

Rapport du Projet d'Étude et de Développement (PED)

Alexis Couture, Marc Sarraute, Judicaël Grasset, Corentin Salingue

29 mars 2016

Master Informatique

Études et optimisations des performances d'un logiciel de traitement d'images utilisant les données d'eye-tracking provenant des lunettes Tobii.

université
de **BORDEAUX**



Professeur encadrant : Pr. Jenny Benois-Pineau

Dernière révision le 28 mars 2016

Remerciements

Nous souhaitons particulièrement remercier le professeur **Jenny Benois-Pineau** de l'Université de Bordeaux pour ses conseils et son suivi ainsi que pour avoir adapté son sujet de projet à nos compétences.

Nous remercions **Philippe Pérez De San Roman**, stagiaire en master recherche au LaBRI pour son aide et sa coordination sur le projet.

Nous tenons à remercier **Vincent Buso**, ATER¹ dans le groupe AIV² pour son aide dans l'analyse des outils d'annotation.

1. Attaché Temporaire d'Enseignement et de Recherche
2. Analyse et Indexation Vidéo

Table des matières

Remerciements	1
1 Présentation du contexte	3
2 Objectifs	3
3 Protocole expérimental	4
4 Outils d'annotation	4
5 Mesures des performances	4
5.1 Méthode opératoire	4
5.2 Première mesure	5
5.3 Résultats des performances en version initiale	6
6 Optimisations	8
6.1 Usage de la mémoire GPU	8
6.2 Calcul de la carte de chaleur	9
6.2.1 Calcul unique de la LUT	9
6.2.2 Portage sur GPU	9
6.3 Résultat des performance après optimisations	10
7 Perspectives d'évolution	12
7.1 Calcul de la carte de chaleur	12
7.2 La perspective du « tout GPU »	12
8 Conclusion	12

1 Présentation du contexte

Il existe aujourd'hui des prothèses dites myoélectriques qui consistent en un ensemble d'appareils et de capteurs permettant de détecter les impulsions électriques des muscles. Afin d'améliorer les déplacements de ces prothèses, un ambitieux travail de recherche est mené sur l'attention visuelle.

En effet, lorsque l'on souhaite se saisir d'un objet, notre regard va balayer la scène à la recherche de l'objet en question, puis le fixera longuement en voulant le prendre.

Mariam Cheikh Rouhou et le groupe composé de Marin Gutierrez, de Cyril Hatchi, d'Antoine Pitaud et de Mylene Tahar, stagiaires dans le groupe AIV, ont décrit dans leurs rapports [7, 5] le mode opératoire pour montrer l'attention visuelle d'une personne à l'aide des lunettes Tobii qui permettent de capter la position du regard dans une scène. Mariem Cheikh Rouhou a aussi développé un logiciel permettant de traiter les flux de données provenant des lunettes, aujourd'hui repris par Philippe Perez De San Roman, stagiaire en master recherche.

2 Objectifs

Initialement, le projet devait porter sur la synchronisation de capture vidéo entre les lunettes Tobii [1] et une caméra GoPro 3D [4]. Néanmoins, ayant peu d'expérience avec les compétences exigées par le sujet initial, celui-ci a été réorienté par notre encadrante Jenny Benois-Pineau vers le calcul haute performance, en lien avec notre formation du semestre précédent.

Nous nous intéressons donc à l'application développée au LaBRI calculant pour chaque image d'une vidéo la carte de saillance à partir des données d'eye-tracking des lunettes Tobii.

La carte de saillance peut être vue comme un filtre qui met en valeur ce que regarde l'utilisateur (hauts coefficients) et supprime le reste (coefficients nuls). Une carte de chaleur est ensuite calculée, où les couleurs chaudes correspondent au regard de l'utilisateur (voir exemple en figure 1).

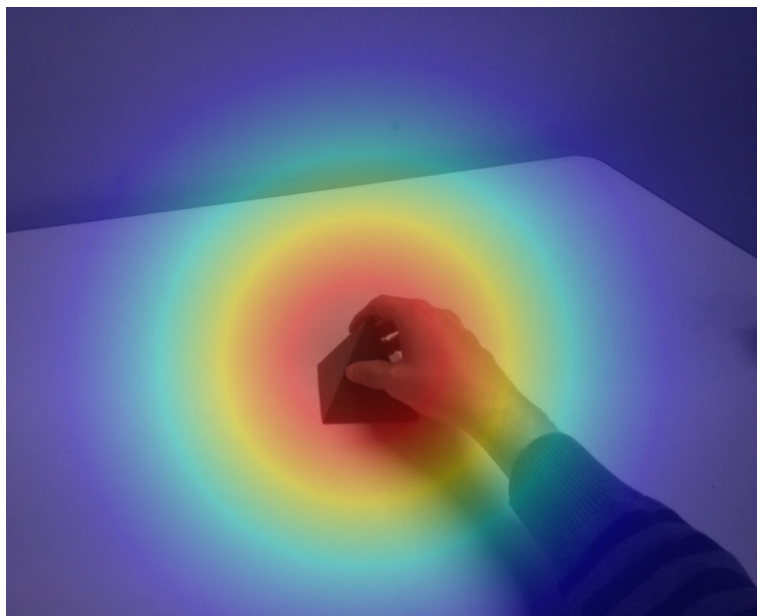


FIGURE 1 – Exemple de carte de chaleur

Notre objectif principal est d'effectuer tous les calculs en *temps réel*, c'est-à-dire, en un temps inférieur à 40 millisecondes (qui est le temps moyen entre deux fixations).

Pour la réalisation, nous avons découpé le projet en quatre grandes étapes :

- Produire des vidéos avec les lunettes Tobii et la caméra GoPro pour fournir des échantillons à mesurer ;
- Mesurer les temps passés dans chaque composante de calcul (interpolation, calcul de la saillance, calcul de la carte de chaleur) ;
- Optimiser le temps de calcul d’une trame ;
- Mesurer ces temps d’annotation et les comparer aux temps d’annotation image par image avec un second logiciel développé par Vincent Buso.

3 Protocole expérimental

Tout au long du projet, nous avons capturé des vidéos à l’aide des lunettes Tobii et de la caméra GoPro. Les vidéos ont toutes été prises suivant le même protocole.

Le sujet porte les lunettes Tobii ainsi que deux caméras GoPro sur la partie droite de son buste. Au début de l’expérience le sujet à les yeux clos. Quatre objets sont posés devant lui sur une table blanche devant un mur blanc.

Pendant la mise en scène, le sujet garde les yeux fermés et reçoit la consigne de prendre un objet particulier. Il doit ensuite ouvrir les yeux, trouver et regarder l’objet en question puis le saisir avec sa main droite. La prise de multiples vidéos permet de produire des échantillons de tests sur lesquels l’application pourra être exécutée et mesurée.

Au total, ce protocole a été appliqué cent cinquante huit fois afin d’obtenir un nombre suffisant d’échantillons, permettant de mieux lisser nos mesures de performance. Nous avons ainsi obtenu un total de plus de 27000 trames (images de vidéo) et de 1000 secondes (voir la figure en annexe).

4 Outils d’annotation

Nous avons annoté les vidéos que nous avons produites avec le logiciel d’annotation de Philippe. Nous produisons en moyenne 1 Go de données par vidéo.

Nous avons essayé d’annoter des vidéos proposées par Vincent Buso avec un outil dont il est l’auteur. Nous nous sommes heurté à un problème technique entre les 2 outils. En effet, le logiciel conçu par Philippe Perez De San Roman ne permet d’annoter la prise que d’un seul objet, alors que le second, permet d’en annoter plusieurs. De plus, les vidéos contenaient la prise de plusieurs objets.

Nous n’avons donc pas pu dans l’état actuel du logiciel développé par Philippe Perez De San Roman, comparer les temps d’annotation et leurs précisions. Néanmoins, on peut donner quelques caractéristiques :

- Le logiciel développé par Philippe Perez De San Roman nous permet « semble-t-il » d’annoter plus rapidement mais avec une précision moindre car le seuil de saillance qui permet de détecter l’objet est unique pour l’ensemble de la vidéo.
- Le logiciel développé par Vincent Buso est plus précis, mais prend plus de temps à annoter car il est nécessaire de dessiner des rectangles autour de chaque objet saisi.

5 Mesures des performances

5.1 Méthode opératoire

Nous nous intéressons aux temps nécessaires pour calculer l’interpolation, la saillance et la carte de chaleur de chaque trame.

Pour cela, nous avons étudié le code et inséré autour des parties à mesurer des *chronos* que nous affichons ensuite dans la sortie standard avec un certain préfixe. Par exemple, pour la chaleur, le préfixe « *Heat :* » était ajouté.

```
1 auto start = std::chrono::steady_clock::now();
2 //code a mesurer
3 auto end = std::chrono::steady_clock::now();
4 std::cout << "Heat:" << std::chrono::duration_cast<std::chrono::microseconds>(end -
    start).count() << "\n";
```

Ensuite, nous avons créé un script bash qui effectue le rôle de parseur en fonction des différents paramètres que l'utilisateur peut lui fournir. Ainsi, ce script peut générer des graphes à l'aide de Gnuplot et calculer des moyennes, des variances et des écarts-types.

Pour lancer efficacement nos mesures sur l'ensemble de la population de vidéos, nous avons créé un second script bash qui exécute les mesures sur toutes les vidéos présentes dans un dossier. Ce script possède lui aussi des options qui lui permettent, par exemple, de lire les vidéos en résolution originale (1080p) ou en taille réduite. Cependant, il faut respecter un format de stockage de vidéo particulier : le fichier JSON et le fichier MP4 doivent être dans le même dossier.

De plus, nous avons tenu une feuille de calcul via « LibreOffice Calc » qui recense toutes les vidéos et fournit quelques statistiques comme le temps total et le nombre total de trames. Voir figure annexe

Enfin, l'ensemble de nos mesures de performances ont été effectuées sur des PC de la salle 203 du CREMI, orientée calcul parallèle. Ces PC embarquent un processeur Intel Xeon E5-2620 v2 avec 24 coeurs cadencés à 2.10GHz et un GPU Nvidia Quadro 4000.

5.2 Première mesure

Lors des premières mesures des logiciels, nous nous avons obtenu des résultats plutôt surprenants : chaque fois que le logiciel s'exécutait, nous avions un temps anormal lors des calculs sur la première trame de chaque vidéo (voir la figure 2. Nous avons alors cherché à comprendre d'où venait ce problème de performance. Après lecture du code, nous n'avons rien trouvé qui puisse expliquer une aussi mauvaise performance lors du premier lancement.

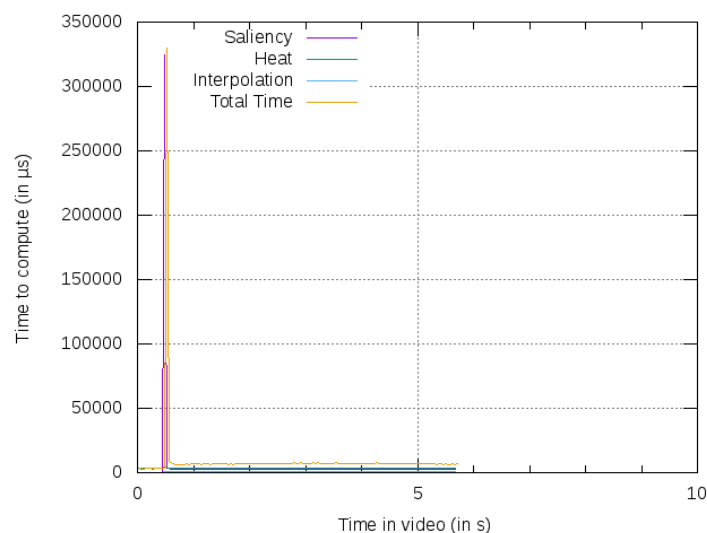


FIGURE 2 – Avant déplacement du pic

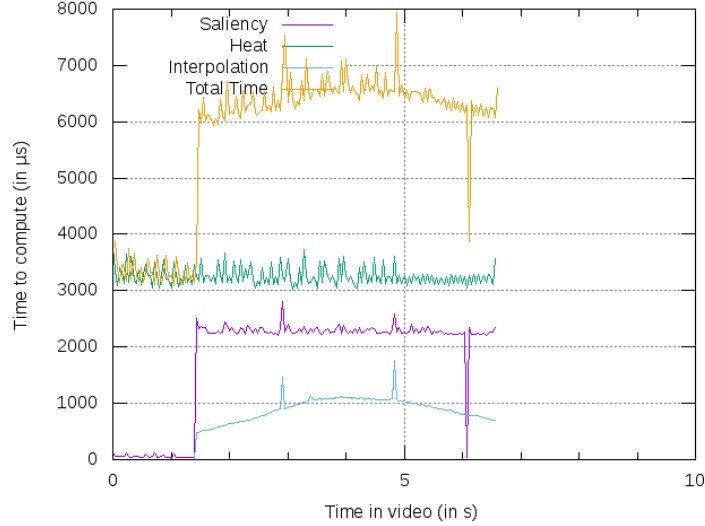


FIGURE 3 – Après déplacement du pic

C'est en lisant la documentation d'OpenCV que nous avons découvert d'où provenait le problème. Le code CUDA généré par OpenCV n'était en fait pas compilé en même temps que le reste du logiciel, mais lors de la première exécution du code[6]. La première itération de la boucle avait donc un temps extrêmement élevé et faussé, puisqu'elle comprenait le temps de compilation du code spécifique à la carte graphique.

Pour résoudre ce problème, nous avons inséré un code effectuant une opération inutile sur la carte graphique dès le lancement du logiciel (une multiplication de matrice de 1x1) :

```
1 //Launch a small computation in order to compile the cuda code
2 cv::Mat cm1(1, 1, CV_32FC1, cv::Scalar(0, 0, 255));
3 cv::Mat cm2(1, 1, CV_32FC1, cv::Scalar(0, 0, 255));
4 cv::gpu::GpuMat m1, m2;
5 m1.upload(cm1);
6 m2.upload(cm2);
7 cv::gpu::gemm(m1, m2, -4, m1, 2, m2);
```

La compilation de tout le code destiné à la carte graphique s'effectuant alors à cet instant, ainsi à aucun moment les calculs utiles ne sont pénalisés.

Une possible amélioration serait de forcer la compilation du code destiné à la carte graphique pendant la compilation du code destiné au processeur. D'après la documentation d'OpenCV cela devrait être possible en spécifiant une architecture de carte graphique particulière dans le fichier CMake d'OpenCV via les variables CUDA_ARCH_BIN et CUDA_ARCH_PTX. Pour une raison qui nous est inconnue, nous n'avons malheureusement pas réussi à obtenir le gain attendu après avoir effectué cette modification, le code destiné à la carte graphique semblant tout de même être compilé lors du premier appel à celle-ci².

5.3 Résultats des performances en version initiale

Avant d'effectuer toute optimisation, nous avons commencé par mesurer les performances obtenues par les versions CPU et GPU de l'application qui nous a été fournie, afin de détecter quelles parties du calcul prennent le plus de temps. Les valeurs présentées ici sont les temps de calcul moyens par trame obtenu sur chaque échantillon vidéo. En outre, les mesures ne prennent en compte que les trames où le sujet de la vidéo a les yeux ouverts, le temps de calcul de saillance étant nul dans le cas contraire.

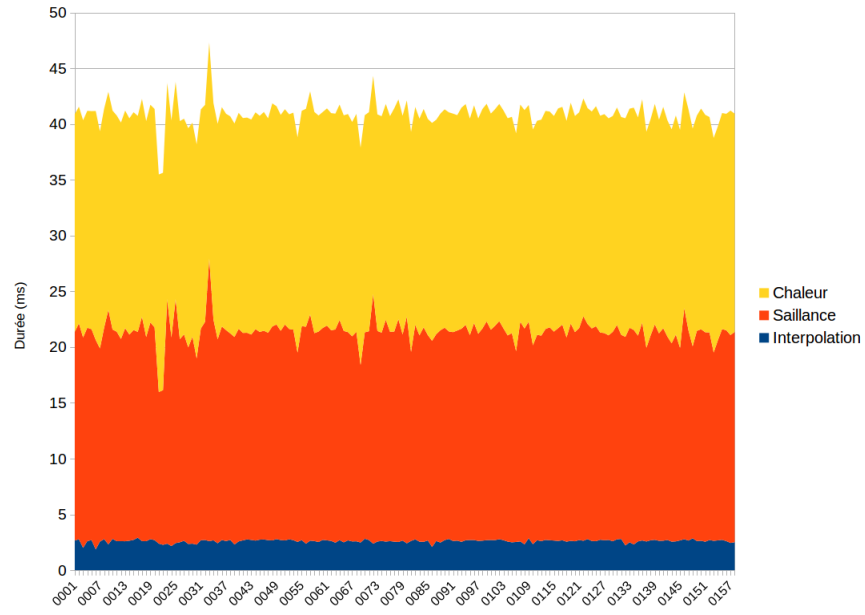


FIGURE 4 – Temps d'exécution (empilés) des différentes phases de calcul sur la version CPU initiale

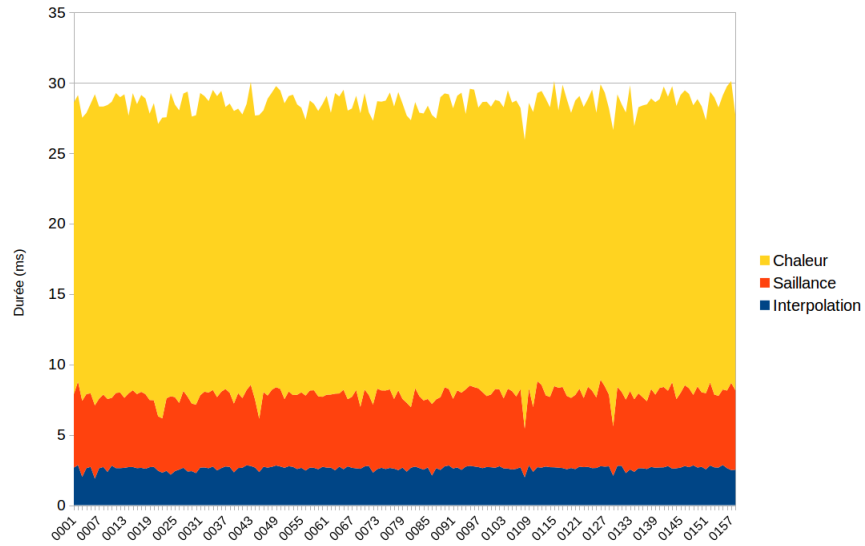


FIGURE 5 – Temps d'exécution (empilés) des différentes phases de calcul sur la version GPU initiale

Les figures 4 et 5 montrent le gain de temps obtenu en passant une partie du calcul sur GPU. Le seul gain notable est néanmoins du côté du calcul de la carte de saillance, dont l'exécution a été rendue 3 à 4 fois plus rapide (voir figure 6).

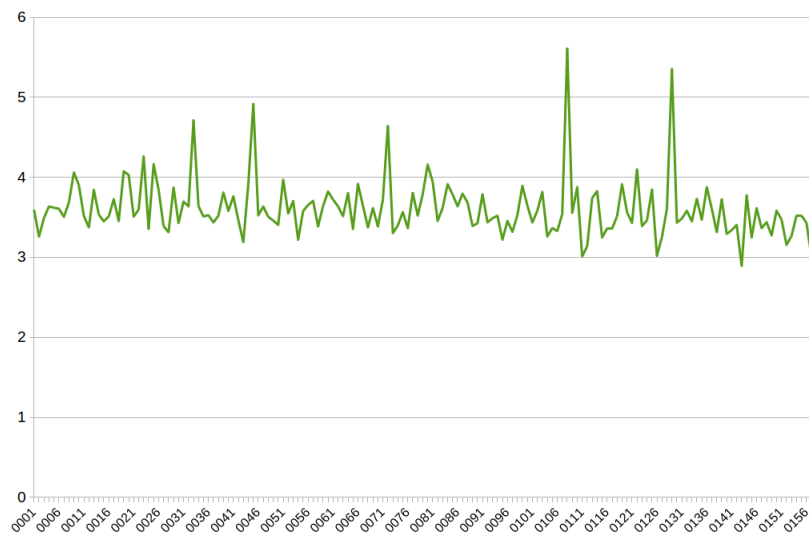


FIGURE 6 – Accélération obtenue sur le calcul de la carte de saillance (GPU par rapport à CPU)

Le calcul de la carte de chaleur occupe la majeure partie du calcul. Néanmoins, la carte de chaleur sert uniquement pour l'être humain à mieux visualiser la carte de saillance, autrement que par des coefficients. L'optimisation des 2 autres phases (saillance et interpolation) est en revanche plus cruciale dans la mesure où ces calculs doivent être réalisés image par image, dans le cadre d'une synchronisation entre 2 appareils différents (lunettes Tobii et caméra GoPro).

6 Optimisations

La finalité du projet étant de traiter en temps réel les flux vidéos parvenant des lunettes Tobii et de la caméra GoPro, il est essentiel de réduire au mieux les temps de calcul par trame, de l'application calculant la saillance sur le flux vidéo des lunettes Tobii.

6.1 Usage de la mémoire GPU

Nous avons dans un premier temps optimisé l'usage de la mémoire pour le calcul de la saillance sur chaque trame. Initialement, la mémoire nécessaire au calcul de la saillance sur GPU était allouée puis détruite à chaque trame. Celle-ci est maintenant allouée uniquement à la première trame et détruite en fin de programme.

Nous avons pour cela ajouté un « *static* » devant la déclaration des pointeurs sur la zone mémoire du GPU. L'allocation de la mémoire est ainsi faite lors du calcul de la première image par OpenCV, et détruite en fin de programme.

```
1 //persistent matrices
2 static cv::gpu::GpuMat gpuGaussian1DX, gpuGaussian1DY, gpuSaliency;
```

En considérant une trame d'une vidéo en couleurs RGB de résolution 1080p, c'est à dire 1 920 x 1 080 pixels, on évite ainsi l'allocation par le GPU d'une matrice de taille (3 x 1920 x 1080) et de 2 matrices de taille (1080 x 1) et (1920 x 1) à chaque calcul de la saillance.

6.2 Calcul de la carte de chaleur

6.2.1 Calcul unique de la LUT

Le calcul de la carte de chaleur nécessite l'utilisation d'une LUT (Look-Up Table) pour pouvoir convertir une échelle de valeurs en couleurs afin d'obtenir les résultats de la figure 1. Dans la version initiale du code, OpenCV construisait et détruisait à chaque trame cette LUT.

Nous avons donc recopié et modifié les fonctions d'OpenCV pour adapter le comportement de création de la LUT. De plus, dans la version GPU, nous avons ajouté un « *static* » devant la déclaration de la zone mémoire de la LUT afin de bénéficier d'une seule allocation pour toute l'exécution du programme.

```
1 // Create LUT with color cv::COLORMAP_JET
2 static Frame lut;
3 //If uninitialized because lut is **static**
4 if (lut.rows == 0)
5     lut = getLutColorMapJet();
```

6.2.2 Portage sur GPU

Nous avons tenté de calculer la carte de chaleur sur GPU. Cependant, nous nous sommes heurtés à un problème technique. En effet, si la conversion de l'échelle de valeurs de la saillance en couleurs fut portée assez facilement sur le GPU, l'intégration de la carte de chaleur sur l'image fut plus compliquée. La ligne de code suivante représente le travail nécessaire à effectuer :

```
1 //alpha = 0.6
2 image_with_heat = alpha * image + (1-alpha) * heatMap;
```

Ce code représente un **DGEMM**³ entre 2 matrices. Il devrait donc naturellement se porter assez facilement sur GPU. Cependant, les matrices *image* et *heatMap* sont des matrices avec 3 canaux (Rouge, Vert et Bleu) et OpenCV ne prend en charge que des matrices avec 1 ou 2 canaux sur ses DGEMM.

Nous avons alors découpé chaque matrice en 3 sous-matrices de 1 canal, puis nous avons effectué un DGEMM sur chaque sous-matrice avant de les ré-assembler. Le code ci-dessous résume le travail :

```
1 //Sub-matrices...
2 static cv::gpu::GpuMat gpuChannelsImage[3];
3 static cv::gpu::GpuMat gpuChannelsHeatMap[3];
4
5 //Split each matrix in 3 channels
6 cv::gpu::split(gpuImage, gpuChannelsImage);
7 cv::gpu::split(gpuHeatMap, gpuChannelsHeatMap);
8
9 //Convert them for DGEMM
10 gpuChannelsImage[0].convertTo(gpuChannelsImage[0], CV_32FC1);
11 gpuChannelsImage[1].convertTo(gpuChannelsImage[1], CV_32FC1);
12 gpuChannelsImage[2].convertTo(gpuChannelsImage[2], CV_32FC1);
13
14 gpuChannelsHeatMap[0].convertTo(gpuChannelsHeatMap[0], CV_32FC1);
15 gpuChannelsHeatMap[1].convertTo(gpuChannelsHeatMap[1], CV_32FC1);
16 gpuChannelsHeatMap[2].convertTo(gpuChannelsHeatMap[2], CV_32FC1);
17
18 //Upload an identity matrix needed to fill the DGEMM function
19 gpuIdentity.upload( Frame::eye(gpuChannelsImage[0].rows, gpuChannelsImage[0].rows,
20                               CV_32FC1) );
21
22 //Apply gemm on each channel
23 cv::gpu::gemm(gpuIdentity, gpuChannelsImage[0], alpha, gpuChannelsHeatMap[0],
24              (1.0-alpha), gpuChannelsHeatMap[0]);
25 cv::gpu::gemm(gpuIdentity, gpuChannelsImage[1], alpha, gpuChannelsHeatMap[1],
26              (1.0-alpha), gpuChannelsHeatMap[1]);
27 cv::gpu::gemm(gpuIdentity, gpuChannelsImage[2], alpha, gpuChannelsHeatMap[2],
28              (1.0-alpha), gpuChannelsHeatMap[2]);
```

3. double general matrix multiplication

```

26 //Convert them for the following computes
27 gpuChannelsHeatMap[0].convertTo(gpuChannelsHeatMap[0], CV_8UC1 );
28 gpuChannelsHeatMap[1].convertTo(gpuChannelsHeatMap[1], CV_8UC1 );
29 gpuChannelsHeatMap[2].convertTo(gpuChannelsHeatMap[2], CV_8UC1 );
30
31 //Merge in HeatMap to save memory
32 cv::gpu::merge(gpuChannelsHeatMap, 3, gpuHeatMap);

```

Cependant, cette version n'est pas très performante car l'ensemble du travail est fait en séquentiel. Nous avons alors intégré des **streams** ou flux de calculs pour essayer d'effectuer en parallèle certaines actions. Les résultats obtenus montrent des performances moins bonnes (de l'ordre d'un coefficient de 3) qu'une version par CPU. Une idée sera présentée dans la section présentant les perspectives d'évolution.

6.3 Résultat des performance après optimisations

On mesure les gains obtenus précédemment en effectuant les mêmes mesures que celles présentées en section 5.3. On peut constater sur la figure 7 un net gain sur le temps de calcul de la phase d'interpolation, qui devient dès lors négligeable (inférieur à 1 milliseconde). Ce gain est en grande partie du aux options de compilation que nous avons ajoutées durant la phase d'optimisation. Les autres calculs gardent le même ordre de grandeur, sachant que nous n'avons pas optimisé le calcul de la carte de chaleur.

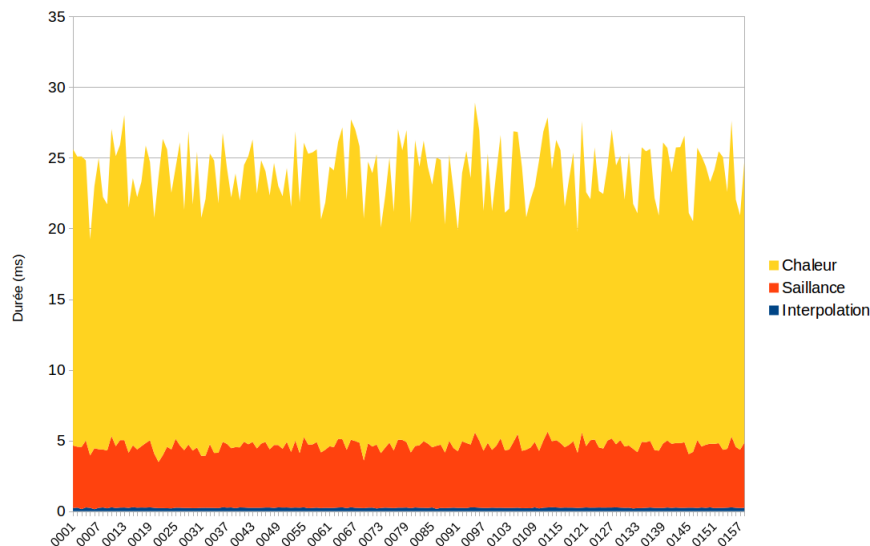


FIGURE 7 – Temps d'exécution (empilés) des différentes phases de calcul sur la version GPU optimisée

Le calcul de saillance a en revanche bénéficié d'une accélération d'environ 20% (figure 8), due aux optimisations listées précédemment.

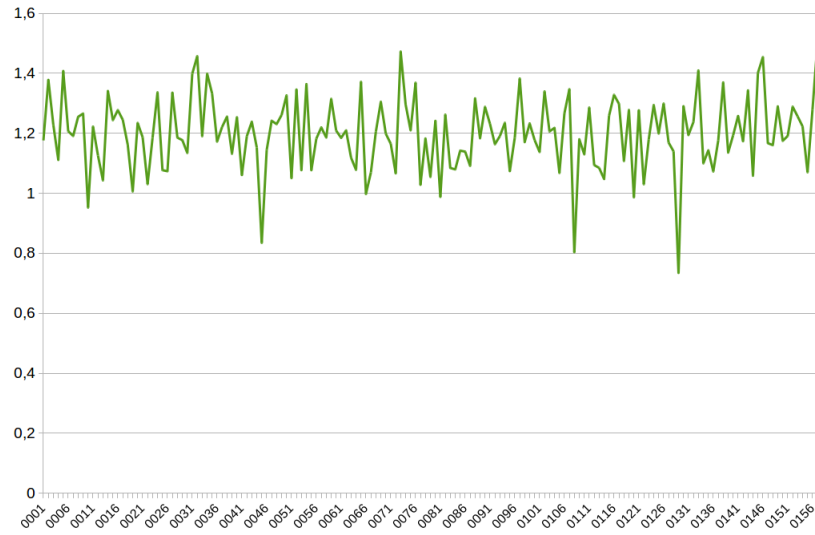


FIGURE 8 – Accélération obtenue sur le calcul de la carte de saillance sur GPU après optimisations

Globalement, nos optimisations ont permis d'accélérer le temps global d'exécution de 10%. En ne considérant pas le calcul de la carte de chaleur, le gain obtenu est d'environ 70% (figure 9). Ce gain correspond à environ 5 millisecondes par trame (figure 10).

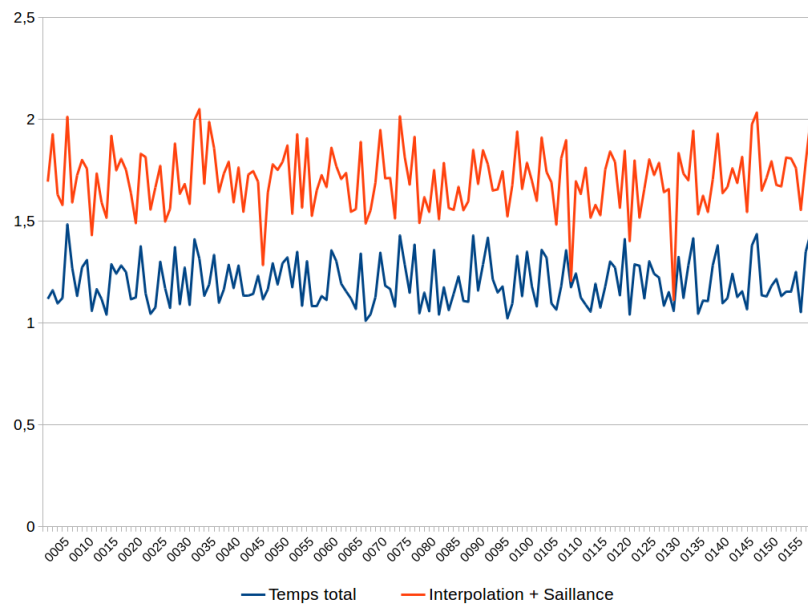


FIGURE 9 – Accélérations du temps global d'exécution et du temps cumulé des calculs de saillance et d'interpolation

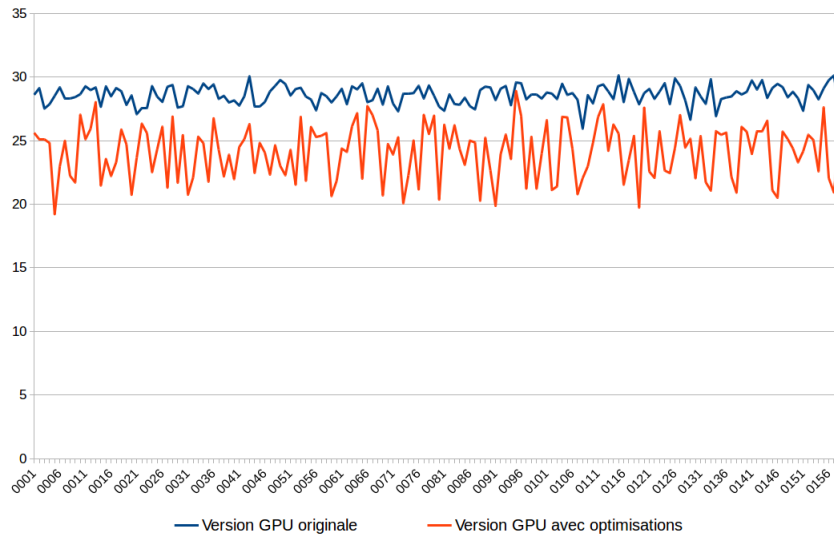


FIGURE 10 – Comparaison des temps d’exécution de la version GPU, avant et après optimisations

7 Perspectives d’évolution

7.1 Calcul de la carte de chaleur

Le calcul de la carte de chaleur sur GPU comme indiqué dans la section traitant des optimisations, nécessite de décomposer l’image en 3 sous-matrices car la fonction DGEMM d’OpenCV ne supporte pas ce format.

OpenCV étant Open-source, il est possible de consulter son code et d’en modifier une partie. Nous avons envisagé de coder une fonction DGEMM supportant les images en 3 canaux. Malheureusement, par manque de temps, cette solution n’a pas été menée à son terme. La principale difficulté sera essentiellement de trouver comment sont stockées les images selon le format **CV_32FC3** par la bibliothèque.

7.2 La perspective du « tout GPU »

Actuellement, la séparation du calcul de la saillance et de la carte de chaleur entraîne le rapatriement de la carte de saillance du GPU vers la mémoire RAM à la fin de son calcul, puis son envoi vers le GPU depuis la RAM au début du calcul de la carte de chaleur.

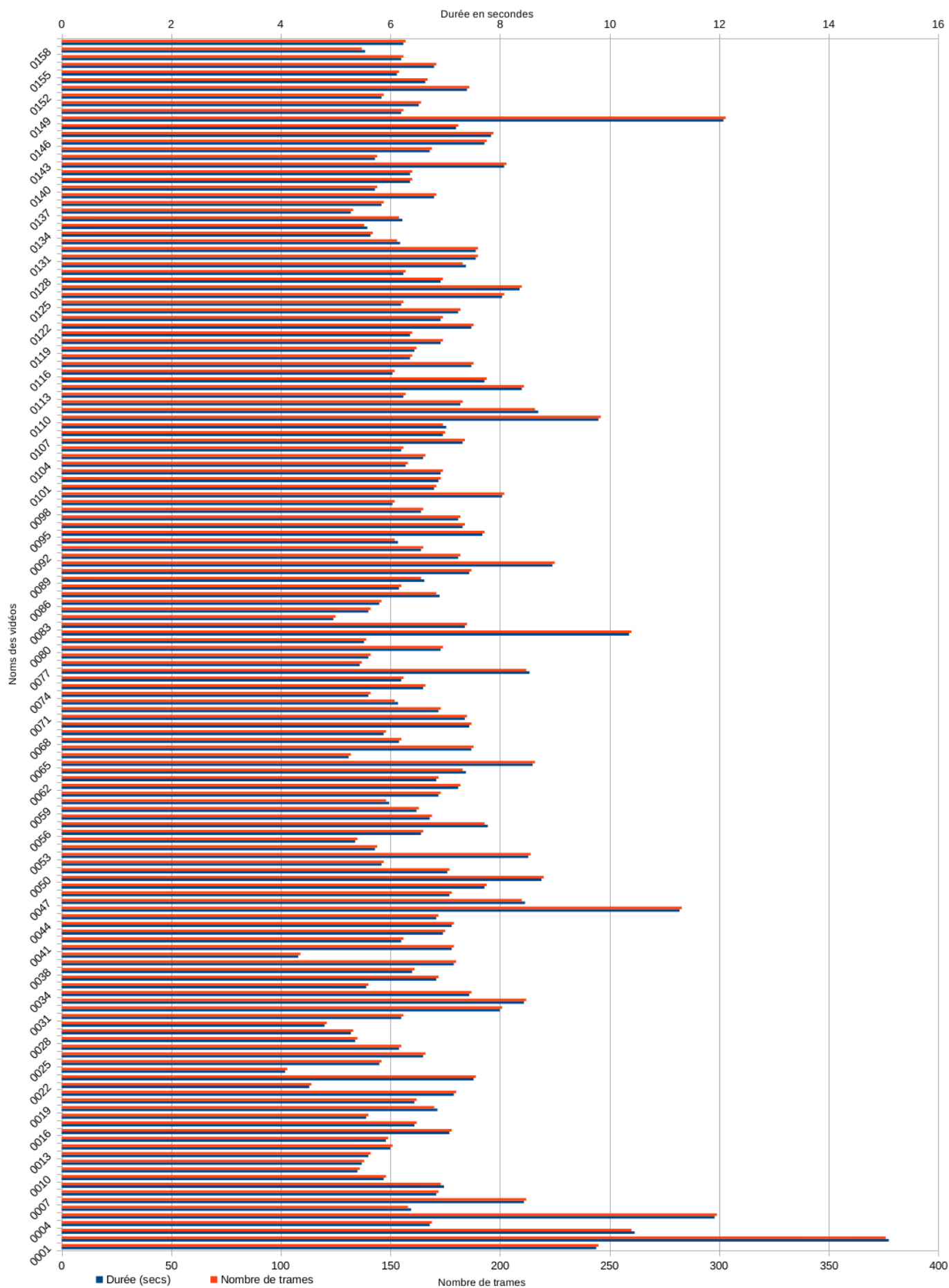
Lorsque l’ensemble des calculs seront portés efficacement sur le GPU, il sera fortement envisageable de transférer un pointeur vers les données présentes sur la carte graphique sans les rapatrier.

8 Conclusion

Ce projet fut très intéressant et possède encore de bonnes perspectives d’évolution en termes de performances. Cependant, nous aurions apprécié pouvoir disposer d’un accès direct au logiciel de version du projet pour suivre les évolutions. En effet, nous recevions les nouveautés des logiciels par clé USB environ une fois par semaine et la fusion des codes nous a fait perdre un temps assez précieux. Il était de plus difficile de comprendre en quoi consistaient et servaient les modifications puisque nous n’avions ni les messages de commit, ni l’historique des modifications.

Références

- [1] Tobii AB. Product description tobii pro glasses 2. <http://www.tobiipro.com/siteassets/tobii-pro/product-descriptions/tobii-pro-glasses-2-product-description.pdf>, 2015. [Online; last accessed 2016-03-28].
- [2] Vincent Buso, Jenny Benois-Pineau, and Jean-Philippe Domenger. Geometrical cues in visual saliency models for active object recognition in egocentric videos. *Multimedia Tools Appl.*, 74(22) :10077–10095, 2015.
- [3] Vincent Buso, Iván González-Díaz, and Jenny Benois-Pineau. Goal-oriented top-down probabilistic visual attention model for recognition of manipulated objects in egocentric videos. *Sig. Proc. : Image Comm.*, 39 :418–431, 2015.
- [4] GoPro Inc. Hero3+ silver manual spécifications. http://cbcdn2.gp-static.com/uploads/product_manual/file/198/UM_H3PlusSilver_FR_REVB_WEB.pdf, 2015. [Online; last accessed 2016-03-28].
- [5] Antoine Pitaud Mylene Tahar Marin Gutierrez, Cyril Hatchi. Biomimetic control of a prosthesis from egocentric videos to reach objects. Master’s thesis, Institut Polytechnique de Bordeaux, 2016.
- [6] OpenCV. Gpu module introduction. <http://docs.opencv.org/2.4/modules/gpu/doc/introduction.html#compilation-for-different-nvidia-platforms>, 2016. [Online; last accessed 2016-03-24].
- [7] Mariem Cheikh Rouhou. Analyse du flux video pour l’amélioration des taches de saisies par neuro-prothèses. Master’s thesis, École Nationale d’Ingénieurs de Sfax, 2015.



Annexe : Durée et nombre de trames des enregistrements