Hes·so Master

MSE | MASTER OF SCIENCE IN ENGINEERING

# Design of Embedded Hardware and Firmware

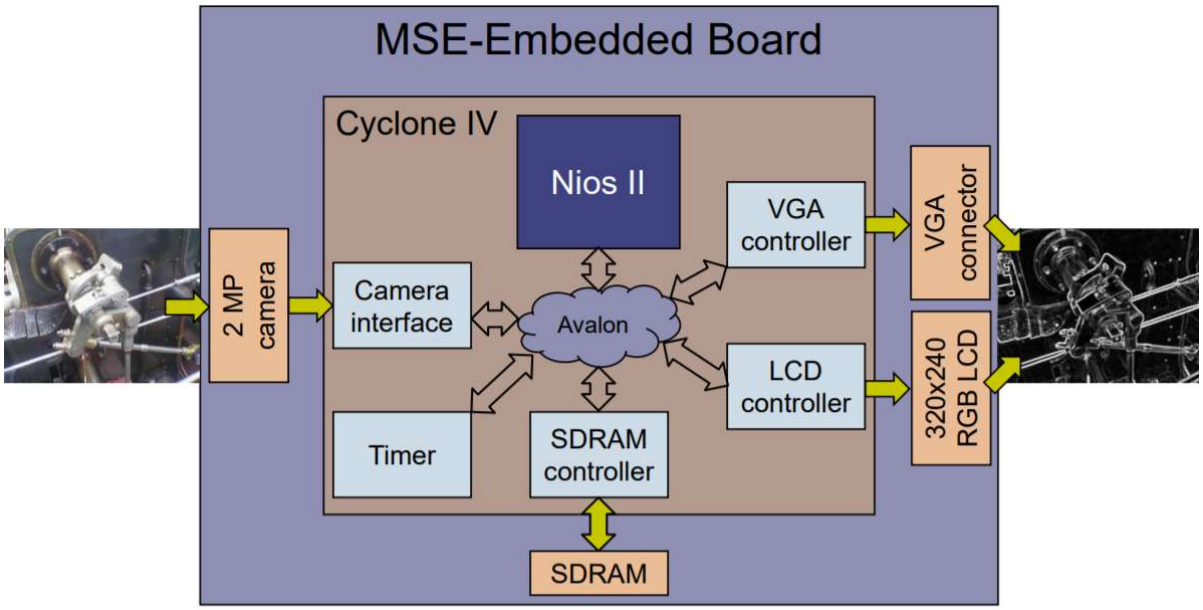
## Sobel


| | |
|---|---|
| Auteur : | Spinelli Isaia |
| Prof : | Andres Upegui |
| | Andrea Guerrieri |
| | Fabien Vannel |
| Assist : | Berthet Quentin |
| Date : | 18.12.2020 |
| Salle : | A2 – Lausanne |
| Classe : | T-EmbHardw |

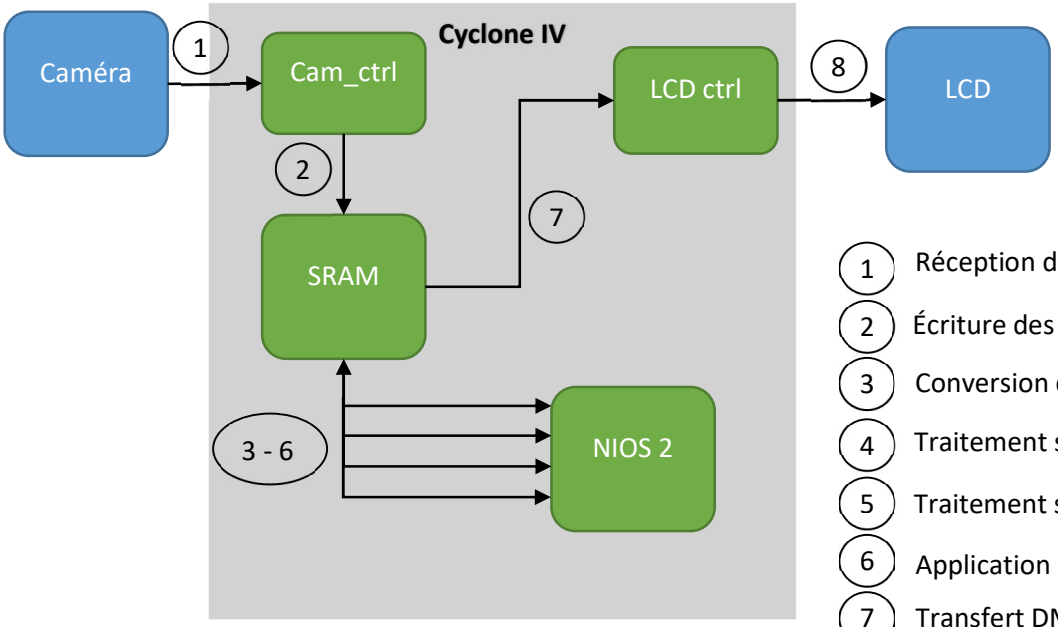
# Table des matières

## Structure du projet

Nous pouvons voir ci-dessous, l'image fourni dans le cours expliquant bien la structure du système complet.




Afin de mieux comprendre le flux des données dans le système fourni (VHDL + C), j'ai réalisé un schéma :

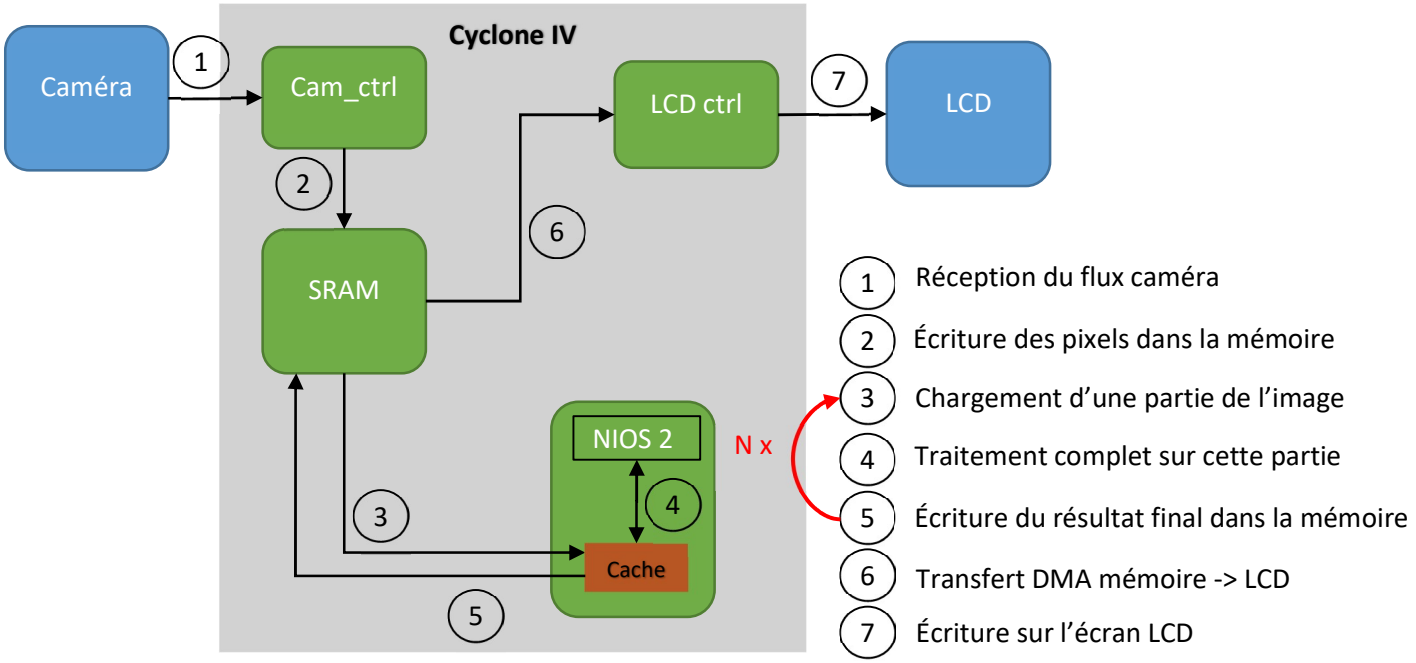



1. Réception du flux caméra
2. Écriture des pixels dans la mémoire
3. Conversion des pixels en gris
4. Traitement sobel_x
5. Traitement sobel_y
6. Application du threshold
7. Transfert DMA mémoire -> LCD
8. Écriture sur l'écran LCD

La première étape consiste à récupérer les pixels de la caméra via le module VHDL fourni. Ensuite, ce module place ces pixels dans la mémoire SRAM. En fonction du mode choisis avec les switches,

différents traitement seront effectués sur les pixels. Pour réaliser ces traitements, les pixels devront être ramenés depuis la mémoire SRAM jusqu'au CPU (registre/cache). Après le traitement, les pixels sont de nouveau placés en mémoire. Ensuite, grâce au module « contrôleur LCD » fourni, ces données seront lues en mode DMA puis renvoyées directement sur l'écran LCD.

On peut constater que les données naviguent beaucoup entre la mémoire SRAM et le CPU pour effectuer du traitement. L'objectif est donc de réduire tous ces échanges. Pour ce faire, voici un nouveau schéma qui permettrait de réduire le temps de transition des données :




La grande différence avec le schéma précédent est que nous allons éviter de passer par la SRAM entre chaque traitement et tenté d'utiliser le plus efficacement la mémoire cache. Pour ce faire, il est nécessaire de découper l'image en plusieurs petites images afin d'utiliser au mieux la mémoire cache. Les traitements seront tous effectués sur une petites images en utilisant un maximum la cache, puis finalement placé dans la mémoire SRAM.

# Les optimisations effectuées

Plusieurs types d'optimisation ont été effectuées sur ce projet afin d'améliorer au maximum les performances du projet. L'objectif est d'avoir une bonne qualité d'image avec le taux maximum de rafraîchissement possible. Pour ce faire, voici la liste des optimisations effectuées :

1. Dérouler les boucle (unrolling loop)
2. Éviter les appels de fonction (Inline functions)
3. Argument du compilateur (O0 - O3)
4. Modification de la fonction « rgb -> gray » (conv_grayscale)
5. Ajout de cache
6. Division de l'image en plusieurs petites images
7. Traitement uniquement sur les parties de l'mages visible
8. Instruction custom

Le déroulage de boucle et la diminution des appels de fonction sont des optimisations simples et indépendantes des architectures.

L'optimisation via les paramètres du compilateur est la plus simple à effectuer et est extrêmement efficace.

La modification de la fonction « conv_grayscale » nécessite un peu plus de réflexion mais reste relativement simple et efficace. Cependant, elle peut engendre une diminution de la qualité du résultat final.

L'ajout de la cache est une modification rarement applicable facilement. Étant donné que nous utilisons un soft core, il nous est possible d'ajouter de la cache. Cette optimisation est très efficace mais a des désavantages tels que ; Augmentation du nombre de logique et diminution de la fréquence maximum.

La division de l'image en plusieurs petites images est une optimisation plus complexe mais permet d'utiliser de façon plus efficace la mémoire cache. De plus, dans notre cas, nous affichons qu'une partie de l'image sur l'écran, il est donc possible d'éviter d'effectuer le traitement sur tous les pixels mais uniquement sur les pixels affichés.

Finalement, les instructions customisées peuvent être des optimisations relativement complexes mais sont extrêmement efficaces. Dans le cadre de ce projet, deux instructions ont été ajoutées :

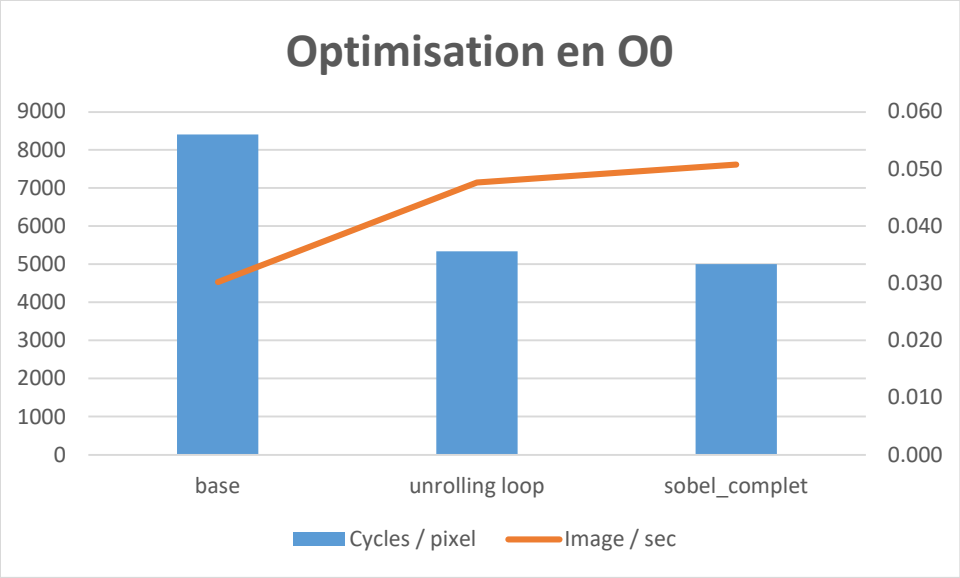
1. Conversion d'un pixel rgb en un pixel gris.
2. Application du filtre « threashold ».

# Évolution des performances

Les démonstrations d'évolution des performances sont présentées sous une forme graphique avec deux échelles. La premières échelles se situe à gauche des graphiques, elle représente le nombre de cycle par pixel. La deuxième échelle est à droite du graphique, elle représente le nombre d'image par seconde.

## Optimisation en O0

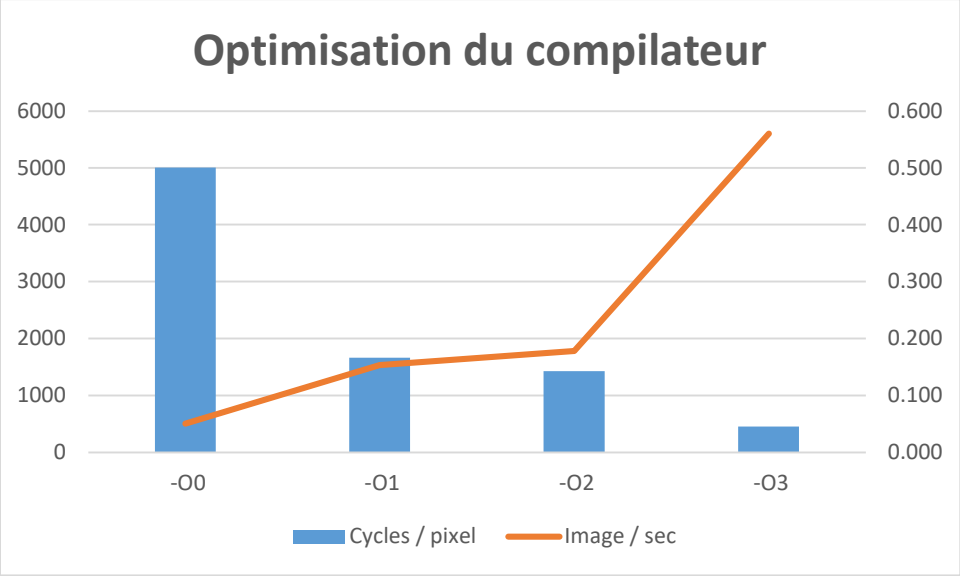
Le graphique suivant représente l'évolution des performances des premières optimisations apportées au projet sans optimisation du compilateur (-O0) :




Nous pouvons constater que le déroulement des boucles apporte une bonne amélioration d'environ 40%. En évitant les appels de fonction en regroupant toutes les fonctions de sobel dans une seule « sobel_complet », une petite amélioration est visible.

## Optimisation du compilateur

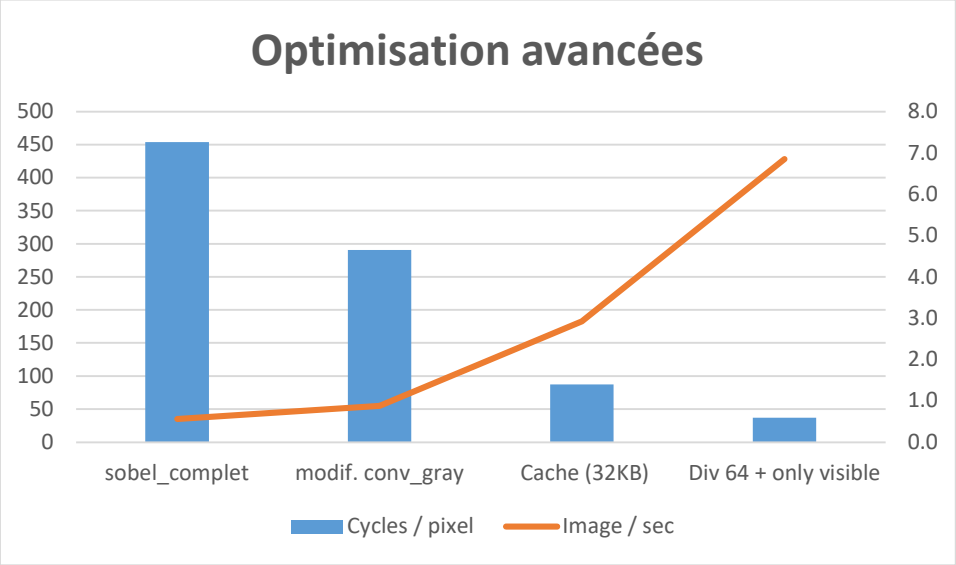
En partant de l'état précèdent du projet, les différentes optimisation du compilateur ont été ajoutées :



La première optimisation (-O1) améliore d'environ 3x les performances de rafraichissement. L'optimisation -O2 améliore encore un peu les performances mais pas beaucoup plus. Finalement, l'optimisation maximale du compilateur (-O3) améliore encore plus de 3x les performances précédentes et plus de 10x les performances de base.

## Optimisation avancées

La graphique ci-dessous, représente l'amélioration des performances des optimisations plus avancés effectuées.

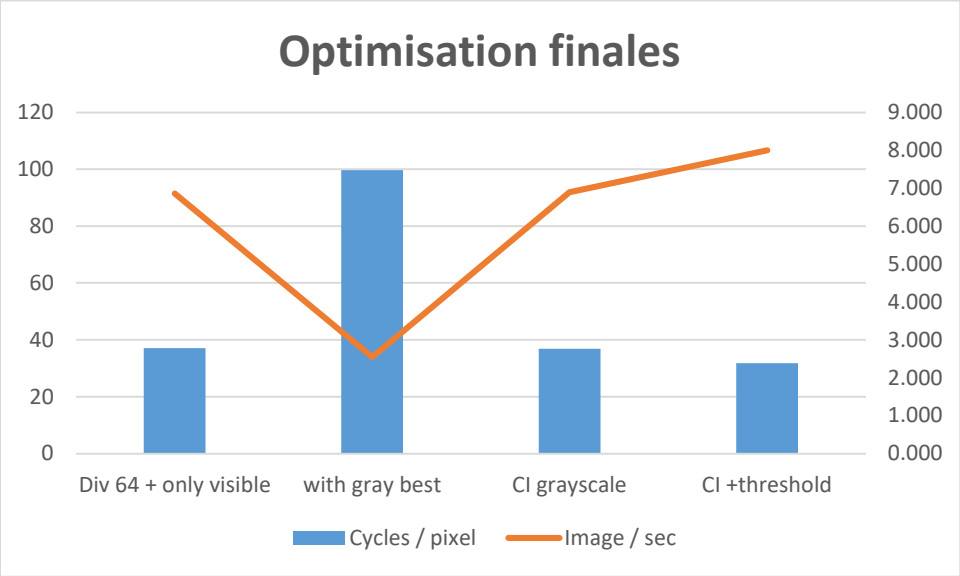



1. Sobel_complet : dernier état du projet.
2. Modif. Conv_gray : en modifiant la fonction « conv_gray » pour améliorer le taux de rafraîchissement d'image en perdant un peu de qualité.
3. Cache (32KB) : En ajoutant 32 KB de cache au processeur Nios 2.
4. Div 64 + only visible : en divisant l'image en 64 petites images et en traitant uniquement les images visible sur l'écran.

On peut constater que toutes ces optimisations sont effectivement très efficaces. Au final, un peu moins de 7 images par secondes est possible.

## Optimisation finales

Personnellement, 7 images par secondes me semblent relativement acceptable à l'œil. De ce fait, l'objectif maintenant est de gardé ce taux de rafraîchissement mais en retrouvant la qualité de résultat de base. Pour ce faire, des instructions customisées ont été ajoutées.




1. Div 64 + only visible : dernier état du projet.
2. With gray best : en utilisant la fonction de conversion « conv_gray » originelle.
3. CI grayscale : en ajoutant une instruction customisé afin de convertir un pixel rgb en gris.
4. CI +threshold : en ajoutant une instruction customisé afin d'appliquer le threshold.

En utilisant la fonction de base « conv_gray », on obtient un meilleur résultat mais le taux de rafraîchissement est retombé en dessous de 3, ce qui n'est pas convenable.

En remplaçant la conversion rgb en gris en une instruction customisé, on peut constater que les performances précédentes sont de retour. En ajoutant encore une instruction customisé afin d'appliquer le threshold, une amélioration a encore pu être observée.

# Conclusion

En gardant une optimisation de compilateur de -O3, les performances ont pu être améliorer de 628 cycles par image en 32 cycles par image, ce qui fait une amélioration d'environ 20x. En prenant compte l'état du projet de base sans aucune modification et sans optimisation du compilateur, les performances ont pu être améliorer de 8408 cycles par image en 32 cycles par image, ce qui fait une amélioration d'environ 263x.

Personnellement, j'ai trouvé ce laboratoire extrêmement intéressant. Le fait de reprendre un projet fonctionnel et de l'améliorer en appliquant toutes les différentes méthodes vue en cours est très instructif.

Finalement, je pense avoir acquis quasiment tous les sujets du cours grâce à ce laboratoire.

# Annexes

Les annexes se trouvent dans le dossier rendu :

1. Code VHDL
2. Code C
3. Final bitstream
4. ELF files

Date : 16.12.20

Nom de l'étudiant : Spinelli Isaia