

# Rapport projet détection de mouvement

**Réalisé par**

Rayan BOUCHERRAB

Yanis HALIT

**Sous la direction de**

Quentin MEUNIER

Lionel LACASSAGNE



# **SOMMAIRE**

## **1) Réalisations, organisation et exécution**

## **2) Présentations et comparaisons**

- Optimisations bas niveaux sur la version basique de max/min
- Optimisations bas niveaux sur optimisations haut niveaux
- Optimisations des formats de calculs et de stockage en mémoire

## **3) Synthèse**

- Expérimentations sur la séquence "car3"


## **4) Conclusion**

# I) Réalisations, organisation et exécution

## Réalisations

Dans le cadre de ce projet, nous avons réalisé une série de travaux afin d'implémenter et d'optimiser la chaîne de traitement dont il est question dans l'énoncé pour enfin pouvoir les comparer de la façon la plus représentative possible.

1) SD+morpho en scalaire sans optimisations. 


2) Transformations de bas niveaux (transformations de boucles) 

(reg/rot/red/ilu3/ilu3\_red/elu2\_red/elu2\_red\_factor/ilu3\_elu2\_red/ilu3\_elu2\_red\_factor).

3) Transformation de haut niveaux avec **pipeline** et **fusion** d'opérateurs   
avec toutes leurs différentes optimisations de bas niveaux pour l'ouverture et la fermeture :

-**Pour le pipeline** : (basic/red/ilu3\_red/elu2\_red/elu2\_red\_factor/ilu3\_elu2\_red/ilu3\_elu2\_red\_factor).


-**Pour la fusion** : (basic/ilu5\_red/ilu5\_elu2\_red/ilu5\_elu2\_red\_factor).

-Optimisation des formats de calcul et de stockage en mémoire avec l'utilisation de **SWP 8/16/32** 

**Pour morpho max/min** avec la totalité des transformations bas niveaux

**Pour le pipeline d'opérateurs** avec la totalité des transformations bas niveaux

**Pour la fusion d'opérateurs** avec l'ensemble des transformations bas niveaux, excepté ilu5\_elu2\_red\_factor et ilu15\_red.

-Benchmark quantitatifs en faisant varier les tailles des images   
et en traçant les courbes cpp (cycle par point) en fonction de la taille de celle-ci avec les diverses optimisations bas niveaux appliquées.

-**Pour morpho max et ouverture.**

-**Pour les versions SWP (8/16/32) de morpho max** avec le pack et l'unpack comptabilisés et non comptabilisés dans le bench .

-**Pour les versions SWP (8/16/32) de l'ouverture** avec le pack et l'unpack comptabilisés et non comptabilisés dans le bench.

-**Pour motion detection** avec utilisation des transformations bas niveaux (érosion -> dilatation -> dilatation -> érosion) et des transformations hauts niveaux (ouverture -> fermeture) en SWP8 sur la séquence "car3", avec un test de débit (nombre de pixels traités en une seconde).

# Organisation

-Pour l'organisation, que ce soit pour morpho\_min/max, ouverture, fermeture les versions SWP 8/16/32 de chaque optimisation bas/hauts niveaux, ont été ajoutés dans le même fichier que les versions sans cette optimisation des formats de calculs et de stockage en mémoire.

Donc, dans les fichiers ./src/morpho\_max.c | ./src/morpho\_min.c | ./src/morpho\_ouverture.c | ./src/morpho\_fermeture.c et leur fichiers ".h" correspondants.

-Pour les tests et benchmarks des versions SWP, nous avons pris le soin d'ajouter les fichiers "./src/swp\_test.c" et "./include/swp\_test.h" qui se présentent sous la même forme que "./src/morpho\_test.c" et réalisant les mêmes opérations/tests.

-Pour le fichier "./src/motions.c", seules les fonctions de bench ont été ajoutés selon les versions à tester.

# Exécution

Pour l'exécution des différents algorithmes/tests/benchmarks réalisés, il faudra au préalable dé-commenter dans le fichier "./src/main.c" dans la fonction "main" le test à réaliser

```
// =====  
int main(int argc, char *argv[])  
// =====  
{  
    test_morpho(argc, argv);  
    // test_swp(argc, argv);  
    // test_motion(argc, argv);  
    return 0;  
}
```

Qui selon la la fonction appelée appelle le fichier "morpho\_test.c", "swp\_test.c" ou "motion\_test.c" et donc implicitement "motion.c"

Il s'agira ensuite de dé-commenter l'appel de fonction désiré.

```
void motion_detection_morpho(void)  
// =====  
{  
    // motion_detection_morpho_v1(); .....  
    // bench_motion_detection_morpho(); .....  
    bench_motion_detection_morpho_SWP8(); .....  
    // bench_motion_detection_morpho_SWP16(); .....  
    // bench_motion_detection_morpho_SWP32(); .....  
    test_PGM();  
}
```

```
int test_swp(int argc, char *argv[])  
{  
    puts("=== test_swp ===");  
    // test_swp_morpho_max();  
    // test_swp_morpho_min();  
    // test_swp_morpho_ouverture();  
  
    // MAX  
    // BENCH IN  
    bench_swp8_morpho_max_in(128, 512, 8);  
    bench_swp16_morpho_max_in(128, 512, 8);  
    bench_swp32_morpho_max_in(128, 512, 8);  
  
    bench_swp8_morpho_max_out(128, 1024, 8);  
    bench_swp16_morpho_max_out(128, 1024, 8);  
    bench_swp32_morpho_max_out(128, 1024, 8);  
  
    // BENCH OUT  
    bench_swp8_morpho_max_out(128, 512, 8);  
    bench_swp16_morpho_max_out(128, 512, 8);  
    bench_swp32_morpho_max_out(128, 512, 8);  
  
    bench_swp8_morpho_max_out(128, 1024, 8);  
    bench_swp16_morpho_max_out(128, 1024, 8);  
    bench_swp32_morpho_max_out(128, 1024, 8);  
  
    // OUVERTURE
```

```
int test_morpho(int argc, char *argv[])  
{  
    puts("=== test_morpho ===");  
    // test_morpho_max();  
    // test_morpho_min();  
    // test_morpho_ouverture();  
  
    // OUVERTURE  
    bench_morpho_ouverture(128, 512, 8);  
    bench_morpho_ouverture(128, 1024, 8);  
  
    // MAX  
    bench_morpho_max(128, 512, 8);  
    bench_morpho_max(128, 1024, 8);  
  
    // AUTRES  
    test_set_str();  
    test_wikipedia();  
}
```

# Matériels et conditions de travaux

La machine utilisée pour l'ensemble du projet présente ces caractéristiques



Parmi les macros utilisées on trouve :

"swp.h"

"nrdef.h"

```
// mettre les macro left/right ici
// -----
#define i8left(a, b) (a >> (8-1)) | (b << 1)
#define i16left(a, b) (a >> (16-1)) | (b << 1)
#define i32left(a, b) (a >> (32-1)) | (b << 1)
#define i8right(b, c) (b >> 1) | (c << (8-1))
#define i16right(b, c) (b >> 1) | (c << (16-1))
#define i32right(b, c) (b >> 1) | (c << (32-1))
```

```
#define load(X, i) X[i]
#define load2(X, i, j) X[i][j]
#define store1(X, i, x) X[i] = x
#define store2(X, i, j, x) X[i][j] = x
#define min3(x, y, z) x & y & z
#define min2(x, y) x & y
#define max3(x, y, z) x | y | z
#define max2(x, y) x | y
```

Les optimisations du compilateur ont été mises en O(0).

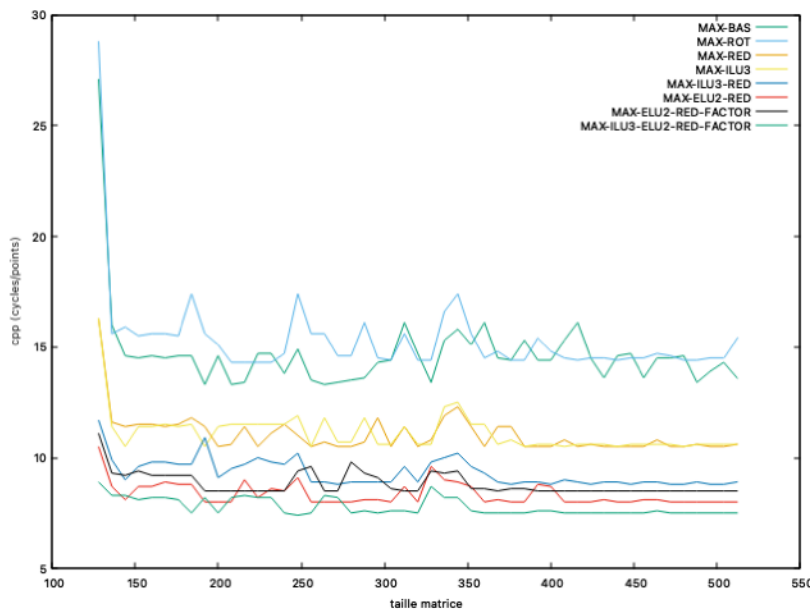
```
# -- Flags -----
#C_OPTIMISATION_FLAGS = -std=c99 -O3 -fstrict-aliasing -Wno-comment
C_OPTIMISATION_FLAGS = -std=c99 -O0 -fstrict-aliasing -Wno-comment
C_ARCH_FLAGS = -mtune=native -march=native
C_ARCH_FLAGS = -g
C_INC_FLAGS = -I$(INC_PATH)
```

## II) Présentations et comparaisons

### 1) Optimisations bas niveaux sur la version basique de max/min

*Consistant en des transformations de boucles afin d'en optimiser la durée (rot/red/ilu3..).*

**Ici, le bench est réalisé sur des matrices carrés de tailles allant de 128x128 à 512x512.**



On remarque qu'au début, toutes les versions présentent une certaine "latence" liée au chargement des premiers pixels dans le cache car n'y étant pas présents, une fois que les données y sont présentes, celles-ci sont plus rapidement accessible par le CPU ce qui s'explique par une courbe décroissante (moins de cycles pour effectuer un/des load/s une fois les données présentes car, chargées précédemment en ligne dans le cache).

Le même phénomène est observable avec l'augmentation progressive de la taille des matrices expliquant l'apparition de "pics" qui correspondent au chargement des données/instructions directement dans la mémoire (suite à un MISS de cache) qui selon le niveau d'optimisation appliquée influence la performance de traitement d'un pixel.

En effet le traitement de chaque pixel dépend des 8 autres pixels qui lui sont voisins, les différences de cpp dépendent donc du nombre d'accès mémoire et du nombre d'instructions/opérations réalisées (ici max3 donc deux "or" logiques).

La version MAX\_BAS se trouve plus performante que la version MAX\_ROT simple quand l'information à charger est déjà présente dans le cache, en effet la version MAX\_ROT pose un problème de latence générée par la dépendance de variable.

La version MAX\_RED quand à elle limite le nombre d'opérations (instructions) effectuées et donc chargées dans le cache d'instructions, ce qui explique ses meilleures performances,

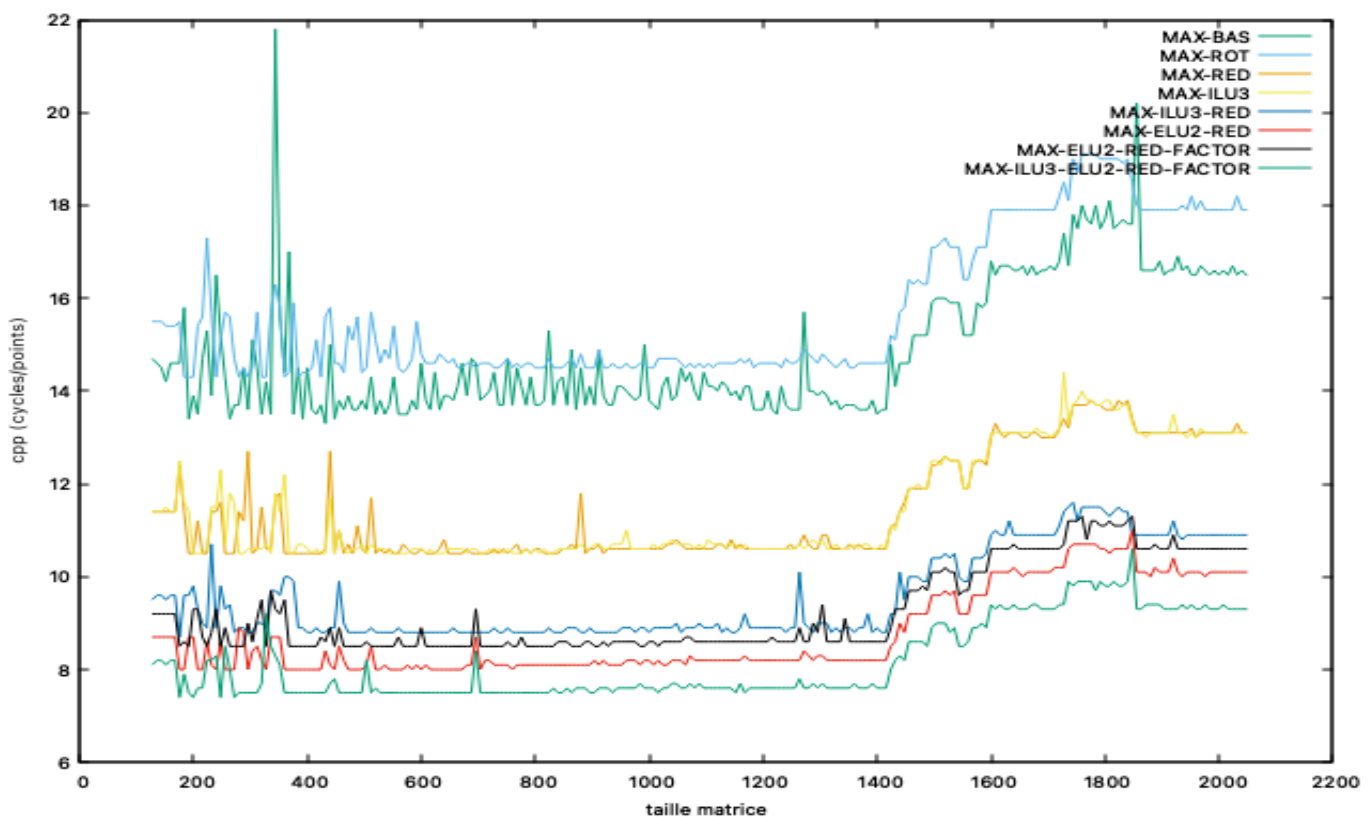
La version MAX\_ILU3 (déroulage de boucle) supprime les rotations des variables ce qui réduit la latence en plus d'accélérer la boucle car, traitant 3 pixels par itérations diminuant donc la taille de la boucle (on souhaite y rester le moins longtemps possible). Cette dernière, couplée à l'optimisation MAX\_RED permet en plus, de réduire le nombre d'opérations effectuées en réutilisant les résultats déjà calculés.

La version MAX\_ELU2\_RED (traitement de deux lignes par appel) effectue moins de loads que la version MAX\_ILU3\_RED qui, malgré la présence de rotation est plus performante que MAX\_ILU3\_RED.

En comparant MAX\_ELU2\_RED et ELU2\_RED\_FACTOR on peut constater que la factorisation d'opérateur ne présente pas de grand intérêt car utilisant un nombre plus élevé de variables et donc de registres..

Naturellement la combinaison de MAX\_ILU3\_RED et ELU2\_RED qui sont les plus performantes nous donne la "meilleure" version avec MAX\_FACTOR.

**Ici, le bench est réalisé sur des matrices carrés de tailles allant de 128x128 à 2048x2048.**



Plus globalement en observant les résultats obtenues sur des tailles plus grandes de matrices, on constate qu'une fois la taille 500x500 dépassée les fluctuations de cpp sont assez constantes jusqu'à la taille 1400x1400 (les pics liés au chargement data/instructions sur cet intervalle peuvent être négligés).

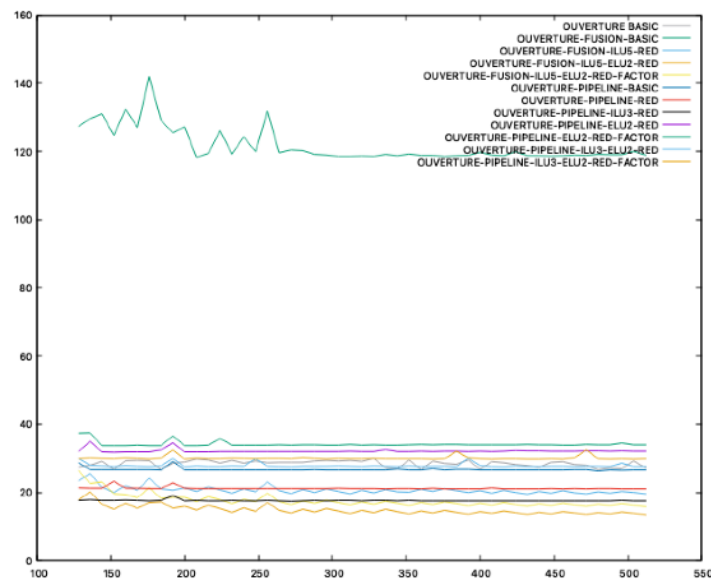
Le cache L1 (instruction/data) étant limité en taille, peut se trouver saturé induisant à la suppression puis le chargement de nouvelles données/instructions. Cela est observable à partir de la taille 1400x1400 où le cpp augmente pour toutes les versions mais à des allures différentes selon le degré d'optimisation appliqué.

## 2) Optimisations bas niveaux sur optimisations haut niveaux (ouverture)

Consistant en des transformations de boucles (rot/red/ilu3..) superposé à l'utilisation de pipeline et fusion d'opérateurs qui sont effectués en 3X3.

En sachant que le pipeline d'opérateurs permet de réduire la durée des accès mémoire en maximisant la réutilisation du cache (typiquement on aura moins de pic cars nous effectuons un nombre plus ou moins important de "HITs" (informations présente dans le cache)), tandis que la fusion d'opérateurs quant à elle permet de limiter le nombre d'accès à la mémoire.

Ici, le bench est réalisé sur des matrices carrés de tailles allant de 128x128 à 512x512.

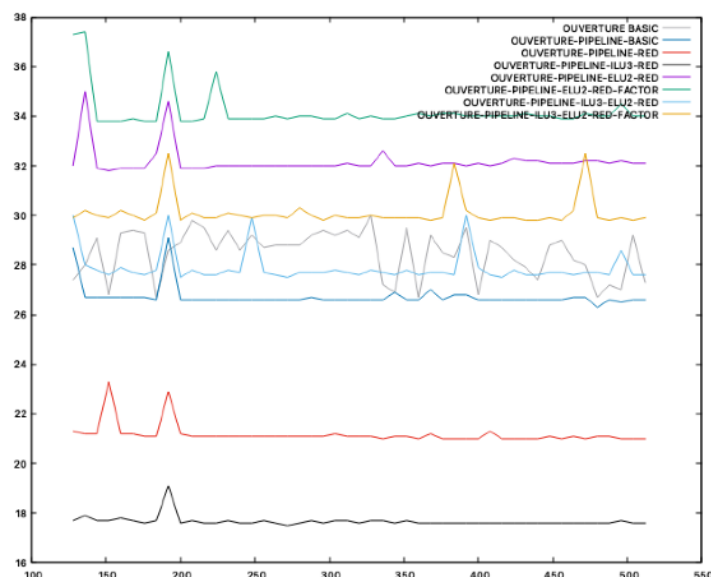


On constate aisément que la version ouverture fusion basique est très peu performante par rapports aux autres versions.

En effet celle-ci, sans autres optimisations de bas niveaux effectue un grand nombre de loads et d'instructions.

Pour comparer, le traitement d'un pixel avec la version `OUVERTURE_BASIC` effectue 9x2 loads sur chaque traitement d'un pixel, tandis que la version `OUVERTURE_FUSION_BASIC` en effectue 81..

Ici, nous avons supprimé les résultats obtenus par la fusion d'opérateurs.



Pour des matrices de tailles réduites, on constate que certaines optimisations bas niveaux du **pipeline** sont inutiles voir, moins performantes que la version `OUVERTURE_BASIC` (ici en gris) on y retrouve :

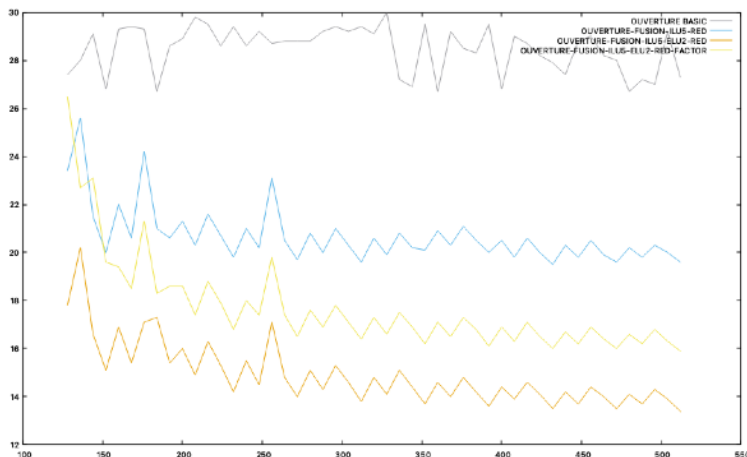
`OUVERTURE_PIPELINE_EL2_RED_FACTOR`,  
`OUVERTURE_PIPELINE_EL2_RED`,  
`OUVERTURE_PIPELINE_ILU3_EL2_RED_FACTOR`.

Donc, on peut dire que pour le pipeline, les versions ELU2 ne maximisent pas l'utilisation du cache pouvant s'expliquer par la manière dont sont chargées les données/instructions.

Pour le pipeline on ne gardera donc que les versions `OUVERTURE_PIPELINE_ILU3_RED`, `OUVERTURE_PIPELINE_RED` et `OUVERTURE_PIPELINE_ILU3_EL2_RED`



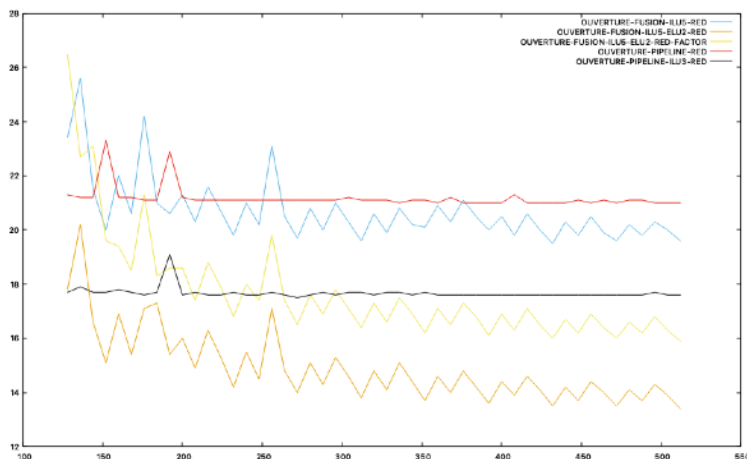
Ici, nous avons supprimé les résultats obtenus par le pipeline d'opérateurs.



On remarque que quelque soit la version de fusion **couplé** à des transformations de bas niveaux, celle-ci se trouve plus performante en terme de cpp par rapport à la version OUVERTURE\_BASIC, en effet celles-ci permettent de limiter le nombre d'accès mémoire et donc le nombre de cycles pour charger les données/instructions et donc, traiter un pixel.

La version OUVERTURE\_FUSION\_ILU5\_ILU2\_RED\_FACTOR n'ajoute que 5 accès mémoires et 15 instructions supplémentaires dans la boucle pour traiter un **nouveau pixel** de lignes contiguë ce qui représente une optimisation non négligeable (courbes orange (ilu5\_elu2\_red présentant de meilleurs résultat que la courbe bleu (ilu5\_red) ).

Ici, nous allons comparer les meilleures versions du pipeline et de la fusion

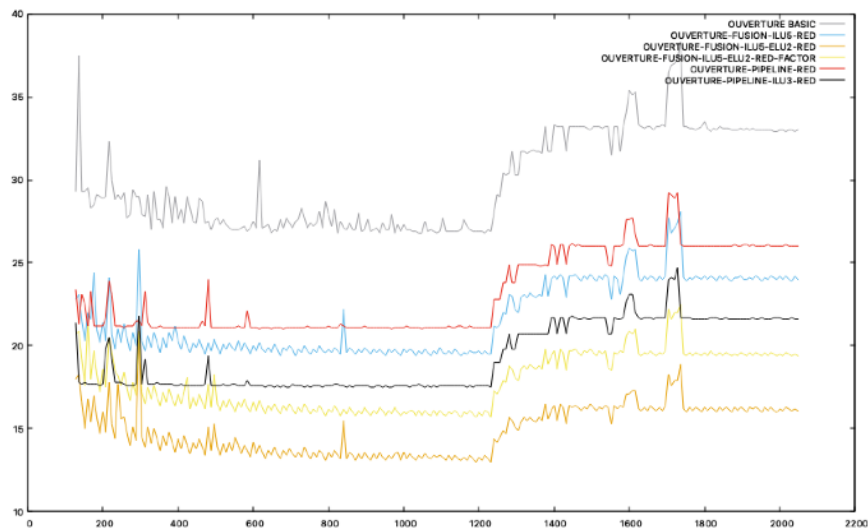


On sait déjà que plus généralement, le pipeline d'opérateurs est moins efficace que la fusion et cela est visible sur ce graphique.

En comparant OUVERTURE\_PIPELINE\_ILU3\_RED et OUVERTURE\_FUSION\_ILU5\_ILU2\_RED on voit que la version pipeline présente moins de pics ( utilisation plus accrue du cache mémoire ), contrairement à la version fusion qui n'optimise pas l'utilisation du cache mais, réalisant moins de chargement de données/instructions.

On peut donc en déduire qu'une optimisation visant à diminuer le nombre de chargement des données/instructions peut être plus performante qu'une optimisation qui favorise et maximise l'utilisation du cache malgré sa proximité au CPU (moins de cycles pour charger des informations).

**Ici, le bench est réalisé sur des matrices carrés de tailles allant de 128x128 à 2048x2048.**



Les observations précédentes s'appliquent pour des tailles allant de 400x400 à 1200x1200 (constance du cpp, pics réduit pour le pipeline..) .

À partir des tailles supérieures à 1200x1200, les cpp se voient augmentés (sortie du cache après saturation) pour toutes les versions mais à des proportions différentes selon le niveau d'optimisation appliqué.

Nous gardons le même constat/conclusion que pour les matrices de petites tailles :

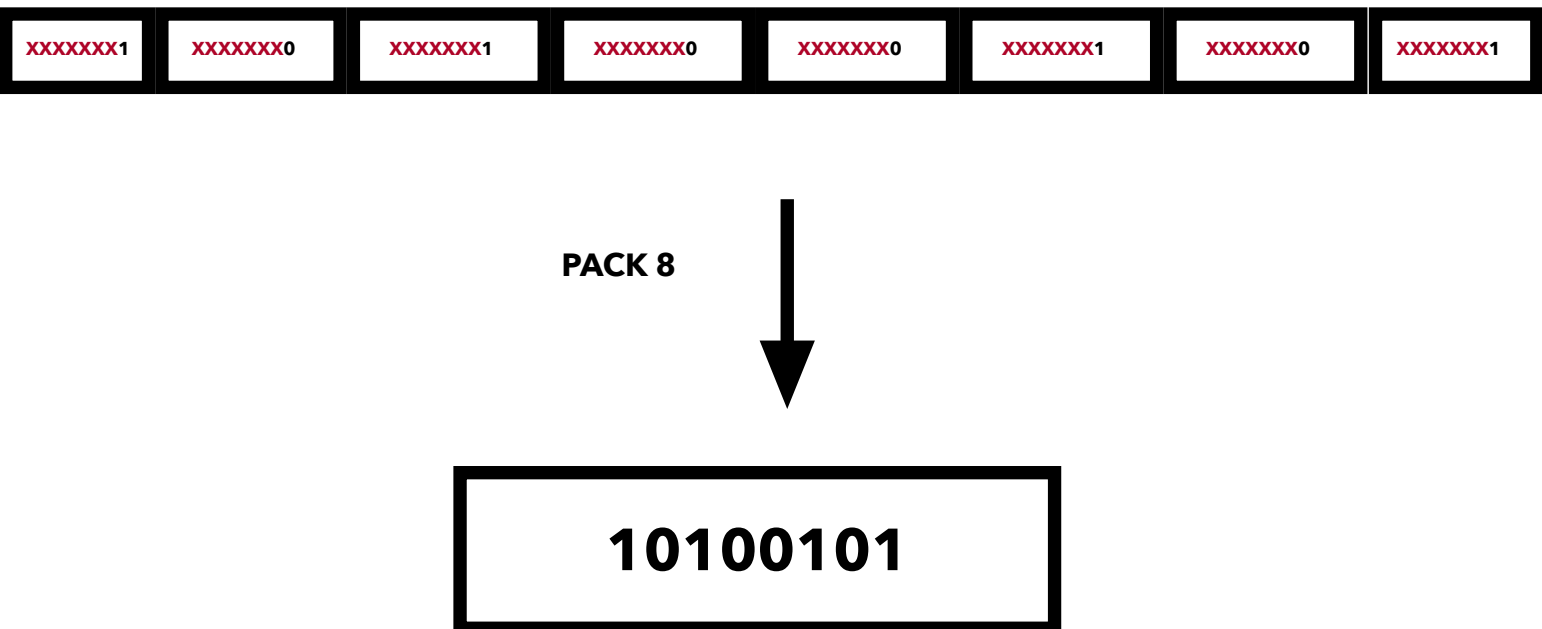
**La fusion se trouve plus performante que le pipeline  
( limiter les accès mémoires > favoriser l'utilisation du cache )**

### 3) Optimisations des formats de calculs et de stockage en mémoire (SWP 8/16/32)

*Dans les optimisations vu précédemment, nous avons utilisé des matrices où chaque ligne est un tableau de valeurs 8-bit correspondant chacune à UN pixel nous donnant l'information sur 1 bit. Avec ce modèle d'implémentation, nous sommes amenés à charger et à effectuer des calculs sur des cases de 8 bits, dont l'information utile n'est que sur le bit de poids faible.*

*L'intérêt du SWP est de transformer nos matrices de 8bits dont 1bit est utile en une matrice moins "large" qui, selon la version (8/16/32) nous renvoie après le "pack" une matrice où chaque valeurs (case) contient 8, 16 ou 32 bits utiles, représentant chacun un pixel différent.*

*On diminue donc considérablement le nombre d'accès mémoire ainsi que le nombre d'instructions (un or au lieu de 8 16 ou 32 ).*

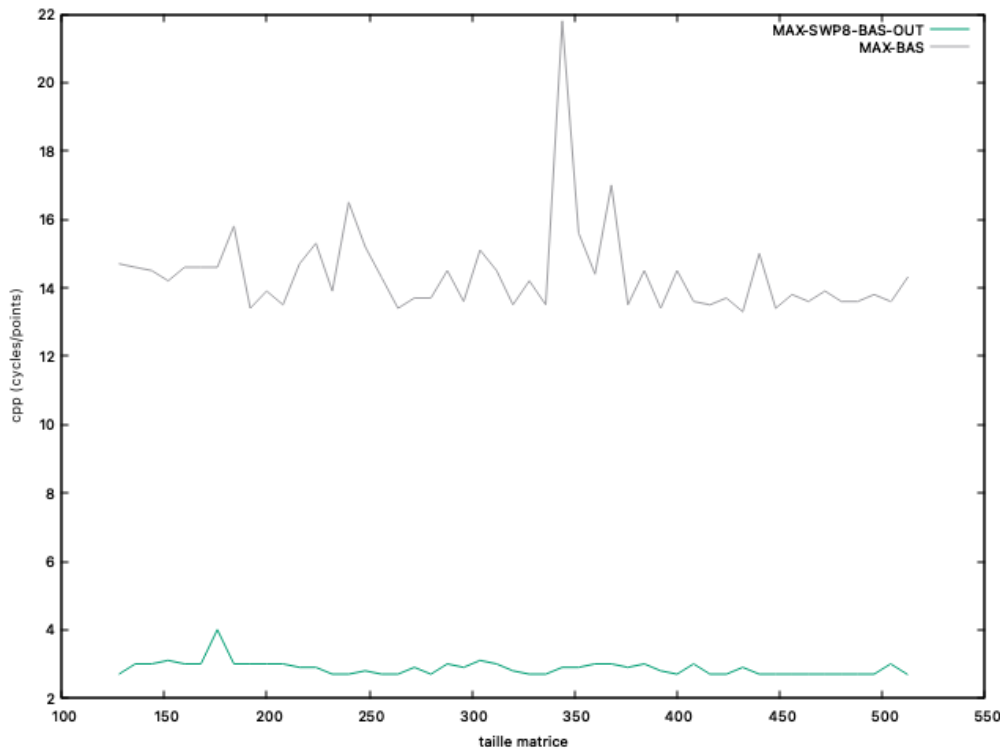


**En théorie en utilisant SWP 8/16/32 pour les mêmes optimisations bas niveaux/haut niveaux le temps de traitements doit être divisé par 8/16/32 par rapport aux versions n'utilisant pas SWP.**

**L'utilisation de SWP 16 divise le temps de traitement par 2 par rapport à SWP 8, celle de SWP 32 divise le temps de traitement par 2 par rapport à SWP 16..**

## SWP MAX avec appel de pack et unpack non compris dans le bench

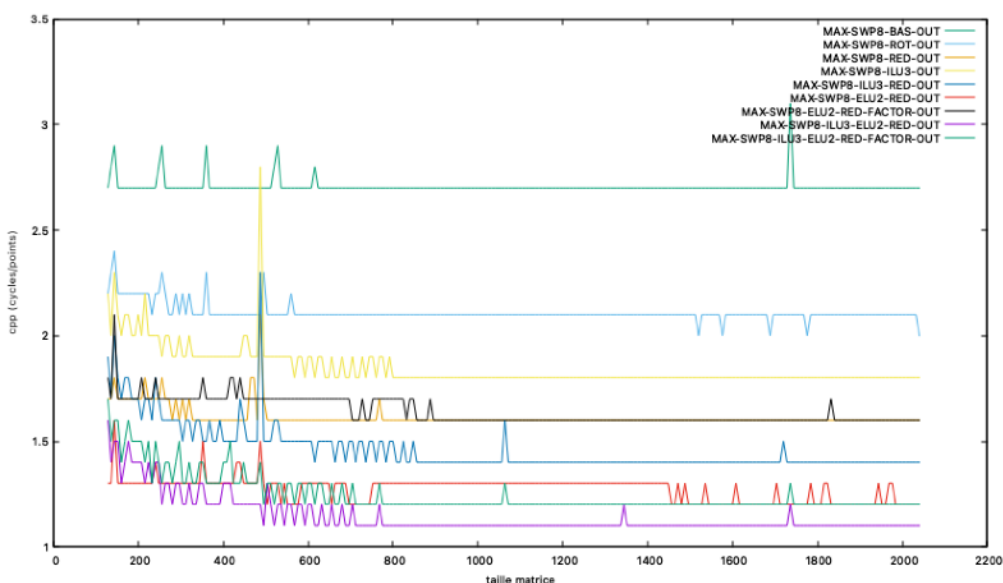
Ici, le bench est réalisé sur des matrices carrées de tailles allant de 128x128 à 512x512.



On va considérer la version basique du traitement morpho MAX (ici en gris) ainsi que la version SWP8 basique pour le même traitement (ici en vert) sans autres optimisations de bas/haut niveaux.

On observe que le cpp a été divisé par un facteur compris entre 6-7 avec l'utilisation de SWP8 ce qui est cohérent avec l'hypothèse de départ.

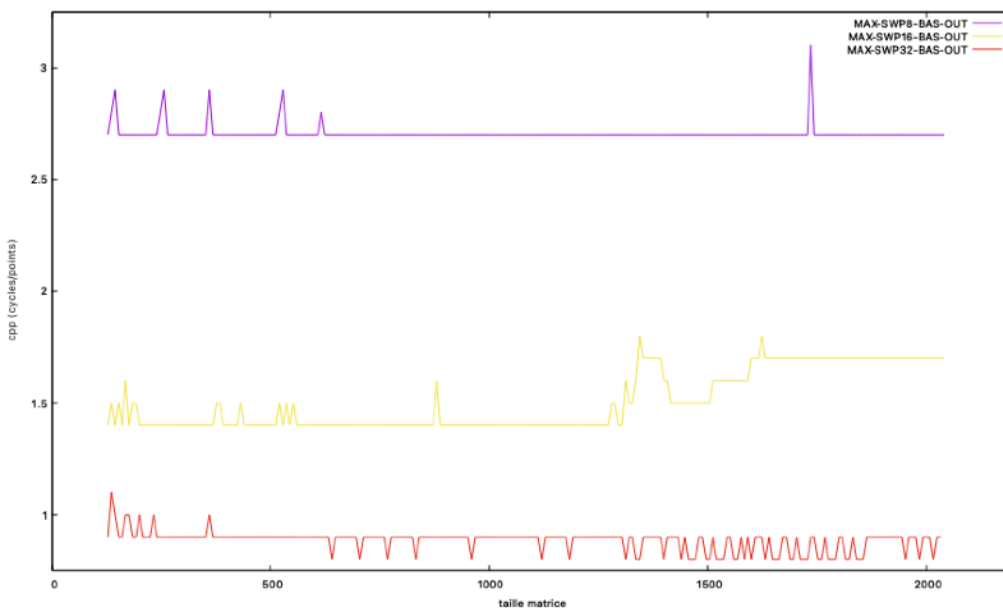
Ici, le bench est réalisé sur des matrices carrées de tailles allant de 128x128 à 2048x2048 avec SWP8.



Ici on peut voir que les optimisations bas\_niveaux apportent les mêmes différences de performance que le bench effectué sur le traitement sans SWP.

On constate par ailleurs que l'intervalle où le cpp est plus ou moins constant est plus étendu que pour les versions sans SWP. On peut imaginer que le sortie du cache (suite à sa saturation) s'effectuera plus loin pour des matrices de tailles plus importantes (approximativement 2048x8 X 2048x8).

Ici, le bench est réalisé sur des matrices carrées de tailles allant de 128x128 à 2048x2048 pour comparer les versions basiques de SWP8, SWP16 et SWP32

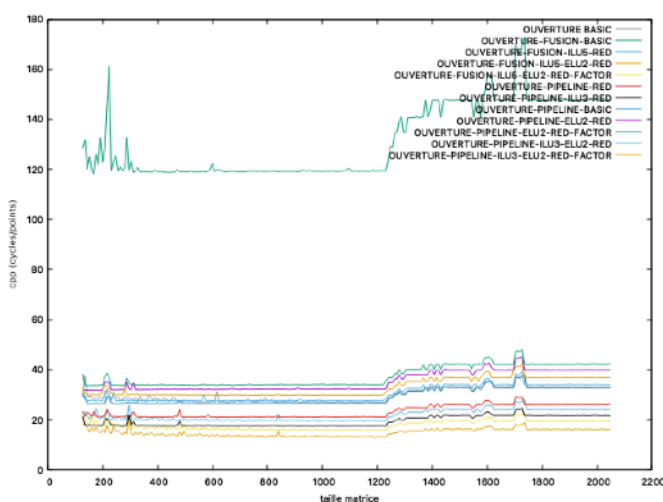


On voit bien l'amélioration apportée par chaque version de SWP, la courbe violette correspondant au SWP8 voit son cpp divisé par 2 en utilisant SWP16 courbe jaune (passant de 2.7 à 1.5 ce qui donne un facteur de 1.8).

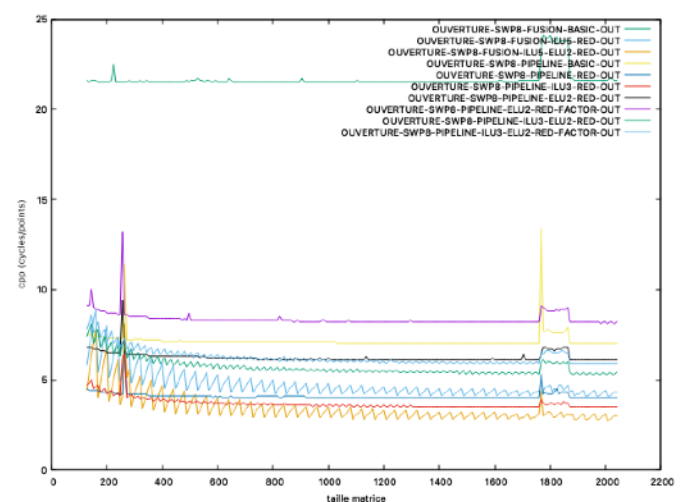
On observe la même chose en comparant SWP16 (courbe jaune) et SWP32 (courbe rouge) (Passant d'un temps cpp de 1.5 à 0.8 ce qui donne un facteur 1.875).

## SWP OUVERTURE avec appel de pack et unpack non compris dans le bench

Ici nous devons obtenir les mêmes améliorations apportées par les optimisations bas niveaux sur le pipeline et la fusions d'opérateurs en utilisant SWP 8/16/32 mais avec un temps de traitement réduit à l'instar de l'application de SWP\_MAX.



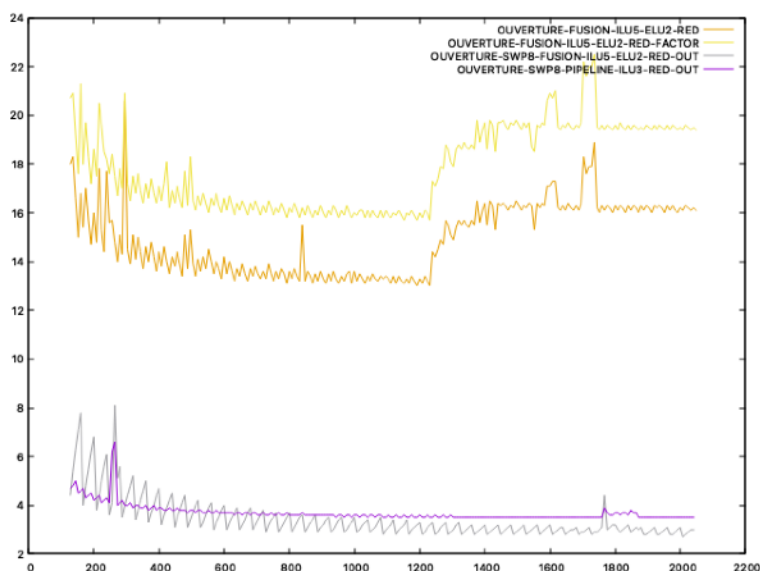
Sans SWP



SWP 8

**Donc, nous effectuerons la comparaison en nous basant sur les versions les plus performantes du pipeline et de la fusion d'opérateur partagé en SWP et non SWP.**

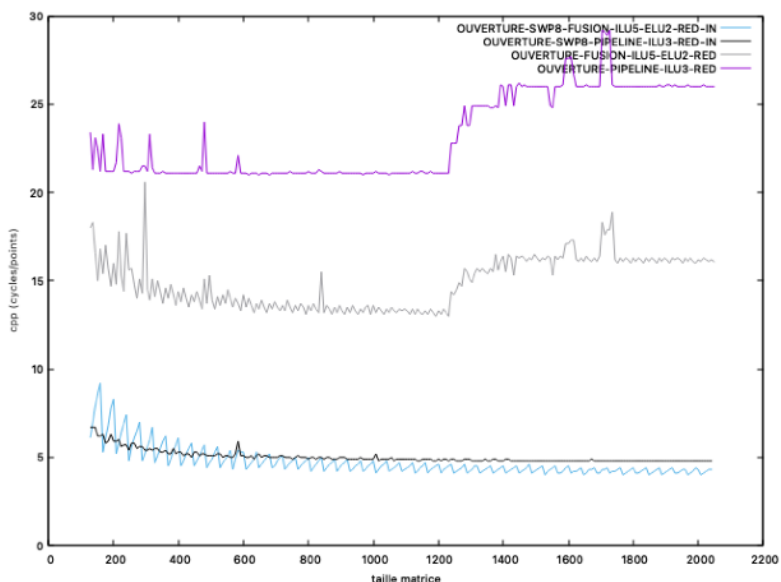
**Ici, le bench est réalisé sur des matrices carrées de tailles allant de 128x128 à 2048x2048 pour comparer les versions OUVERTURE\_SWP8\_FUSION\_ILU5\_ELU2\_RED\_OUT et OUVERTURE-SWP8-PIPELINE-ILU3-RED-OUT en SWP8 à ces mêmes version sans SWP**



On remarque la même réduction (division) du cpp pour le traitement entre la version sans SWP8 et celle avec SWP8 pour les mêmes optimisations de haut et bas niveaux.

### **Comparaison pour l'ouverture entre les versions FUSION ILU5 ELU2 RED OUT et PIPELINE-ILU3-RED-OUT sans SWP et avec SWP8 avec appel de pack et unpack compris dans le bench**

*Ce test permettant de montrer plus explicitement le gain de performance des version SWP car comprenant ici le traitement de pack et un pack dans le bench (l'handicapant par conséquence).*



En calculant le facteur de division entre versions sans SWP et celles avec SWP8 on constate que ce dernier diminue à cause de l'inclusion du pack unpack dans le calcul du cpp lors du bench.

Cette version reste néanmoins plus efficace que la version classique n'utilisant pas SWP.

# III) Synthèse

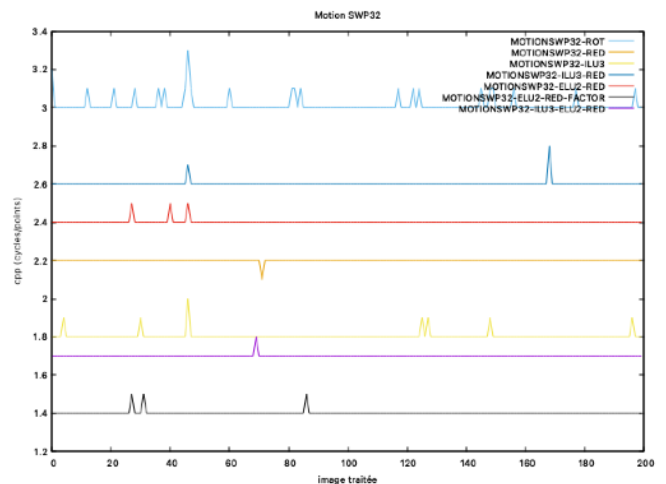
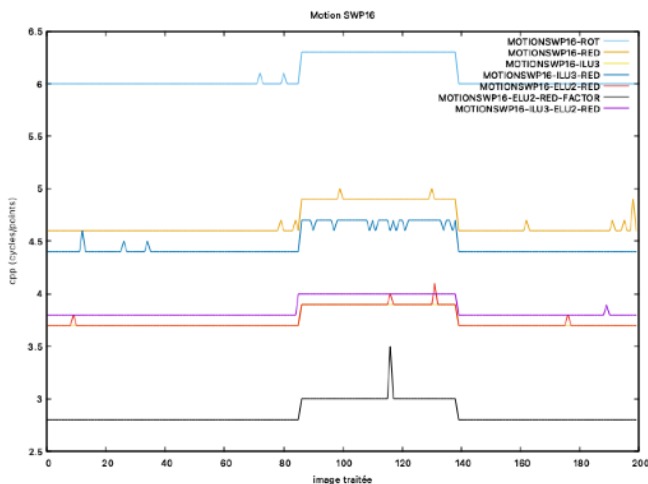
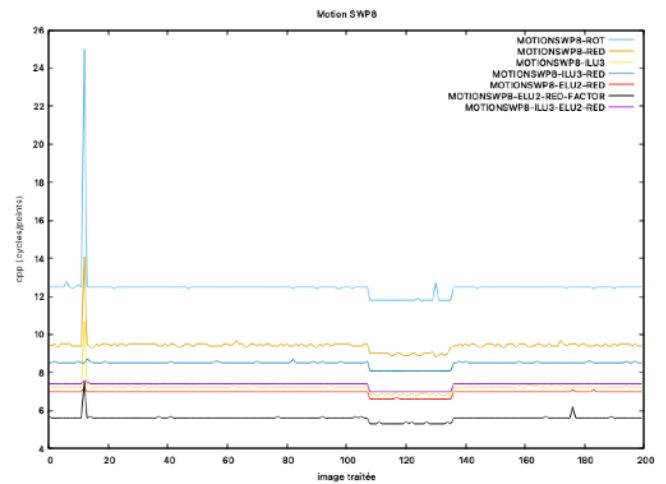
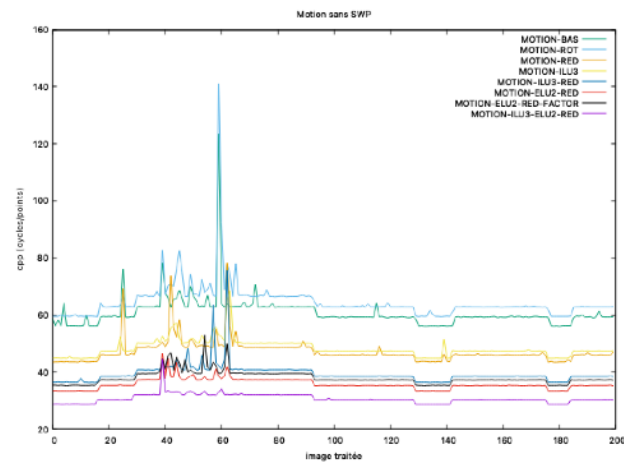
Dans un premier temps, nous allons appliquer des benchs sur la séquence "car3" en effectuant le traitement :

- SD
- min -> max (ouverture)
- max -> min (fermeture)

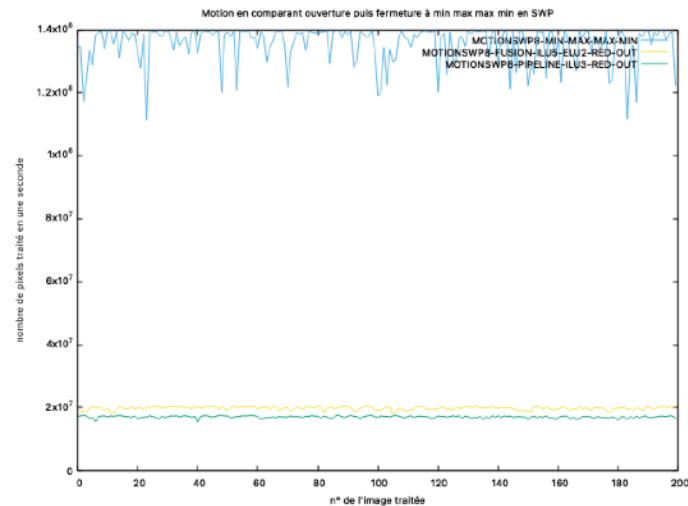
Nous utiliseront directement les fonctions max3 et min3 avec leurs optimisations de bas niveaux.

Les images traitées sont au nombre de 200 et son relativement petites (320x240), les versions utilisées sont :

*SANS SWP* : BAS, ROT, RED, ILU3, ILU3\_RED, ELU2\_RED, ELU2\_RED\_FACTOR  
*AVEC SWP 8/16/32* : BAS, ROT, RED, ILU3, ILU3\_RED, ELU2\_RED, ELU2\_RED\_FACTOR



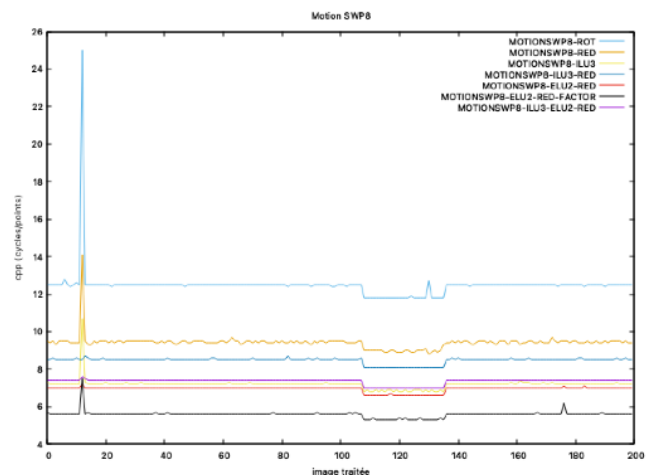
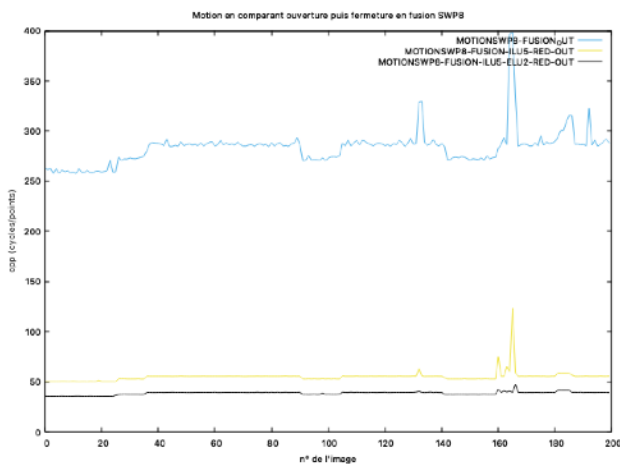
Dans un second temps nous allons appliquer le bench sur la même séquence d'images "car3" mais en utilisant cette fois-ci la version la plus optimisée de la fusion et du pipelines d'opérateurs pour SWP8 donc OUVERTURE\_SWP8\_FUSION\_ILU5\_ELU2\_RED\_OUT (resp FERMETURE) et OUVERTURE-SWP8-PIPELINE-ILU3-RED-OUT que l'on comparera au même traitement mais en utilisant  $min3\_swp\_ui8matrix\_ilu3\_elu2\_red\_factor \rightarrow max3\_swp\_ui8matrix\_ilu3\_elu2\_red\_factor \rightarrow max3\_swp\_ui8matrix\_ilu3\_elu2\_red\_factor \rightarrow min3\_swp\_ui8matrix\_ilu3\_elu2\_red\_factor$  sur la séquence "car3".



Ici, on peut remarquer que l'application de la transformation sans utiliser d'optimisation haut niveaux (pipeline/fusion) est largement préférable, traitant  $1.4 \times 10^8$  pixels par seconde contre  $2 \times 10^7$  en appliquant une ouverture puis une fermeture avec la meilleure version pipeline et la meilleure version fusion.

Pour comparer, nous avons analysé le cpp pour une ouverture puis fermeture en utilisant la fusion avec optimisations de bas niveaux en SWP8 qui nous donne au mieux un cpp de 40 cycles/points, tandis que la pire version en utilisant min puis max puis max et enfin min en SWP8 (ici version en rotation) nous donne un cpp de 13 cycles/points.

Cela pourrait s'expliquer d'une part par la taille des nos images qui sont invariables ici et relativement petites, d'autre part en explorant le code de plus près on s'aperçoit que l'enchaînement  $min \rightarrow max \rightarrow max \rightarrow min$  en ILU3\_ELU2\_RED\_FACTOR effectue moins de loads et d'instructions que l'ouverture avec optimisation pipeline ou fusion.





**En conclusion nous pouvons dire que le gain apporté par chaque optimisation logiciel (de haut niveau plus particulièrement) peut toujours être discuté selon la nature du problème que l'on doit traiter, et ce malgré des observations effectuées sur des tests avec des données aléatoires (et donc variés) et de tailles différentes.**

**Une optimisation logiciel donnée ne peut donc, être considérée comme l'optimisation "parfaite" et cela se justifie notamment à travers ce dernier test enchainant ouverture puis fermeture.**

**Il est donc toujours nécessaire de tester et d'évaluer les performance de diverses optimisations avant de tirer des conclusions.**

**À contrario les optimisations dites matériels comme le SWP, reposant sur la modification des structures manipulées (cf page 11), semblent être une valeur sûr pour l'amélioration des performances en diminuant le nombre d'accès mémoire par rapport au nombre d'opérations.**