 cases qui l'entourent.

1	1	1
1	1	0
1	1	1

en ajoutant les bords cela devient :

0	**0**	**0**	**0**	**0**
0	1	1	1	**0**
0	1	1	0	**0**
0	1	1	1	**0**
0	**0**	**0**	**0**	**0**

*On peut ensuite faire le **MAX** des 9 cases pour chaque case de chaque ligne*:

0	**0**	**0**	0	0
0	1	**1**	1	0
0	**1**	**1**	0	0
0	1	1	1	0
0	0	0	0	0

-

0	**0**	**0**	**0**	0
0	**1**	1	**1**	0
0	**1**	**1**	**0**	0
0	1	1	1	0
0	0	0	0	0

-

0	0	**0**	**0**	**0**
0	1	**1**	1	**0**
0	1	**1**	**0**	**0**
0	1	1	1	0
0	0	0	0	0

On obtiens finalement comme matrice, le résultat suivant :

1	1	1
1	1	1
1	1	1

Figure. Explication de Dilatation.

Toutes les fonctions ont été implémentées dans **morpho_dilatation.c**. Aussi, toutes les optimisations vu pour dilatation sont les mêmes que pour érosion, le principe est le même à la différence de mettre un **MAX** au lieu d'un MIN. Le gain en performance reste donc la même.

3.3 Ouverture

A partir des 2 opérateurs vu précédemment, **Érosion** et **Dilatation** il est possible d'en créer deux autres : **Fermeture** et **Ouverture** qui conservent la taille (discrète) des régions.
Dans notre cas, nous allons nous intéresser juste au deuxième : l'ouverture, qui permet d'agrandir les trous a l'intérieur des composantes connexes (si leur rayon est inférieur à celui de l'élément structurant).

En partant d'une matrice source (X), on applique une **Erosion**, que l'ont stock dans une matrice (Y) temporaire et où on appliquera une **Dilatation** pour obtenir le résultat de notre **Ouverture** dans (Z).

```
149 // ------------------------------------------------------------
150 void ouverture3_ui8matrix_basic(uint8 **X, int i0, int i1, int j0, int j1, uint8 **Y, uint8
        **Z)
151 // ------------------------------------------------------------
152 {
153     erosion3_ui8matrix_basic(X, i0-1, i1+1, j0-1, j1+1, Y);
154     dilatation3_ui8matrix_basic(Y, i0,   i1,   j0,   j1,   Z);
155 }
```

(1)

0	**0**	**0**	**0**	**0**
0	1	1	1	**0**
0	1	1	1	**0**
0	1	1	1	**0**
0	**0**	**0**	**0**	**0**

(2)

0	0	0
0	1	0
0	0	0

(3)

1

(1) *Matrice 5x5 avant ouverture,*
(2) *Matrice temporaire après Érosion,*
(3) *Case obtenu après Dilatation sur la matrice temporaire, elle correspond au résultat de ouverture sur (1).*
Les deux fonctions erosion et dilation "cachent" 2 boucles imbriqués. Ainsi, on remarque déjà que cet version naïve parcours 2 fois les même cases.

Toutes les fonctions sont implémentées dans **ouverture.c**, l'optimisation de **Ouverture** passe principalement par l'élimination d'une des deux boucles imbriqués. Pour cela, nous allons principalement voir les deux optimisations suivantes : **Pipeline d'opérateurs** et **Fusion d'opérateurs** auxquels nous leur rajouterons les optimisations déjà vu en **3.2**.

3.3.1 Fusions d'opérateurs

Fusion d'opérateurs comme son nom l'indique consiste à **fusionner** en une boucle line_erosion et line_dilatation. C'est à dire que à chaque itération ; on va à partir d'une matrice de taille 5x5 de la source (uint8 **X) calculer et stocker dans des variables locales le minimum par convolution 3x3, ce qui va nous permettre d'obtenir une matrice 3x3 et sur cette matrice, toujours dans la même itération, on peut maintenant à partir de ces 9 valeurs intermédiaires obtenues calculer la case courante en calculant le maximum entre ces 9 valeurs.

(1)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(t00)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(t01)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(t02)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(t10)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(t11)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(t12)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(t20)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(t21)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(t22)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(2)

0	0	0
0	1	0
0	0	0

(3) **1**

(1) *Matrice 5x5 sur laquelle on va appliquer les opérations.*
(2) *Matrice 3x3 obtenu suite aux MIN9 des cases colorés pour (t00)(t01)(t02) (t10)(t11)(t12) (t20)(t21)(t22)*
(3)*Case obtenue à la fin de l'itération suite au MAX du (2)*

3.3.2 Fusion Basique

La version basique de cette méthode sera notre version de référence. Elle consiste à réécrire naïvement le principe de **3.3.8**.

3.3.3 Fusion et Réduction de colonnes(fusion_red)

La réduction de colonnes a été vu précédemment en **3.2.5**, il suffit de l'appliquer à la fusion et cela consiste à pré-calculé (t00)(t01), (t10)(t11) et (t20)(t21) avant l'entrée de boucle et a chaque itération recalculé t02, t12 et t22. En fin d'itération, on oublie pas de faire une **rotation** de colonnes et de var locales.

3.3.4 Fusion avec déroulage de boucle interne de degré 5 et réduction de colonnes(fusion_ilu5_red)

Contrairement au **3.2.7** qui fait référence à ilu3_red, nous avons, ici, un déroulage de boucle interne de degré 5 et non de degré 3 mais le principe reste le même.
Dans le **prologue**, on va calculer les 4 premières colonnes qui vont nous permettre de calculer les variable temporaires pour l'érosion, on détermine aussi l'intervalle de j.
Si on rentre dans la boucle, a **chaque itération** on commence par déterminer la dernière colonne de la matrice 5x5 pour obtenir les 3 dernières variables locales et ensuite faire le MAX9, puis on fait la rotation de colonnes et des variables locales (temporaires), on répète ce processus 4 fois de suite car déroulage interne de degré 5.
Si l'intervalle parcourut par j n'est pas un multiple de 5 alors on ajoute un prologue permettant de calculer les cases restantes.

3.3.5 Fusion avec déroulage de boucle interne de degré 15 et réduction de colonnes(fusion_ilu15_red)

Ici, on fait la même chose que vu plus haut (ilu5_red) mais au lieu de faire 5 fois le processus dans la boucle on le fera **15 fois** car déroulage de boucle interne de degré 15.

3.3.6 Fusion avec déroulage de boucle externe de degré 2 et réduction de colonnes (fusion_elu2_red)

Pour obtenir un déroulage de boucle externe de degré 2, on doit rajouter une ligne pour avoir une matrice 6x5 et non plus faire une écriture mémoire mais **deux** par itération pour deux lignes.

(1)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0
0	0	0	0	0

Nouvelle matrice 5x6

Cela implique, que nous devons rajouter 3 variables locales de plus.

(t30)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0
0	0	0	0	0

(t31)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0
0	0	0	0	0

(t32)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0
0	0	0	0	0

(t30) et **(t31)** vont être calculés dans le prologue avec les 6 autres variables locales, et (t32) va être calculé à chaque itération. On aura donc 2 matrice 3x3 obtenue suite aux MIN9.

(2.1)

0	0	0
0	1	0
0	0	0

(2.2)

0	1	0
0	0	0
0	0	0

```
156             y0 = MAX9(  t00, t01, t02,     | y1 = MAX9(  t10, t11, t12,
157                         t10, t11, t12,     |             t20, t21, t22,
158                         t20, t21, t22);    |             t30, t31, t32);
```

(3.1) **1** (3.2) **1**

(2.1) *et* ***(2.2)*** *correspondent aux 2 matrices obtenus a la suite des deux MIN9.*
(3.1) *et* ***(3.2)*** *correspondent aux 2 cases obtenue des 2 lignes a la fin de chaque itération.*

3.3.7 Fusion avec déroulage de boucle interne de degré 5, déroulage de boucle externe de degré 2 et réduction de colonnes (fusion_ilu5_elu2_red)

Dernière optimisation pour fusion, celle de fusion_ilu5_elu2_red. Avec tout ce qu'on a déjà vu, cela n'est pas bien compliqué, on commence avec **fusion_ilu5_red** et sur cette fonction on implémente la méthode de elu_2. On rajoute une ligne supplémentaire pour avoir une matrice 6x5 et on ajoute un deuxième registre pour un accès mémoire vers la deuxième ligne.
En rajoutant tout ça, nous devons changer en conséquence, a savoir dans le prologue, calculer les 4 colonnes et les 2 variables locales supplémentaire. L'intervalle de j ne change pas.
A chaque tour de boucle, nous allons donc calculer les deux dernières colonnes et une variable locale en conséquence pour ainsi calculé les cases de chaque ligne sans oublier de faire la rotation aussi sur les colonnes et les variables locales ajoutées.

3.3.8 Fusion avec déroulage de boucle interne de degré 5, déroulage de boucle externe de degré 2 et réduction de colonnes factorisé (fusion_ilu5_elu2_red_factor)

Si on reprend nos schémas vu en **3.3.1** et en **3.3.6** (ilu5_red en lui rajoutant elu2_red) on remarque deux choses :

- D'une case a une autre, on a 6 cases en commun :

(t00)
0	**0**	**0**	0	0
0	**1**	**1**	1	0
0	**1**	**1**	1	0
0	1	1	1	0
0	0	0	0	0

(t01)
0	**0**	**0**	0	0
0	**1**	**1**	1	0
0	**1**	**1**	1	0
0	1	1	1	0
0	0	0	0	0

en **gras** les cases en commun

- d'une matrice a une autre on a aussi 6 cases en commun

(t00)
0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(t10)
0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

En jaune les cases en communs.

Pour cela, on va utiliser une variable f qui va nous permettre de factoriser en prenant la valeur du MIN des 2 cases en commun par colonnes :

:

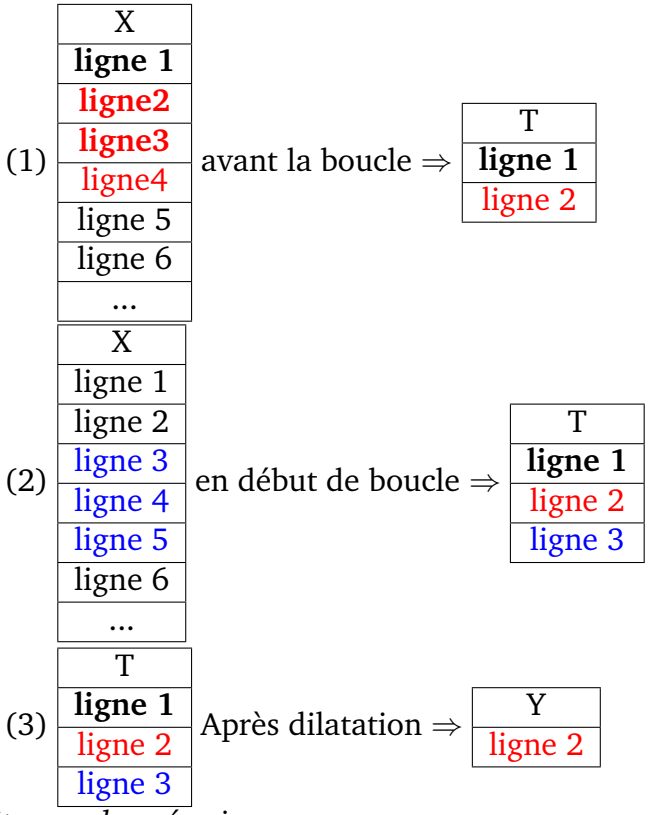
```
163     f   = MIN(  a10, a20);
164     c00 = MIN(  a00, f);
165     c10 = MIN(  f, a30);
166
167     f   = MIN(  a11, a21);
168     c01 = MIN(  a01, f);
169     c11 = MIN(  f, a31);
170
171     f   = MIN(  a12, a22);
172     c02 = MIN(  a02, f);
173     c12 = MIN(  f, a32);
174
175     f   = MIN(  a13, a23);
176     c03 = MIN(  a03, f);
177     c13 = MIN(  f, a33);
```

et ensuite du MIN des 2 colonnes en commun :

```
178     f   = MIN(  c01, c02);
179     t00 = MIN(  c00, f);
180     t01 = MIN(  f, c03);
181
182     f   = MIN(  c11, c12);
183     t10 = MIN(  c10, f);
184     t11 = MIN(  f, c13);
```

3.3.9 Pipeline d'opérateurs

Parfois il est impossible de faire de la fusion d'opérateurs, on réalise alors un **Pipeline d'opérateurs**, c'est moins efficace qu'une fusion mais ça améliore la persistance des données en cache donc cela reste très performant. On suppose ici que les boucles internes ne sont pas fusionnables ou contre productive.
Pour réaliser cette optimisation, on va commencer par les 2 premiers lignes sur lesquelles on applique érosion (car on veut une ouverture) avant d'entrer dans la boucle. On obtient ainsi deux lignes qu'on va écrire dans une matrice intermédiaire **T** à l'aide des fonctions **line_erosion** présent dans morpho_erosion.c.
Ensuite, une fois dans la boucle externe, on refait une érosion pour obtenir une troisième ligne dans T et une fois les trois lignes dans T on lui applique une dilatation pour obtenir le résultat voulu.



(1)*Avant la boucle, suite aux deux érosion*
(2) *3e ligne de T avec line_erosion*
(3) *Résultat dans Y a la fin de chaque itération.*

3.3.10 Pipeline Basique

Première version de pipeline, la version basique, une application naïve du **3.3.9**.

```
185 // ------------------------------------------------------------
186 void ouverture3_ui8matrix_pipeline_basic(uint8 **X, int i0, int i1, int j0, int j1, uint8
        **T, uint8 **Y)
187 // ------------------------------------------------------------
188 {
```

```
189     int i;
190     int b = 1;
191     i =i0-b;
192     line_erosion3_ui8matrix_basic(X, i   , j0-b, j1+b, T);
193     line_erosion3_ui8matrix_basic(X, i+1, j0-b, j1+b, T);
194     for(i=i0; i<=i1; i++)
195     {
196         line_erosion3_ui8matrix_basic(X, i+1, j0-b   , j1+b   , T);
197         line_dilatation3_ui8matrix_basic(T, i   , j0-b+1, j1+b-1, Y);
198     }
199 }
```

3.3.11 Pipeline et Réduction de colonnes(pipeline_red)

Pour l'optimisation des réduction de colonnes appliqué au pipeline, il suffit de recopier le code précèdent vu en **3.3.10** dans la version basique et de remplacer :
- line_erosion3_ui8matrix_basic par **line_erosion3_ui8matrix_red**
- line_dilatation3_ui8matrix_basic par **line_dilatation3_ui8matrix_red**

3.3.12 Pipeline avec déroulage de boucle interne de degré 3 et réduction de colonnes(pipeline_ilu3_red)

On part du même principe que pour pipeline_red, on remplace simplement :
- line_erosion3_ui8matrix_basic par **line_erosion3_ui8matrix_ilu3_red**
- line_dilatation3_ui8matrix_basic par **line_dilatation3_ui8matrix_ilu3_red**

3.3.13 Pipeline avec déroulage de boucle externe de degré 2 et réduction de colonnes (pipeline_elu2_red)

La technique est toujours la même sauf que pour un déroulage de boucle **externe** de degré 2 on doit en plus déterminer l'intervalle de j et rajouter un épilogue si jamais on rentre pas dans la boucle.

3.4 Subword Parallelism (SWP)

Jusque là, nous utilisions le bit de poids faibles d'un uint8 pour encoder une case or les opérateurs logiques que nous utilisons sont des opérateurs bits à bits. Cela signifie que nous n'exploitons pas 7 calculs sur 8. Il serait donc intéressant de faire un paquetage de nos uint8 dans des valeurs de 8, 16, 32 ou encore 64 bits.

Voyons comment cela est possible avec l'exemple d'un paquetage sur 8 bits dans une architecture little-endian:

Soit 3 lignes paquettés dont on veut calculé le minimum :

$X[i][j-1]$	a7	a6	a5	a4	a3	a2	a1	a0
$X[i][j+0]$	b7	b6	b5	b4	b3	b2	b1	b0
$X[i][j+1]$	c7	c6	c5	c4	c3	c2	c1	c0

Réarrangeons maintenant ces bits à l'aide des macros shifts vu plus haut :

b6	b5	b4	b3	b2	b1	b0	a7
b7	b6	b5	b4	b3	b2	b1	b0
c0	b7	b6	b5	b4	b3	b2	b1

Lorsqu'on va maintenant appliquer un AND ou un OR bit à bit, la valeur obtenue encodera le résultat de l'opérateur pour chaque colonne.
Dans notre cas nous avons implémenté juste les fonctions de **swp_64**. Pour passer à swp_8 ou swp_32, il suffit tout simplement de faire une édition de texte par remplacement. Cependant, le débogage quant à lui est difficile notamment à cause des dépassements (Overflow) engendrés par les macros opérateurs en mode debug qui deviennent des additions. Pour simplifier le débogage, il est judicieux de commencer par SWP_64 en faisant des paquets de 8 ou 16.

3.4.1 SWP_64 Érosion et SWP_64 Dilatation

Tout d'abord, pour swp_64 le principe est le même que avec 8 bits et les figures vu plus haut, on décale de 63 bits au lieu de 7 bits.
Pour ce qui est Érosion ou Dilatation, la seule différence avec ce qui a été vu plus tôt sont **les décalages a gauche et à droite de la case centrale avec la case de gauche et celle de droite respectivement.**
Exemple : Prenons le cas d'une matrice 3x3.

a00	a01	a02
a10	a11	a12
a20	a21	a22

on lui applique les décalages, on obtiens la nouvelle matrice :

l0 = LEFT_64(a00,a01)	a01	r0 = RIGHT_64(a01,a02)
l1 = LEFT_64(a10,a11)	a11	r1 = RIGHT_64(a11,a12)
l2 = LEFT_64(a20,a21)	a21	r2 = RIGHT_64(a21,a22)

C'est sur cette matrice qu'on va faire le MIN9 pour Érosion et MAX9 pour Dilatation. Voici le code pour la version line basic de érosion en version basique : (mettre MAX au lieu de MIN pour dilatation)

```
200 // ------------------------------------------------------------------------------
201 void line_erosion3_ui64matrix_swp64_basic(uint64 **X, int i, int j0, int j1, uint64 **Y)
202 // ------------------------------------------------------------------------------
203 {
204     int j;
205     for (j = j0; j <= j1; j++){
206         Y[i][j]=MIN9(
207                 LEFT_64(X[i-1][j-1], X[i-1][j]),X[i-1][j], RIGHT_64(X[i-1][j],X[i-1][j+1]),
208                 LEFT_64(X[i  ][j-1], X[i  ][j]),X[i  ][j], RIGHT_64(X[i  ][j],X[i  ][j+1]),
209                 LEFT_64(X[i+1][j-1], X[i+1][j]),X[i+1][j], RIGHT_64(X[i+1][j],X[i+1][j+1])
210                 );
211     }
212 }
```

Pour ce qui est des optimisations, le principe reste inchangé a **la seule différence** que nous devons toujours garder en tête que les cases a gauche et a droite de la base sont obtenues par les décalages.

3.4.2 Ouverture SWP_64 (fusion et pipeline)

Maintenant que nous avons implémenté érosion et dilatation swp_64 nous pouvons faire ouverture. Tout d'abord, pour **Pipeline d'opérateurs**, il suffit tout simplement de **copier coller** les fonctions sans swp et de changer le noms des fonctions d'érosion et de dilatation avec celles de swp_64, il n'y a aucun changement.
Pour fusion, cela n'est pas aussi simple, comme il est écrit plus haut nous devons toujours garder en tête que les cases a gauche et a droite de la base sont obtenues par les décalages.
C'est a dire, si nous appliquons le principe de **fusion** en prenant en compte les décalages,pour chaque tout de boucle, on trouve pour la première ligne:

Les 3 variables locales :

(t00)

l0	0	r0	0	0
l1	1	r1	1	0
l2	1	r2	1	0
0	1	1	1	0
0	0	0	0	0

(t01)

0	l0	0	r0	0
0	l1	1	r1	0
0	l2	1	r2	0
0	1	1	1	0
0	0	0	0	0

(t02)

0	0	l0	0	r0
0	1	l1	1	r1
0	1	l2	1	r2
0	1	1	1	0
0	0	0	0	0

Ensuite on fait l0 = LEFT_64(t00,t01) et r0 = RIGHT_64(t01,t02)
en faisant cela pour les 3 lignes on obtiens finalement la matrice 3x3 :

l0	t01	r0
l1	t11	r1
l2	t21	r2

4 Résultats

Après l'implémentation de ces algorithmes avec leurs différents optimisations, étudions leur complexité.

4.1 Résultats théoriques

4.1.1 Erosion - Dilatation :

Version	MIN ou MAX	Load + Store	OPparPoint	ratio
basique	8	(2*9) + 2 = 20	28/1	18:2 = 9:1
reg	8	9 + 1 = 10	18/1	9:1
rot	8	3 + 1 = 4	12/1	3:1
red	4	3 + 1 = 4	8/1	3:1
ilu3	27	5 + 3 = 8	35/3 = 12	5:3
ilu3_red	12	9 + 3 = 12	24/3 = 8	9:3 = 3:1
elu2_red	8	4 + 2 = 6	14/2 = 7	4:2 = 2:1
elu2_red_factor	7	4 + 2 = 6	13/2 = 6.5	4:2 = 2:1
ilu3_elu2_red	24	12 + 6 = 18	42/6 = 7	12:6 = 2:1
ilu3_elu2_red_factor	21	12 + 6 = 18	39/6 = 6.5	12:6 = 2:1
basique_swp64	8	(2*9) + 2 = 20	28/64 = 0.45	9:64
reg_swp64	8	9 + 1 = 10	18/64 = 0.3	9:64
rot_swp64	8	3 + 1 = 4	12/64 = 0.2	3:64
red_swp64	4	3 + 1 = 4	8/64 = 0.125	3:64
ilu3_swp64	27	5 + 3 = 8	35/192 = 0.18	5:192
ilu3_red_swp64	12	9 + 3 = 12	24/192 = 0.125	3:64
elu2_red_swp64	8	4 + 2 = 6	14/128 = 0.11	2:64 = 1:32
elu2_red_factor_swp64	7	4 + 2 = 6	13/128 = 0.1	2:64 = 1:32
ilu3_elu2_red_swp64	24	12 + 6 = 18	42/384 = 0.11	2:64 = 1:32
ilu3_elu2_red_factor_swp64	21	12 + 6 = 18	39/384 = 0.1	2:64 = 1:32

Interprétation : La première optimisation significative intervient pour la version réduction de colonne. Le nombre d'opération logique effectué dans la boucle est divisé par 2 par rapport à la version basique. De plus, le nombre de load quant à lui est divisé par 3. Ce qui permet d'avoir 8 opérations significatives par point. La prochaine amélioration impactant arrive pour elu2_red_factor, où le nombre d'opération par point atteint 6.5 qui partage ce palier avec ilu3_elu2_red_factor sauf que comme vu précédemment, dans la partie érosion le déroulage de boucle interne permet de réduire le nombre de saut et d'éliminer les rotations de variable.
De même, l'interprétation au dessus s'applique pour la partie SWP 64. Cependant, ici, chaque store nous permet de produire 64 points. Ainsi, ilu3_elu2_red_factor_swp64 fait en moyenne 0.1 opérations par point.

4.1.2 Ouverture - Fermeture :

Pipeline :

Version	MIN ou MAX	Load + Store	OPparPoint	ratio
basique	2*8 = 16	2*18 + 2*2 = 40	56/1	18:1
red	2*4 = 8	2*3 + 2*1 = 8	16/1	6:1
ilu3_red	2*12 = 24	2*9 + 2*3 = 24	48/1	18:3 = 6:1
elu2_red	2*8 = 16	2*4 + 2*2 = 12	28/2 = 14	8:2 = 4:1
ilu3_elu2_red	2*24 = 48	2*12 + 2*6 = 36	84/6 = 14	24:6 = 4:1
ilu3_elu2_red_factor	2*21 = 42	2*12 + 2*6 = 36	78/6 = 13	24:6 = 4:1
basique_swp64	2*8 = 16	2*18 + 2*2 = 40	56/64 = 0.875	18:64 = 9:32
red_swp64	2*4 = 8	2*3 + 2*1 = 8	16/64 = 0.25	6:64 = 3:32
ilu3_red_swp64	2*12 = 24	2*9 + 2*3 = 24	48/64 = 0.75	18:64 = 9/32
elu2_red_swp64	2*8 = 16	2*4 + 2*2 = 12	28/128 = 0.22	4:64 = 1:16
elu2_red_factor_swp64	2*7 = 14	2*4 + 2*2 = 12	26/128 = 0.2	4:64 = 1:16
ilu3_elu2_red_swp64	2*24 = 48	2*12 + 2*6 = 36	84/384 = 0.22	4:64 = 1:16
ilu3_elu2_red_fac_swp64	2*21 = 42	2*12 + 2*6 = 36	78/384 = 0.2	4:64 = 1:16

Fusion :

Version	MIN ou MAX	Load + Store	OPparPoint	ratio
basique	9*8 + 8 = 80	9*9 + 1 = 82	162/1	81:1
red	6*2 + 8 = 20	5 + 1 = 6	26/1	5:1
ilu5_red	(6*2 +8)*5 = 100	5*5 + 5 = 30	130/5 = 26	25:5 = 5:1
elu2_red	4*2*2 + 2 * 8 = 32	6 + 2 = 8	40/2 = 20	8:2 = 4:1
elu2_red_factor	2*7 = 14	2*4 + 2*2 = 12	26/2 = 13	8:2 = 4:1
ilu5_elu2_red	5*(2*8 + 2*8) = 160	5*(2*12 + 2*6) = 180	340/10 = 34	120:60 = 2:1
ilu5_elu2_red_factor	5*(6+8+5+3+3) = 125	5*(6 + 2) = 40	165/10 = 16.5	30:10 = 3:1
basique_SWP64	9*8 + 8 = 80	9*9 + 1 = 82	162/64 = 2.5	81:64
red_SWP64	6*2 + 8 = 20	5 + 1 = 6	26/64 = 0.4	5:64
ilu5_red_SWP64	(6*2 +8)*5 = 100	5*5 + 5 = 30	130/320 = 0.4	5:64
elu2_red_SWP64	4*2*2 + 2 * 8 = 32	6 + 2 = 8	40/124 = 0.32	4:64 = 1:16
elu2_red_factor_SWP64	2*7 = 14	2*4 + 2*2 = 12	26/124 = 0.2	4:64 = 1:16
ilu5_elu2_red_SWP64	5*(2*8 + 2*8) = 160	5*(2*12 + 2*6) = 180	340/640 = 0.5	1:32
ilu5_elu2_red_factor_SWP64	5*(6+8+5+3+3) = 125	5*(6 + 2) = 40	165/640 = 0.26	3:64

Interprétation :

les interprétations précédentes se tiennent aussi dans le cas d'ouverture et fermeture. Cependant, il est intéressant de comparer la version pipeline et fusion. En effet, en se basant seulement sur les opérations par point. Il serait tentant de dire que pipeline est plus rapide que fusion mais fusion fait beaucoup moins de load par point. Cela signifie que dépendant de plusieurs facteurs comme la taille du bus de donnée, la taille du cache etc... Pipeline pourrait être plus rapide que fusion et vice versa. Dans les ordinateurs modernes, les bus de donnée acceptent des flux relativement larges ce qui signifierait que pipeline serait plus rapide mais ce n'est pas forcément vrai dans un système embarqué avec un architecture plus restreint et moins énergivore.

4.2 Résultats expérimentals

4.2.1 Configuration et environnement

CPU :
Intel(R) Core(TM) i5-8264U CPU @ 1.60Ghz
Coeur(s) : 4
Thread(s) : 8

Fréquences :
Vitesse Coeur : 600 Mhz Mult. : x6.0(4-39)
Vitesse de bus 99.94 Mhz

Cache :
Données L1 : 4 * 32 Ko, 8 voies, 64 octets par ligne @ 10^5 MB/s
Instr. L1 : 4 * 32 Ko, 8 voies, 64 octets par ligne @ $6.5 * 10^4$ MB/s
Niveau L2 : 4 * 256 Ko, 4 voies, 64 octets par ligne @ $5 * 10^4$ MB/s
Niveau l3 : 6 Mo, 12 voies

OS:
Ubuntu 20.04.2 LTS
Architecture 64 bits

Compilateur:
CC : gcc version 9.3.1 (Ubuntu 9.3.0-17ubuntu1 20.04)
FLAGS : -std=c99 -O3 -Wno-comment
TARGET : x86_64-linux-gnu
Executable lancé offline

4.2.2 Ouverture_fusion

i	ouverture_basic	fusion_basic	fusion_red	fusion_ilu5_red	fusion_elu2_red	fusion_ilu5_elu2_red	fus_ilu5_elu2_red_fac	fusion_ilu15_red
i = 128	14.80	29.67	14.13	8.93	8.66	7.54	6.93	11.55
i = 192	13.75	29.17	13.03	9.01	9.68	7.89	7.34	10.77
i = 256	13.92	29.13	14.36	8.83	9.72	5.68	5.71	6.12
i = 320	8.05	16.68	7.76	4.83	5.32	6.08	6.88	9.05
i = 384	13.18	29.05	13.23	8.62	6.43	4.63	4.00	5.55
i = 448	13.48	28.86	13.65	9.03	9.06	7.76	4.54	5.36
i = 512	13.74	28.41	8.79	4.95	5.47	7.20	6.58	10.06
i = 576	9.65	28.43	13.79	8.97	9.58	4.90	3.93	5.34
i = 640	11.95	18.60	13.96	8.76	9.72	4.78	3.96	7.96
i = 704	8.36	28.66	8.78	5.02	9.94	7.83	6.84	9.52
i = 768	12.04	18.65	11.73	5.15	9.72	7.78	6.71	9.42
i = 832	8.31	28.78	9.84	8.74	9.66	7.85	3.79	4.64
i = 896	8.93	17.78	13.53	8.73	5.41	3.75	3.26	4.43
i = 960	9.87	19.86	13.53	5.28	4.84	3.61	3.27	4.37
i = 1024	10.10	26.35	7.55	4.16	4.58	3.60	3.21	4.34
i = 1088	10.52	19.71	6.92	4.16	4.63	3.63	3.22	4.37
i = 1152	10.82	18.72	6.70	4.15	4.58	3.60	3.19	4.36
i = 1216	11.14	17.01	6.61	4.11	4.58	3.60	3.19	4.35
i = 1280	13.34	15.98	6.53	4.11	4.58	3.59	3.22	4.35
i = 1344	12.47	15.62	6.51	4.11	4.60	3.59	3.20	4.34
i = 1408	12.82	14.19	6.51	4.14	4.58	3.59	3.20	4.34
i = 1472	12.64	14.07	6.51	4.11	4.62	3.61	3.18	4.34
i = 1536	10.98	14.03	6.51	4.11	4.58	3.59	3.18	4.35
i = 1600	9.79	13.96	6.51	4.10	4.58	3.59	3.21	4.33
i = 1664	12.61	13.96	6.52	4.10	4.58	3.58	3.20	4.33
i = 1728	11.32	13.96	6.52	4.10	4.58	3.59	3.21	4.32
i = 1792	11.72	13.96	6.52	4.10	4.58	3.59	3.18	4.34
i = 1856	10.99	13.94	6.52	4.10	4.58	3.63	3.18	4.32
i = 1920	11.05	13.97	6.52	4.10	4.58	3.58	3.20	4.32
i = 1984	12.40	13.95	6.51	4.10	4.57	3.58	3.19	4.31
i = 2048	12.31	13.94	6.51	4.10	4.57	3.58	3.22	4.32

4.2.3 Ouverture_pipeline

i	ouverture_basic	pipeline_basic	pip_red	pip_ilu3_red	pip_elu2_red	pip_elu2_red_fac	pip_ilu3_elu2_red	pip_ilu3_elu2_red_fac
i = 192	13.75	13.96	6.18	4.20	6.92	5.12	3.99	4.29
i = 128	14.80	13.92	4.97	4.65	6.57	4.68	3.18	3.27
i = 256	13.92	8.63	3.55	3.06	4.46	3.03	2.23	2.24
i = 320	8.05	13.48	5.56	3.97	6.81	4.22	3.70	3.57
i = 384	13.18	9.81	3.39	2.82	4.04	3.51	2.94	3.41
i = 448	13.48	8.31	5.16	3.71	6.79	4.59	3.69	3.52
i = 512	13.74	13.75	5.73	4.11	6.74	4.68	3.35	3.51
i = 576	9.65	12.77	5.73	3.99	6.60	4.64	3.48	3.42
i = 640	11.95	13.67	5.51	3.93	6.54	4.58	3.48	3.41
i = 704	8.36	9.41	3.46	4.00	6.72	4.57	3.63	3.59
i = 768	12.04	9.91	5.64	4.02	6.80	4.61	3.58	3.46
i = 832	8.31	7.03	2.82	2.11	3.26	2.37	1.81	1.77
i = 896	8.93	6.98	2.82	2.11	3.26	2.34	1.81	1.77
i = 960	9.87	6.98	2.81	2.11	3.29	2.34	1.80	1.77
i = 1024	10.10	6.97	2.81	2.12	3.26	2.34	1.80	1.77
i = 1088	10.52	6.97	2.81	2.10	3.26	2.34	1.80	1.77
i = 1152	10.82	7.02	2.81	2.10	3.26	2.33	1.80	1.77
i = 1216	11.14	6.97	2.81	2.11	3.26	2.33	1.80	1.76
i = 1280	13.34	6.97	2.81	2.10	3.27	2.33	1.80	1.77
i = 1344	12.47	6.97	2.84	2.10	3.26	2.33	1.79	1.76
i = 1408	12.82	6.97	2.80	2.09	3.25	2.33	1.79	1.76
i = 1472	12.64	6.98	2.80	2.09	3.25	2.33	1.79	1.76
i = 1536	10.98	6.98	2.80	2.09	3.25	2.33	1.79	1.77
i = 1600	9.79	6.96	2.80	2.09	3.25	2.33	1.79	1.76
i = 1664	12.61	6.98	2.80	2.09	3.27	2.33	1.79	1.76
i = 1728	11.32	6.97	2.80	2.09	3.25	2.33	1.80	1.76
i = 1792	11.72	6.96	2.80	2.09	3.28	2.33	1.79	1.76
i = 1856	10.99	6.96	2.80	2.09	3.25	2.33	1.79	1.76
i = 1920	11.05	6.97	2.80	2.09	3.25	2.33	1.79	1.76
i = 1984	12.40	6.96	2.80	2.09	3.26	2.33	1.79	1.76
i = 2048	12.31	6.97	2.80	2.09	3.25	2.33	1.79	1.76

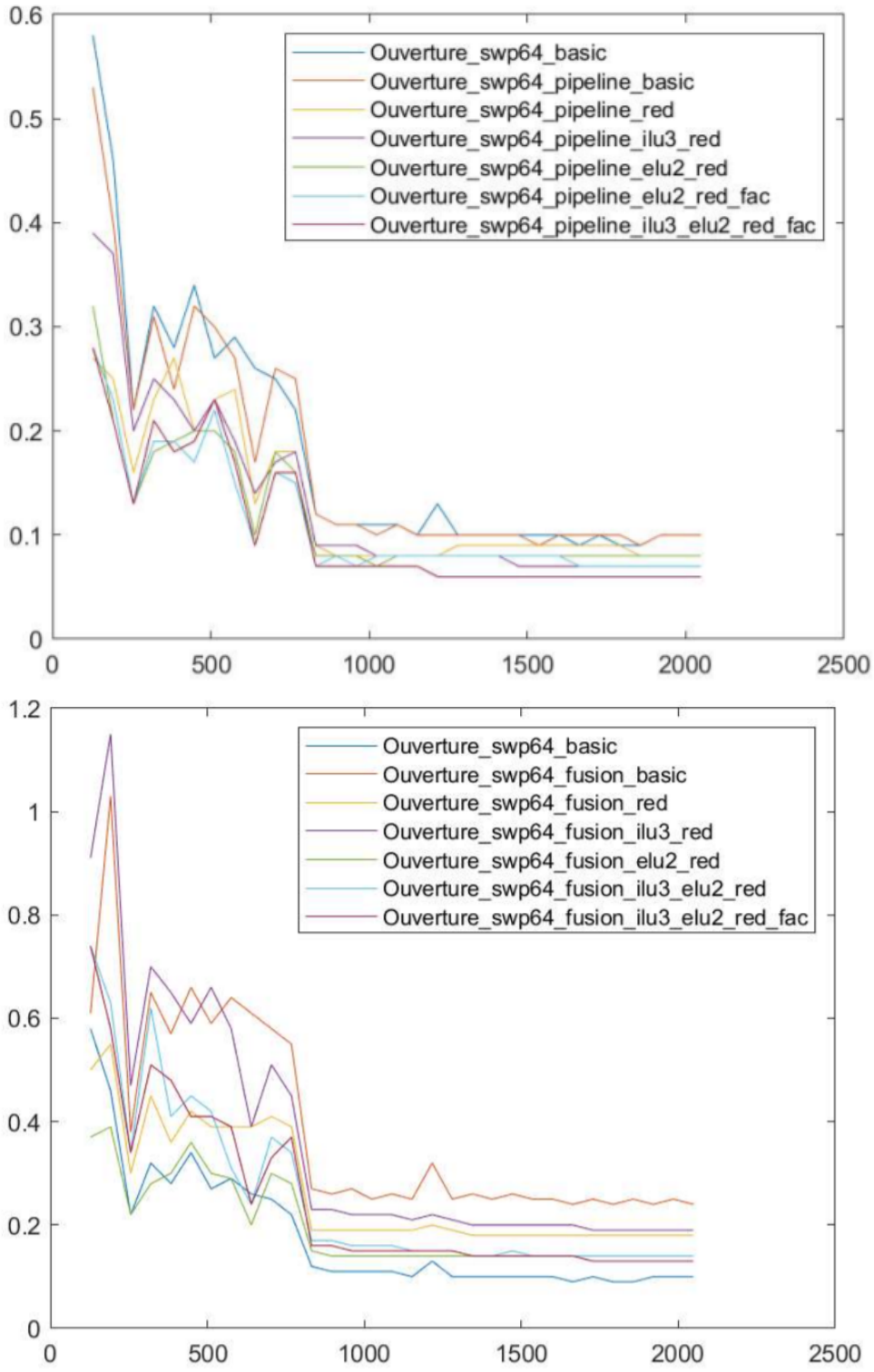
4.2.4 Ouverture_swp64_fusion

i	ouverture_swp64_basic	fus_swp64_basic	fus_swp64_red	fus_swp64_ilu3_red	fus_swp64_elu2_red	fus_swp64_ilu3_elu2_red	swp64_ilu3_elu2_red_fac
i = 128	0.58	0.61	0.50	0.91	0.37	0.74	0.74
i = 192	0.46	1.03	0.55	1.15	0.39	0.63	0.58
i = 256	0.22	0.38	0.30	0.47	0.22	0.34	0.34
i = 320	0.32	0.65	0.45	0.70	0.28	0.62	0.51
i = 384	0.28	0.57	0.36	0.65	0.30	0.41	0.48
i = 448	0.34	0.66	0.42	0.59	0.36	0.45	0.41
i = 512	0.27	0.59	0.39	0.66	0.30	0.42	0.41
i = 576	0.29	0.64	0.39	0.58	0.29	0.31	0.39
i = 640	0.26	0.61	0.39	0.39	0.20	0.24	0.24
i = 704	0.25	0.58	0.41	0.51	0.30	0.37	0.33
i = 768	0.22	0.55	0.39	0.45	0.28	0.34	0.37
i = 832	0.12	0.27	0.19	0.23	0.15	0.17	0.16
i = 896	0.11	0.26	0.19	0.23	0.14	0.17	0.16
i = 960	0.11	0.27	0.19	0.22	0.14	0.16	0.15
i = 1024	0.11	0.25	0.19	0.22	0.14	0.16	0.15
i = 1088	0.11	0.26	0.19	0.22	0.14	0.16	0.15
i = 1152	0.10	0.25	0.19	0.21	0.14	0.15	0.15
i = 1216	0.13	0.32	0.20	0.22	0.14	0.15	0.15
i = 1280	0.10	0.25	0.19	0.21	0.14	0.15	0.15
i = 1344	0.10	0.26	0.18	0.20	0.14	0.14	0.14
i = 1408	0.10	0.25	0.18	0.20	0.14	0.14	0.14
i = 1472	0.10	0.26	0.18	0.20	0.14	0.15	0.14
i = 1536	0.10	0.25	0.18	0.20	0.14	0.14	0.14
i = 1600	0.10	0.25	0.18	0.20	0.14	0.14	0.14
i = 1664	0.09	0.24	0.18	0.20	0.14	0.14	0.14
i = 1728	0.10	0.25	0.18	0.19	0.14	0.14	0.13
i = 1792	0.09	0.24	0.18	0.19	0.14	0.14	0.13
i = 1856	0.09	0.25	0.18	0.19	0.14	0.14	0.13
i = 1920	0.10	0.24	0.18	0.19	0.14	0.14	0.13
i = 1984	0.10	0.25	0.18	0.19	0.14	0.14	0.13
i = 2048	0.10	0.24	0.18	0.19	0.14	0.14	0.13

Interprétation : Remarquons, ici, que la version basique est beaucoup plus rapide que les autres optimisations. Plusieurs explications sont possibles : notre bus de donnée étant assez large les loads ont un coût assez faible. De plus, dans la version basique, il n'y a pas de variable locale or dans les optimisations suivantes, il y a un nombre important de variable locale que la fonction doit charger et décharger depuis le stack, le nombre de registre est trop faible pour autant de variables locales provoquant une disparité spatiale. Il est donc très difficile pour le compilateur d'optimiser en registre ces variables locales.

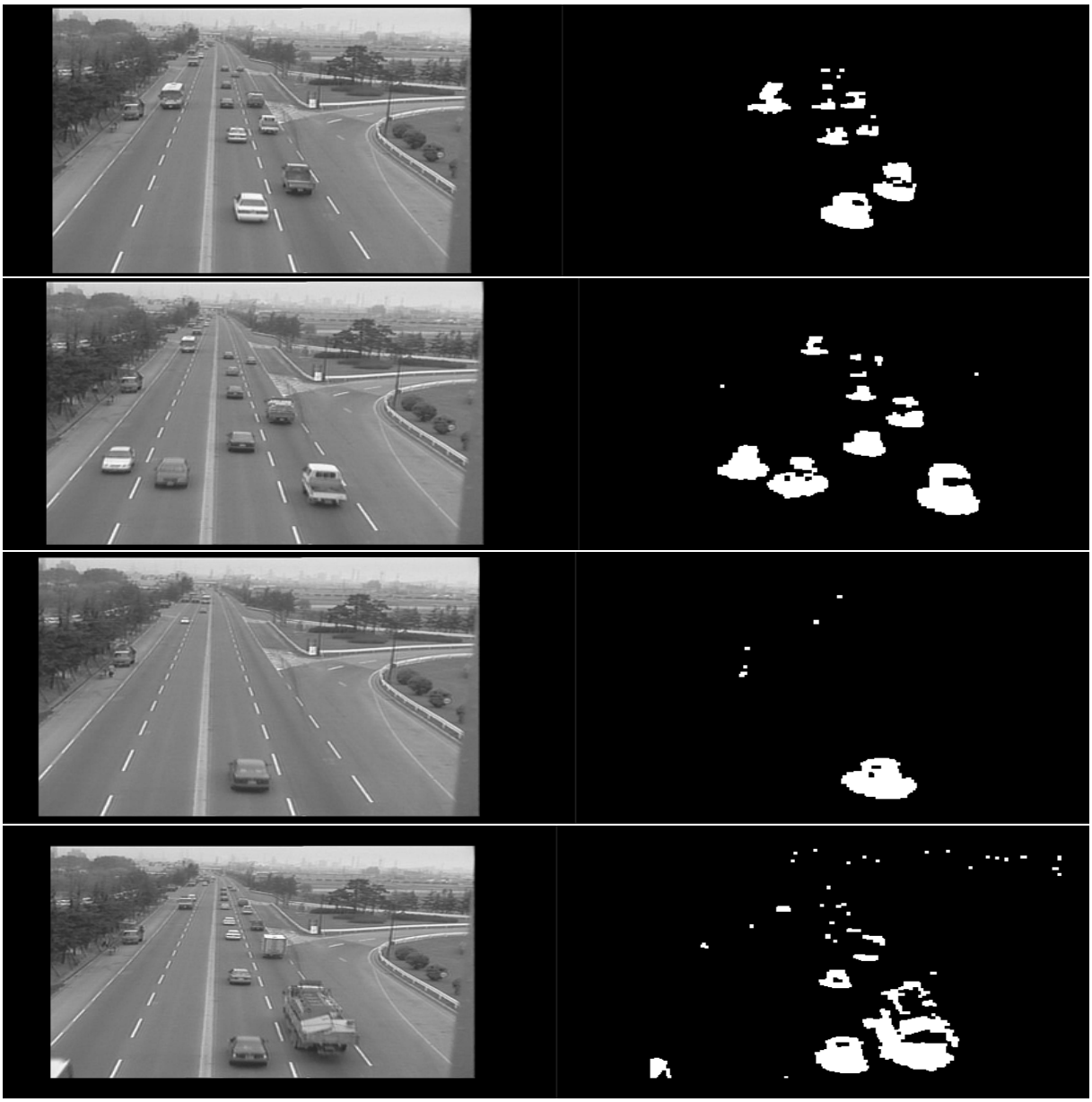
4.2.5 Ouverture_swp64_pipeline

i	ouverture_swp64_basic	pip_swp64_basic	pip_swp64_red	pip_swp64_ilu3_red	pip_swp64_elu2_red	pip_swp64_elu2_red_fac	swp64_ilu3_elu2_red_fac
i = 128	0.58	0.53	0.27	0.39	0.32	0.28	0.28
i = 192	0.46	0.40	0.25	0.37	0.21	0.23	0.21
i = 256	0.22	0.22	0.16	0.20	0.13	0.13	0.13
i = 320	0.32	0.31	0.23	0.25	0.18	0.19	0.21
i = 384	0.28	0.24	0.27	0.23	0.19	0.19	0.18
i = 448	0.34	0.32	0.20	0.20	0.20	0.17	0.19
i = 512	0.27	0.30	0.23	0.23	0.20	0.22	0.23
i = 576	0.29	0.27	0.24	0.19	0.18	0.15	0.17
i = 640	0.26	0.17	0.13	0.14	0.10	0.09	0.09
i = 704	0.25	0.26	0.18	0.17	0.18	0.16	0.16
i = 768	0.22	0.25	0.18	0.18	0.16	0.15	0.16
i = 832	0.12	0.12	0.09	0.09	0.08	0.07	0.07
i = 896	0.11	0.11	0.08	0.09	0.08	0.08	0.07
i = 960	0.11	0.11	0.08	0.09	0.08	0.07	0.07
i = 1024	0.11	0.10	0.08	0.08	0.07	0.08	0.07
i = 1088	0.11	0.11	0.08	0.08	0.08	0.08	0.07
i = 1152	0.10	0.10	0.08	0.08	0.08	0.08	0.07
i = 1216	0.13	0.10	0.08	0.08	0.08	0.08	0.06
i = 1280	0.10	0.10	0.09	0.08	0.08	0.08	0.06
i = 1344	0.10	0.10	0.09	0.08	0.08	0.08	0.06
i = 1408	0.10	0.10	0.09	0.08	0.08	0.08	0.06
i = 1472	0.10	0.10	0.09	0.07	0.08	0.08	0.06
i = 1536	0.10	0.09	0.09	0.07	0.08	0.08	0.06
i = 1600	0.10	0.10	0.09	0.07	0.08	0.08	0.06
i = 1664	0.09	0.10	0.09	0.07	0.08	0.07	0.06
i = 1728	0.10	0.10	0.09	0.07	0.08	0.07	0.06
i = 1792	0.09	0.10	0.09	0.07	0.08	0.07	0.06
i = 1856	0.09	0.09	0.08	0.07	0.08	0.07	0.06
i = 1920	0.10	0.10	0.08	0.07	0.08	0.07	0.06
i = 1984	0.10	0.10	0.08	0.07	0.08	0.07	0.06
i = 2048	0.10	0.10	0.08	0.07	0.08	0.07	0.06



Interprétation : Comme supposer plus haut, pipeline semble être plus rapide que fusion. En effet, nous sommes dans une configuration où le bus de donnée est assez large pour que les loads aient un impact aussi significatif que les opérations morphologiques.

4.3 Test motion



Voici 4 exemples de l'algorithme naïve appliqué sur des voitures en mouvement. On remarque que les masques générés correspondent bien à un objet en mouvement. Cela permet de valider partiellement notre chaîne de traitement, mais ne vaut pas un test unitaire.En ce qui concerne les images, les trois premières images distingue bien les voitures par de gros blocs de cases blanches, or pour la 4e image, vers le fond de l'image on est un peu confus et plusieurs partis du camion sont supprimées.

5 Conclusion

Pour conclure sur ce projet, nous avons, avant d'implémenter les algorithmes et optimisations, adopté un workflow qui nous permet un débogage efficace et facile d'utilisation grâce aux macros.
Lors de l'implémentation des algorithmes et des optimisations, nous avons procédé par étape : en commençant par l'implémentation d'une version naïve pour servir de référence. Cette version est la plus importante car elle permet de valider les optimisations que l'on fera par la suite. Ainsi, il est crucial de le valider avec des tests unitaires après la moindre modification.
Le projet étant symétrique (i.e. Erosion - Dilatation et Fermeture - Ouverture), une attention particulière portée au nommage des macros couplée à l'édition de texte des éditeurs de code moderne, a permis un développement rapide de notre projet.
Les différentes optimisations ne sont pas construit d'avoir une optimisation progressive mais plutôt l'exploration de plusieurs optimisations différentes (i.e. ilu15) pour juger leur impact sur la partie architecture et logicielle. Il est ensuite judicieux de les combiner pour éliminer les limitations éventuelle (i.e. rotation de variable).
Avant de parler des résultats, il est important de préciser que les résultats obtenus son spécifique à notre environnement d'exécution et difficilement généralisable à des systèmes embarqués qui ont des contraintes non seulement en terme de performance mais aussi de mémoire et d'énergie.
Nos hypothèses sur l'impact de l'optimisation sont pour la plupart vérifiées par expérimentation. Cependant, il persiste quelques anomalies (i.e. ouverture swp64 fusion) qui sont explicables par notre environnement d'exécution et les optimisations effectuées par notre compilateur. Des facteurs qui changent d'une machine à une autre et de façon significative entre ordinateur portable et un processeur embarqué.
Pour répondre à la problématique du temps réel, il est difficile d'y répondre sans avoir le chaîne de traitement complet à savoir acquisition du flux d'image (ffmpeg), génération du masque motion (SD + morpho), détection de l'objet en mouvement (agrégation des points de masque connexes) et enfin le tracking d'un overlay labellant l'objet (i.e. rectangle de couleur). Cependant, la génération du masque est une étape critique qui demande le plus de ressource. Ainsi, vu les optimisations importantes effectués, il est possible de prédire qu'une chaîne de traitement complète est possible en temps réel.