Rapport GPGPU – Détection de mouvement en CUDA

Login : `Lucas Collemare | Cyprien Deruelle | Antoine Malmezac | Charles Prioux` Groupe 2

Contexte

L'objectif du projet est de réussir à traiter en temps-réel un flux vidéo via un pipeline de détection de mouvement. Le but est donc de séparer le fond des éléments présent sur ce flux vidéo. Nous avons produit une première implémentatation en C++ naïf qui va nous servir de référence. Nous avons ensuite créé une vectorisation CPU (SIMD) et implémentations GPU (CUDA) avec optimisations algorithmiques et mémoire pour accélérer le processus.

Répartition des tâches

Membre	Tâches
Cyprien Deruelle	Baseline C++
Lucas Collemare	Baseline C++ SIMD + Interpretation résultat
Antoine Malmezac	Baseline Cuda + iteratif et BFS + optimisation
Charles Prioux	Résultats global / Benchmark + rapport + slides

Implémentations et optimisations majeures

Pour répondre à cette problématique, nous avons implémenté une pipeline de detection de mouvement sur un flux vidéo. Elle va donc détecter, nettoyer et mettre en évidence les zones en mouvement dans une vidéo en comparant chaque image à un fond de référence, puis en filtrant le bruit :

1. Estimation / mise à jour du fond et différence frame–background : calcul d'un masque de mouvement à partir de la différence entre l'image courante et le background.
2. Seuils (low/high) + hystérésis : création de masques faible/fort puis propagation des pixels faibles connectés à des pixels forts.
3. Nettoyage du masque (morphologie) : ouverture morphologique (érosion + dilatation) pour supprimer le bruit.
4. Overlay / affichage : application d'un overlay sur l'image (mise en évidence des zones en mouvement).

À partir de ce pipeline de référence, nous avons développé plusieurs implémentations plus optimisées :

1) SIMD CPU – Différence Background (`opt_cpu_simd`)

Vectorisation de la différence frame–background via instructions SIMD (traitement de plusieurs pixels en parallèle). Points clés : forte amélioration CPU, sans transferts CPU↔GPU, bonne localité mémoire (plans R/G/B séparés). Limites : code moins portable et optimisation partielle (une étape).

2) GPU – Différence + Fusion overlay (`gpu_diff + kernel_fusion`)

Fusion des opérations (diff + masques + overlay) pour limiter : (i) le nombre de kernels et (ii) les relectures/écritures globales. Très rentable à haute résolution ; moins intéressant sur petites images où l'overhead (lancement/transfer) domine.

3) GPU – Hystérésis (`gpu_hysteresis`)

Propagation parallèle des pixels faibles connectés à des pixels forts. Deux approches sont discutées :

— *Itérative* : itérations globales jusqu'à convergence (avec un flag global `changed`).

— *BFS* : propagation via file d'attente pour éviter les itérations inutiles.

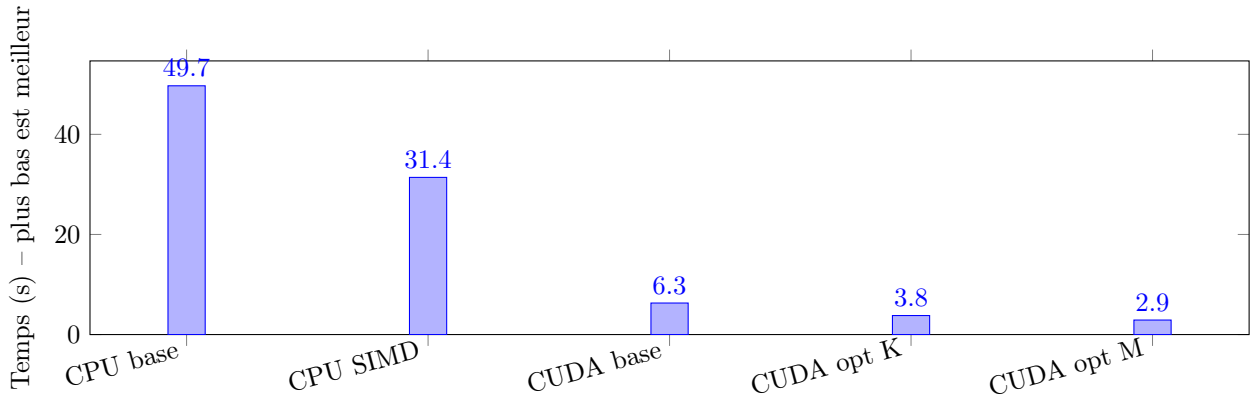
Bénéfices : gain majeur vs CPU ; limites : synchronisations et/ou contention (atomics/queue) selon l'approche.

4) GPU – Morphologie (érosion + dilatation)

Ouverture morphologique en deux kernels. Très efficace sur GPU (opérations de voisinage). Limites actuelles : nombreux accès globaux ; améliorable via *shared memory* (tuilage + halo).

Résultats / Benchmarks
Voici les mesures (vidéo de référence + FPS webcam) résumées

Version	Temps (vidéo ref)	FPS webcam	Speedup vs CPU base
CPU Baseline C++	49.7 s	1.41	1.0×
CPU SIMD	31.4 s	2.08	1.70×
CUDA Baseline	6.3 s	15	7.9×
CUDA optimisé (kernels)	4.8 s	20	11.1×
CUDA optimisé (mémoire)	3.9 s	25	15.1×



On peut lire sur le tableau qeu le SIMD apporte déjà un facteur ≈ 2× sur CPU (pipeline partiellement vectorisé). Le passage GPU rend le traitement compatible temps réel (15–30 FPS sur webcam selon optimisations). Les gains supplémentaires GPU proviennent d'abord de la réduction du surcoût kernel (fusion) puis de la réduction des surcoûts d'allocation (mémoire persistante).

Analyse des performances et bottlenecks (Nsight / nvprof)
Nous avons trois sources principales de coût côté GPU : les transferts CPU↔GPU, les synchronisations imposées par l'hystérésis itérative et des kernels de voisinage memory-bound.

— Transferts H→D / D→H : upload_current_frame() fait un cudaMemcpy2D à chaque frame dès que gpu_diff est actif. Si gpu_overlay est activé, le résultat est rapatrié par un cudaMemcpy2D (frame final) pour l'affichage, ce qui peut dominer à faible résolution. De plus, dès qu'on mixe CPU/GPU entre deux étapes, sync_*_masks() déclenche des copies de masques supplémentaires.

— Diff + seuils (+ overlay fusionné) : diff_kernel calcule les masques low/high et peut appliquer l'overlay si kernel_fusion est activé. Les seuils sont en mémoire constante (__constant__), et la fusion réduit le nombre de kernels, mais le coût de transfert du frame reste présent.

— Hystérésis GPU : synchronisation par itération : hysteresis_gpu() lance jusqu'à 10 itérations. À chaque tour, un cudaMemcpy du flag d_changed vers l'hôte force une synchronisation CPU↔GPU. Le kernel hysteresis_propagate_kernel utilise aussi un atomicExch (contention possible en cas de nombreuses activations).

— Mise à jour du background : après update_background_kernel, on rapatrie le background complet par cudaMemcpy2D vers le CPU pour synchroniser l'état hôte, ce qui ajoute un coût fixe non négligeable lors des frames de mise à jour.

Conclusion
Le pipeline passe d'une version C++ naïve (1.41 FPS) à une version CUDA optimisée atteignant 25 FPS sur webcam. Les gains proviennent : du parallélisme massif GPU par pixel, de l'optimisation du coût fixe (fusion de kernels, allocations persistantes), et d'améliorations algorithmiques (réduction du travail de l'hystérésis via approche BFS). Les principaux axes restants concernent surtout la réduction des accès mémoire globaux (tuilage shared) et la minimisation des synchronisations/atomics dans l'hystérésis.