
Projet GPGPU

Septembre 2024 - Novembre 2024

ÉCRIT PAR

BAPTITE BELLAMY
EITHAN NAKACHE
GASPARD SALIOU
GAUTIER GALLY

Table des matières

1 Présentation Du Sujet	2
2 Répartition des tâches	2
3 Implémentation	3
3.1 C++	3
3.1.1 Background Estimation	3
3.1.2 Motion Mask	3
3.1.3 Mask Cleaning	4
3.1.4 Seuil d'hystérésis	5
3.1.5 Masquage	6
3.2 Cuda	7
3.2.1 Initialisation	8
3.2.2 Disque Kernel	8
3.2.3 Background Estimation	8
3.2.4 Morphological Opening	9
3.2.5 Seuillage d'hystérie	10
4 Benchmarks et Analyse	12
4.1 Temps d'exécution	12
4.2 Analyse	13
5 Analyse des Résultats du Profilage	13
5.1 Activités GPU	13
5.2 Appels API CUDA	14

1 Présentation Du Sujet

Le projet **GPGPU** vise à développer un **plugin GStreamer** permettant la séparation entre le fond et les objets en mouvement dans des vidéos. Ce traitement, crucial pour de nombreuses applications de vision par ordinateur, sera implémenté et optimisé à l'aide de **CUDA** afin d'accélérer les calculs sur GPU.

La solution se divise en trois étapes :

1. **Estimation du fond** : Analyse pixel par pixel pour identifier les zones de changement dans une séquence vidéo, basée sur des critères de distance dans l'espace colorimétrique Lab.
2. **Génération de masques binaires** : Détection des mouvements à partir des variations identifiées entre l'image actuelle et le background.
3. **Nettoyage et post-traitement** : Amélioration de la qualité des masques via des opérations morphologiques (ouvertures) et un seuillage d'hystérésis pour éliminer le bruit et renforcer la précision.

Le plugin sera intégré dans un pipeline GStreamer. Une version de base en **C++** servira de référence avant l'implémentation des optimisations GPU. Ce projet a également pour ambition d'évaluer les gains de performance en termes de qualité des résultats et de vitesse de traitement grâce aux calculs parallèles des GPUs.

2 Répartition des tâches

Nom	BE	MM	OM	SH	CUDA	BENCH
Gaspard	X	X		X	X	X
Gautier	X	X	X		X	
Baptiste	X	X		X	X	
Eithan	X	X			X	X

TABLE 1 – Répartition des tâches par étape du projet

Légende des colonnes :

- **BE** : Background Estimation - Implémentation et optimisation des algorithmes d'estimation du fond.
- **MM** : Motion Mask - Génération et gestion du masque de mouvement.
- **OM** : Opérations Morphologiques - Nettoyage du masque via érosion et dilatation.
- **SH** : Seuillage d'Hystérésis - Application du seuillage pour améliorer la qualité du masque.
- **CUDA** : Implémentation et optimisation CUDA pour accélérer les calculs sur GPU.
- **BENCH** : Benchmarks - Création des outils de benchmarking et analyse des performances.

3 Implémentation

3.1 C++

3.1.1 Background Estimation

L'objectif est de séparer le fond statique des objets en mouvement dans une vidéo. Pour cela, on adopte une approche pixel-wise, c'est-à-dire que chaque pixel est traité individuellement, tout en prenant en compte une fenêtre 3×3 autour de lui pour renforcer l'analyse locale.

À chaque itération, la couleur du pixel cible est comparée à celle de ses voisins dans la fenêtre. Une couleur moyenne est calculée à partir de ces voisins pour réduire les effets du bruit et fournir une valeur représentative de la zone locale. La couleur du pixel est ensuite comparée à celle du fond en utilisant l'espace Lab, qui évalue les différences de couleur. Si la différence est inférieure à un seuil, le pixel est considéré comme appartenant au fond. Sinon, il est classé comme faisant partie d'un objet en mouvement.

Lorsqu'un pixel est détecté comme ne correspondant pas au fond, il est traité comme un candidat pour devenir un nouveau fond. Ce candidat n'est intégré qu'après avoir montré une certaine stabilité temporelle, évitant ainsi les erreurs dues à des mouvements temporaires. Si le pixel correspond déjà au fond, sa couleur est ajustée progressivement en combinant ses anciennes valeurs avec les nouvelles, ce qui permet de s'adapter aux changements progressifs dans la scène.

Pour assurer une mise à jour continue, un compteur de temps est associé à chaque pixel. Ce mécanisme contrôle les transitions entre les états et prévient les mises à jour trop rapides ou incorrectes. Grâce à cette approche, le modèle de fond reste stable et précis, tout en s'adaptant aux changements lents dans la scène et en excluant efficacement les objets en mouvement.

3.1.2 Motion Mask

Dans cette étape, nous traduisons la distance calculée entre la couleur estimée du fond et la couleur actuelle en une intensité visuelle dans le masque de mouvement.

La couleur est attribuée au canal rouge de chaque pixel, où l'intensité est proportionnelle à la distance mesurée. Un facteur d'échelle est appliqué pour ajuster la sensibilité.

Ce traitement met en évidence les zones de mouvement dans l'image en les affichant dans différentes nuances de rouge, où les intensités élevées indiquent des variations importantes, tandis que les faibles variations apparaissent plus sombres. Cependant, malgré la génération du masque de mouvement, des artefacts et du bruit persistent dans l'image, nécessitant une étape supplémentaire de nettoyage pour améliorer la qualité et la précision du résultat.



FIGURE 1 – Résultat de l'estimation du fond et du masque de mouvement

3.1.3 Mask Cleaning

Nous avons implémenté des opérations morphologiques d'ouverture (érosion suivie de dilatation) pour améliorer les masques binaires, permettant ainsi de mieux isoler les objets en mouvement tout en réduisant le bruit.

- **Érosion** : Attribue la valeur minimale au pixel dans une fenêtre centrée sur celui-ci.
- **Dilatation** : Attribue la valeur maximale au pixel dans une fenêtre centrée sur celui-ci.

Pour les opérations d'ouverture, nous avons opté pour un noyau en disque pour plusieurs raisons :

- **Isotropie** : Le noyau en disque traite les pixels de manière uniforme dans toutes les directions, ce qui préserve mieux les contours des objets par rapport aux noyaux rectangulaires, qui peuvent favoriser certaines directions.

- **Préservation des contours** : Cette forme permet de mieux conserver les bords des objets tout en éliminant les petits artefacts, ce qui est essentiel pour des applications de détection de mouvement.

Pour adapter automatiquement la taille du noyau à la résolution des images, nous utilisons un calcul dynamique du rayon :

- Le rayon est déterminé comme étant **0.5%** de la plus petite dimension de l'image, garantissant ainsi un nettoyage proportionnel à la taille de celle-ci. Cette valeur a été trouvée de façon empirique.
- Un paramètre `minradius` fixe une taille minimale pour s'assurer que le noyau reste efficace, même pour des images de faible résolution.

Ce calcul permet de trouver un équilibre entre l'élimination du bruit et la préservation des objets d'intérêt, optimisant ainsi la qualité du traitement.

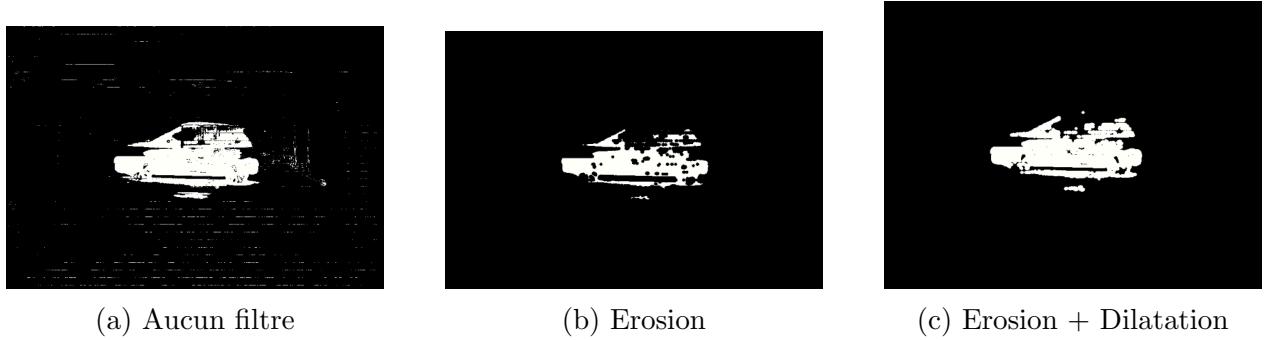


FIGURE 2 – Processus de réduction des bruits

3.1.4 Seuil d'hystérisis

Nous avons implémenté la fonction de seuil d'hystérisis, qui permet de détecter les contours/bords. Elle repose sur deux seuils : un seuil bas et un seuil haut. Les pixels dont l'intensité dépasse le seuil haut sont immédiatement considérés comme des contours, tandis que ceux inférieurs au seuil bas sont ignorés. Les pixels entre les deux seuils sont retenus uniquement s'ils sont connectés à des pixels déjà retenus.

Voici comment nous l'avons implémentée :

- Classification des pixels :
 - Les pixels avec une intensité supérieure ou égale à un seuil haut (`highThreshold`) sont marqués comme contours forts (blancs 255, 255, 255).
 - Les pixels avec une intensité inférieure à un seuil bas (`lowThreshold`) sont marqués comme arrière-plan (noirs 0, 0, 0).
 - Les pixels entre les deux seuils sont marqués comme indéterminés (gris 127, 127, 127).

- Propagation des contours : Les contours forts propagent leur état aux pixels indéterminés adjacents si leur intensité est comprise entre les seuils bas et haut. Ces pixels sont alors mis en blanc s'ils sont adjacents à un pixel blanc.
- Nettoyage final : Tous les pixels restants indéterminés (gris) sont convertis en arrière-plan (noirs).

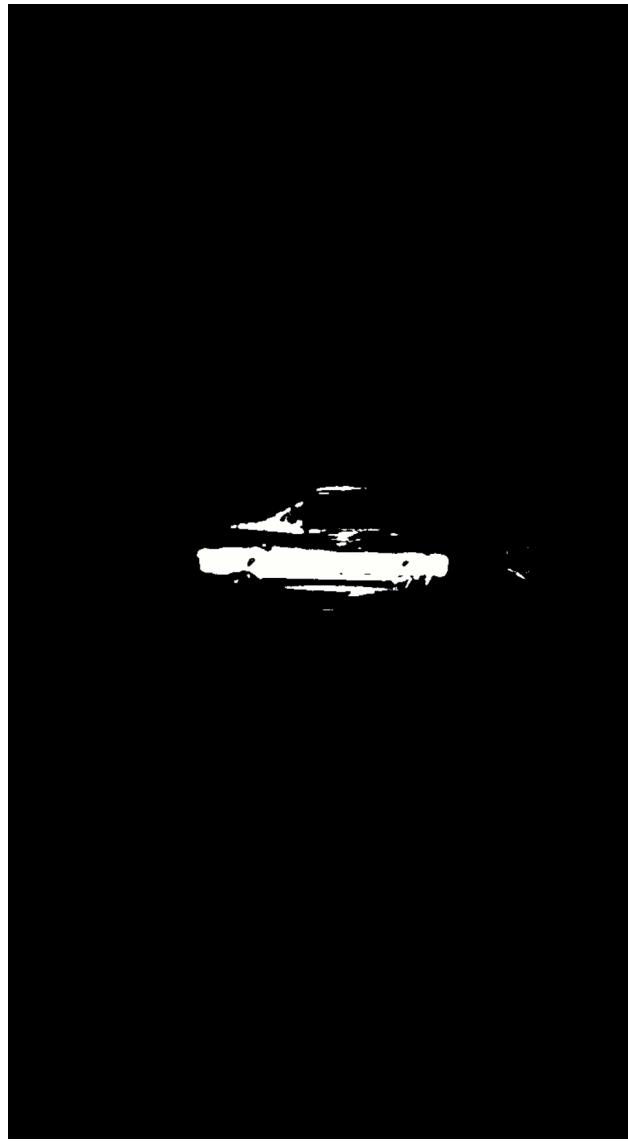


FIGURE 3 – Résultat du seuillage d'hystérie

3.1.5 Masquage

Enfin, une fonction de masquage est appelée pour appliquer une transformation à l'image originale. Elle applique un masque sur le canal rouge qui représente le déplacement.

Pour cela, pour chaque pixel blanc du seuil d'hystérosis, une valeur calculée à partir de 0.5×255 est ajoutée à l'intensité initiale du canal rouge (en utilisant l'image initialPixels), sans dépasser 255. Les pixels noirs restent identiques à l'image initiale.



FIGURE 4 – Résultat final pour la partie cpu

3.2 Cuda

Nous avons repris les fonctions réalisées en C++ et les avons adaptées en CUDA. Nous distinguons 5 parties principales

3.2.1 Initialisation

Nous initialisons toutes les variables nécessaires et les transférons en mémoire GPU (device).

- Les variables `device_logo`, `pixel_time_counter`, `device_background`, et `device_candidate` sont créées et initialisées avec les données appropriées depuis la mémoire hôte ou avec des valeurs par défaut (par exemple, des zéros).
- Les paramètres CUDA sont définis : un bloc GPU est configuré en grille de 16x16 threads, et les dimensions des grilles (grid) sont calculées en fonction des dimensions de l'image d'entrée pour couvrir toute la surface.

3.2.2 Disque Kernel

- Le calcul du Disque Kernel est effectué directement dans la fonction `compute_cu`.
- Les paramètres et éléments utilisés pour calculer le Disque Kernel sont similaires à ceux en C++ (dans la version originale), mais ici, le Disque Kernel est représenté sous forme de `int*` (pointeur entier) au lieu d'un `std::vector`

3.2.3 Background Estimation

Le procédé de calcul pour l'estimation du fond reste identique à celui de la fonction C++ affiliée.

- Les mêmes algorithmes et étapes sont utilisés pour déterminer l'estimation du fond de l'image.
- Cependant, les opérations sont adaptées pour s'exécuter sur le GPU, tirant parti du traitement parallèle offert par CUDA.

De plus, nous avons réimplémenté la fonction `swap` pour l'adapter à CUDA, en prenant en compte les spécificités de la gestion de la mémoire et des buffers sur le GPU. Cette réimplémentation permet d'effectuer efficacement les échanges directement en mémoire GPU.



FIGURE 5 – Résultat de l'estimation du fond et du masque de mouvement en cuda

3.2.4 Morphological Opening

La fonction de morphological opening diffère de l'implémentation en C++ de la manière suivante :

- Les fonctions erode et dilate ont été regroupées en une seule fonction, qui prend en argument un booléen Erode. Ce booléen détermine si l'opération effectuée est une érosion ou une dilatation.
- Les fonctions standards min et max utilisées en C++ ont été réimplémentées pour CUDA afin de fonctionner efficacement dans le contexte des kernels GPU.

Pour appliquer le morphological opening, la fonction unifiée est appelée deux fois :

1. Une première fois avec le booléen Erode défini à true, pour effectuer l'érosion.

2. Une deuxième fois avec Erode défini à false, pour effectuer la dilatation.

Cette approche réduit la duplication de code et optimise le traitement sur GPU.

3.2.5 Seuillage d'hystérie

Le seuillage d'hystérie a été amélioré par rapport à la version C++ et est divisé en trois fonctions principales :

1. Initialisation des pixels. Cette fonction attribue une couleur aux pixels en fonction des seuils :

- Blanc si l'intensité dépasse le seuil haut.
- Noir si l'intensité est inférieure au seuil bas.
- Gris si l'intensité est entre les deux.

2. Propagation des pixels

- Cette fonction s'occupe de propager les modifications sur le GPU.
- Elle est appelée de manière répétée tant que des changements sont détectés dans l'image.
- L'algorithme a été adapté pour CUDA avec une amélioration : Une condition supplémentaire vérifie le nombre de pixels voisins d'un pixel donné. Si le nombre de voisins blancs est inférieur à un seuil (par exemple, 4), le pixel est marqué en noir.

3. Fonction principale de seuillage d'hystérie

- Cette fonction gère l'ensemble du processus.
- Elle marque les pixels restants indéterminés comme appartenant au fond.

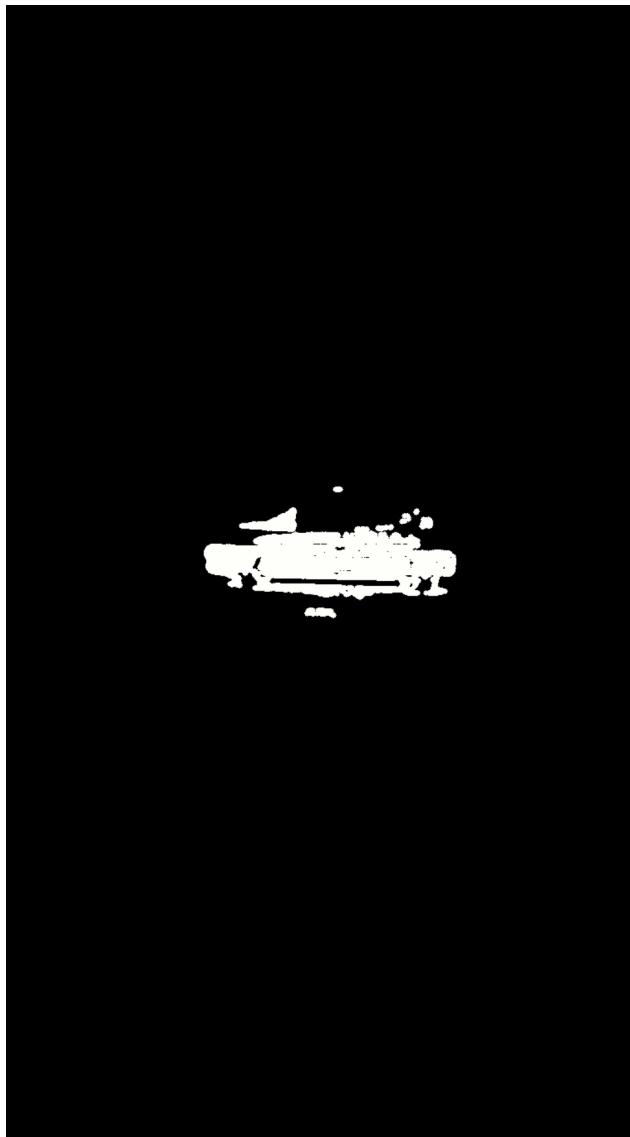


FIGURE 6 – Résultat du seuillage d'hystérie

Ces améliorations rendent l'algorithme plus efficace et précis, tout en exploitant pleinement les capacités parallèles du GPU grâce à CUDA.

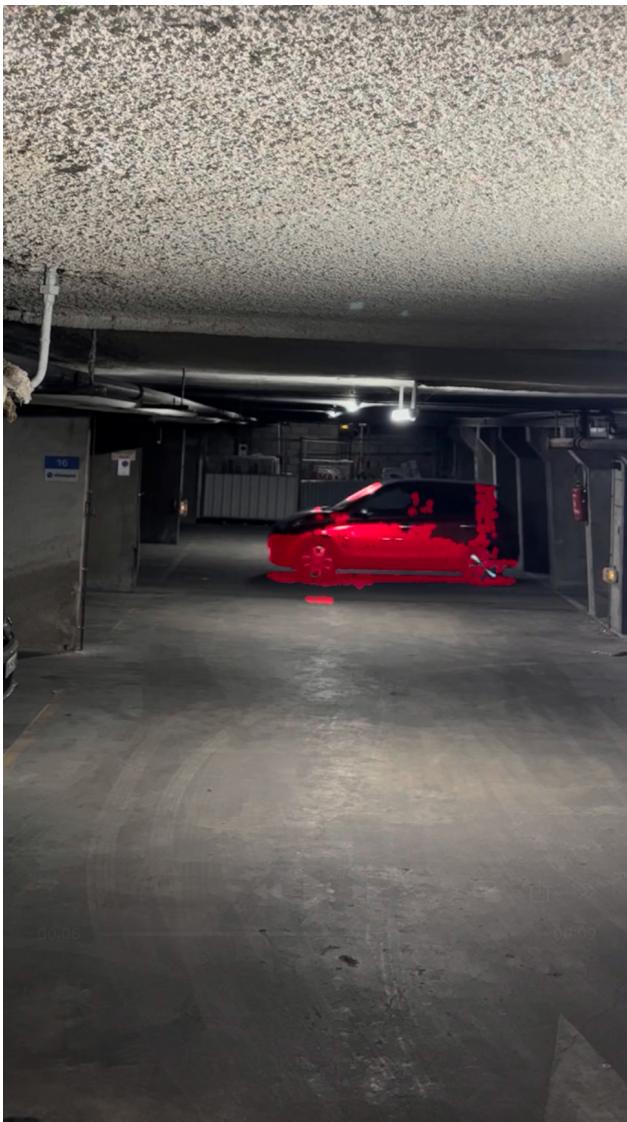


FIGURE 7 – Résultat Final pour la partie cuda

4 Benchmarks et Analyse

4.1 Temps d'exécution

Le tableau ci-dessous présente les temps d'exécution pour plusieurs vidéos traitées par les versions CPU et GPU de l'algorithme :

Vidéo	Temps CPU (s)	Temps GPU (s)	Accélération
Nuits Blanches	847.572	11.309	$\times 74.94$
ACET	277.735	5.857	$\times 47.44$
Lil Clown	818.374	11.217	$\times 72.95$

TABLE 2 – Comparaison des temps d'exécution entre les versions CPU et GPU, et le facteur d'accélération.

4.2 Analyse

Les résultats des benchmarks montrent une amélioration significative des performances grâce à l'utilisation du GPU. Voici les principaux constats :

- Pour la vidéo **Nuits Blanches**, l'exécution sur GPU est environ 75 fois plus rapide que sur CPU, avec un temps de traitement réduit de 847.572 secondes à 11.309 secondes.
- La vidéo **ACET** bénéficie également d'une accélération importante, avec un facteur d'environ 47, réduisant le temps de 277.735 secondes à 5.857 secondes.
- Enfin, pour la vidéo **Lil Clown**, le GPU offre une accélération de près de 73 fois, diminuant le temps de traitement de 818.374 secondes à 11.217 secondes.

Ces résultats confirment que l'optimisation GPU permet de gérer efficacement les traitements lourds nécessaires au calcul du masque de mouvement, notamment dans des vidéos complexes. Le facteur d'accélération varie en fonction de la taille et du contenu des vidéos, mais il reste systématiquement élevé.

5 Analyse des Résultats du Profilage

5.1 Activités GPU

Fonction	Temps (%)	Temps total (s)	Appels	Moy (ms)
background_estimation_process	82.63%	4.88889	268	18.242
CUDA memcpy HtoD	7.91%	0.46812	807	0.58008
CUDA memcpy DtoH	5.25%	0.31077	804	0.38653
morphologicalOpening	3.16%	0.18718	536	0.34921
applyRedMask_cuda	0.47%	0.027653	268	0.10318
hysteresis	0.43%	0.025254	268	0.09423

TABLE 3 – Profiling des activités GPU

D'après les résultats du profilage, la répartition des activités du GPU est la suivante :

- La fonction **background_estimation_process** consomme **82.63%** du temps total, avec un temps moyen d'environ **8.16 ms** par appel.
- Les opérations morphologiques (**morphologicalOpening**) représentent **3.16%** du temps.
- Le seuillage par hystérésis (**hysteresis**) en consomme **4.08%**.

background_estimation_process

Cette fonction effectue de nombreux accès mémoire, ce qui explique son temps d'exécution conséquent. Une optimisation possible serait de stocker temporairement des blocs d'image dans la **mémoire partagée**, ce qui permettrait d'y accéder plus rapidement et de réduire les latences associées aux accès à la mémoire globale.

5.2 Appels API CUDA

API Call	Temps (%)	Temps total (s)	Appels	Moy (ms)
cudaDeviceSynchronize	70.63	5.13887	1340	3.8350
cudaMemcpy2D	13.09	0.95268	1075	0.88621
cuCtxCreate	4.38	0.31846	2	159.23
cuMemcpy2DAsync	4.32	0.31399	536	0.58580
cudaMallocPitch	4.04	0.29393	808	0.36377
cudaFree	1.48	0.10802	1072	0.10076
cuCtxDestroy	1.09	0.079577	2	39.789
cudaLaunchKernel	0.35	0.025471	1340	0.019008
cuLaunchKernel	0.10	0.0075285	282	0.026696
cuMemFree	0.10	0.0069929	62	0.11279
cudaMemcpy	0.07	0.0050844	268	0.018971

TABLE 4 – Profiling des appels API CUDA

Les appels API qui impactent le plus les performances sont :

- `cudaDeviceSynchronize` : représente **11.03%** du temps total.
- `cudaMemcpy` (Host-to-Device et Device-to-Host) : ces transferts de mémoire prennent également une part significative du temps d'exécution, ce qui indique que la bande passante mémoire est un facteur limitant.

cudaDeviceSynchronize

Nous utilisons `cudaDeviceSynchronize` pour nous assurer que tous les threads se terminent avant de continuer l'exécution du programme. Cet appel de fonction est crucial, car il garantit que les pixels de notre image ne seront pas modifiés avant d'avoir été complètement traités.

Optimisation des Transferts de Données

- Réduire le nombre de transferts `cudaMemcpy` en conservant les données sur le GPU autant que possible.
- Allouer des buffers persistants sur le GPU si les données doivent être réutilisées plusieurs fois, afin d'éviter des copies redondantes entre le CPU et le GPU.

Conclusion et Perspectives

En conclusion, le projet **GPGPU** nous a permis de concevoir un plugin GStreamer performant pour la séparation entre le fond et les objets en mouvement, en exploitant CUDA pour accélérer les traitements. Nos résultats montrent une accélération jusqu'à 75 fois par rapport à la version CPU, tout en maintenant une qualité de détection élevée. Pour aller plus loin, nous envisageons de réduire les transferts CPU-GPU avec des buffers persistants, d'optimiser les accès mémoire via la mémoire partagée et des registres pour les variables locales.

Nous pourrions également adapter le plugin pour des traitements en temps réel et introduire des filtres avancés pour un meilleur nettoyage des masques, ce qui permettrait d'étendre les applications à des domaines tels que la surveillance vidéo ou l'imagerie médicale.

Références

- [1] GStreamer COMMUNITY. *GStreamer Plugin Documentation*. 2024. URL : <https://gstreamer.freedesktop.org/documentation/gstplugin.html?glanguage=c>.
- [2] NVIDIA CORPORATION. *CUDA C Programming Guide*. 2024. URL : <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>.
- [3] EASYRGB. *Mathematical Formulas for Color Conversion*. 2024. URL : <https://www.easyrgb.com/en/math.php>.