COMPTE RENDU
Projet Motion

Par Papa Talla Dioum

Table des matieres

1. Introduction / Présentation

Ce projet s'inscrit dans le cadre de l'optimisation d'une chaîne complète de détection et de suivi d'objets en mouvement dans des séquences vidéo. L'application étudiée, nommée *Motion*, repose sur une succession de traitements classiques de vision par ordinateur : détection du mouvement par l'algorithme Sigma–Delta (ΣΔ), filtrage morphologique, étiquetage de composantes connexes (CCL), analyse des régions d'intérêt (CCA), filtrage par surface, association par k-plus proches voisins (k-NN) et suivi temporel.

L'objectif principal du projet est d'augmenter le débit global de l'application, mesuré en images par seconde (FPS), tout en conservant une stricte équivalence fonctionnelle des résultats (images binaires, labels, RoIs et trajectoires). L'optimisation ne se limite pas à un unique noyau de calcul, mais concerne l'ensemble de la chaîne, avec une approche incrémentale, méthodique et mesurée.

2. Description détaillée des optimisations

2.1 Baseline et point de départ

La baseline retenue pour ce projet est motion2, qui correspond à une version de référence de l'application avant optimisation.

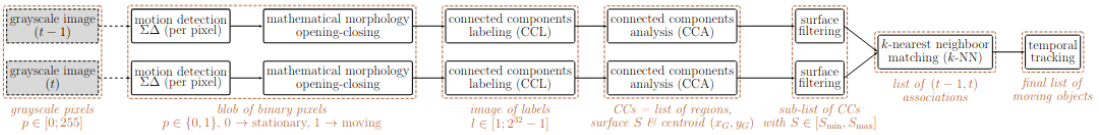


Fig 1: Graphe de la detection motion detection et du tracking processing

Ce code est lancé avec la commande

**./bin/motion2 --vid-in-path ./traffic/1080p_day_street_top_view_snow.mp4 **

--flt-s-min 2000 --knn-d 50 --trk-obj-min 5 --vid-out-play --vid-out-id

---

Performance mesurée :

<u>motion2</u> : 59.102 ms par image → 16.92 FPS

Cette version effectue un traitement redondant entre les instants t-1 et t, ce qui induit un surcoût inutile dans plusieurs étapes de la chaîne.

Donc nous allons partir de cette base pour écrire un nouvelle version motion.c simplifiée qui va ensuite servir de baseline pour nos optimisations.

2.2 Simplification de motion2 : suppression du traitement explicite à t-1

Dans motion2, la chaîne est historiquement appliquée à la fois sur l'image courante t et sur l'image précédente t-1, ce qui double des calculs. L'optimisation consiste à ne calculer la chaîne complète que pour l'image t, puis à « mémoriser » t comme étant t-1 pour l'itération suivante par permutation (swap) des pointeurs des buffers (IG0/IG1, IB0/IB1, L10/L11, RoIs0/RoIs1) et des compteurs n_RoIs0/n_RoIs1.

Concrètement : on supprime la section « Processing at t-1 » et on place la permutation des RoIs (et de n_RoIs) à la fin de chaque itération, après l'affichage/écriture des résultats. Cette optimisation réduit le nombre total d'opérations par frame sans changer la sémantique.

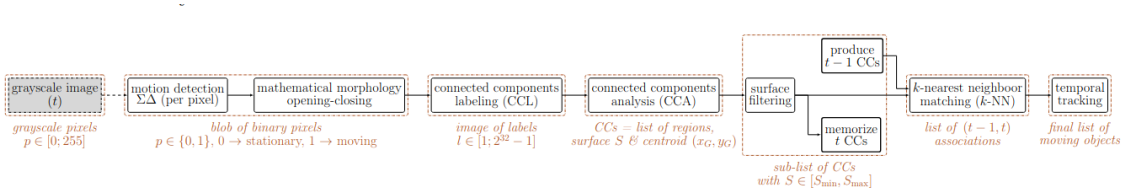


Fig 2 : Graphe de Motion simplifiée

Performances après optimisation :

```
# Average latencies:
# -> Video decoding =    0.200 ms
# -> Sigma-Delta    =   14.686 ms
# -> Morphology     =   11.004 ms
# -> CC Labeling    =    2.525 ms
# -> CC Analysis    =    1.179 ms
# -> Filtering      =    0.003 ms
# -> k-NN           =    0.006 ms
# -> Tracking       =    0.003 ms
# -> *Logs*         =    0.000 ms
# -> *Visu*         =    0.000 ms
# => Total          =   29.605 ms [~33.78 FPS]
```

- **motion (baseline optimisée)** : **29.605 ms → 33.78 FPS**

Gain immédiat : ×2 par rapport à motion2

En regardant de plus près les temps de latence, nous remarquons que les parties Sigma-Delta et Morphology prennent encore beaucoup de temps, nous allons donc par la suite essayer d'appliquer des techniques d'optimisation pour réduire au mieux le temps qu'elles prennent.

2.3 Première phase : parallélisation OpenMP progressive

2.3.2 Parallélisation OpenMP ciblée du Sigma–Delta

Le Sigma–Delta est identifié comme un premier goulot d'étranglement.

Les optimisations appliquées :

- ajout de "**#pragma omp for"** sur les boucles par lignes,

Résultat : **omp-sd Total = 21.840 ms [~45.79 FPS]**

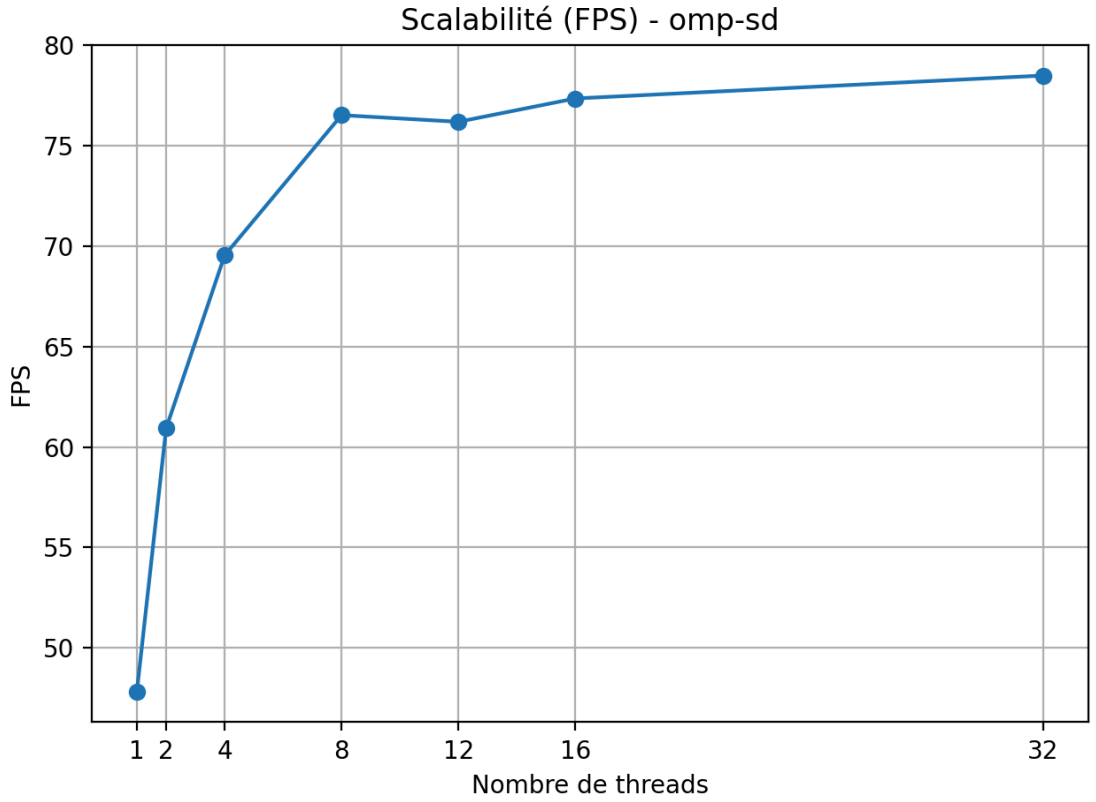
- regroupement de plusieurs boucles dans une région parallèle unique

Résultat : **omp-sd-v2 Total = 13.033 ms [~76.73 FPS]**

Cette étape montre déjà un gain significatif, mais met aussi en évidence le coût des multiples passes mémoire.

Nous testons ensuite Open MP pour différents nombres de Threads nous obtenons les resultats suivants:

Threads	**1**	**2**	**4**	**8**	**12**	**16**	**32**
Nombre de FPS	47.84	60.93	69.53	76.51	76.18	77.34	78.48



On voit bien que le résultat s'améliore nettement lorsqu'on fixe un nombre de thread plus grand.

2.3.3 Parallélisation OpenMP de la morphologie

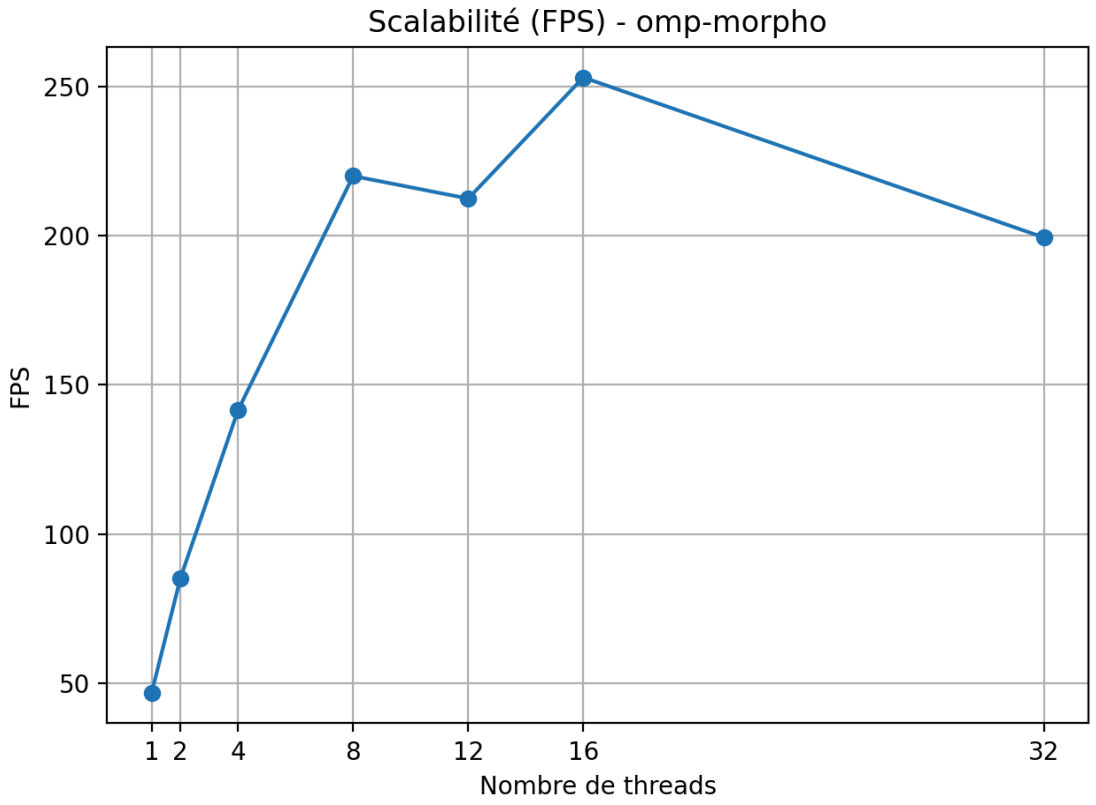
Pour la morphologie on applique exactement la même méthode qu'avec sigma delta c'est à dire ajouter des "**#pragma omp for"** sur toutes les boucles des opérateurs morphologiques (érosion et dilatation).

Resultats immédiat: **5.270 ms [~189.75 FPS]**

Ensuite on supprime les parallélisations inutiles sur les bordures et on fait une parallélisation OpenMP ciblée uniquement sur le cœur de l'image, le FPS monte à 224.78**.**

Ensuite on fait des tests sur différents threads et on obtient les résultats suivants:

	Threads	1	2	4	8	12	16	32
	Nombre de FPS	46.93	85.28	141.52	219.97	212.42	253.00	199.38



On remarque que le FPS augmente bien avec le nombre de threads jusqu'à 16 puis baisse lorsqu'on passe à 32. Cela pourrait s'expliquer par l'architecture matérielle de notre système. EN effet, sur Dalek (i9-13900H), on a :

- 14 cœurs physiques
- 20 threads matériels
- 1 socket, mémoire partagée
- Caches :
    - L1 / L2 par cœur
    - L3 partagé par tous les cœurs

A partir d'un certain nombre de threads :

- les threads partagent les mêmes unités d'exécution
- ils se disputent les caches
- ils ne doublent plus la puissance de calcul

La morphologie atteint donc un maximum de performance autour de 16 threads, ce qui correspond au nombre de cœurs physiques exploitables efficacement sur Dalek.
Au-delà, l'utilisation de threads supplémentaires n'apporte plus de parallélisme réel et introduit des phénomènes de contention (Hyper-Threading, saturation mémoire et cache), entraînant une légère dégradation des performances.

2.4 Optimisation mémoire par row pointers

Cette section décrit l'optimisation la plus déterminante du projet en termes de performances.
Elle repose sur une réorganisation des accès mémoire, sans modifier l'algorithme, mais en changeant profondément la manière dont les pixels sont parcourus en mémoire.

2.4.1 Problème initial : accès 2D coûteux et inefficaces

Dans les versions initiales du code, les images sont manipulées sous la forme de matrices 2D, avec des accès du type : `img[i][j]`

Ce type d'accès implique, à chaque pixel :

- un calcul d'adresse complet ,
- plusieurs dépendances mémoire,
- une mauvaise exploitation du cache,
- une difficulté accrue pour le compilateur à vectoriser automatiquement le code.

Ces surcoûts sont particulièrement pénalisants dans :

- le Sigma–Delta, parcouru à chaque image,
- la morphologie mathématique, qui effectue plusieurs lectures voisines par pixel (fenêtre 3×3).

2.4.2 Principe général de l'optimisation *row pointer*

Le principe consiste à transformer les accès 2D en accès 1D, en travaillant directement avec des pointeurs sur les lignes de l'image.

Au lieu d'écrire :

- `img[i-1][j-1]`
- `img[i][j]`
- `img[i+1][j+1]`

on introduit explicitement des pointeurs de lignes :

- `const uint8_t* r0 = img[i-1];`
- `const uint8_t* r1 = img[i];`
- `const uint8_t* r2 = img[i+1];`

et on accède ensuite aux pixels via :

- `r0[j-1], r0[j], r0[j+1]`
- `r1[j-1], r1[j], r1[j+1]`
- `r2[j-1], r2[j], r2[j+1]`

2.4.3 Application au Sigma–Delta

Avant optimisation

Dans la version initiale du Sigma–Delta, chaque pixel était traité via des accès 2D répétés :

- sd_data->M[i][j]
- img_in[i][j]
- sd_data->O[i][j]
- sd_data->V[i][j]

Ces accès sont réalisés plusieurs fois par pixel, dans des boucles successives.

Après optimisation (*row pointers*)

Chaque ligne est d'abord chargée une seule fois :

- uint8_t* Mi = sd_data->M[i];
- const uint8_t* Ini = img_in[i];
- uint8_t* Oi = sd_data->O[i];
- uint8_t* Vi = sd_data->V[i];
- uint8_t* Out = img_out[i];

Puis la boucle interne devient strictement linéaire :

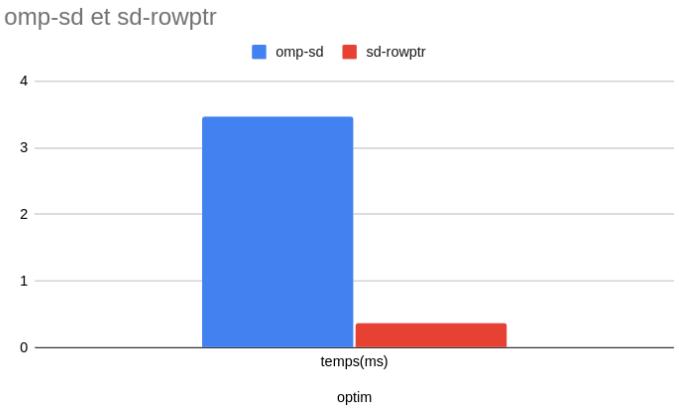
```
for (int j = j0; j <= j1; j++) {

   // opérations sur Mi[j], Ini[j], Oi[j], Vi[j], Out[j]

}
```

Bénéfices concrets

- suppression des recalculs d'adresses 2D,
- accès mémoire parfaitement contigus,
- excellente exploitation du cache L1,
- vectorisation SIMD facilitée,
- meilleure efficacité du parallélisme OpenMP.



Le Sigma–Delta devient une étape quasi négligeable dans le temps total avec un temps passant de **3.47 à 0.36 ms et un FPS de 390.15.**

2.4.4 Application à morphology

Spécificité de morphology

Les opérateurs d'érosion et de dilatation 3×3 effectuent :

- 9 lectures par pixel,
- des opérations logiques simples (AND / OR),
- sur des données binaires.

Cela rend cette étape extrêmement sensible à la qualité des accès mémoire.

Avant optimisation

Les accès étaient réalisés via :

```
img[i-1][j-1], img[i][j], img[i+1][j+1]
```

avec recalcul d'adresse pour chaque voisin.

Après optimisation (*row pointers*)

On introduit explicitement les trois lignes :

```
const uint8_t* r0 = img_in[i - 1];
const uint8_t* r1 = img_in[i];
const uint8_t* r2 = img_in[i + 1];
uint8_t* out = img_out[i];
```

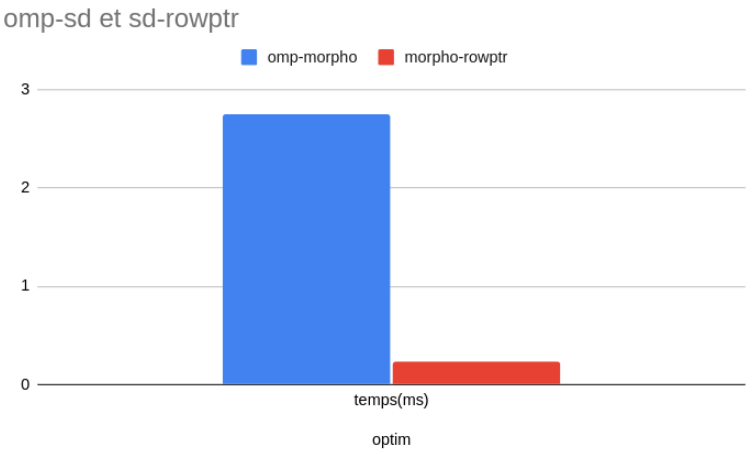
Puis on calcule :

```
uint8_t c0 = r0[j - 1] & r0[j] & r0[j + 1];
uint8_t c1 = r1[j - 1] & r1[j] & r1[j + 1];
uint8_t c2 = r2[j - 1] & r2[j] & r2[j + 1];
out[j] = c0 & c1 & c2;
```

Bénéfices spécifiques

- lecture séquentielle en mémoire,
- réduction drastique des dépendances mémoire,
- pipeline CPU plus efficace,
- très bonne affinité avec le cache L1/L2,
- base idéale pour une vectorisation SIMD ultérieure.

2.4.5 Résultats expérimentaux



Cette optimisation à elle seule permet un gain de plusieurs centaines de FPS avec un temps de morpho qui passe de **2.748 à 0.237 ms et un FPS de 555.90**

2.4.6 Analyse et conclusion

L'optimisation par *row pointers* :

- ne modifie pas l'algorithme,
- ne dépend ni du nombre de threads ni du SIMD,
- améliore simultanément :
    - la localité mémoire,
    - la prédictibilité des accès,
    - la vectorisation,
    - le scaling OpenMP.

2.5. Optimisation du CCL (Connected Component Labeling)

2.5.1 Analyse du code initial

Le Connected Component Labeling (CCL) a pour objectif d'attribuer un identifiant unique à chaque composante connexe de l'image binaire issue de la morphologie.

L'implémentation utilisée repose sur l'algorithme LSL (Line Segment Labeling), qui se décompose en plusieurs étapes :

1. **Détection de segments ligne par ligne**
   Chaque ligne est analysée indépendamment pour détecter des segments continus de pixels actifs.

2. **Construction des équivalences entre lignes consécutives**
   Les segments d'une ligne sont comparés à ceux de la ligne précédente afin de déterminer les connexions verticales.

3. **Résolution des classes d'équivalence**
   Cette étape agrège les labels relatifs pour produire des labels absolus.

4. **Labellisation finale de l'image**
   Chaque pixel reçoit son label définitif.

Contrainte majeure :
La construction et la résolution des équivalences introduisent des dépendances entre lignes successives, ce qui empêche une parallélisation complète de l'algorithme.

Dans la version initiale :

- la détection de segments était séquentielle,
- la labellisation finale parcourait l'image de manière non parallèle,
- de nombreuses écritures mémoire intermédiaires étaient effectuées,
- certaines boucles contenaient peu de travail par itération, limitant l'efficacité du parallélisme.

2.5.2 Optimisations appliquées

1. Parallélisation complète de la détection de segments

Elle a été parallélisée avec OpenMP :

```
#pragma omp parallel for schedule(static)
for (int i = i0; i <= i1; i++) {
   _LSL_segment_detection(CCL_data_er[i],
                          CCL_data_rlc[i],
                          &CCL_data_ner[i],
                          img[i],
```

```
                j0, j1);
}
```

Bénéfices :

- parallélisation parfaite (grain suffisant),
- aucun risque de data race,
- excellent scaling sur plusieurs threads.

2. Maintien volontaire de la construction d'équivalences séquentielle (_LSL_equivalence_construction(...))

Cette partie est conservée séquentielle, car elle représente une fraction limitée du temps total, et sa parallélisation serait contre-productive.

Cette étape dépend :

- de la ligne courante,
- de la ligne précédente,
- et modifie une structure globale d'équivalences (CCL_data_eq).

Toute tentative de parallélisation introduit :

- des synchronisations lourdes,
- ou des risques de corruption des labels.

3. Parallélisation de la labellisation finale (_LSL_compute_final_image_labeling(...))

Cette étape parcourt les segments déjà labellisés et écrit les labels finaux dans l'image.

Elle est totalement indépendante ligne par ligne, et a donc été parallélisée :

```
#pragma omp parallel for schedule(static)
for (int i = i0; i <= i1; i++) {
    ...
```

Bénéfices :

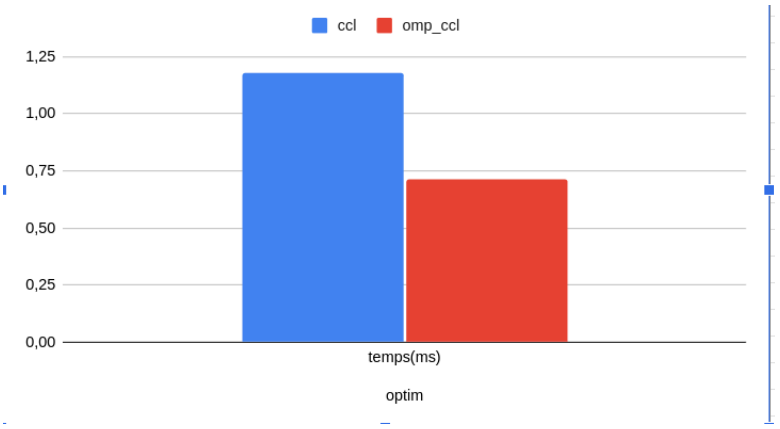
- forte réduction du temps total,
- bonne exploitation du cache,
- aucune dépendance entre threads.

4. Réduction des écritures mémoire inutiles

- fusion de certaines étapes (conversion labels relatifs → absolus),

- suppression d'écritures intermédiaires redondantes,
- utilisation directe des segments (RLC) pour remplir l'image finale.

Ces optimisations font passer le temps du ccl de 1.179 à 0.713 ms.



2.6 Optimisation du CCA (Connected Component Analysis)

2.6.1 Rôle du CCA

Le Connected Component Analysis (CCA) calcule, pour chaque composante connexe :

- la surface,
- le barycentre,
- la bounding box,
- divers moments statistiques.

Cette étape parcourt tous les pixels labellisés et met à jour des structures associées aux RoIs.

2.6.2 Problème initial : accumulations concurrentes

Dans l'implémentation initiale, plusieurs threads pouvaient vouloir mettre à jour simultanément les mêmes RoIs :

```
RoIs_S[r]  += 1;

RoIs_Sx[r] += j;

RoIs_Sy[r] += i;

...
```

Cela impose l'utilisation d'opérations atomiques :

```
#pragma omp atomic

RoIs_S[r] += 1;
```

Conséquence directe :

- un seul thread peut modifier une RoI donnée à la fois,
- contention massive,
- très mauvais scaling OpenMP,
- performance dominée par la synchronisation, et non par le calcul.

2.6.3 Stratégie retenue

1. Buffers locaux par thread

Chaque thread alloue ses propres buffers :

- RoIs_S_local
- RoIs_Sx_local
- RoIs_Sy_local
- RoIs_xmin_local, RoIs_xmax_local, etc.

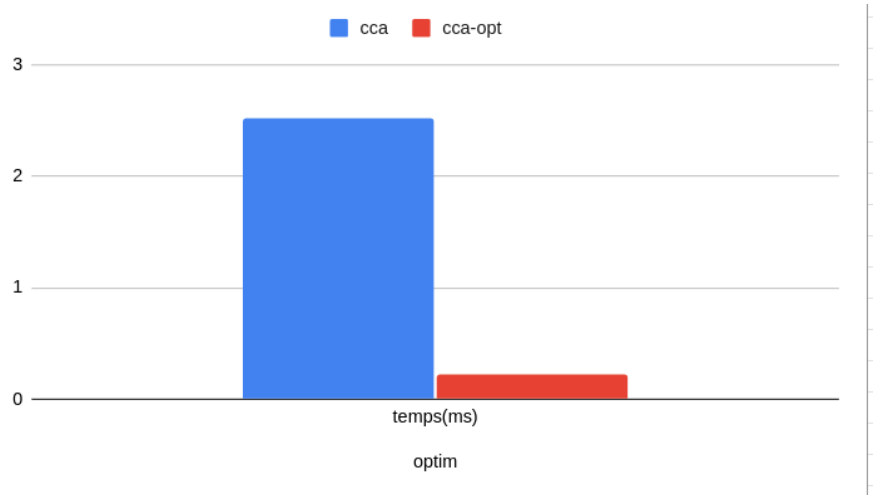
Indexés par :

[thread_id][roi_id]

2. Boucle pixel sans atomiques

3. Fusion finale parallèle

Ces optimisations font passer le temps du cca de 2.525 à 0.225 ms.



2.7 Vectorisation SIMD

2.7.1 Parties vectorisées du Sigma–Delta

La vectorisation a été appliquée à l'intérieur de la boucle sur les colonnes, en conservant la parallélisation OpenMP sur les lignes.

Les opérations suivantes ont été vectorisées :

1. Mise à jour du modèle M

Opération scalaire :

- si $M < I \rightarrow M{+}{+}$
- si $M > I \rightarrow M{-}{-}$

Version SIMD :

- comparaison vectorielle $M < I$ et $M > I$,
- génération de masques SIMD,

- addition et soustraction conditionnelles sur l'ensemble du vecteur.

2. Calcul de la différence O

Opération :

$O = |M - I|$

Version SIMD :

- calcul du maximum et du minimum vectoriels,
- soustraction vectorielle :

$O = \max(M, I) - \min(M, I)$

Cette formulation évite les branchements et s'adapte parfaitement au SIMD.

3. Mise à jour du seuil V

Opération scalaire :

- comparaison V avec $N \times O$,
- incrément ou décrément conditionnel,
- saturation entre $vmin$ et $vmax$.

Version SIMD :

- calcul vectoriel du seuil,
- comparaisons vectorielles,
- mise à jour conditionnelle par masques,
- clamp vectoriel entre vmin et vmax.

4. Binarisation finale

Opération :

output = (O < V) ? 0 : 255

Version SIMD :

- comparaison vectorielle O < V,
- sélection vectorielle (mask_blend) entre 0 et 255,
- écriture groupée des pixels.

2.7.2 Schéma d'implémentation (principe)

Sans entrer dans le détail du code, la structure est la suivante :

- chargement vectoriel des pixels (load),
- calcul vectoriel :
    - comparaisons,
    - masques,
    - additions / soustractions conditionnelles,
    - saturation,
- écriture vectorielle (store),
- boucle scalaire de finition pour les pixels restants si la largeur de l'image n'est pas multiple de la largeur SIMD.

Cette approche garantit :

- la correction des résultats,
- une compatibilité avec toutes les tailles d'images.

Ces améliorations ont fait passer le FPS à **583.88 FPS.**

```
# Average latencies:
# -> Video decoding =    0.173 ms
# -> Sigma-Delta    =    0.356 ms
# -> Morphology     =    0.238 ms
# -> CC Labeling    =    0.713 ms
# -> CC Analysis    =    0.225 ms
# -> Filtering      =    0.003 ms
# -> k-NN           =    0.004 ms
# -> Tracking       =    0.002 ms
# -> *Logs*         =    0.000 ms
# -> *Visu*         =    0.000 ms
# => Total          =    1.713 ms [~583.88 FPS]
```

2.9 Influence du nombre de threads et de l'allocation CPU

Sur Dalek, l'allocation explicite des cœurs a un impact majeur. En effet, durant tout le projet je n'avais pas pris en compte le fait de réserver des coeurs physiques au moment de se mettre sur un noeud. Ainsi lorsqu'on utilise la commande srun --cpus-per-task=16 et qu'on combine avec nos optimisations à 16 threads on atteint notre pic de FPS à 1821.83.

```
# Average latencies:
# -> Video decoding =    0.093 ms
# -> Sigma-Delta    =    0.038 ms
# -> Morphology     =    0.079 ms
# -> CC Labeling    =    0.269 ms
# -> CC Analysis    =    0.064 ms
# -> Filtering      =    0.002 ms
# -> k-NN           =    0.002 ms
# -> Tracking       =    0.001 ms
# -> *Logs*         =    0.000 ms
# -> *Visu*         =    0.000 ms
# => Total          =    0.549 ms [~1821.83 FPS]
```

Conclusion:

Ce projet a permis d'optimiser en profondeur une chaîne complète de détection de mouvement, en combinant des améliorations algorithmiques, mémoire, parallèles et vectorielles. À partir d'une version de référence limitée à moins de 20 FPS, les différentes optimisations ont conduit à un débit dépassant 1800 FPS sur la machine Dalek. Les gains les plus significatifs proviennent de la réorganisation mémoire (row pointers), de la suppression des contentions dans le CCA et de la vectorisation SIMD du Sigma–Delta. Ces résultats illustrent l'importance d'une approche globale de

l’optimisation en calcul haute performance, intégrant à la fois l’architecture matérielle et la structure des algorithmes.