Rapport Projet IRGPU

Jules-Victor LÉPINAY, Nathan SUE, Thomas GALATEAU

20 Juin 2025

Table des matières

1 Introduction

1.1 Objectif du projet

Ce projet vise à développer un plugin GStreamer dédié à la séparation automatique entre le fond et les objets mobiles dans des flux vidéo. Cette fonctionnalité est une étape déterminante dans de nombreux pipelines de traitement vidéo, notamment pour la vidéosurveillance ou les applications de réalité augmentée.

1.2 Approche technique

L'algorithme implémenté suit un processus en quatre étapes :
— Estimation du fond : Modélisation du fond de référence
— Création du masque de mouvement : Détection des variations par rapport au fond
— Nettoyage morphologique : Ouverture morphologique puis seuillage d'hystérésis pour éliminer le bruit
— Application du masque : Isolation des objets mobiles sur l'image courante

1.3 Optimisation GPU

La nature locale des opérations de traitement d'image fait de ce projet un candidat pour l'optimisation GPU. Cette approche permet d'exploiter le parallélisme des processeurs graphiques pour traiter efficacement les volumes de données importants générés par les flux vidéo en temps réel.

1.4 Répartition des tâches

Jules-Victor :
— GPU Version 2
— GPU Version 3

Nathan :
— CPU
— GPU Version 1
— Slides

Thomas :
— CPU
— GPU Version 3
— Benchmarks
— Rapport

2 Benchmark des performances

2.1 Performances CPU

Voici les performances du CPU avec le code C++ en termes de FPS en moyenne des vidéos de format mp4 à notre disposition.

TABLE 1 – Analyse des fps moyen pour les vidéos mp4 - Version C++

Maximum	**Minimum**	**Moyenne**	**Ecart-type**
41.69	0.03	3.7	9,21

2.2 Performances GPU

2.2.1 Version initiale

La première implémentation en CUDA est très naïve, c'est une transposition du code CPU vers du CUDA. Il n'y a encore aucune optimisation. On remarque cependant que toutes les fonctions n'auront pas besoin d'être optimisées de la même manière. Dans la Table 2, on remarque très vite que les cudaMemcpy et notre ouverture morphologique sont très coûteux. Cela s'explique par l'utilisation de nombreuses boucles for, qui ne tirent aucun avantage de la parallélisation massive du GPU. À ce stade, sur certaines vidéos, le code CPU est meilleur.

TABLE 2 – Analyse des temps d'exécution CUDA - Version 1

% Time	**μ Time (ms)**	**σ Time**	**μ Inst.**	**σ Inst.**	**Category**	**Operation**
49.9 %	28,818.963	72,149.330	658.1	191.3	cuda_api	cudaMemcpy
24.2 %	13,976.052	35,924.947	328.5	95.7	cuda_kernel	morphoDilationKernel
23.9 %	13,806.038	35,254.661	328.5	95.7	cuda_kernel	morphoErosionKernel
0.5 %	282.944	310.310	328.5	95.7	cuda_kernel	detectMotionKernel
0.4 %	257.304	266.547	328.5	95.7	memory_op	[memcpy Device-to-Host]
0.4 %	256.098	269.398	329.5	95.7	memory_op	[memcpy Host-to-Device]
0.3 %	159.130	6.705	4.0	0.0	cuda_api	cudaMalloc
0.2 %	97.482	107.233	328.5	95.7	cuda_kernel	hysteresisKernel
0.1 %	79.694	86.348	328.5	95.7	cuda_kernel	applyMotionMaskKernel
0.0 %	15.797	5.025	1,642.6	478.3	cuda_api	cudaLaunchKernel
	57,749.503		**4,606.4**			

TABLE 3 – Analyse des opérations mémoire - Version 1

μ Bytes (MiB)	**σ Bytes**	**μ Count**	**σ Count**	**Operation**
1,942.31	1,966.17	328.5	95.7	Device-to-Host
1,971.35	1,987.97	329.5	95.7	Host-to-Device
3,913.66		**658.1**		

2.2.2 Version optimisée n°1

Dans cette première version optimisée, de nombreux changements ont été apportés, dont deux optimisations majeures. La première est une optimisation de l'instanciation de la donnée de la première frame qui compose notre background de référence. Nous avons fait un mapping de la frame directement sur le GPU ce qui minimise le temps de copie. La deuxième grande optimisation concerne les opérations morphologiques. Cette optimisation est inspirée du papier de recherche *Parallel Implementation of Morphological Operations on Binary Images Using CUDA* [1], qui consiste à faire les opérations morphologiques en 1D avec plusieurs passages. On effectue donc une opération en row major, puis column major, et de nouveau row major. Cela permet de faire un mapping et de parcourir la donnée de manière coalescente.

De plus, nous avons supprimé les boucles pour tirer parti de la parallélisation, nous avons rajouté des streams CUDA et optimisé nos cudaMalloc en cudaMallocPitch.

TABLE 4 – Analyse des temps d'exécution CUDA - Version 2

% Time	μ Time (ms)	σ Time	μ Inst.	σ Inst.	Category	Operation
47.5 %	1,081.897	1,027.449	659.1	191.3	cuda_api	cudaMemcpy2DAsync
12.1 %	275.455	287.308	329.5	95.7	memory_op	[memcpy Device-to-Host]
11.7 %	265.849	285.045	329.5	95.7	memory_op	[memcpy Host-to-Device]
6.7 %	153.136	5.380	1.0	0.0	cuda_api	cudaStreamCreate
4.7 %	106.938	114.983	328.5	95.7	cuda_kernel	erosion_column_major
4.7 %	106.783	114.712	328.5	95.7	cuda_kernel	dilation_column_major
3.3 %	75.746	81.005	328.5	95.7	cuda_kernel	dilation_row_major
3.3 %	75.659	80.882	328.5	95.7	cuda_kernel	erosion_row_major
2.5 %	56.902	59.512	328.5	95.7	cuda_kernel	motion_detect
1.3 %	29.807	30.193	328.5	95.7	cuda_kernel	hysterisis
1.2 %	26.439	28.395	328.5	95.7	cuda_kernel	apply_red
1.0 %	22.348	7.083	2,300.7	669.6	cuda_api	cudaLaunchKernel
0.1 %	1.454	0.531	329.5	95.7	cuda_api	cudaStreamSynchronize
0.0 %	0.272	0.035	4.0	0.0	cuda_api	cudaMallocPitch
0.0 %	0.120	0.094	1.0	0.0	cuda_kernel	motion_first_frame
	2,278.806		**6,255.0**			

TABLE 5 – Analyse des opérations mémoire - Version 2

μ Bytes (MiB)	σ Bytes	μ Count	σ Count	Operation
1,947.76	1,970.26	329.5	95.7	Device-to-Host
1,947.76	1,970.26	329.5	95.7	Host-to-Device
3,895.51		**659.1**		

1. https ://www.researchgate.net/publication/304107791$_{P}arallel_{I}mplementation_{o}f_{M}orphological_{O}perations_{o}n_{B}inary_{I}m$

2.2.3 Version optimisée n°2

Pour cette deuxième optimisation, nous avons de nouveau optimisé les opérations morphologiques. Ces opérations étant maintenant en 1D et les données étant coalescentes, il est possible de passer d'une grille 16 par 16 à une grille 256 par 1. Cela permet de faire plus d'opérations en parallèle.
De nouveaux streams pour optimiser le temps de copie des données ont été ajoutés, et les memcpy sont maintenant en asynchrone.

TABLE 6 – Analyse des temps d'exécution CUDA - Version 3

% Time	μ Time (ms)	σ Time	μ Inst.	σ Inst.	Category	Operation
46.7 %	776.095	702.024	659.1	191.3	cuda_api	cudaMemcpy2DAsync
16.5 %	275.160	287.321	329.5	95.7	memory_op	[memcpy Device-to-Host]
16.0 %	265.661	283.456	329.5	95.7	memory_op	[memcpy Host-to-Device]
9.2 %	153.631	2.997	1.0	0.0	cuda_api	cudaStreamCreate
5.0 %	82.599	84.183	328.5	95.7	cuda_kernel	motion_detect
2.5 %	41.044	39.391	328.5	95.7	cuda_kernel	hysterisis
2.4 %	40.167	41.222	328.5	95.7	cuda_kernel	apply_red
1.3 %	20.827	6.457	2,300.7	669.6	cuda_api	cudaLaunchKernel
0.1 %	1.656	1.207	328.5	95.7	cuda_kernel	erosion_row_major
0.1 %	1.595	1.176	328.5	95.7	cuda_kernel	erosion_column_major
0.1 %	1.437	0.514	329.5	95.7	cuda_api	cudaStreamSynchronize
0.1 %	1.415	1.156	328.5	95.7	cuda_kernel	dilation_column_major
0.1 %	1.414	1.160	328.5	95.7	cuda_kernel	dilation_row_major
0.0 %	0.275	0.045	4.0	0.0	cuda_api	cudaMallocPitch
0.0 %	0.120	0.095	1.0	0.0	cuda_kernel	motion_first_frame
	1,663.096		**6,255.0**			

TABLE 7 – Analyse des opérations mémoire - Version 3

μ Bytes (MiB)	σ Bytes	μ Count	σ Count	Operation
1,947.76	1,970.26	329.5	95.7	Device-to-Host
1,947.76	1,970.26	329.5	95.7	Host-to-Device
3,895.51		**659.1**		

3 Analyse des performances

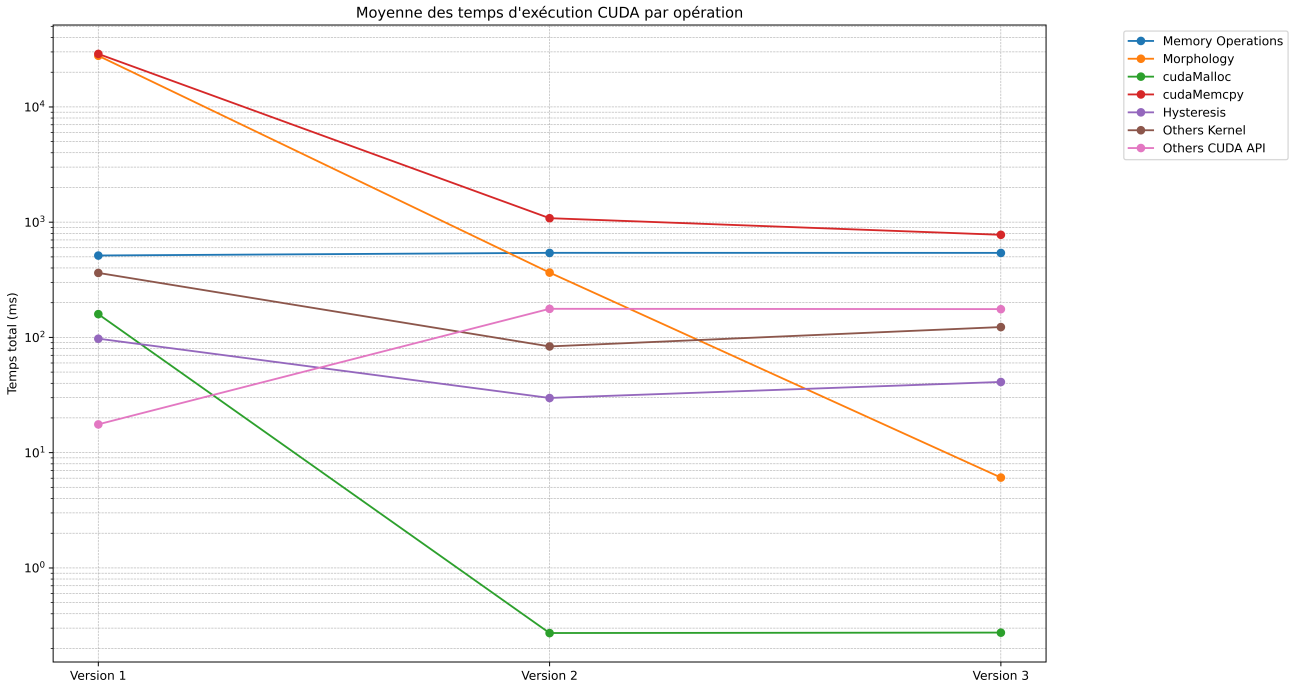

FIGURE 1 – Moyenne des temps d'exécution CUDA par opération

La Figure 1 met en parallèle les trois versions GPU implémentées sur les performances en temps d'exécution sur des groupes d'opérations.
On remarque que la majorité du temps dans la version 1 est prise par les opérations morphologiques et les cudaMemcpy. Nous avons donc optimisé ces deux aspects dans la version 2 qui ont été divisés par 100 en vitesse d'exécution. Cela entraîne la chute du temps d'exécution du cudaMalloc qui est effectué maintenant une seule fois.
On remarque cependant que le temps d'exécution des API CUDA augmente, à cause de l'introduction des streams. Cette augmentation nous permet une diminution plus forte des temps de copie.
Dans la version 3, l'optimisation est surtout apportée sur les opérations morphologiques, mais elle est très significative.

Nous observons cependant que l'optimisation est limitée par le CPU, notamment dans l'attente des threads futex et poll pour récupérer la donnée de GStreamer et du GPU.

Il existe d'autres outils pour travailler avec le GPU en direct depuis la source de streaming tels que cuda-gst.

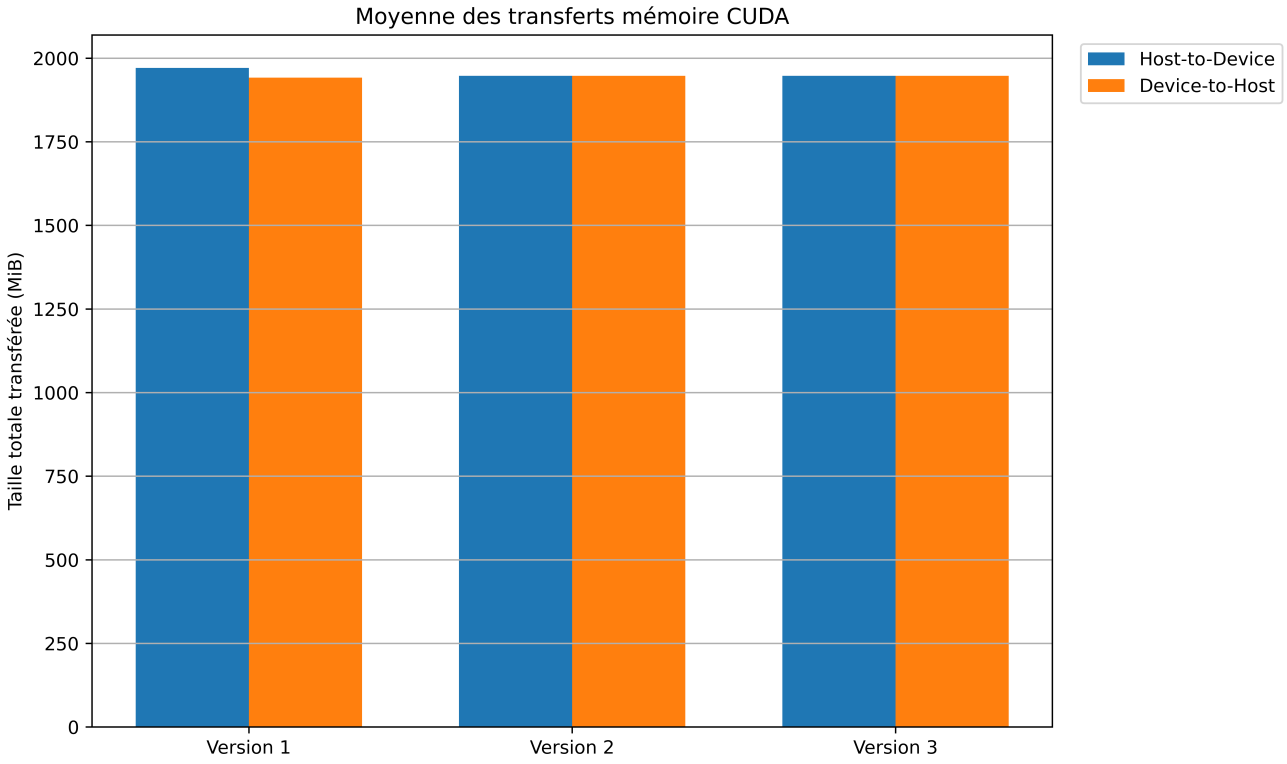

FIGURE 2 – Moyenne des transferts mémoire CUDA