IRGPU RAPPORT

Param Dave
Pierre Litoux
Mohamed Amine El Maghraoui
Hugo Deplagne

Novembre 2023

Sommaire

1 Description du projet

1.1 Contexte et Objectif

Le projet a pour but de développer un algorithme pour séparer le fond et les objets mobiles dans des vidéos. Cette fonctionnalité est essentielle pour la surveillance vidéo, la reconnaissance d'objets, et les effets spéciaux dans le cinéma. L'objectif est de créer un système rapide et efficace en utilisant CUDA.

1.2 Approche et Méthodologie

Le projet, utilisant GStreamer pour le développement du plugin, se concentre sur les étapes suivantes pour le traitement des séquences vidéo :

- **Calcul du masque** : Ce processus implique quatre étapes clés pour distinguer les objets en mouvement du fond.
  1. *Différence entre l'image courante et l'image de fond* : Utilisation d'un algorithme de soustraction de l'arrière-plan avec un niveau de luminosité (lb) pour identifier les changements.
  2. *Ouverture morphologique* : Application de techniques morphologiques pour affiner les résultats de la soustraction, en éliminant le bruit et en améliorant la qualité des contours des objets détectés.
  3. *Seuillage par hystérésis* : Cette étape permet de définir des seuils pour identifier clairement les objets en mouvement, en se basant sur la différence d'intensité ou de couleur.
  4. *Création du masque* : Génération d'un masque binaire qui identifie les régions de mouvement dans l'image.
- **Mise à jour du modèle de fond** : Initialement établi comme la première image de la séquence, le modèle de fond est régulièrement mis à jour à chaque trame. Cette mise à jour se fait en calculant la moyenne des modèles de fond précédents, permettant une adaptation continue aux changements d'environnement dans la vidéo.
  Le calcul de la médiane est censé donner de meilleurs résultats. Nous l'avons donc essayé mais pour cela il faut garder en mémoire les images précédentes et calculer à chaque nouvelle image la médiane sur chaque pixel à partir de l'historique des images. Ceci est très coûteux et plombe nos performances, nous avons donc employé la moyenne.

$$\text{new\_bg\_model} = \frac{n \times \text{bg\_model} + \text{im}}{n+1}$$

  A chaque itération le modèle de fond est mis à jour, nous faisons cette mise-à-jour après avoir calculé le masque car nous voulons mettre à jour le modèle de fond uniquement au niveau des pixels que nous avons détéctés comme faisant partie du fond, c'est à dire la ou notre masque est égal à 0.

Ces étapes forment le cœur de l'algorithme et sont essentielles pour le développement et l'optimisation du plugin en vue d'une intégration efficace avec GStreamer.

1.3 Étapes de Développement

1.3.1 Développement initial en C++ et implémentation du projet, suivi du débogage et de la vérification des résultats.

La première étape cruciale du projet a été de comprendre, s'approprier et implémenter les différents algorithmes en C++. Durant cette étape nous avons du donc mettre en place nos environnements de travail et de debug. La version C++ a demandé beaucoup de debug notamment au niveau de la gestion du modèle de fond mais aussi au niveau des algorithmes 'basiques' comme hystérésis ou érosions/dilatation ce qui nous a demandé de revenir dessus souvent.
Comme vidéo de référence pour le débogage nous somme resté sur la vidéo de base comprenant la voiture sortant du garage comme choix numéro 1 pour nos debugs.

1.3.2 Mise en œuvre initiale en CUDA, de manière très basique.

Sur cette étape nous nous sommes concentrés sur une conversion de notre code C++ en CUDA fonctions par fonctions. C'est-à-dire, nous avons d'abord convertie le calcul de la distance LAB entre deux images consécutives en CUDA 'simple' (partie calcul dans un kernel) ce qui a grandement amélioré les performances sachant que la distance LAB est celle contenant le plus de calcul. Puis nous sommes passés à l'implémention de l'érosion et de la dilatation pour finir sur l'hystérésis.
En faisant toutes ces conversions nous avons déjà grandement amélioré le temps de process des vidéos à voir dans la partie benchmark mais beaucoup peut être fait car nous avons une utilisation non optimisé de la mémoire que ce soit en créant des copies inutiles dans les fonctions hôtes ou en n'optimisant pas l'utilisation de la mémoire coalescente dans les kernels.

1.3.3 Optimisation progressive au fur et à mesure de l'implémentation CUDA.

Nous constatons clairement que le code nécessite une agrégation entre chaque appel de kernel afin de réutiliser directement les résultats. Cela réduirait le nombre d'appels à cudaMemcpy, car ces appels ont un impact significatif sur le temps d'exécution. La version finale de notre code CUDA est optimisée au maximum sur cet aspect, éliminant les copies inutiles et rendant chaque appel significatif. Cet aspect constituera la plus grande optimisation après la conversion du C++ en CUDA.

Un autre aspect amélioré est l'augmentation de la mémoire coalescente et de l'ILP (Instruction Level Parallelism) dans les méthodes kernel. Nous avons déroulé chaque boucle pouvant apparaître dans les kernels pour augmenter l'ILP, ce qui a entraîné une légère amélioration.

Ensuite, nous avons cherché à utiliser au maximum des variables '__constant__' pour accéder à la mémoire constante, notamment dans la distance LAB.

Une grande amélioration de la mémoire coalescente a été réalisée en remplaçant l'utilisation des 'uint8_t' par 'uchar3' dans toutes les copies d'images sur le dispositif. En effet, ce type regroupe efficacement trois valeurs de 8 bits dans une seule unité de 24 bits, économisant de la mémoire par rapport à l'utilisation de variables 'uint8_t' distinctes pour chaque canal.

L'utilisation de types vectoriels tels que 'uchar3' lors de l'accès à des emplacements mémoire voisins conduit à une meilleure coalescence de la mémoire, améliorant les motifs d'accès mémoire et les performances globales.

2 Repartition

	Pierre	Param	Amine	Hugo
Dev-CPP	★★★★★	★★★☆☆	★★☆☆☆	★★★★☆
- Lab Distance	☆☆☆☆☆	★★★☆☆	☆☆☆☆☆	★★★★★
- Ouverture Morpho	★★★☆☆	☆☆☆☆☆	☆☆☆☆☆	★★★★★
- Hystérésis	★★★★★	★★★☆☆	★★★☆☆	☆☆☆☆☆
- Mask	★☆☆☆☆	★★★★☆	★★★★☆	☆☆☆☆☆
- Gestion Modèle de Fond	★★★★★	☆☆☆☆☆	☆☆☆☆☆	☆☆☆☆☆
CUDA Baseline	★☆☆☆☆	★★★★☆	☆☆☆☆☆	★★★☆☆
CUDA Optimisation	☆☆☆☆☆	★☆☆☆☆	☆☆☆☆☆	★★★★★
Benchmarking	★☆☆☆☆	★★★★★	☆☆☆☆☆	★☆☆☆☆
Script de Build	★☆☆☆☆	★★★★★	☆☆☆☆☆	☆☆☆☆☆

Table 1: Répartition des tâches du projet

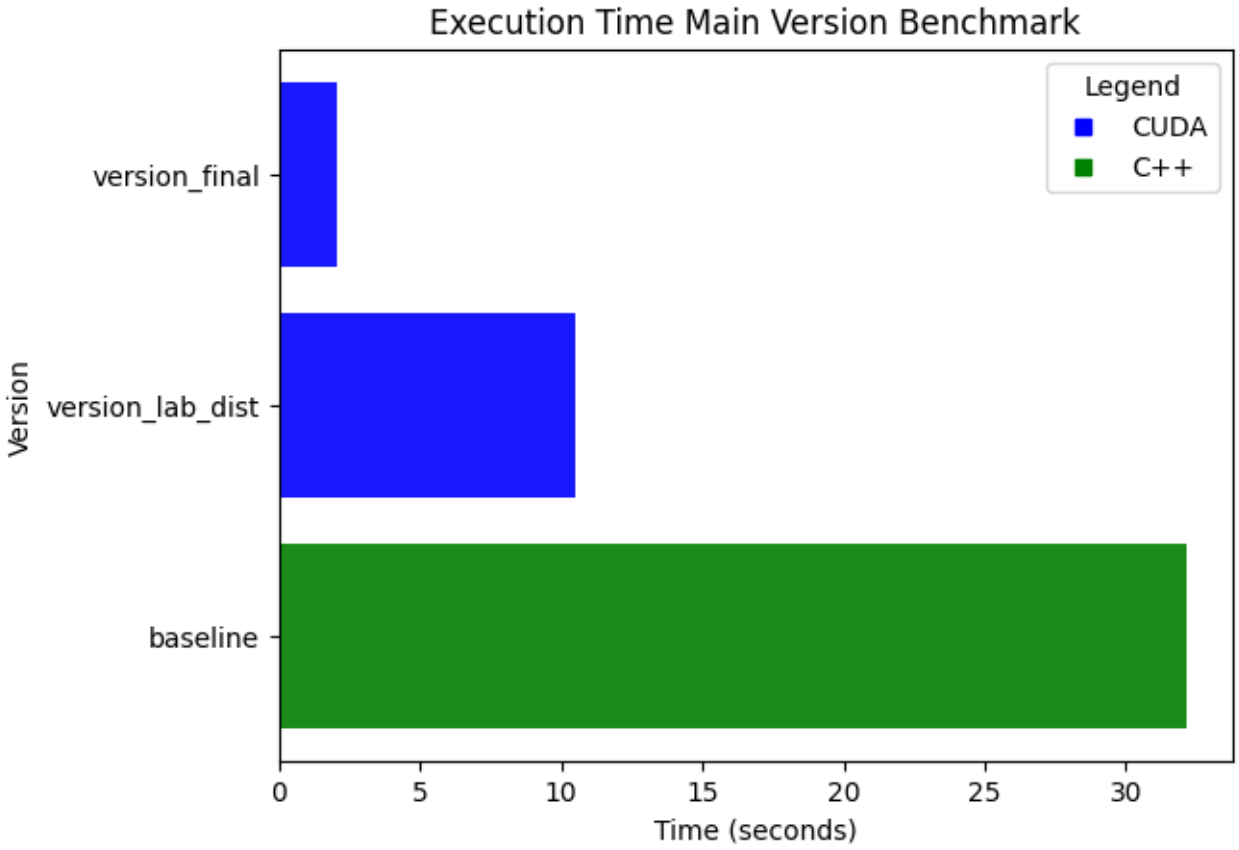
3 Benchmark

Les benchmarks ont été lancés sur plusieurs versions du code. Principalement, le code final en *C++*, l'implémentation naïve en *CUDA* et la version optimisée en *CUDA*.

Nous avons également benchmarker plusieurs versions intermédiaires afin de pouvoir avoir une analyse plus poussée des résultats et de voir la progression en continu de notre projet.
Tout les résultats suivants ont été prit après execution du code de la vidéo de la voiture qui sort du parking.

3.1 Implémentations principales

Voici le benchmark pour les versions principales de notre projet mentionnées plus haut.



En vert, **baseline** est notre implémentation finale en *C++*, dont son execution time dure aux alentour de **32 secondes**.

En bleu, nos implémentations en *CUDA*.

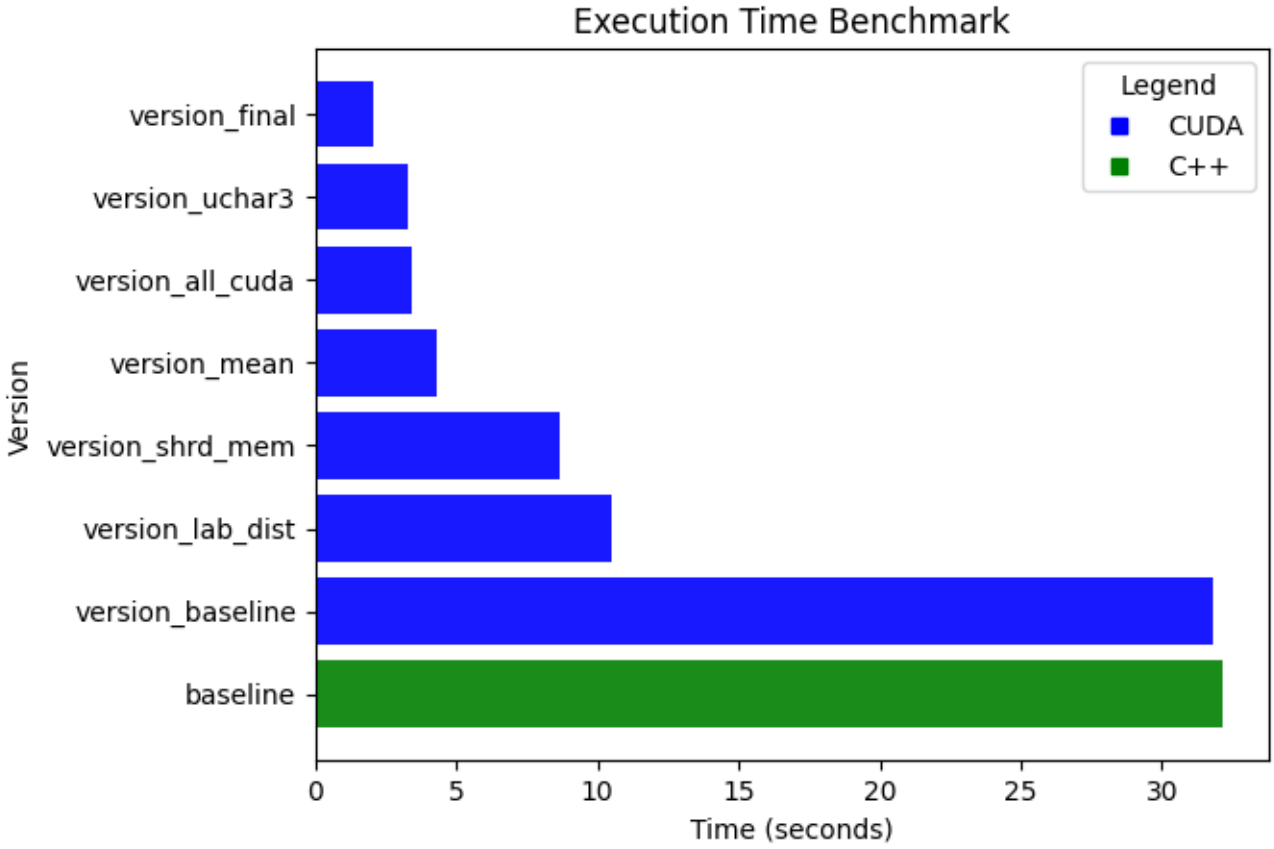
version_lab_dist est la version naïve de notre code où nous avons légèrement optimisé les fonctions de calcul de la distance *LAB*. Ce choix de version vient du fait que c'est la première vraie optimisation que nous avons fait et nous avons estimé cela juste de l'incorporé dans les versions principales à benchmark. Cette version à un execution time d'environ **10 secondes**.

version_final est la version final de notre projet. Son temps d'execution est aux alentours de **1.9 secondes**.

Nous observons un gain de **93%** en temps entre le code final *C++* et *CUDA*.

3.2 Les différents paliers

Grâce à l'utilisation de **GitLab**, nous avons pu facilement récuperer les les versions du projet que nous avons juger nécessaire a analyser. A noter que ci-dessous sont que les versions choisies arbitrairements, pleins de feature ont ete en realite implementer sur d'autre *commits*. Voici le benchmark des différentes versions *CUDA*:



- **version_final**: Passage en **HostToDevice** pour certaines variables.
- **version_uchar3**: Marque le passage des variables de **uint8_t** à **uchar3**. Prend moins d'espace.
- **version_all_cuda**: Première version où tout le code passe par des kernels (optimisation partout).
- **version_mean**: Passage du calcul de l'image de fond de la médiane à la moyenne.
- **version_shrd_mem**: Utilisation de la *shared memory*.
- **version_baseline**: Version sans changement avec le code en *C++*.

Pour regarder plus en détail ces améliorations, faisons une analyse du **profiling**.

4 Analyse

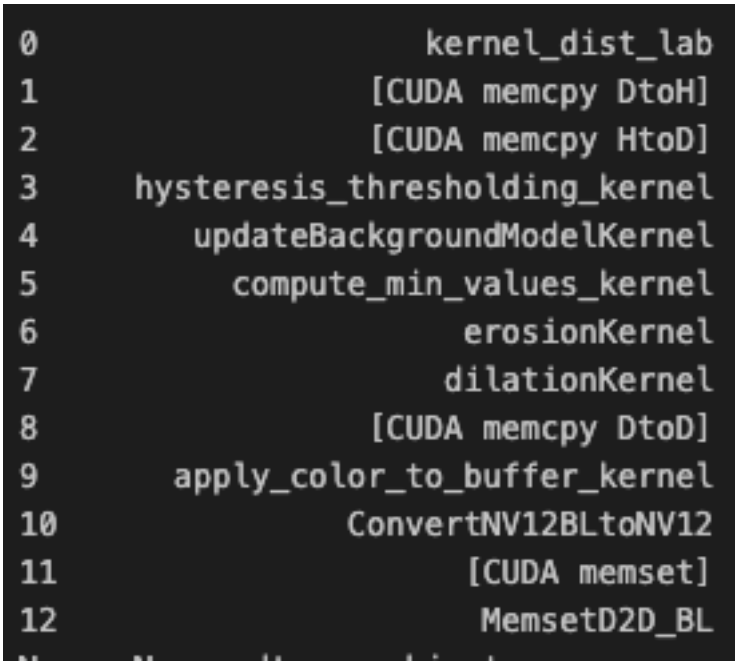
Pour analyser plus en profondeur les performances et les goulots d'étranglement, nous avons décidé d'utiliser **nvprof**.

4.1 Observations

nvprof nous permet d'observer l'activité GPU et les appels API de notre code en *CUDA*.

4.1.1 L'activité GPU

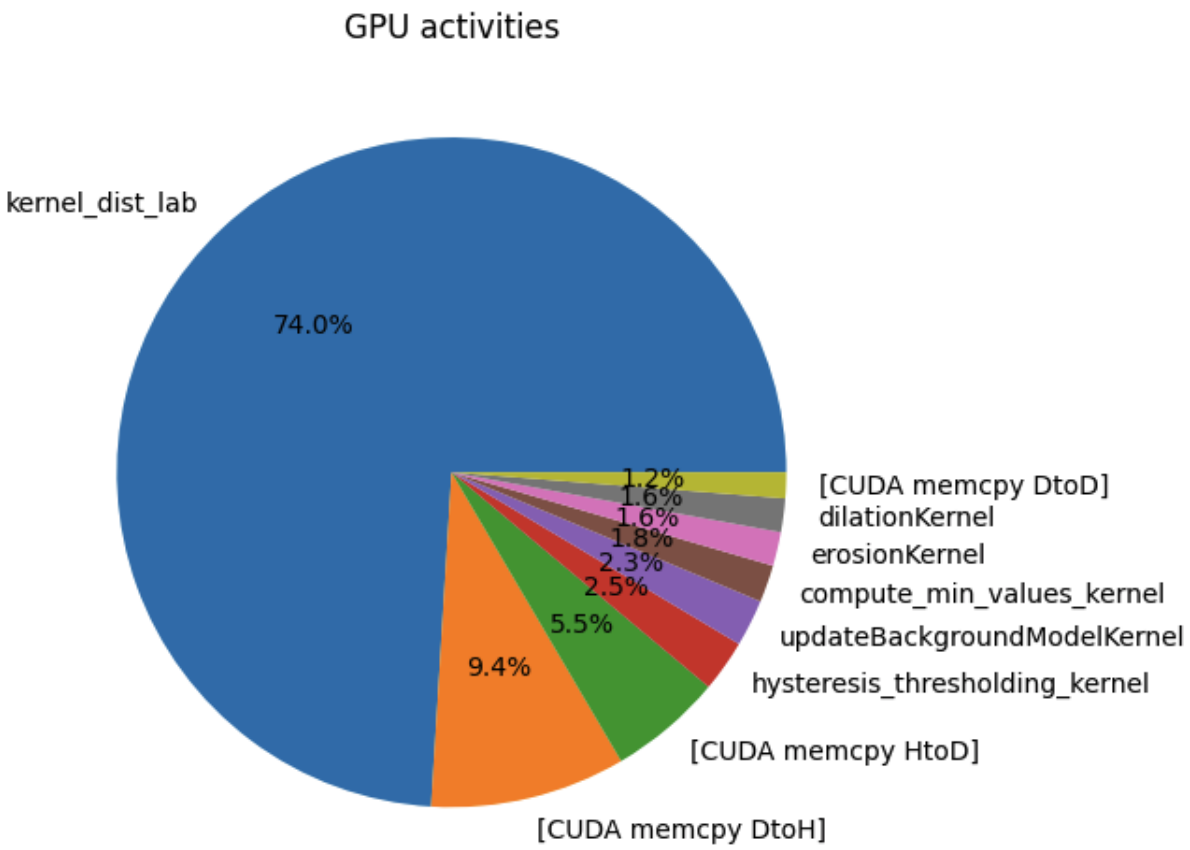
Voici les différentes sources d'activité du GPU:



kernel_dist_lab, **hsysteresis_thresholding_kernel**, **updateBackgroundModelKernel**, **compute_min_values_kernel**, **erosionKernel**, **dilationKernel** et **apply_color_to_buffer_kernel** sont les kernels que nous avons codé nous même.

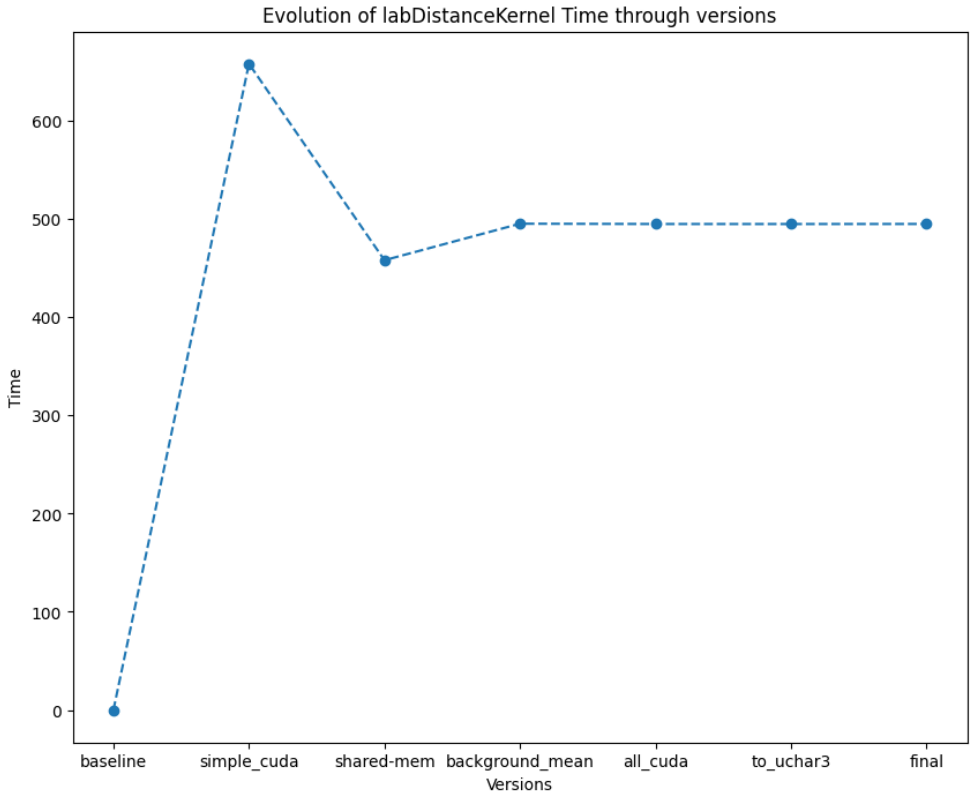
Nous trouvons ensuite les différents types de **CUDA memcpy** et d'autres fonctions.

Regardons l'activité GPU de notre version finale:



Voici un graphique représentant le pourcentage du temps d'exécution de chaque kernel (au-delà d'un seuil de 1%) pour la **version_finale**.

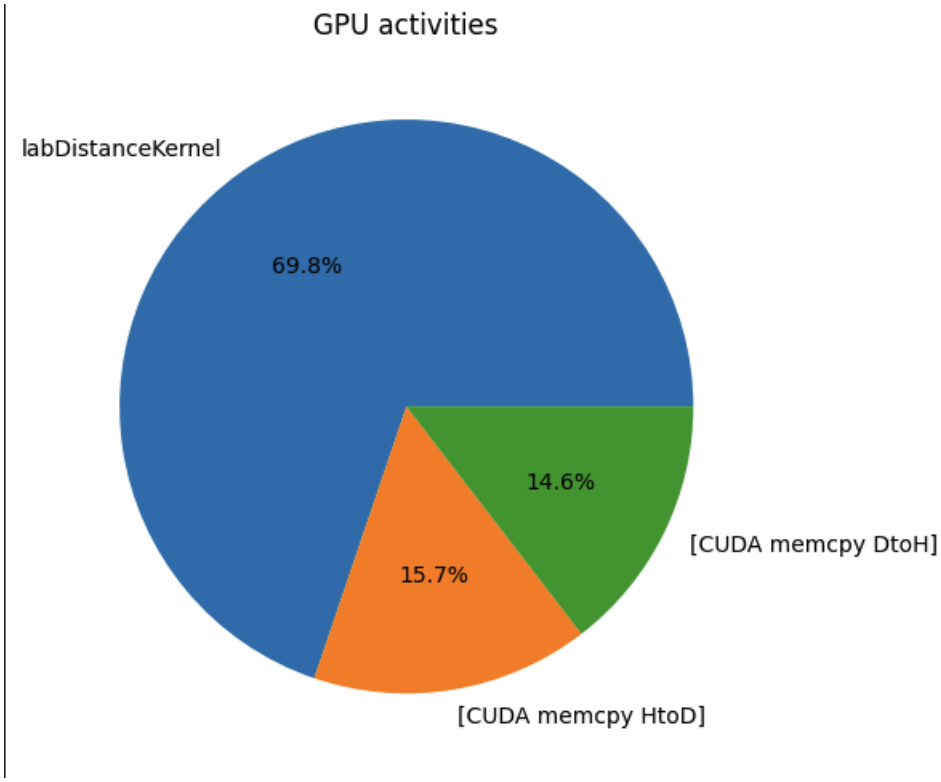
Nous remarquons que le kernel dédié au calcul de la distance LAB représente les trois quarts du temps d'exécution des activités GPU.



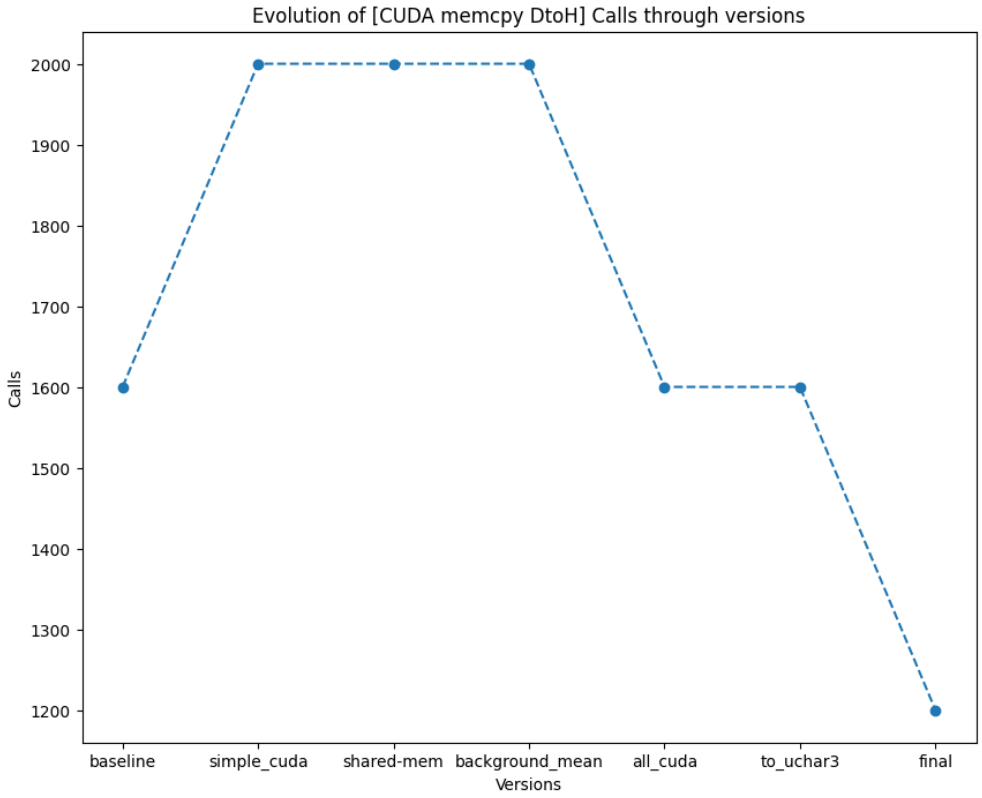
Le temps d'execution du kernel pour la distance LAB est constant à un peu moins de 500ms. Nous remarquons que ce temps augmente au passage entre la version **shared_mem** et **background_mean**. Ceci est du au fait que nous avons enlevé la memoire partagé. On en deduit que l'utilisation de la memoire partage etait correcte et que nous devions la garder.

La deuxième source principale de l'utilisation du GPU est l'utilisation de **CUDA memcpy de Device to Host**.

Voici le graphique pour la **version_lab_dist**:

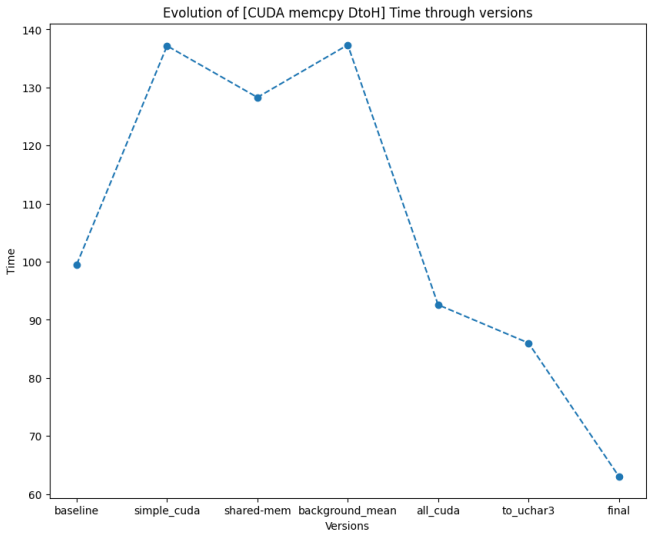


On peut constater que les fonctions **CUDA memcpy** prennent énormément de temps. Pour optimiser notre code, nous avons donc essayé de réduire le nombre d'appels à **CUDA memcpy de Device to Host** autant que possible.



Nous avons réussi à diminuer le nombre d'appels à ces fonctions de manière intelligente et avons ainsi gagné en vitesse d'exécution.

En effet, le nombre d'appels et le temps d'exécution de cette source sont corrélés.

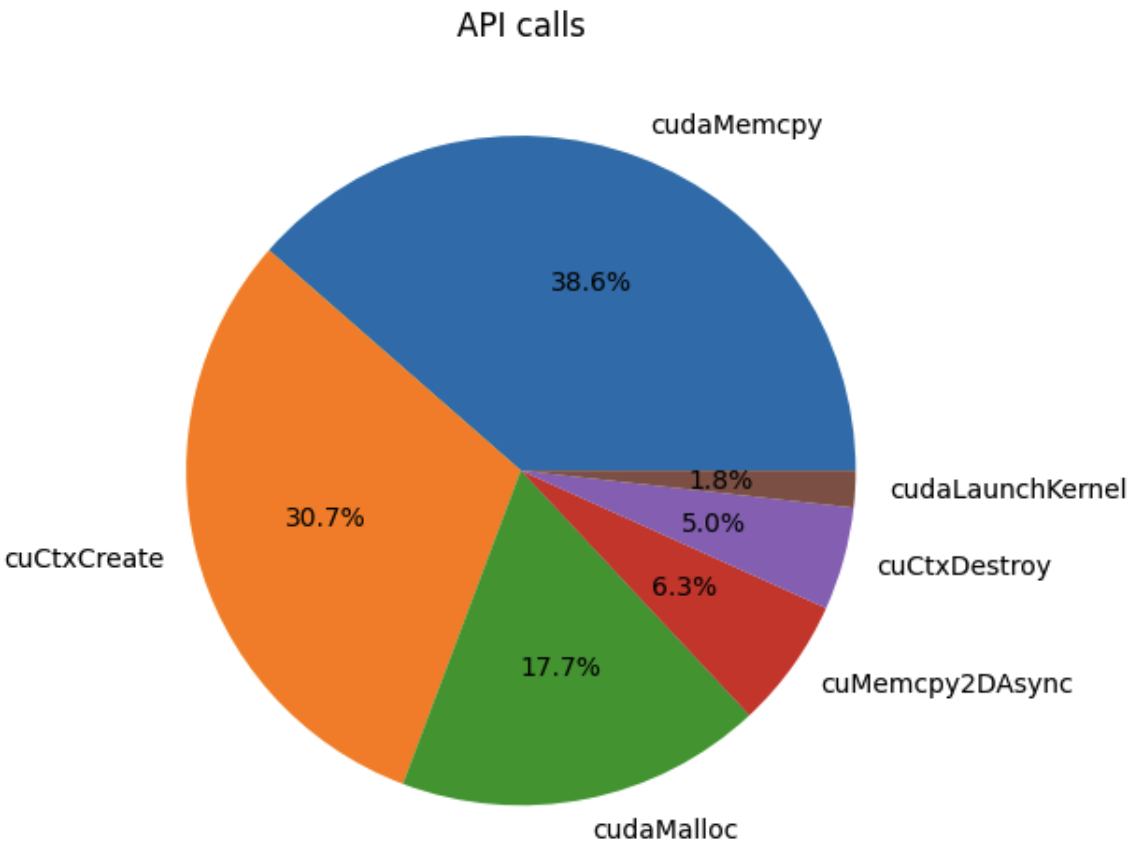


De la même manière, nous avons optimisé l'utilisation de **CUDA memcpy de Host to Device** et avons obtenu des résultats similaires.

4.1.2 Les calls API

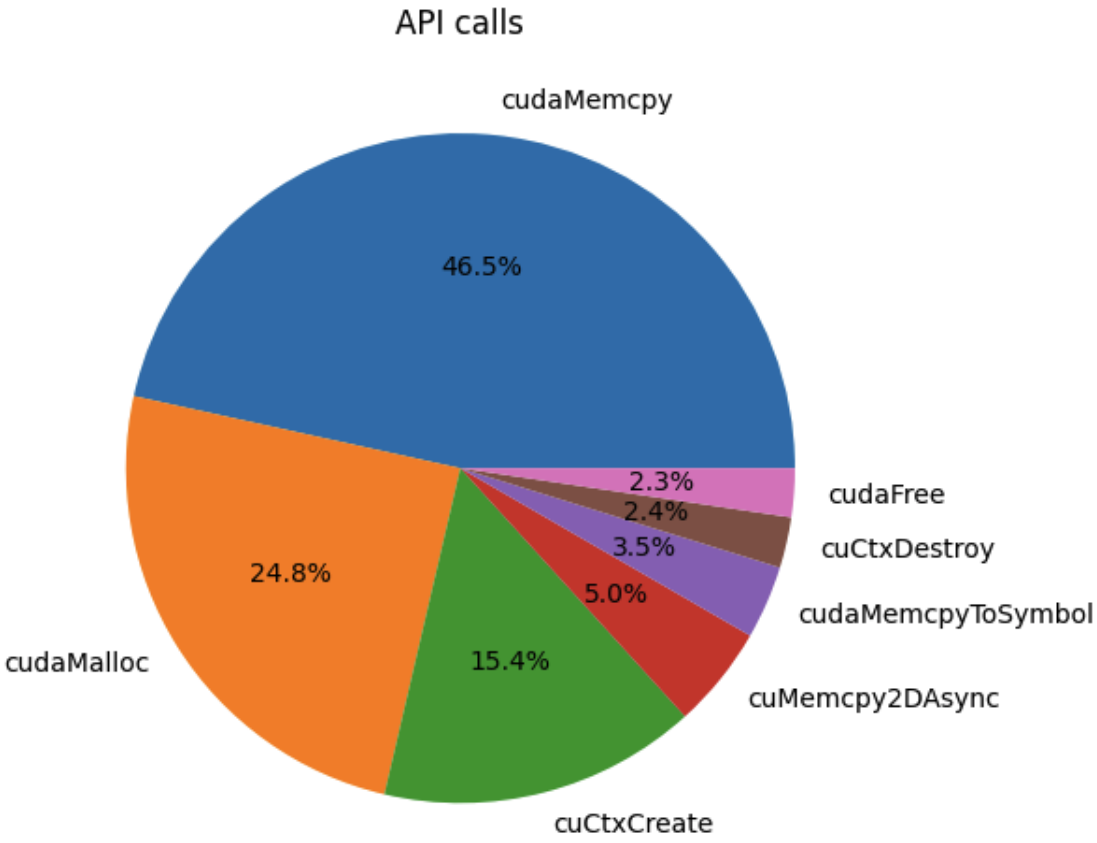
Les appels API sont d'autres fonctions qui impactent le temps d'exécution, il est donc nécessaire d'implémenter notre code en évitant autant que possible ces appels API de manière intelligente.

Voici un graphique représentant le pourcentage du temps d'exécution de chaque appel API (au-delà d'un seuil de 1%) pour la **version_finale**.



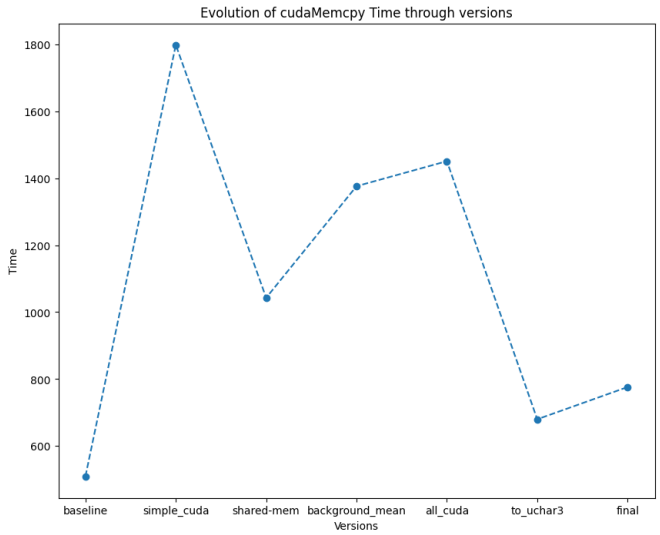
Nous remarquons que les calls API les plus conséquents sont les calls **cuCtxCreate**, **cudaMemcpy** et **cudaMalloc**.

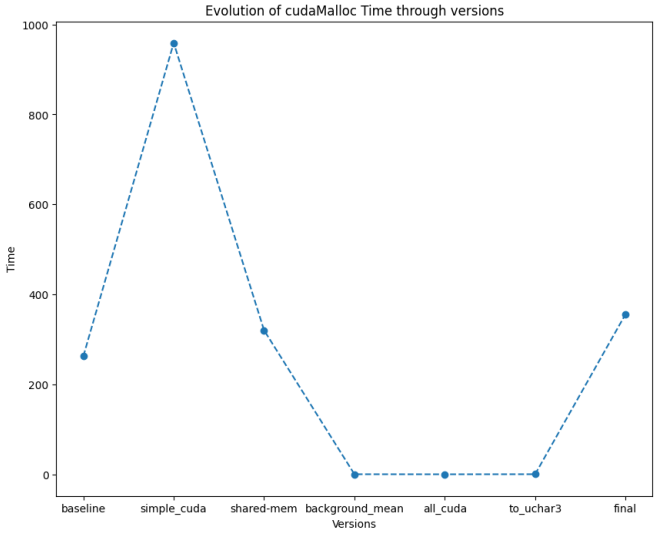
Voici le même graphique pour **version_lab_dist**:



Nous avons réussi à réduire le pourcentage occupé par **cudaMemcpy** et **cudaMalloc**.

Voici l'évolution du temps d'exécution de ces deux fonctions:



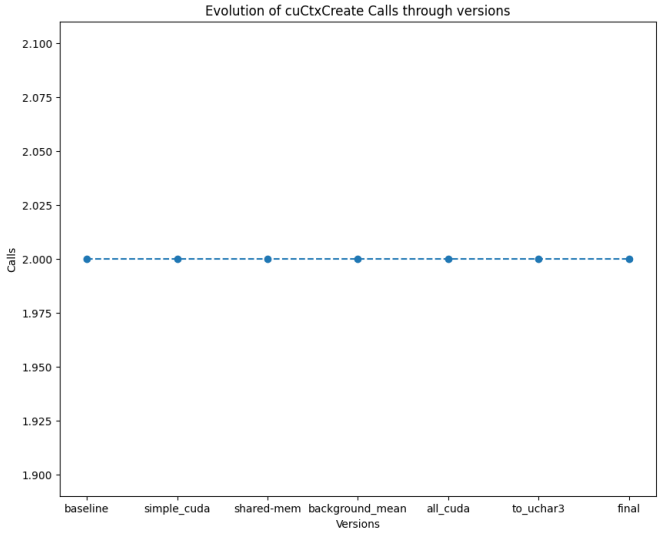

Evolution of cudaMalloc Time through versions
Time
1000
800
600
400
200
0
baseline
simple_cuda
shared-mem
background_mean
all_cuda
to_uchar3
final
Versions

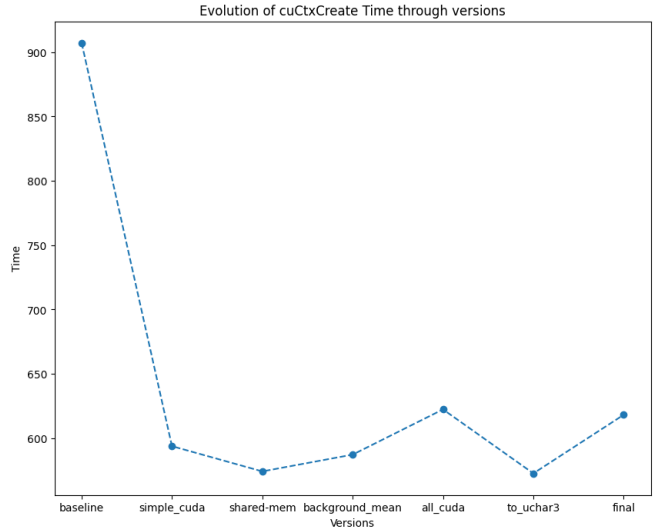
4.2 Bottleneck

Nous avons indentifié plusieurs bottlenecks qui nous empecherai d'optimiser mieux à l'avenir.

out d'abord, l'appel API **cuCtxCreate**. Cette fonction permet de créer un nouveau contexte *CUDA*. Nous ne semblons pas parvenir à gagner du temps sur cette fonction.

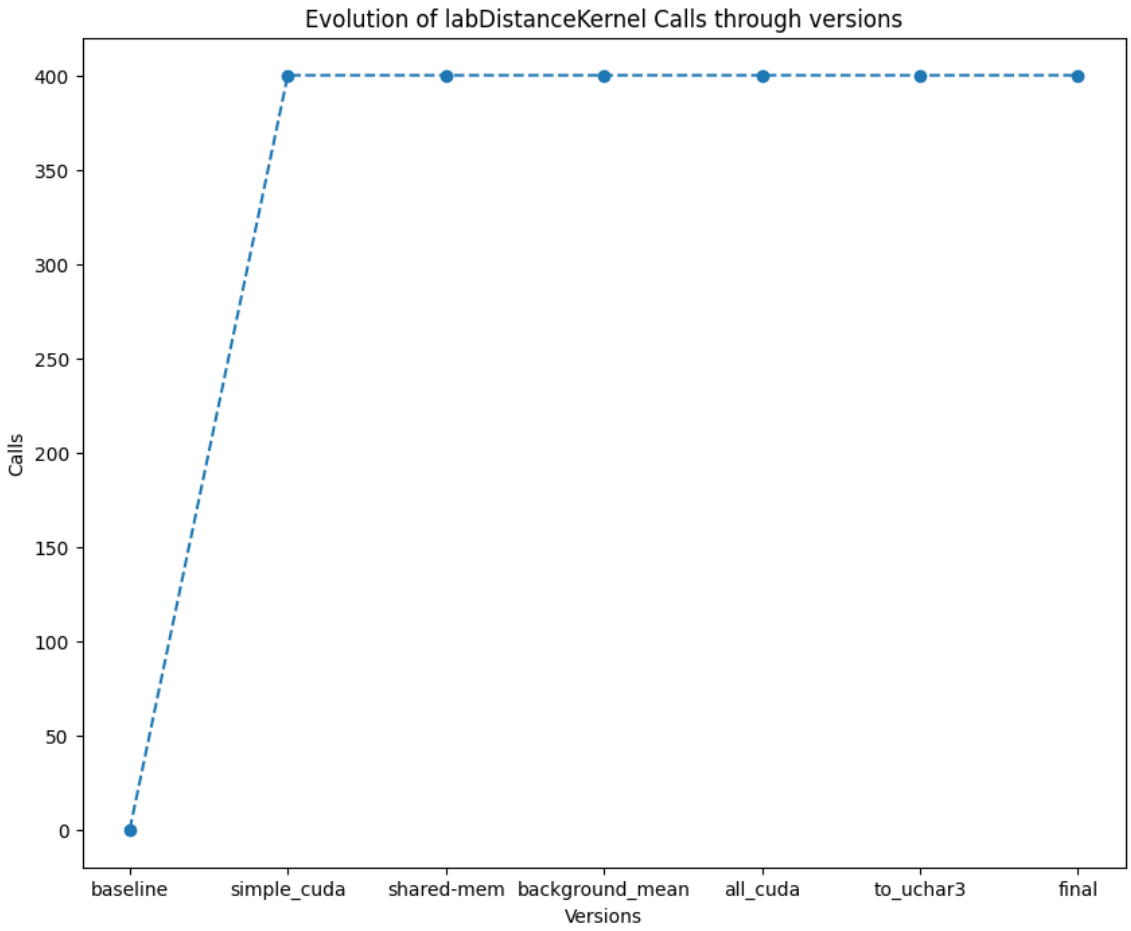
En effet, le nombre d'appel à cette fonction et son temps d'execution varie très peu sur nos dernières versions.





Un autre bottleneck conséquent est notre plus grosse source d'activité GPU: **kernel_dist_lab**.

Malgré sa lenteur par rapport aux autres, nous n'avons pas réussi à réduire le nombre d'appels à cette fonction.


Evolution of labDistanceKernel Calls through versions
Calls
0
50
100
150
200
250
300
350
400
baseline
simple_cuda
shared-mem
background_mean
all_cuda
to_uchar3
final
Versions

5 Conclusion

Ce rapport a exposé le développement d'une solution de traitement vidéo pour séparer le fond des objets en mouvement, utilisant principalement GStreamer. Le projet a démarré avec une approche basée sur C++ et s'est progressivement orienté vers une optimisation en CUDA, entraînant une amélioration significative des performances. Le temps nécessaire au traitement d'une vidéo a été considérablement réduit, passant de 32 secondes à 1,9 secondes grâce à des techniques d'optimisation judicieuses.
Malgré les défis rencontrés, en particulier en ce qui concerne l'optimisation des fonctions CUDA, nous avons réussi à surmonter ces obstacles et à atteindre nos objectifs. En fin de compte, ce projet a mené à la mise au point d'une solution efficace et performante pour le traitement vidéo, répondant efficacement aux besoins spécifiés au départ.