Sorbonne Université

LIP6

Algorithmes
Architectures
Optimisations

Implémentation d'une chaine de détection de mouvement En temps-réel sur un processeur multi-coeurs SIMD

1 Objectifs

Ce projet a 3 objectifs :

- algorithmique : implémenter une chaine de traitement représentative d'applications en traitement d'images pour les systèmes embarqués.
- optimisation : réaliser un ensemble d'optimisations (algorithmiques, logicielles et architecturales) afin d'accélérer l'exécution de cette chaine de traitement et comprendre leur apport séparé et combiné. Un point particulier sera fait sur l'utilisation des principaux *Design Patterns* SIMD.
- méthodologie : développer une méthodologie d'analyse des résultats d'un point de vue qualitatif et quantitatif afin de réaliser des compromis selon des critères opposés de vitesse de traitement et de performance algorithmique.

Ce travail doit être un travail *original* pour chaque binôme. L'utilisation de fragments de code existants est strictement interdite tout comme le partage de code entre binômes. Le plagiat conduira systématiquement à une procédure disciplinaire (voir documents officiels de Polytech').

2 Chaine de traitement

La chaine de traitement est composée de deux parties. La première partie est l'algorithme de détection de mouvement qui à partir d'une séquence d'images en niveaux de gris fourni une séquence d'images (de masques) binaires indiquant quels sont les pixels en mouvement (appartement à une région en mouvement) et quels sont les pixels immobiles (appartenant au fond fixe).

Le problème de la détection du mouvement consiste à séparer dans chaque image les zones en mouvement et les zones statiques. A chaque instant, chaque pixel doit ainsi être étiqueté par un identifiant binaire fixe/mobile. Lorsque la caméra est fixe, on peut effectuer une telle détection à partir des différences temporelles calculées pour chaque pixel. Sur le plan algorithmique, l'objectif est d'étudier deux techniques de détection, de critiquer leurs performances et d'étudier leur coût d'implantation.

Soient les notations suivantes :

- t : instant de temps courant, servant à indicer les images
- I_t : image source en niveau de gris à l'instant t et I_{t-1} : l'image source à l'instant $t-1$,
- M_t : image de fond (image de moyenne),
- O_t : image de différence, en niveau de gris
- V_t : image de variance (d'écart type en fait) calculée pour tout pixel,
- E_t : image d'étiquettes binaires (mouvement / fond), $E_t(x) = \{0, 1\}$ ou $E_t(x) = \{0, 255\}$ pour coder {fond, mouvement}
- x : le pixel courant de coordonnées (i,j).

La plupart des techniques de détection du mouvement dans la séquence d'images $I_t(x)$ se fondent sur une estimation du module du gradient temporel $|\frac{\partial I}{\partial t}|$. Si les conditions d'éclairage de la scène varient lentement (sont *constantes* entre deux images consécutives) alors une variation significative du niveau de gris d'un pixel (supérieur à un seuil θ) entre deux images impliquera qu'il y a un mouvement en ce point.

2.1 Algorithme Frame-Difference (FD)

L'algorithme de différence d'images est l'un des premiers algorithmes inventé pour détecter un mouvement. C'est aussi le plus simple : pour un pixel donné, si la différence entre les valeurs de ce pixel pour deux images successives est supérieure à un seuil fixe (et constant dans toute l'image), alors ce pixel est en mouvement. Mais c'est aussi le moins robuste car il fait l'hypothèse que le bruit (la variation du niveau de gris d'un pixel - sans qu'il y ait mouvement) est constant dans toute l'image (c'est pour cela que θ est constant).

Algorithm 1: Frame Difference (FD)

foreach *pixel* x **do** // step #1 : O_t computation
  $O_t(x) = |I_t(x) - I_{t-1}(x)|$
foreach *pixel* x **do** // step #2 : O_t thresholding and $\hat{E}_t$ estimation
  if $O_t(x) < \theta$ **then** $\hat{E}_t(x) \leftarrow 0$
  else $\hat{E}_t(x) \leftarrow 1$

2.2 Algorithm Sigma-Delta (SD ou $\Sigma\Delta$)

L'algorithme $\Sigma\Delta$ fait au contraire l'hypothèse que le niveau de bruit (écart type) peut varier en tout point. Pour cela le niveau de gris d'un pixel est modélisé par une moyenne $M_t(x)$ et une variance (un écart type en fait) $V_t(x)$. Si la différence entre l'image courante et l'image de fond est supérieure à un N fois l'écart type alors il y a mouvement. La valeur de N est un paramètre connu et fixé à l'avance. Typiquement N vaut 2 ou 3, éventuellement 4.
Il s'agit d'une détection de mouvement fondée sur une estimation des statistiques du fond statique basée sur la modulation Σ-Δ : c'est une méthode itérative de conversion analogique/numérique qui incrémente ou décrémente d'un bit la valeur numérisée en fonction du résultat de la comparaison entre la valeur analogique et la valeur numérisée courante.

L'initialisation de l'algorithme pour $t = 0$ est la suivante : $M_0(x) = I_0(x)$ et $V_0(x) = V_{min}$. L'algorithme est appliqué à la séquence d'images à partir de $t = 1$. Les valeurs V_{min} et V_{max} sont des constantes permettant de restreindre les valeurs possibles de V_t. Typiquement, $V_{min} = 1$ et $V_{max} = 254$.

Algorithm 2: $\Sigma\Delta$

foreach *pixel* x **do** // step #1 : M_t estimation
  if $M_{t-1}(x) < I_t(x)$ **then** $M_t(x) \leftarrow M_{t-1}(x) + 1$
  if $M_{t-1}(x) > I_t(x)$ **then** $M_t(x) \leftarrow M_{t-1}(x) - 1$
  otherwise do $M_t(x) \leftarrow M_{t-1}(x)$
[step #2 : difference computation]
foreach *pixel* x **do** // step #2 : O_t computation
  $O_t(x) = |M_t(x) - I_t(x)|$
foreach *pixel* x **do** // step #3 : V_t update and clamping
  if $V_{t-1}(x) < N \times O_t(x)$ **then** $V_t(x) \leftarrow V_{t-1}(x) + 1$
  if $V_{t-1}(x) > N \times O_t(x)$ **then** $V_t(x) \leftarrow V_{t-1}(x) - 1$
  otherwise do $V_t(x) \leftarrow V_{t-1}(x)$
  $V_t(x) \leftarrow max(min(V_t(x), V_{max}), V_{min})$ // clamp to $[V_{min}, V_{max}]$
foreach *pixel* x **do** // step #4 : $\hat{E}_t$ estimation
  if $O_t(x) < V_t(x)$ **then** $\hat{E}_t(x) \leftarrow 0$
  else $\hat{E}_t(x) \leftarrow 1$

2.3 Morphologie Mathématique

Dans ce projet nous utiliserons des éléments structurants carrés B de taille 3×3 ou 5×5. Soit X l'ensemble des pixels de l'image associés à l'élément structurant B. Il existe deux opérations de base : la dilatation de X notée $\delta_B(X)$ et l'érosion de X noté $\epsilon_B(X)$.
L'application des opérateurs de morphologie mathématique est similaires aux opérateurs de filtrage, mais avec des opérations non linéaires.

Pour des images binaires, la dilatation consiste à calculer un OR sur le voisinage B dans l'image source et à l'écrire dans l'image destination. Inversement l'érosion consiste à calculer un AND sur le voisinage.

Ainsi, il suffit qu'un point soit à 1 dans le voisinage pour que la dilatation produise un 1 (puisque `x OR 1 = 1`), dilatant ainsi la composante connexe binaire (un ensemble de pixels à 1 dans l'image). Inversement, il suffit qu'un pixel soit à 0 dans le voisinage B pour que l'érosion produise un 0 (puisque `x AND 0 = 0`) érodant ainsi la composante connexe.

L'érosion sert à réduire le bruit dans les images : si l'on considère qu'un petit groupe de pixels est du bruit que l'on cherche à supprimer, alors l'application d'une érosion avec un élément structurant B de taille 3×3 fera disparaitre tout groupe de pixels inférieur à cette taille.

Soit r le rayon et $d = 2r + 1$ le diamètre d'un élément structurant carré B, alors une érosion de rayon r enlève à toute composante connexe une épaisseur de r pixels de contour tandis qu'une dilatation de rayon r ajoute une épaisseur de r pixels au contour (Fig. 2, la logique est inversée : les pixels à 1 sont noirs tandis que les pixel à 0 sont blancs)

Les opérations de morphologie binaire sont étendues aux images en niveaux de gris en remplaçant le `OR` par un *max* et le `AND` par un *min*.

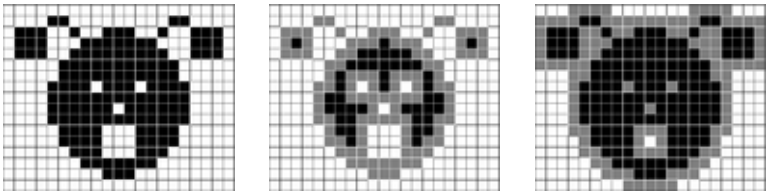

FIGURE 1 – A gauche, image binaire initiale. Au centre, image érodée par un carré 3×3 : les pixels gris sont supprimés de l'ensemble résultant. A droite, image dilatée par un carré 3×3 : les pixels gris sont ajoutés à l'ensemble résultant. Source wikipedia (`https://fr.wikipedia.org/wiki/Morphologie_mathmatique`)

A partir de ces deux opérateurs, il est possible d'en définir deux autres : la fermeture $\phi_B(X) = \epsilon_B(\delta_B(X))$ et l'ouverture $\gamma_B(X) = \delta_B(\epsilon_B(X))$. La fermeture permet de réduire (voire fermer complètement) les trous à l'intérieur des composantes connexes (si leur rayon est inférieur à celui de l'élément structurant) tandis que l'ouverture fait l'inverse et permet d'agrandir ces mêmes trous.

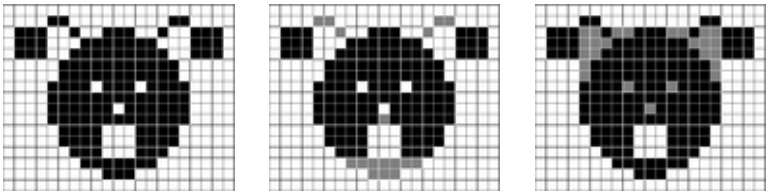

FIGURE 2 – A gauche, image binaire initiale. Au centre, image ouverte par un carré 3×3 : les pixels gris sont supprimés de l'ensemble résultant. A droite, image fermée par un carré 3×3 : les pixels gris sont ajoutés à l'ensemble résultant. Source wikipedia (`https://fr.wikipedia.org/wiki/Morphologie_mathmatique`)

Un des intérêts de l'ouverture et de la fermeture est qu'elles conservent la taille (discrète) des régions, contrairement à l'érosion qui la diminue ou à la dilation qui l'augmente. En fonction des besoins, on

choisira une fermeture ou une ouverture. Comme ces opérateurs sont idempotents, les appliquer plusieurs fois ne change pas le résultat (qui sera identique à celui obtenu après une seule application). Par contre il est possible de les enchainer (ouverture puis fermeture ou fermeture puis ouverture) pour améliorer l'image résultat (diminution du bruit, remplissage des trous, ...). En faisant augmenter progressivement leur rayon, on obtient les filtres alternés séquentiel, particulièrement efficaces (et avec une complexité particulièrement élevée) pour supprimer du bruit.

3 Travaux à réaliser

Sont listés l'ensemble des travaux possibles. Pour chaque thématique, il n'est pas nécessaire de tout faire, sauf si cela est explicitement indiqué.

3.1 Travail algorithmique

Dans une première étape, il s'agit de réaliser la chaine de traitement : SD+morpho, en scalaire mono-thread et sans optimisation afin d'obtenir une version de référence. Dans une seconde étape il s'agit d'optimiser cette chaine de traitement pour qu'elle soit le plus rapide possible, en codant les opérateurs en SIMD, en paralélisation le code sur les coeurs du processeur, en appliquant des optimisations logicielles et surtout en appliquant des transformations de haut niveau : pipeline d'opérateurs, fusion d'opérateurs. Le traitement morphologique à implémenter est la séquence érosion - dilatation - dilatation - érosion. La séquence d'images à utiliser est `car3.pgm`.

3.2 Travail d'optimisation

Sont listés dans cette section les différentes optimisations possibles (codage SIMD et pipeline ou fusion sont obligatoire). L'évaluation se fera sur la performance du code.

3.2.1 Optimisation SIMD

L'objectif principal est de coder la chaine SD+morpho en SIMD (SSE). Le codage en SSE2 est obligatoire.

Pour chaque étape de l'algorithme SD, fournir des tests unitaires commentés et validant les différents *Design Patterns* (obligatoire). Toute la chaine peut être codée en SSE2, mais il est possible de faire mieux en utilisant l'ensemble des jeux d'instructions SSE disponibles (SSE 4.1 et 4.2).

Les principales évolutions par rapport à cette version de référence sont :
- Augmentation du parallélisme SIMD en codant en AVX2 (256 bits) ou AVX512 (512 bits).
- Portage sur une plateforme embarquée en codant en SIMD Neon sur carte ARM Cortex A (RaspberryPi3, Nvidia JetsonTK1, TX1, ...)

3.2.2 Optimisations *Domain Specific* et fusion / factorisation d'opérateurs

Les opérateurs 2D de morphologie mathématique utilisés sont séparables, factorisables, associatifs et commutatifs :
- Un opérateur avec un élément structurant 2D $d \times d$ est séparable en deux opérateurs 1D d'élément structurant respectif $d \times 1$ et $1 \times d$ et vice-versa. Par exemple (Fig. 3) $(3 \times 3) \leftrightarrow (3 \times 1) \circ (1 \times 3)$ et $(5 \times 5) \leftrightarrow (5 \times 1) \circ (1 \times 5)$.
- Deux opérateurs d'élément structurant carré de rayon r donnent un opérateur d'élément structurant carré de rayon $2r$ et vice-versa. Par exemple (Fig. 4) $(3 \times 3) \circ (3 \times 3) \leftrightarrow (5 \times 5)$.

3.2.3 Optimisations logicielles : pipeline d'opérateurs

- Une fois que chaque opérateur de morphologie mathématique a été optimisé avec les transformations précédentes, il est possible d'en optimiser l'enchainement. Plutôt que d'appliquer un opérateur à une image entière puis de faire de même avec l'opérateur suivant, il est possible de les pipeliner (et de pipeliner la détection avec le post-traitement).

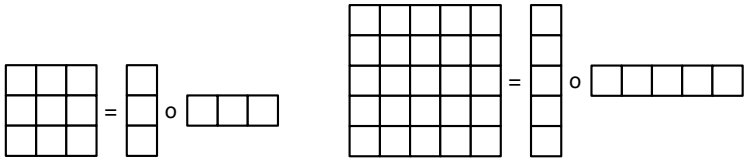

FIGURE 3 – Gauche : décomposition d'un élément structurant (3×3) en deux éléments structurants (3×1) et (1×3). Droite : décomposition d'un élément structurant (5×5) en deux éléments structurants (5×1) et (1×5).

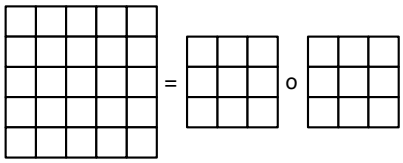

FIGURE 4 – Décomposition d'un élément structurant (5×5) carré de rayon 2 en deux éléments structurants carré (3×1) de rayon 1.

- Les opérateurs sur les images sont décomposés en opérateurs "ligne". Ce sont ces opérateurs "ligne" qui sont pipelinés. Cela permet d'améliorer (d'un point de vue probabiliste) la persistance des données dans les caches proches du processeurs. Cette optimisation est particulièrement efficace dans un context multithreadé. On parle alors de *Cache Level Parallelism* qui est une version améliorée du *Thread Level Parallélism* (qui ne s'intéresse qu'aux calculs).
- Cette optimisation est applicable en scalaire et en SIMD.
- Cela nécessite éventuellement un prologue et un épilogue.

3.2.4 Optimisation mémoire : entrelacement mémoire

La façon dont les données sont stockées en mémoire s'appelle le *memory layout*. Aujourd'hui, sur les processeurs multi-coeurs SIMD, l'optimisation du *memory layout* est un point majeur dans la performance d'une application, car la majorité des opérateurs sont *memory-bound*.

- Si initialement les images sont séparées, il est possible de les entrelacer pour réduire le nombre de références actives. Cela peut avoir un impact significatif pour une version multi-coeurs (pthread ou OpenMP). On parle de conversion *SoA* (*Structure of Arrays*) en *AoS* (*Array of structures*).
- Cela fonctionne aussi avec des images SIMD. Si en scalaire, l'entrelacement des données est 1 : 1 en SIMD, l'entrelacement est $c : c$ (avec c le cardinal du registre SIMD). Par exemple, si on doit entrelacer des valeurs 8 bits en SIMD 128 bits (parallélisme de 16), on entrelacera 16 valeurs du premier tableau puis 16 valeurs du second tableau et ainsi de suite. Cette optimisation est parfois appellée *AoSoA* : *Array of (Structure of Arrays*).
-

3.2.5 Optimisation des formats de calcul et de stockage en mémoire

- En SIMD il est primordial de garder le parallélisme maximal pour l'ensemble des calculs. Cela est faisable pour la totalité de la chaine de traitement. On a ainsi un parallélisme de 16 en 128 bits, 32 en 256 bits.
- Pour la morphologie mathématique binaire, il est possible de faire mieux : au lieu d'utiliser un octet pour stocker un pixel, il est possible de stocker 8 pixels binaires par octet. Cela fait 128 pixels en SSE et Neon, 256 en AVX2.
- Cela est aussi applicable en scalaire : 64 pixels binaires peuvent être codés dans un `long int` (ou plus simplement un `long`).
- Enfin, si les données occupent moins de place en mémoire (image 1 bit versus image 8 bits) elles sont plus rapidement chargées/écrites en mémoire. Cette optimisation permet de réduire le temps de traitement en réduisant les temps d'accès mémoire.

3.2.6 Optimisations matérielles combinés SIMD × OpenMP (ou pthread)

Le fait de combiner les deux parallélismes matériels que sont le SIMD et le multi-coeurs nécessite d'optimiser fortement les accès mémoire car les données vont être consommées à très grande vitesse. Trois possibilités :

- Diminuer le nombre d'accès mémoire en fusionnant des opérateurs. On peut envisager de fusionner des étapes de SD avec les opérateurs "lignes" de morpho-math.
- Maximiser la persistance des données dans les caches pour diminuer la durée des transferts en pipelinant les opérateurs. On peut là aussi envisager de pipeliner des étapes de FD les opérateurs "lignes" de morpho-math.
- Combiner fusion et pipeline. La combinaison "ultime" étant la fusion/pipeline de l'ensemble des opérateurs de morpho 1 bit en SIMD×OpenMP.

3.3 Codage

Aucun code C n'est fourni en plus des bibliothèques d'allocation mémoire au format NRC. Néanmoins, `nrutil` contient les routines pour lire et écrire des images au format PGM.

3.3.1 Nom et types

Les variables et les tableaux doivent être fortement typés et à la norme NRC. Le nom des variables, des tableaux et des fonctions doit être simple sans être ambigu. Il est recommandé d'utiliser les mêmes noms de tableaux que ceux utilisés dans cet énoncé. Voici quelques exemples possibles :

- `routine_FrameDifference`,
- `routine_FrameDifference_SSE2`,
- `SigmaDelta_step0` pour l'initialisation,
- `SigmaDelta_1step` pour les itérations suivantes.
- `SigmaDelta_step0_SSE2` et `SigmaDelta_1step_SSE2` our les versions codées en SSE2.

Il en va de même pour les noms de fichiers (obligatoire) :

- `mouvement.c` contiendra l'ensemble des algorithmes scalaires de détection de mouvement (ainsi que leur différentes implantations optimisées).
- `mouvement_SSE2.c`, pour les implémentation SSE2
- `morpho.c` et `morpho_SSE2.c` pour la morphologie mathématique
- `test_mouvement.c`, `test_mouvement_SSE2.c`, `test_morpho.c`, `test_morpho_SSE2.c` pour les tests unitaires et d'integration.
- `bench_mouvement.c`, `bench_mouvement_SSE2.c`, `bench_morpho.c`, `bench_morpho_SSE2.c` pour l'évaluation de performance.

En particulier, pour les tests unitaire SIMD, il est conseillé de commencer par tester un bout de code sur des valeurs choisies à la main (initialisation via l'instruction `_mm_setr_epi8`), afin de s'assurer du fonctionnement de chaque instruction SIMD et de chaque algorithme sans la paire de doubles boucles permettant balayer les images. Il est aussi conseiller d'utiliser des macros (avec un nom adéquat) pour cacher la complexité de lecture de certains emsemble d'instruction SIMD.

3.3.2 Codage logique et codage binaire

Il y a deux possibilités pour coder les images de morphologie mathématique. Soit sur {`0,1`}, soit sur {`0,255`}. Le second choix est plus simple car permet une visualisation et une mise-au-point plus rapide. Le premier choix permet du stocker les images booléennes sur 1 bit. Il est conseillé de faire une fonction de conversion d'un format vers l'autre pour avoir les deux à la fois et faire du debug.

3.4 Instructions SIMD

Les instructions dont vous aurez besoin sont regroupées par fonctionnalités :

Les accès mémoire sont alignés et se font via les instructions :

- `_mm_load_si128`,

— _mm_store_si128

Les opérations arithmétiques sont :
— l'addition : _mm_add_epi8 et _mm_add_epu8,
— la soustraction : _mm_sub_epi8 et _mm_sub_epu8,
— l'extraction du maximum et du minimum : _mm_max_epu8 et _mm_min_epu8
— comparaisons : _mm_cmpeq_epi8, _mm_cmpgt_epi8, _mm_cmplt_epi8
— opérateurs logiques bit à bit : _mm_and_si128, _mm_andnot_si128, _mm_or_si128

Une alternative à l'utilisation des 16 bits (perte de parallélisme, ajout d'instructions de conversion) est d'utiliser l'arithmétique saturée, empêchant les *overflow* :
— _mm_adds_epi8,
— _mm_adds_epu8
Ainsi les opérations de multiplications seront implantées par une série d'instructions d'addition.

Les initialisations de variables se font via les instructions suivantes :
— initialisation avec 16 valeurs différentes : _mm_set_epi8,
— initialisation avec une valeur unique : _mm_set1_epi8,
— initialisation simulant un chargement mémoire : _mm_setr_epi8

Pour l'implémentation compacte où chaque octet contient 8 pixels codés sur 1 bit, d'autres instructions peuvent être nécessaires.

4 Benchmark quantitatif

Il vous revient les choix pour valider et mettre en valeur les optimisations réalisées. Ce peut être des courbes, des tableaux, des images. Dans l'optique d'une comparaison avec une implémentation sur FPGA, il est conseillé d'utiliser les métriques suivantes :
— le temps de traitement en seconde (ou en ms) de chaque opérateur et de la chaine complète,
— le *cpp* (cycle par point) et le débit (en pixel par seconde)

Il est aussi conseillé de faire varier la taille des images pour analyser les performances des codes SIMD : partir d'images petites tenant dans les caches à des images trop grandes pour tenir dans les caches. Dans ce cas, ce sont le *cpp* et le débit qui sont intéressants à tracer sur une courbe.

La séquence car3 est disponible à l'adresse suivante : `http://www-soc.lip6.fr/~lacas/ImageProcessing/MotionDetection/car3.zip`.