

RAPPORT DU PROJET VISION (UE L03IN401) DéTECTION DE MOUVEMENT

Réalisé par :

M. AHMED ALI Nassim

M. BOUTALEB Mohamed Allaa Eddine

Encadré par :

M. MEUNIER Quentin

Sommaire :

1 Introduction	3
2 Fonctions de bases (MIN/MAX)	4
2.1 Basic, reg, rot et red	4
2.2 ILU3, ILU3 REDUCTION	5
2.3 ELU2, ELU2 FACTOR	6
2.4 ILU3 ELU2 RÉDUCTION ET FACTOR	7
3 Fonctions d'ouverture	8
3.1 PIPELINE	8
3.2 FUSION	11
4 Fonctions SWP	13
5 Traitement de bordure	14
6 Comparaison des fonctions	15
6.1 Fonctions pipeline	15
6.2 Fonctions fusion	16
6.3 Fonctions SWP	16
6.4 Conclusion	18
7 Chaîne de traitement	19
7.1. L'algorithme Sigma-Delta	19
7.2. Ouverture + Fermeture	20
8 Conclusion	21

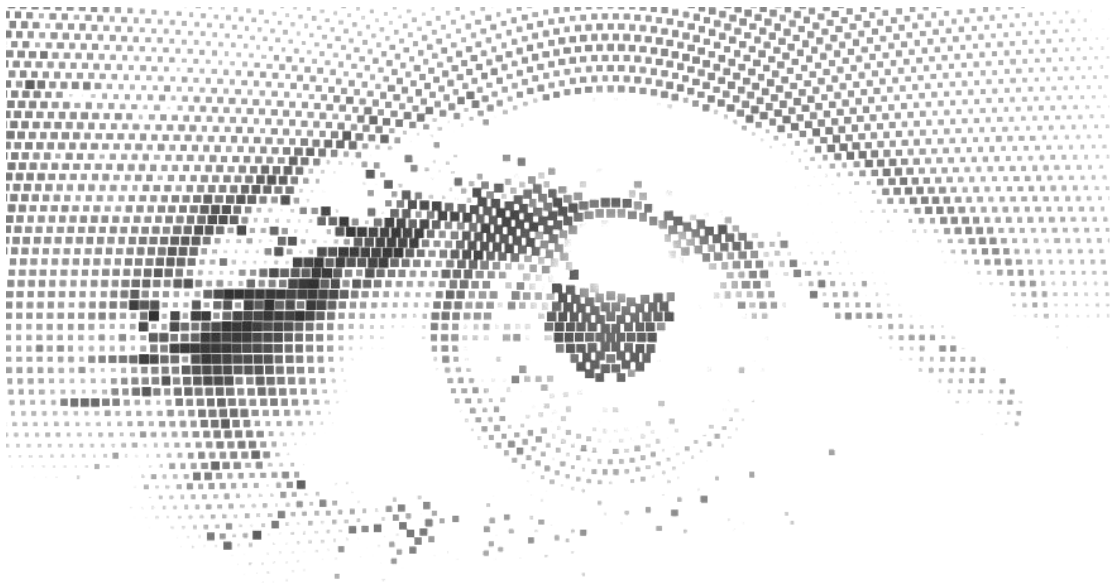
Table des figures:

Figure 1 - Comparaison versions pipeline cpp en fonction de la taille	15
Figure 2 - Comparaison versions fusions cpp en fonction de la taille	16
Figure 3 - Comparaison versions SWP cpp en fonction de la taille	17
Figure 4 - Comparaison versions SWP temps regroupement et séparation inclut cpp en fonction de la taille	18

1. Introduction

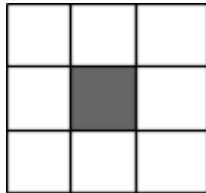
Détecter les objets en mouvement dans une séquence d'images est une tâche de bas niveau importante pour de nombreuses applications de vision par ordinateur telles que la vidéosurveillance, la surveillance du trafic, la signalisation de reconnaissance de la langue... etc. Lorsque la caméra est immobile, une classe de méthodes habituellement employée est la soustraction de bruit de fond. Le principe de ces méthodes est de construire un modèle de la scène statique (c'est-à-dire sans objets en mouvement) appelée l'arrière-plan (background), puis de comparer chaque image de la séquence à ce fond afin de discriminer les régions de mouvement inhabituel, appelé avant-plan (foreground).

Dans le cadre de ce projet, notre mission était de créer une chaîne de traitement d'image afin d'isoler les pixels en mouvement, d'optimiser cette chaîne le plus possible via diverses méthodes et enfin de comparer ces différentes méthodes.



2. Fonctions de bases (MIN/MAX) :

2.1 Basic, reg, rot et red :



BASIC : La version basique est l'application directe de la définition du traitement. C'est-à-dire, pour chaque pixel à traiter, la fonction basic va charger la bordure du pixel, donc 9 pixels en tout (y compris le pixel que nous souhaitons traiter), pour ensuite calculer le min/max entre les différentes valeurs des pixels chargés.

REG : La version registre est identique à la version basique, seulement les valeurs des pixels sont chargées dans des variables avant de calculer le min/max. Cela aidera le compilateur à les optimiser dans des registres.

ROT : La version rotation permet d'éviter de charger les mêmes valeurs entre des itérations consécutives. En effet, si deux itérations sont consécutives, alors les deux itérations ont 6 pixels en commun.

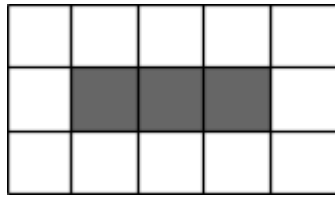


Donc pour éviter de charger plusieurs fois le même pixel, il suffit de faire une rotation de registre à la fin de chaque itération afin de lire seulement 3 pixels par itérations au lieu de 9.

A noter qu'un prologue est obligatoire avant la première itération afin de charger les 6 premiers pixels.

ROT : La version réductions utilise le même principe que la version rotation, sauf que la rotation se fait sur les valeurs min/max calculées au lieu des pixels. A noter que le prologue doit en plus calculer les 2 premières min/max.

2.2 ILU3, ILU3 REDUCTION :



Le principe de ces deux versions consiste à traiter 3 pixels à la fois dans la même itération, c'est-à-dire on fait un déroulage de boucle d'ordre trois lors du traitement de la ligne.

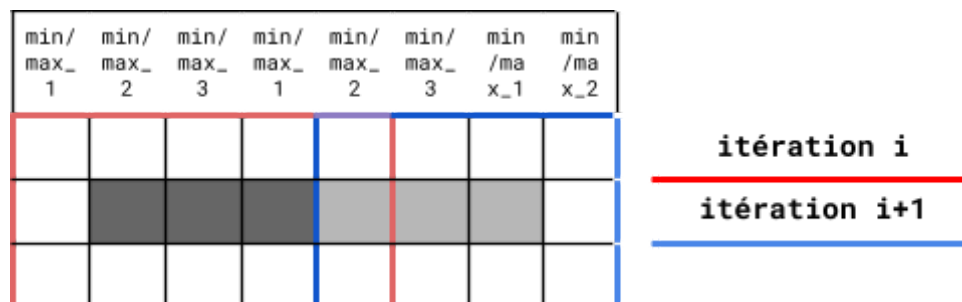
A noter que cette version nécessite un épilogue afin de prendre en compte les cas où le nombre de pixels par ligne n'est pas un multiple de trois.

Le calcul du nombre de case restante se fait en calculant le modulo entre le nombre de case et l'ordre de déroulage.

Le traitement de l'épilogue va se faire en appelant la version **basic_rot** pour la même ligne avec $j0=j1-r$ et $j1=j1$.

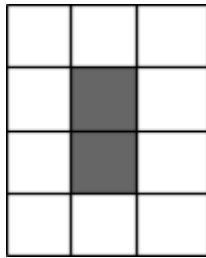
En plus du déroulage de boucle, la version **ILU3_RED**, applique aussi une réduction.

Cette réduction se fait en calculant les min/max dans les variables déjà utilisées afin d'éviter d'une part la rotation de registre, et d'autre part l'utilisation de plus de variables.



A noter que comme précédemment, toute version avec réduction à besoin d'un prologue pour calculer les 2 premiers min/max pour la première itération.

2.3 ELU2, ELU2 FACTOR :



Les versions **ELU2** et **ELU2_factor** consistent à traiter 2 pixels à la fois dans la même itération. La différence avec la version précédente est que les pixels sont sûrs deux lignes différentes. Le déroulage de boucle se fait donc sur la boucle de traitement de la matrice et non de la ligne.

Comme précédemment, cette version nécessite un épilogue afin de prendre en compte les cas où le nombre de lignes n'est pas pair.

Le traitement de l'épilogue se fait en appelant la version ILU3_RED pour la dernière ligne si le nombre de lignes est impair.

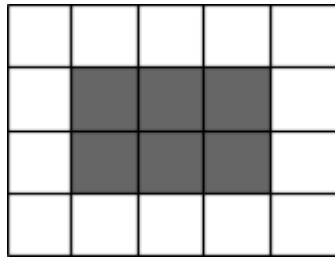
La version factorisée fait la même chose mais au lieu de calculer 2 min/max de trois éléments pour chaque colonne, on calcule d'abord le min/max des éléments en commun puis on calcule les 2 min/max de la colonne.



Version sans factor : Calcul de min/max des pixels 1, 2, 3 et des pixels 2, 3, 4 (4 OR/AND au total)

Version avec factor : Calcul de min/max des pixels 2, 3 dans R ensuite calcul de min/max du pixel 1 et R puis du pixel 4 et R (3 OR/AND au total)

2.4 ILU3 ELU2 RÉDUCTION ET FACTOR :



Ces deux versions sont un mélange des deux versions vu précédemment. C'est-à-dire on traite **3** éléments par lignes et **2** lignes en même temps. On fait donc un déroulage d'ordre **3** sur la boucle de traitement de ligne et un déroulage d'ordre **2** sur la boucle de traitement de la matrice, en appliquant toujours le principe de la réduction et de la factorisation pour la version factor.

On aura donc un prologue pour calculer le nombre de cases restantes par ligne, qu'on va traiter avec la fonction **ELU2** et on aura aussi un épilogue après la boucle de traitement de la matrice, où on va calculer la dernière ligne (si le nombre de ligne est impaire) avec **ILU3_RED**.

3. Fonctions d'ouverture :

L'ouverture est un traitement composé de deux étapes :

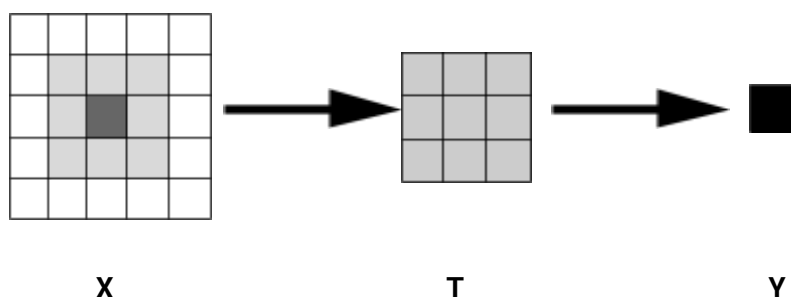
1 - L'érosion (Calcul du MIN) 2 - La dilatation (Calcul du MAX)

La fermeture est tout simplement l'ouverture dans l'ordre inverse, c'est-à-dire la dilatation en premier, puis l'érosion. Dans le cadre de cet UE, nous avons implémenté seulement les différentes fonctions d'ouverture, mais l'implémentation des fonctions de fermeture peut se faire en quelques secondes si les différentes versions d'ouvertures sont codées correctement (On remplace les calculs de MIN par MAX, et les calculs de MAX par MIN).

Les fonctions d'ouverture que nous avons fournis dans le code peuvent être classifiées dans deux catégories principales :

3.1 PIPELINE :

La version Pipeline de l'ouverture est une implémentation du Pipeline Logiciel que nous avons vu en cours. Le Pipeline Logiciel permet de diminuer l'impact des dépendances de données, et consiste à faire un entrelacement de traitements issus de différentes itérations de boucle. La particularité de la version Pipeline est le passage par une matrice intermédiaire '*T*' qui contient les cases en gris clair et foncé dans *X* après érosion, puis la case en noir dans '*Y*' qui contient le résultat du MAX de toutes les cases de '*T*'. Voici un petit schéma qui représente ces étapes :



Par exemple, afin de produire le bon MAX dans **UN pixel** dans la matrice de sortie '*Y*', il faut calculer 9 MINs à partir de la matrice '*X*' et les mettre dans la matrice intermédiaire '*T*'. On se

rend compte que le calcul des 9 MINs pour calculer le **MAX** à la position '**i**', contient **6 MINs** déjà calculés pour produire le **MAX** à la position '**i+1**'. Il est nécessaire de faire des calculs avant la boucle (Prologue) qui vont servir comme données pour la première itération, et après (Epilogue) s'il reste des parties de la matrice que la boucle n'a pas traité. Modélisons ce traitement par un tableau :

PROLOGUE : Erosion de la ligne $i_0 - 1$ et la ligne i_0 (X->T)			
itérations	$i = 0$	$i = 1$	$i = 2$
BOUCLE	Erosion de la ligne $i_0 + 1$ (X->T)	Erosion de la ligne $i_0 + 2$ (X->T)	Erosion de la ligne $i_0 + 3$ (X->T)
	-> Dilatation de la ligne i_0 (T->Y)	-> Dilatation de la ligne $i_0 + 1$ (T->Y)	-> Dilatation de la ligne $i_0 + 2$ (T->Y)
EPILOGUE Seulement dans les versions ELU2			

Le pseudo code des différentes versions d'ouverture Pipeline (sauf pour les versions ELU2) est le suivant :

```
// Prologue
LINE_MIN3_???(X, ligne  $i_0 - 1$ , colonne  $j_0$ , colonne  $j_1$ , T)
LINE_MIN3_???(X, ligne  $i_0$ , colonne  $j_0$ , colonne  $j_1$ , T)

// Boucle
for i =  $i_0$  to  $i_1$  :
    LINE_MIN3_???(X, ligne i + 1, colonne  $j_0$ , colonne  $j_1$ , T)
    LINE_MAX3_???(T, ligne i, colonne  $j_0$ , colonne  $j_1$ , Y)
```

avec '???' étant la version d'érosion (LINE_MIN3) ou de dilatation (LINE_MAX3) correspondante. Par exemple : Pipeline_BASIC utilise LINE_MIN3_BASIC et LINE_MAX3_BASIC. Idem pour : Pipeline_red, Pipeline_ILU3_red.

Pour les versions Pipeline ELU2, nous avons le pseudo code suivant :

```
int r = (i1 - i0 + 1) mod 2

// Prologue
LINE_MIN3_???(X, ligne i0-1, colonne j0, colonne j1, T)

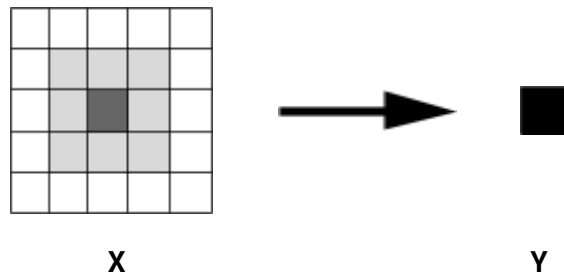
// Boucle
for i = i0 to (i1 - r); i = i + 2 :
    LINE_MIN3_???(X, ligne i + 1, colonne j0, colonne j1, T)
    LINE_MAX3_???(T, ligne i, colonne j0, colonne j1, Y)

// Epilogue
if r != 0 :
    LINE_MAX3_???(T, ligne 'i1 - r + 1', j0, j1, Y)
```

avec '???' étant la version d'érosion (LINE_MIN3) ou de dilatation (LINE_MAX3) correspondante (Versions ELU2 seulement). Par exemple : Pipeline_ELU2_red utilise LINE_MIN3_ELU2_red et LINE_MAX3_ELU2_red. Idem pour : Pipeline_ELU2_red_factor, Pipeline_ILU3_ELU2_red, Pipeline_ILU3_ELU2_red_factor.

3.2 FUSION :

La version Fusion n'utilise pas une matrice intermédiaire 'T', elle passe directement de la matrice d'entrée 'X' a la matrice de sortie 'Y'. Modélisons ca par un schéma :



Nous faisons l'érosion pour toutes les cases en gris clair et foncé dans la matrice 'X', puis nous calculons le MAX entre toutes les nouvelles valeurs et mettre le résultat dans la case correspondante de la matrice 'Y', sans aucun passage par une matrice intermédiaire. A noter que cette fois nous avons besoin d'une bordure de 2 pixels. Les différentes versions de fusion que nous avons implémenté sont les suivantes :

- **line_ouverture3_ui8matrix_fusion :**

Celle-ci est la version basique de l'ouverture par fusion. En prologue, on charge 20 cases (5 cases x 4 colonnes), puis on calcule 3 MINs et on les met dans des variables temporaires pour pouvoir calculer les valeurs des 9 cases après érosion. Après, nous faisons un MAX entre toutes ces valeurs et on stocke le résultat dans la case concernée. Pour la boucle, on charge a chaque fois juste les cases de la colonne 'j+2', car on garde les résultats des calculs précédents. à la fin de la boucle, on fait une rotation de registre sur les MAXs et les MINs des variables temporaires.

- **line_ouverture3_ui8matrix_fusion_ILU5_red :**

Celle-ci a le même prologue que la version précédente. La seule différence est dans le code de la boucle. Tout d'abord, avant d'entrer dans la boucle, on calcule le reste de la division du nombre de colonnes par 5. La boucle commencera de la première colonne, jusqu'à 'Indice de la dernière colonne j1 - reste', avec des pas de 5. Après le calcul de chaque élément, on fait une réduction adéquate pour calculer l'élément

suivant pendant tout le déroulage. à la fin de la boucle, on fait un rotation de registres car 5 n'est pas un multiple de 3. Pour l'épilogue, on utilise la version basique de fusion pour les reste des colonnes (çàd de $j1 - r1 + 1$, jusqu'à $j1$).

- **line_ouverture3_ui8matrix_fusion_ILU5_ELU2_red :**

Même principe que la précédente, sauf que nous calculons 10 cases à chaque itération (5 cases x 2 lignes). Le code de la boucle est presque le même avec une seule différence qui est le calcul de 2 MAXs (çad 2 stockages) soit 10 stockages au total à chaque itération. **ouverture3_ui8matrix_fusion_ILU5_ELU2_red** appelle cette fonction 'line' dans une boucle de pas 2, puisque nous calculons 5 éléments sur chaque ligne pour deux lignes à chaque itération. Un épilogue calcule la dernière ligne si non-calculée avec **ouverture3_ui8matrix_fusion_ILU5_red**.

- **line_ouverture3_ui8matrix_fusion_ILU5_ELU2_red_factor :**

Un squelette de code presque identique à la précédente, sauf qu'il y a plus de variables pour stocker les facteurs en commun des MINs et des MAXs. Cela permet de réduire les calculs mais pas forcément améliorer la performance, nous détaillerons cela après dans la comparaison des performances.

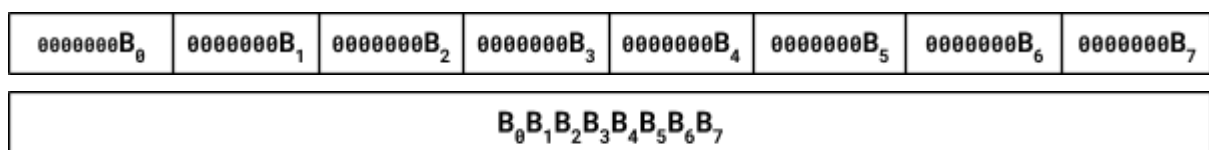
- **line_ouverture3_ui8matrix_fusion_ILU15_red :**

Même principe que **line_ouverture3_ui8matrix_fusion_ILU5_red**, sauf que nous faisons un déroulage de 15 cases par boucle. Le nombre de variables est similaire à celui de ILU5, mais le code de la boucle est 3 fois plus long.

4. Fonctions SWP :

Dans toutes les versions précédentes, les données traitées sont des octets qui représentent des valeurs binaires. Au final, 1 bit suffit pour représenter la donnée traitée par les différentes fonctions. C'est là que les versions SWP entrent en jeu.

Le principe de ces fonctions est de regrouper les octets de la matrice initiale ont un, deux ou quatre octets (selon la version du SWP) afin de calculer le min/max de plusieurs valeurs avec le nombre minimum d'accès mémoire, pour ensuite les séparer pour obtenir la matrice résultante.



A noter qu' un chargement depuis la mémoire permet de charger des mots de 64 bit d'un coup (architecture CPU en 64 bit).

Dans le cadre du projet, nous avons implémenté quatre versions :

- **MIN/MAX SWP8 basic** : Regroupe 8 octets en 1.
- **MIN/MAX SWP16 basic** : Regroupe 16 octet en 2.
- **MIN/MAX SWP32 basic** : Regroupe 32 octet en 4.
- **MIN/MAX SWP32 ELU2** : Regroupe 32 octet en 4 et calcul deux lignes simultanément.

Le calcul des min/max pour un group de pixel se fait de la manière suivante :

- Lecture de la bordure du groupe. (9 lecture au total)

a1	b1	c1
a2	b2	c2
a3	b3	c3

- Min/Max bit à bit entre les mots de la même colonne.

a	b	c
---	---	---

- Min/Max entre B décalé de 1 vers la gauche et A décalé de 7 vers la droite.
- Min/Max entre B décalé de 1 vers la droite et C décalé de 7 vers la gauche.

<code>(a>>7) min/max (b << 1)</code>	<code>b</code>	<code>(b >> 1) min/max (c << 7)</code>
--	----------------	--

- Min/max entre les valeurs trouvées.

Pour chacune des versions, une fonction d'ouverture SWP pipeline est utilisée pour les comparer aux autres fonctions d'ouverture vues précédemment.

5. Traitement de bordure :

Afin de traiter un pixel, nous avons forcément besoin de sa bordure. Cela pose un problème pour les pixels des première et dernière lignes et colonnes.

C'est pour cela que nous avons créé une fonction qui permet d'initialiser les pixels de la bordure à la bonne valeur.

Cette fonction prend en entrée une matrice, des indices de début et fin de ligne et colonnes ainsi que la valeur de la bordure, et va ensuite copier les premières valeurs de la matrice dans la bordure.

Cette fonctions est présente au niveau du fichier **morpho_test.c** et **swp_test.c** (Une version de cette fonction existe pour chaque version du swp)

6. Comparaison des fonctions :

6.1 Fonctions pipeline :

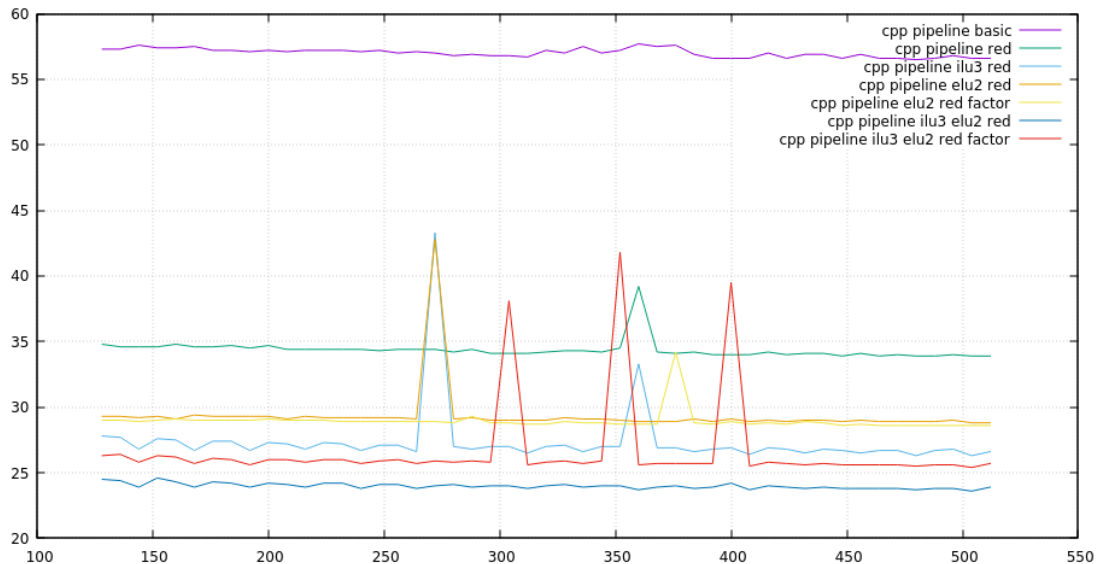


Figure 1 - Comparaison versions pipeline | cpp en fonction de la taille

- Gain de 24cpp en moyenne entre la version basique et red.
- Gain de 7cpp (20%) en moyenne entre ILU3 et red.
- Gain de 5cpp (14%) en moyenne entre ELU2 et red.
- Gain de 11cpp (30%) en moyenne entre ILU3 ELU2 et red.

Malgré la factorisation du calcul dans les versions "factor", nous remarquons une légère diminution de performance par rapport aux versions sans factorisation.

Des causes potentielles de cette perte de performance sont :

- L'incapacité du compilateur d'optimiser les nouvelles variables ajoutées dans des registres, ce qui augmente les accès mémoire.
- Les affectations dans des variables coûtent plus cher que refaire des MIN3/MAX3.

-> **pipeline_ILU3_ELU2_red** donne les meilleures performances.

6.2 Fonctions fusion :

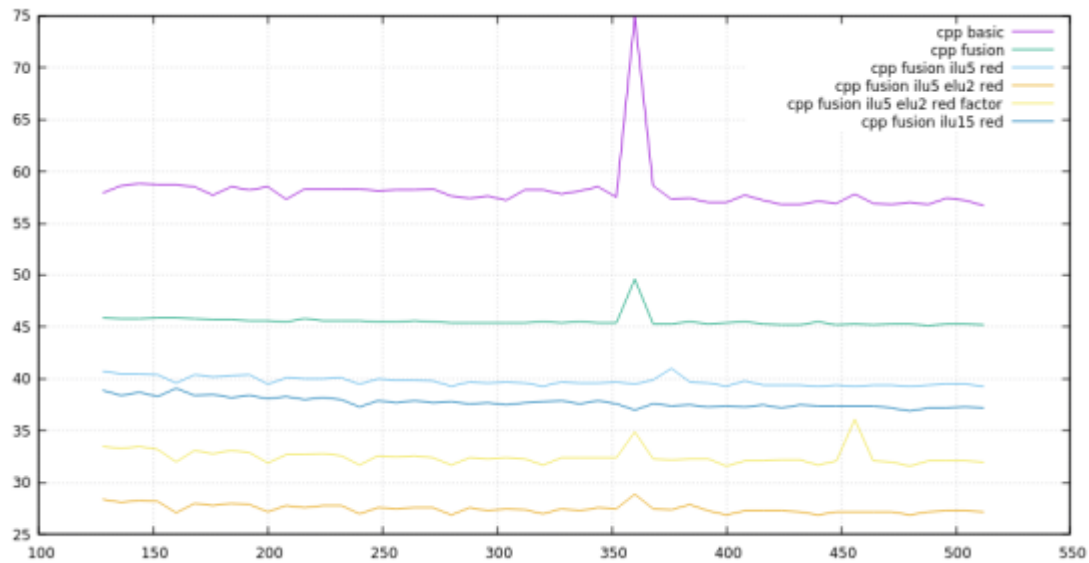


Figure 2 - Comparaison versions fusions | cpp en fonction de la taille

- Gain de 13cpp (22%) entre fusion et basic.
- Gain de 5cpp (10%) entre fusion ILU5 et fusion.
- Gain de 19cpp (41%) entre fusion ILU5_ELU2_red et fusion.
- Gain de 10cpp (21%) entre fusion ILU5_ELU2_red_factor et fusion.
- Gain de 8cpp (17%) entre fusion ILU15 et fusion.

L'ordre de déroulage n'est pas forcément proportionnel au performance. En effet, avoir un grand déroulage complexifie l'optimisation du code par le compilateur.

-> fusion_ILU5_ELU2_red donne les meilleures performances.

Pour les mêmes raisons mentionnées dans la partie Pipeline, les versions factor donnent des performances plus faibles que les versions non-factor.

6.3 Fonctions SWP :

Nous allons d'abord commencer par comparer les différentes versions du SWP entre elles sans inclure le temps pour regrouper et séparer les différents octets de la matrice initiale.

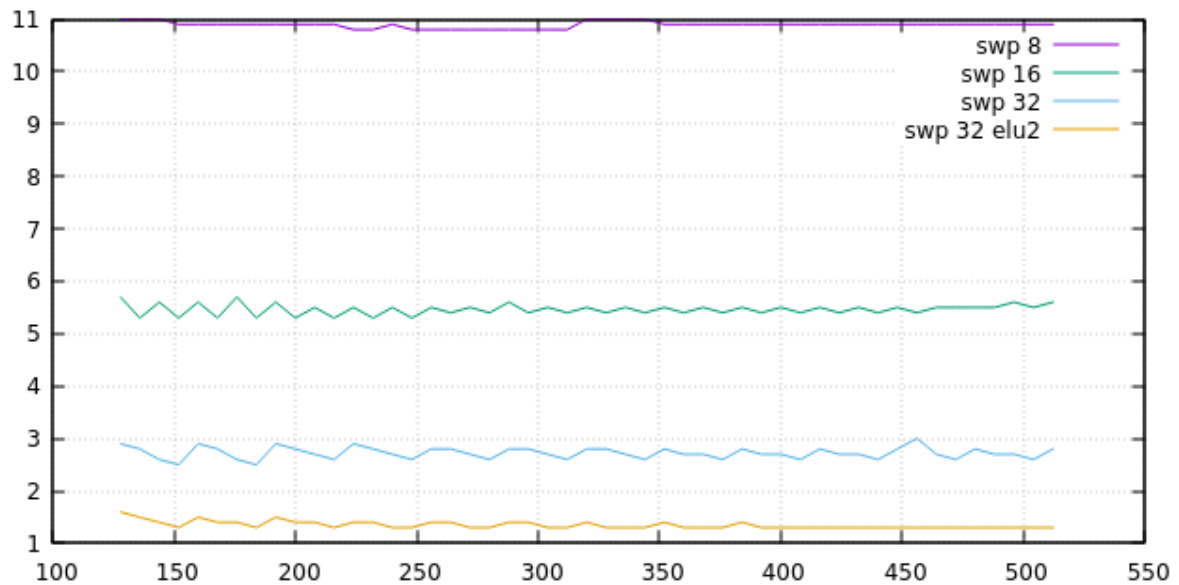


Figure 3 - Comparaison versions SWP | cpp en fonction de la taille

- Les versions SWP sont beaucoup plus rapides que les autres.
- SWP16 est 2 fois plus rapide que la version SWP8.
- SWP32 est 2 fois plus rapide que la version SWP16.
- Gain de 54% pour la version SWP32_ELU2 par rapport à SWP32.

Les versions SWP ont beaucoup moins d'accès mémoire, ce qui implique une plus grande importance pour l'optimisation du temps CPU. D'où le gain conséquent du déroulage de boucle (moins d'instructions de branchement) par rapport aux versions vues précédemment.

-> SWP32_ELU2 donne les meilleures performances.

Nous allons maintenant inclure le temps pour le regroupement et la séparations des octets.

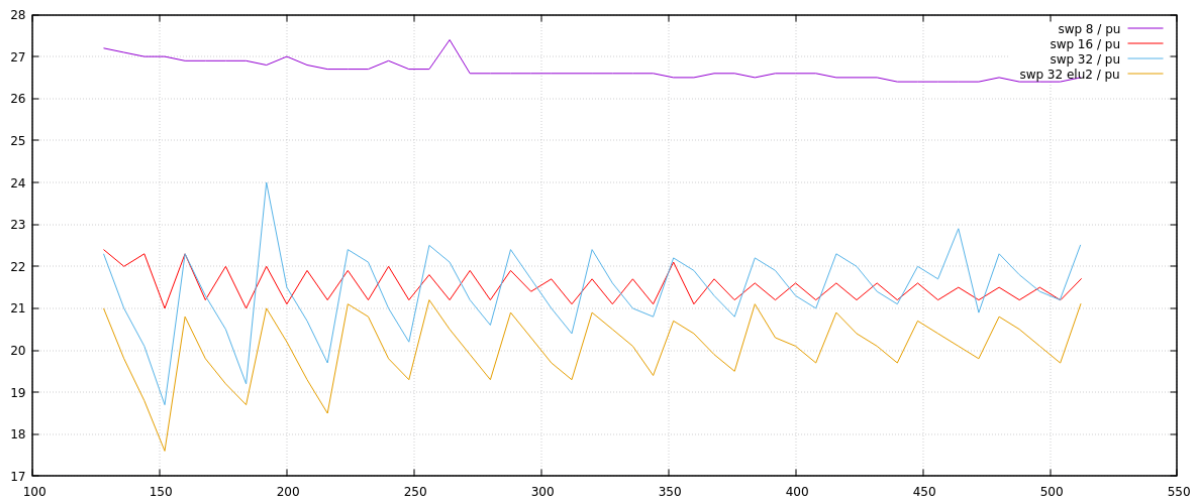


Figure 4 - Comparaison versions SWP temps regroupement et séparation inclut | cpp en fonction de la taille

- Les versions SWP restent plus performantes que la majorité des autres versions.
- Le temps des fonctions de groupement et de séparation des matrices est très dépendant de la taille de la matrice. Ces fonctions prennent moins de temps quand la taille de la matrice est un multiple de l'ordre du SWP. C'est pour cela qu'on observe une régularité au niveau de SWP8.
- Le temps des fonctions de groupement et de séparations représentent la majorité du temps de traitement.

6.4 Conclusion :

- Les versions SWP16, SWP32, SWP32_ELU2 sont les plus performantes même en incluant le temps de regroupement et de séparation des pixels.
- Moins le programme accède à la mémoire, plus l'optimisation du code sera importante en termes de gain de performance, d'où la hausse du pourcentage de gain pour la version SWP32_ELU2 par rapport à pipeline_ELU2.
- Les versions factor, que ce soit dans pipeline ou dans fusion, ne sont pas plus performantes que les versions non-factor.
- La réduction apporte un gain conséquent pour la version pipeline.
- Le degré de déroulage de boucle n'est pas proportionnel au gain de performance.

7. Chaîne de traitement :

7.1. L'algorithme Sigma-Delta :

Cet algorithme de traitement d'image suppose que le niveau de bruit (qui est représenté par un écart type) peut varier en tout point de la matrice. Pour cette raison, on modélise le niveau de gris de chaque pixel par une moyenne $M_t(x)$ et un écart type $V_t(x)$. Reprenons le pseudo code de l'algorithme ici qui va servir comme guide pour les images suivantes.

```
1 foreach pixel  $x$  do // step #1 :  $M_t$  estimation
2   if  $M_{t-1}(x) < I_t(x)$  then  $M_t(x) \leftarrow M_{t-1}(x) + 1$ 
3   if  $M_{t-1}(x) > I_t(x)$  then  $M_t(x) \leftarrow M_{t-1}(x) - 1$ 
4   otherwise do  $M_t(x) \leftarrow M_{t-1}(x)$ 
5 [step #2 : difference computation]
6 foreach pixel  $x$  do // step #2 :  $O_t$  computation
7    $O_t(x) = |M_t(x) - I_t(x)|$ 
8 foreach pixel  $x$  do // step #3 :  $V_t$  update and clamping
9   if  $V_{t-1}(x) < N \times O_t(x)$  then  $V_t(x) \leftarrow V_{t-1}(x) + 1$ 
10  if  $V_{t-1}(x) > N \times O_t(x)$  then  $V_t(x) \leftarrow V_{t-1}(x) - 1$ 
11  otherwise do  $V_t(x) \leftarrow V_{t-1}(x)$ 
12   $V_t(x) \leftarrow \max(\min(V_t(x), V_{max}), V_{min})$  // clamp to  $[V_{min}, V_{max}]$ 
13 foreach pixel  $x$  do // step #4 :  $\hat{E}_t$  estimation
14   if  $O_t(x) < V_t(x)$  then  $\hat{E}_t(x) \leftarrow 0$ 
15   else  $\hat{E}_t(x) \leftarrow 1$ 
16
```

Voici les résultats obtenus en utilisant la séquence d'images "car3" fournie avec le projet :

- Visualisation de $I_t(X)$, $M_t(X)$, $O_t(x)$, $V_t(x)$, $E_t(x)$: [CLIQUEZ ICI](#)

D'après le GIF qui visualise les matrices utilisées pendant les calculs de l'algorithme, nous pouvons remarquer déjà que $M_t(x)$ représente l'arrière-plan, càd la partie de la séquence qui ne bouge pas. $O_t(x)$ représente l'avant-plan, càd la partie de la séquence qui subit à des changements et des mouvements. Pour la visualisation de $E_t(x)$, nous avons légèrement modifié le code de Sigma-Delta pour qu'il mette 255 dans $\hat{E}_t(x)$ au lieu de 1 (ligne 15), pour que l'image PGM affiche des blancs correctement. Si on met 1, on aura l'impression que la séquence de E_t générée est noire à cause de

l'écart presque nul entre les valeurs de 0 et de 1. La modification était seulement faite pour la visualisation. Nous avons gardé la valeur '1' pour les calculs.

7.2. Ouverture + Fermeture:

La matrice $E_t(x)$ va servir comme entrée dans le traitement suivant :

$E_t(x) \rightarrow \text{Erosion 1} \rightarrow \text{Dilatation 1} \rightarrow \text{Dilatation 2} \rightarrow \text{Erosion 2}$

Les étapes en vert représentent l'ouverture.

Les étapes en rouge représentent la fermeture.

- Visualisation de Ero1, Dil1, Dil2, Ero2 : [CLIQUEZ ICI](#)

L'érosion et la dilatation ont tendance à modifier fortement la structure d'une image. Pour réduire cet effet, on les utilise souvent en combinaison : OUVERTURE (érosion puis dilatation) puis FERMETURE (dilatation puis érosion).

La particularité de l'ouverture et la fermeture morphologiques est **l'idempotence**, cela veut dire que le résultat ne change pas même si on applique plusieurs fois ces opérateurs. Ces deux derniers changent relativement peu la forme des grosses particules. Par contre, elles éliminent les petites particules isolées (le bruit) qui est souvent un effet indésirable de la binarisation.

8. Conclusion :

Ce projet fut l'occasion pour nous de découvrir des aspects très intéressants de détection de mouvement. Nous avons vu diverses notions de traitement d'images, avec des exemples concrets de chaînes de traitements qui nous ont permis d'isoler les pixels en mouvements des pixels immobiles. De plus, nous avons eu l'occasion d'améliorer nos connaissances en optimisation, en debugging, en benchmark et d'approfondir nos connaissances du langage C. En plus de cela, l'aspect très pratique de cette UE, nous à permis de penser de manière critique afin de réussir à souligner l'impact de certaines techniques d'optimisation sur la performance.

Nous remercions nos enseignants en TD et en TME pour l'aide qu'ils nous ont fourni pendant tout le semestre.

**MERCI POUR
VOTRE ATTENTION :)**