

Détection de mouvement en CUDA

Optimisations algorithmiques et mémoire

Cyprien Deruelle / Lucas Collemare / Antoine Malmezac / Charles Prioux

Soutenance GPGPU

18/12/2025

Contexte et objectif

- Détection de mouvement sur flux vidéo
- Implémentation initiale en C++
- Accélération par SIMD puis par GPU (CUDA)
- Étude et comparaison de plusieurs optimisations

Pipeline de détection de mouvement

- ① Estimation du fond (Background estimation)
- ② Nettoyage du masque (Mask cleaning)
- ③ Seuils avec hystérésis (Hysteresis thresholding)

Implémenté initialement avec des boucles imbriquées en C++.

Baseline C++

- Implémentation naïve en C++
- Doubles boucles `for`
- Quadruples boucles pour les convolutions
- Faibles performances sur flux vidéo temps réel

Illustration : Baseline C++



Figure – Traitement séquentiel pixel par pixel sur CPU avec boucles imbriquées

Optimisation C++ SIMD

- Utilisation des instructions AVX2
- Optimisation de :
 - Mask cleaning
 - Hysteresis thresholding
- Background estimation non vectorisée (3 canaux)

Illustration : Optimisation C++ SIMD



Figure – Traitement parallèle de plusieurs pixels simultanément à l'aide des instructions AVX2

Performances C++

Version	Temps (vidéo ref)	FPS webcam
Baseline C++	54 s	1.67
C++ SIMD	14 s	4.17
Gain SIMD	+380%	+256%

Accélération : **×2.35** sur le temps de calcul, **×2.23** sur le FPS.

Baseline CUDA

- Même pipeline que la version C++
- Chaque étape implémentée sous forme de kernels CUDA
- Parallélisation massive par pixel

Illustration : Cuda Baseline



Figure – Baseline en Cuda sans optimisations

Optimisations algorithmiques

- Passage de l'hystérésis itérative vers un BFS
- Réduction du nombre d'itérations globales
- Utilisation des fonctions mathématiques CUDA

Hystérésis : Itératif vs BFS

Propagation itérative :

- Mise à jour globale à chaque itération
- Coût élevé jusqu'à convergence

Propagation BFS :

- File d'attente
- Propagation par proximité
- Moins de calculs inutiles

Optimisations mémoire CUDA

- Allocation mémoire persistante
- Suppression des allocations/libérations par frame
- Mémoire stable et prévisible

Illustration : Cuda Optimisations



Figure – Image Cuda avec optimisations

Performances CUDA

Version	Temps (vidéo ref)	FPS webcam
Baseline CUDA	7.9 s	13
CUDA optimisé (kernels)	5.3 s	21
CUDA optimisé (mémoire)	4.9 s	22

Conclusion

- Implémentations multiples :
 - C++ baseline
 - C++ SIMD
 - CUDA baseline
 - CUDA optimisé (kernel et mémoire)
- Forts gains de performances
- Pipeline temps réel moyennement atteint

Limites et perspectives

- Fusion kernels/mémoire : regrouper diff+overlay+hystéresis, loads vectorisés, éviter copies host/device, garder les masques sur GPU.
- Optimisations GPU : tuner blocs/tiling via Nsight Compute, réduire latence mémoire, basculer l'encodage sur NVENC pour sortir du CPU.
- Précision : seuils adaptatifs (variance locale), choix 16/32 bits selon étape, aligner CPU/GPU pour réduire les écarts numériques.