Travail d'Étude et de Recherche

Evaluation d'une architecture SoC (ARM-FPGA) pour des applications de traitement d'images

Rapport des recherches et travaux

DOU Yuhan
GHAOUI Mohamed Anis
ZHANG Boyang

Encadré par : **ELOUARDI Abdelhafid**

Master Électronique, Énergie Électrique et Automatique - Orsay-Cachan
Master Électronique, Énergie Électrique et Automatique - André Ampère

Année : 2018-2019

Table des matières

Table des figures

Introduction générale

Durant les dernières décennies, le besoin d'effectuer des calculs toujours plus importants n'a cessé d'augmenter à un tel point que les améliorations des performances des systèmes à μProcesseur, dits à *architectures classiques*, n'ont pu suivre. Donc ce genre d'architecture devient obsolète.

On a alors d'abord procédé à la parallélisation des calculs sur plusieurs cœurs du même type. Ce type d'architecture est nommée *Architecture Homogène/Symétrique Multi-cœurs*. On a un seul circuit intégré qui porte en lui plusieurs cœurs du même type.

Ce type d'architecture a grandement améliorer les performances pour plusieurs applications. De nos jours, la quasi-totalité des ordinateurs et téléphone contient une architecture multi-cœurs. Toutefois, certains systèmes temps-réel, de sécurités et cyber-physiques requièrent une autre répartition asymétrique des ressources en nombres et type de cœurs alloués. On parle alors d'*Architecture Hétérogène/Asymétrique Multi-cœurs*.

Ces architectures consistent en une puce qui porte tout le système. On parle alors de *System on Chip*. C'est donc un seul circuit intégré qui possède plusieurs cœurs de type et de répartition différentes. On trouve alors des *SoC*s conçues pour des applications plus spécifiques car ce genre d'architecture peut intégrer un ou plusieurs cœurs *CPU*s, *GPU*s, *DSP*s ou *Fast*s ou plusieurs combinaisons de ces cœurs là.

Dans ce rapport, nous allons présenter les résultats des mesures de performances de l'une de ces architectures hétérogènes en y implémentant des programmes de traitement d'images. On appliquera un filtre de Sobel sur des images puis des vidéos. Le but de ce projet n'est pas de étudier le traitement d'images en question ; Mais plutôt, de s'en servir comme un outil d'évaluation de performances des différentes architectures proposées.

On s'intéressera dans un premier temps au traitement d'image sur *CPU ARM*. Ceci, nous permettra de poser des critères d'évaluation adaptés à la comparaison des différentes architectures. Dans une second temps, On procèdera au même test afin de mesurer les performances d'une architecture *CPU+FPGA* utilisant les outils de développement *OpenCL*. On comparera les résultats entre les deux architectures. Par la suite, on appliquera le filtre de Sobel sur des vidéos de différentes résolutions pour chacune des architectures. Enfin, On étudiera la possibilité de la parallélisation du traitement sur les différents cœurs disponibles. Ce projet se base sur le compte rendu bibliographique [1] des recherches menées par notre équipe.

Pour finir, on clôturera par une conclusion et nos perspectives.

Chapitre 1

Généralités sur le traitement d'image et présentation de la carte DE1-SoC

Introduction

Le traitement d'image est une discipline qui requiert des ressources conséquentes pour des algorithmes de plus en plus complexes. Ses applications se font en général sur des systèmes embarqués tels que la vision des voitures autonomes.

1.1 Propriétés et formats d'image

Une image possède plusieurs propriétés qui définissent comment les données sont représentées en mémoire. Pendant la conception d'une architecture dédiée à un certain algorithme de traitement, Il est important de prendre en compte ces propriétés d'encodage qui sont prédéfinis suivants le format de l'image à traiter.

On s'intéresse ici à des images non-compressées (format *ppm*[2]) codées sur 3 octets où chaque canal est codé sur 8 bits. Afin de pouvoir lire et écrire ces fichiers images, on exploitera les outils proposées par la bibliothèque *OpenCV*[3].

1.2 Filtre de Sobel

Le filtre de Sobel consiste en une convolution de une image en niveaux de gris par deux noyaux de dimensions 3×3 pour calculer le gradient horizontal et vertical. Les noyaux sont décrits par les matrices suivantes :

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Ceci permet de calculer les transitions vers la droite et vers le bas. Il suffit alors de pendre la nouvelle valeur du pixel en fonction de la norme quadratique entre G_x et G_y :

$$G = \sqrt{G_x^2 + G_y^2} \qquad \text{; où G est la nouvelle valeur du pixel[4].}$$

Il est important de noter que, pour des raisons de rapidité, il est parfois plus judicieux de faire des approximations et prendre une norme différente telle que la norme absolue :

$$G = |G_x| + |G_y|$$

Ceci permet de solliciter moins de ressources matérielles pour les calcul.

La complexité algorithmique de l'opérateur est linéaire par rapport au nombre de pixels à traiter.

Il est important de noter que les performances du traitement d'une image 1000×2000 diffère de celle du traitement d'une image 2000×1000 même si le nombre de pixels reste le même. Ceci est dû aux différences d'arrangement et d'accès à la mémoire.

1.3 Application sur l'architecture

Une fois qu'un algorithme est développé, il faut l'adapter en fonction de l'architecture ciblée. Pour cela, il faut vérifier si les possibilités logiciels sont en accord avec les caractéristiques matérielles. On vérifie la dimension maximale qui puisse être stockée sur les mémoires de la carte, l'extension *SIMD* [5] est disponible, La possibilité de parallélisation, etc..

1.4 Caractéristique de la carte

Tous les évaluations et opérations seront effectuées sur la carte *Cyclone V : DE1-SoC*[6]. La carte possède tout le système sur la même puce. c-à-d, le *FPGA* et le *HPS (Hardware Processor System)* coexistent côte à côte entourés de tous les périphériques nécessaires comme représenté sur ce schéma :

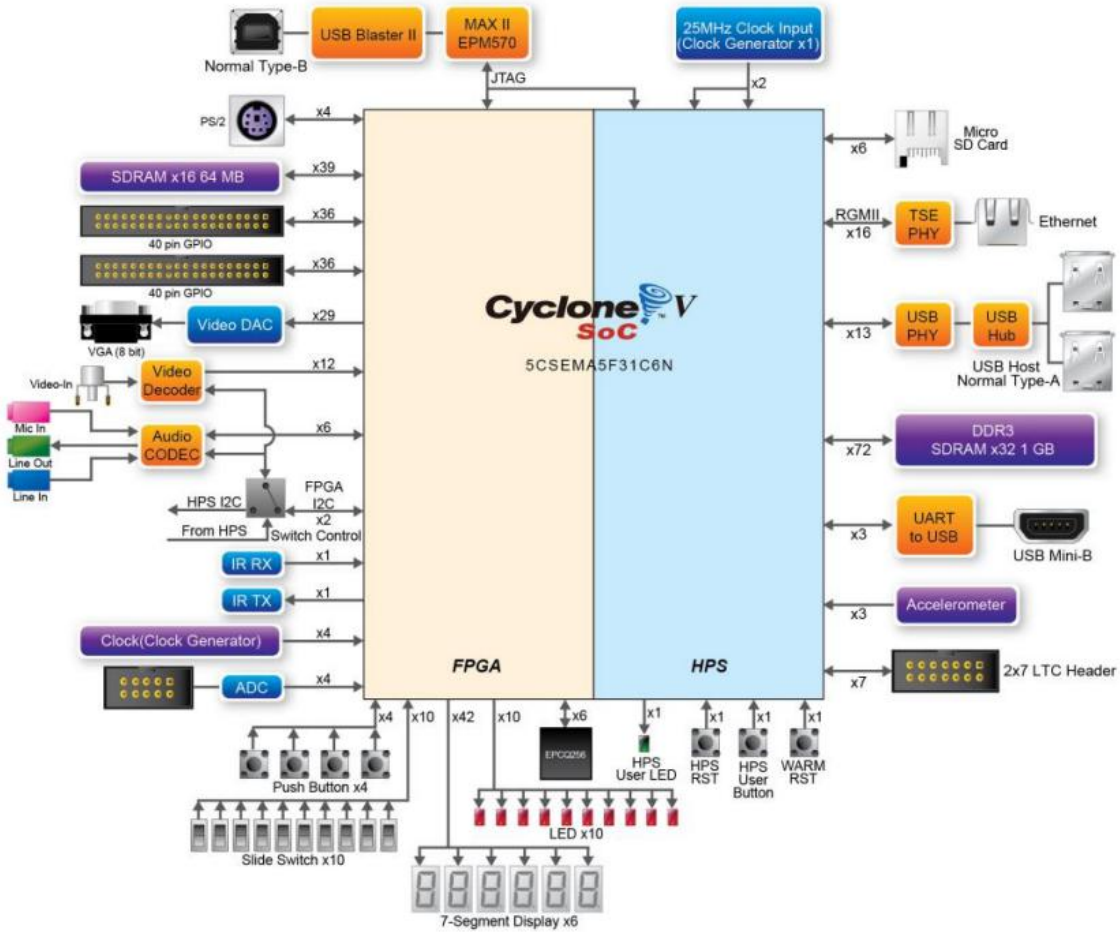


FIGURE 1.1 – Schéma bloc de la carte Cyclone V : DE1 - SoC[6]

On ne listera que les composants utiles à notre étude :

— HPS
  - CPU : ARM Cortex™-A9 MPCore™ Processor Dual-Core 800 MHz.
  - 1 Go DDR3 SDRAM.
  - Lecteur micro-SD.
  - Port UART-USB.
  - Port Ethernet.
  - Ports USB.

— FPGA :

$$50\ MHz \times 4$$

La carte micro-SD possède une image de l'OS à installer. On choisit la distribution Linux fournit par *Intel®/Altera* qui contient les outils *Opencl* nécessaire pour les différentes procédures.

Chapitre 2

Traitement sur ARM

Introduction

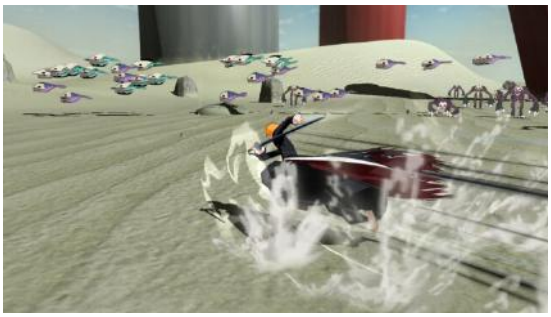
Dans cette partie, On procède à l'application de l'opérateur Sobel sur un ensemble d'image de différentes résolutions. Les traitements sont effectuées entièrement sur le *CPU ARM* présent sur la carte. Cette solution est purement logiciel, on écrit alors un programme en langage *C++* compilé pour cette architecture.

2.1 Procédure de traitement

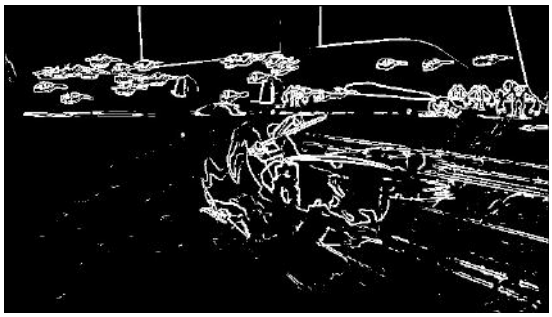
Les fichiers images qui sont traités sont au format *PPM*. On charge alors un tableau d'entiers alloué de taille $Largeur \times Hauteur$ pour chaque image. Il faut alors, dans un premier temps, extraire les valeurs R, G, B et les convertir en valeur grise (i.e. calcul de la moyenne des canaux), Puis, calculer la nouvelle valeur du pixel en appliquant les matrices des noyaux G_x et G_y. Enfin, un seuillage est appliqué pour obtenir le contour. Le traitement se fait donc de manière séquentiel.

2.2 Résultats visuels

Après compilation et exécution du programme sur le *CPU ARM*, on obtient les résultats suivants :

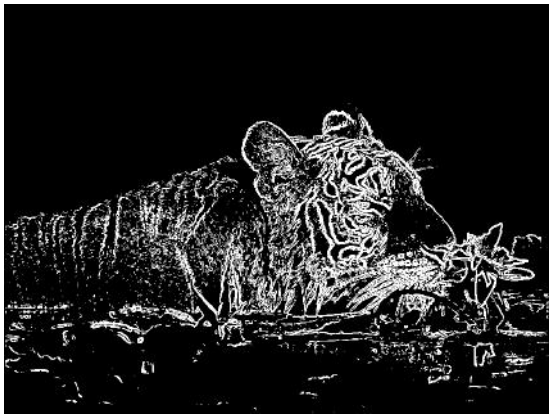


(a) Résolution 640×360



(b) image traitée

(c) Résolution 800×600

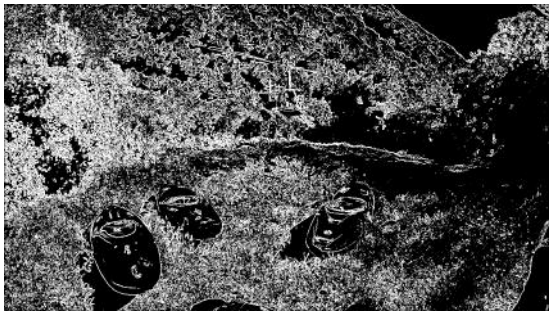


(d) image traitée

(e) Résolution 1280×760

(f) image traitée

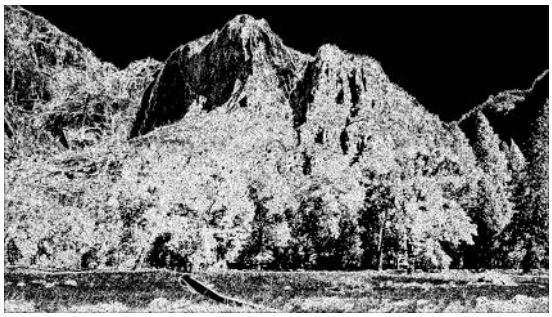
(g) Résolution 1366×768



(h) image traitée

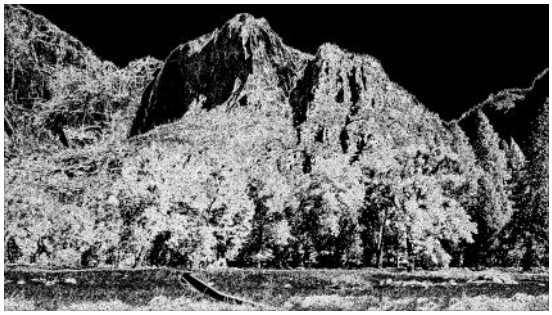
FIGURE 2.1 – Images traitées sur ARM : 1-4

(a) Résolution 1920×1080



(b) image traitée

(c) Résolution 2560×1440

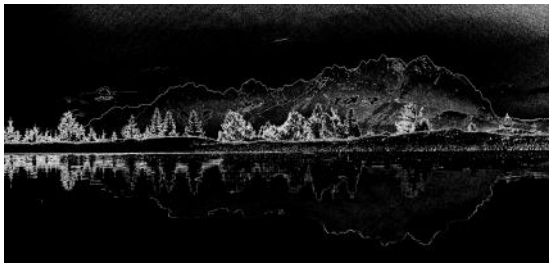


(d) image traitée

(e) Résolution 3840×2160

(f) image traitée

(g) Résolution 6000×2848



(h) image traitée

(i) Résolution 6000×2848

(j) image traitée

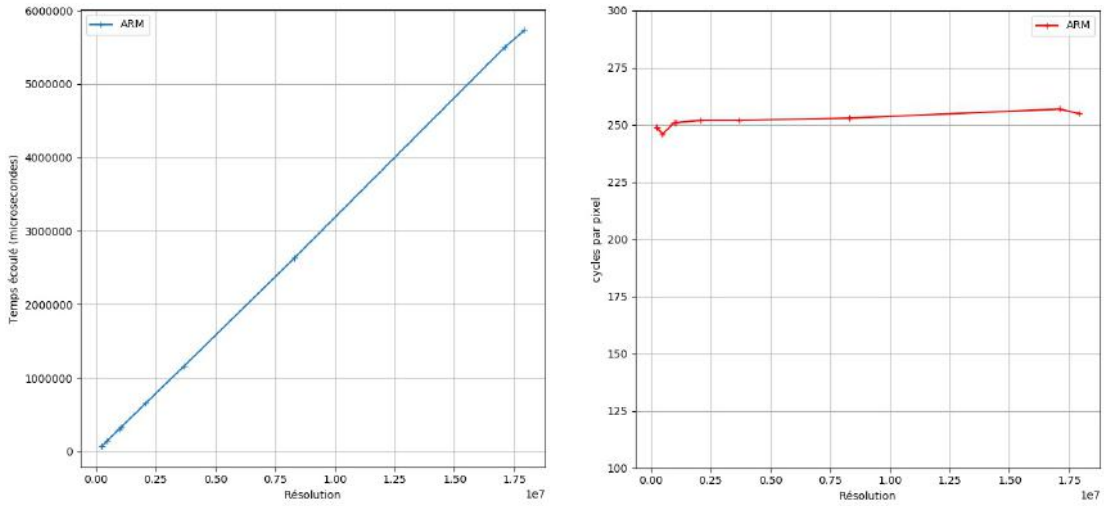
FIGURE 2.2 – Images traitées sur ARM : 5-9

2.3 Analyse des performances

Afin de analyser les performances, on doit mesurer le temps d'exécution de l'algorithme sur toute l'image. On choisit une résolution de l'ordre de la μs afin d'avoir la précision nécessaire pour l'évaluation des performances.
On évalue aussi le nombre de cycles par pixel effectués par le *CPU* :

$$cpp = \frac{t_{exécution} \times fréquence}{Hauteur \times Largeur}$$

Avec le *CPU ARM* qui est cadencé à $800MHz$, on obtient les résultat suivant.



(a) Évolution du temps de traitement (b) Mesures du nombre cycles par pixel

FIGURE 2.3 – Évolution des performances en fonction du nombre de pixels traités

On retrouve bien en Figure 2.3a que l'algorithme est en temps linéaire par rapport au nombre de pixels en entrée. On constate aussi que le nombre de cycles par pixel est quasi-constant, valant une moyenne de 253.7 cycles par pixel.

L'image de résolution maximale que peut traiter le *CPU*, est en théorie une image qui remplirait tout l'espace que possède la *SDRAM* : 1 GO [6]. Il est bien-sûr impossible d'avoir une telle image en mémoire pour 2 raisons : il faut un certain espace mémoire pour le système d'exploitation et les autres processus ; Mais aussi, il faut pouvoir allouer un segment contigu d'un milliard d'octets. Ce cas de figure est très peut probable.

Conclusion

Le traitement sur *CPU ARM* est effectué de manière correcte. Ce chapitre nous a permis de poser les bases de l'évaluation des performances d'une architecture. Il faut donc comparer ces résultats à ceux d'une architecture différente que l'on présente dans le chapitre suivant.

Chapitre 3

Traitement CPU + FPGA avec BSP

Introduction

La solution présentée dans le chapitre précédant repose sur l'usage des mécanismes du *CPU* afin de traiter les images en entrée. Dans ce chapitre, on présente une architecture hétérogène *CPU + FPGA*.

3.1 Mise en place de l'architecture

Dans ce schéma, Le *CPU* charge en mémoire vive une image puis l'envoie au *FPGA*. Le *FPGA* effectue le traitement sur les pixels puis, à la fin des calculs, renvoie l'image traitée au *CPU*.

On conçoit donc 2 programmes *C++* :

— *Host* : s'occupera de l'initialisation/reconfiguration du *FPGA* puis de l'acquisition des images et leurs envois au *FPGA*.

— *Device* : sera traduit en code *HDL* puis synthétisé sur le *FPGA*.

On utilise les outils *Intel SDK for OpenCL* afin de générer un code pour le composant *FPGA* à partir d'un noyau (*Device*). Le programme peut générer et synthétiser les structures correspondantes au code *C++*.

Une fois la génération du code et la compilation finies, on peut téléverser le programme et le noyau exécutable sur la carte. À l'exécution du programme, l'hôte va configurer le *FPGA* et le préparer pour le traitement des données.

On communique alors les pixels de l'image du *CPU* vers *FPGA* en les écrivant dans le *Buffer*. On lance le calcul demandé. Puis, une fois terminé on peut récupérer les pixels traités en lisant le *Buffer*.

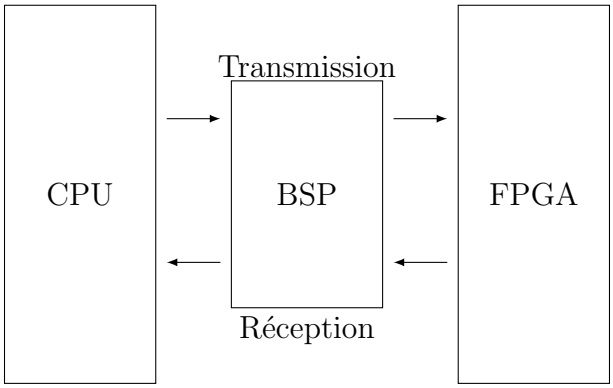


FIGURE 3.1 – Schéma de communication $CPU \rightleftarrows FPGA$

Cette facilité de communication entre *CPU* et *FPGA* est obtenue grâce au *BSP (Board Support Package)* utilisé par *OpenCl* sur la carte. Les *BSP* sont des interfaces propres à chaque carte placés entre le *CPU* et le *FPGA*. Ils permettent d'exploiter les périphériques présents et d'assurer la communication entre ces composants [7].

3.2 Synthèse en HDL

3.2.1 AOC

L'outil *AOC* génère permet de générer un noyau exécutable à partir du code *Cl* fourni. Ceci est fait en synthétisant toute la structure nécessaire pour implémenter le filtre en interprétant les différentes parties du code *C* en *HDL*(Verilog/VHDL). Une fois l'architecture matérielle générée, *AOC* procède à la génération du code machine qui pourra être mis sur le *FPGA* par le *Host* [8].

3.2.2 Déroulage de boule

En général, un compilateur vérifie si une boucle peut être déroulée, c-à-d, au-lieu de générer des instructions à exécuter en une séquence répétée sur la dite boucle, le compilateur "aplatit" la boucle. Cette boucle devient alors une plus longue séquence d'instructions mais elle permet de générer des structures sous forme de blocs. On peut indiquer au compilateur d'essayer de dérouler une boucle grâce au macro *#pragma unroll*. Ce pendant la boucle principale (qui parcourt tous les pixels) ne doit pas être déroulée car elle servira de générateur de structures combinatoires.

3.2.3 Génération en blocs

Pendant la synthèse, Le compilateur analyse les dimensions d'une boucle et, si celle-ci est finie et déterministe, essaie de générer autant de structures en forme de blocs que possible. Ces blocs profitent de la localité spatiale lors du traitement de Sobel. Effectivement, si on traite le pixel $p(i, j)$, on traitera automatiquement le pixel adjacent lors de l'itération suivante. L'idée est donc de simplement décaler tout les pixels du *buffer* jusqu'à la fin de ligne courante.

3.2.4 Chargement des lignes

Le *buffer* contenant toute l'image linéarisé doit pouvoir envoyer 2 lignes + 3 pixels à l'initialisation. Ceci est dû au fait que le premier pixel que l'on peut traiter est $p(2, 2)$ (2ème ligne, 2ème colonne). Mais, il faut aussi avoir tout les pixels 8-adjacents. Il faut donc ajouter 1 ligne + 1 pixel :

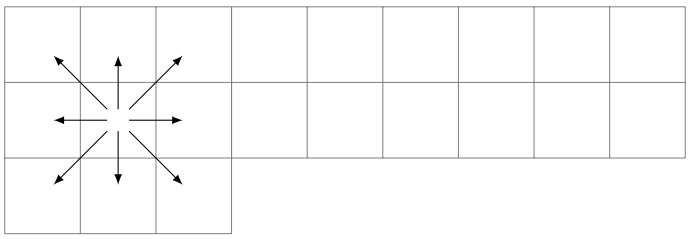
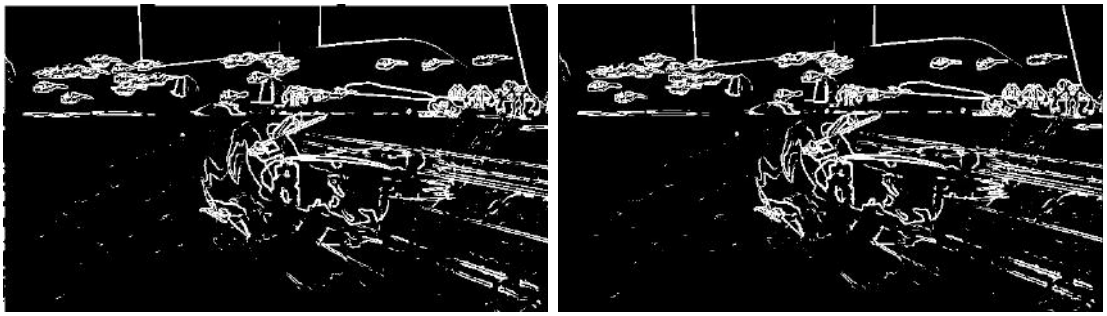


FIGURE 3.2 – Représentation de la lecture du premier pixel

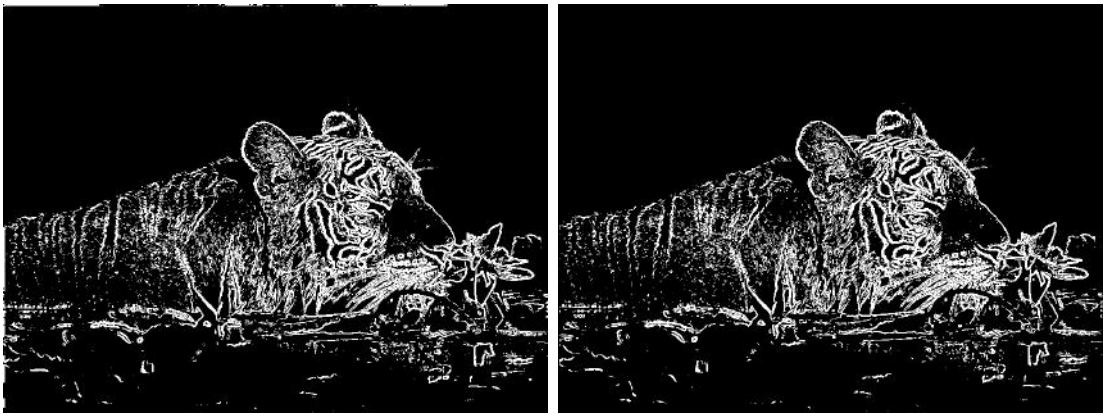
C'est grâce à ces mécanismes que l'on obtient plusieurs structures parallélisées et en pipeline. On peut alors procéder aux tests et mesures de performances.

3.3 Résultats visuels

On obtient les résultats suivants :

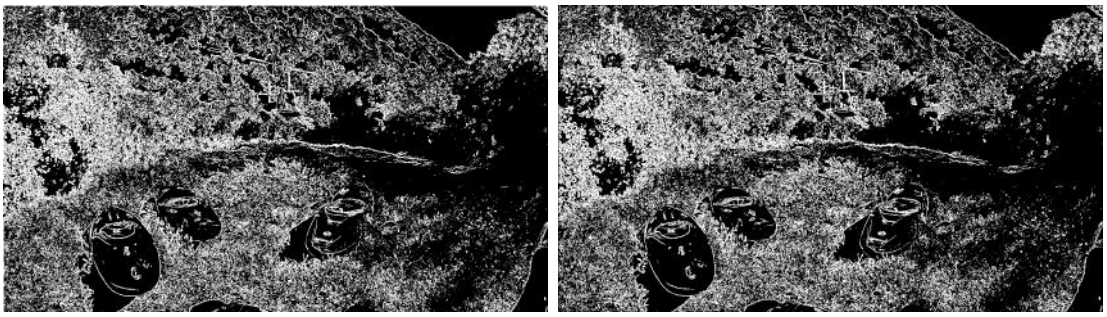


(a) Résolution 640×360 - ARM + FPGA (b) image traitée - ARM



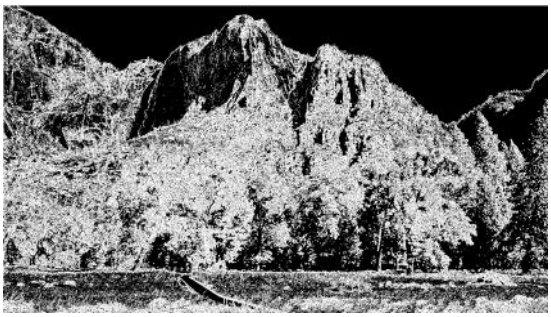
(c) Résolution 800×600 - ARM + FPGA (d) image traitée - ARM

(e) Résolution 1280×760 - ARM + FPGA (f) image traitée - ARM

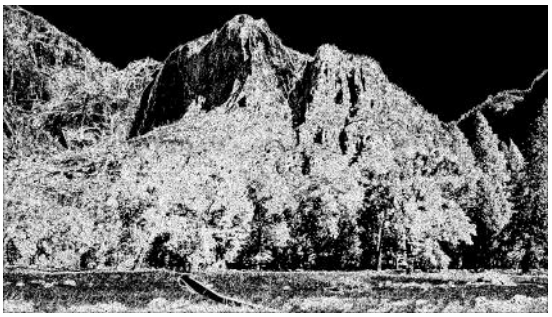


(g) Résolution 1366×768 - ARM + FPGA (h) image traitée - ARM

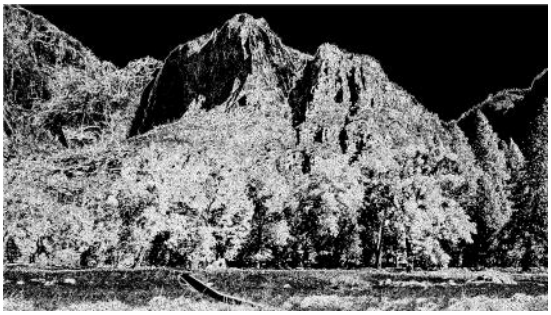
FIGURE 3.3 – Images traitées sur ARM : 1-4



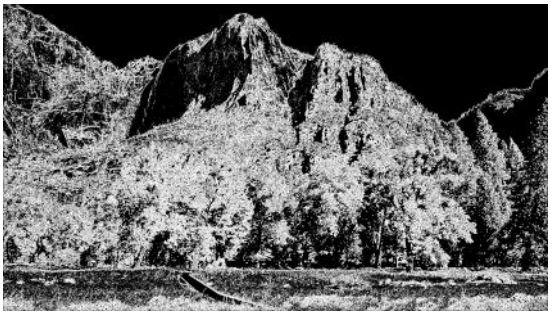
(a) Résolution 1920×1080 - ARM + FPGA



(b) image traitée - ARM



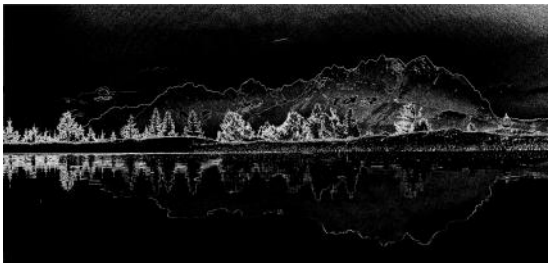
(c) Résolution 2560×1440 - ARM + FPGA



(d) image traitée - ARM

(e) Résolution 3840×2160 - ARM + FPGA

(f) image traitée - ARM



(g) Résolution 6000×2848 - ARM + FPGA

(h) image traitée - ARM

(i) Résolution 6000×2848 - ARM + FPGA

(j) image traitée - ARM

FIGURE 3.4 – Images traitées sur ARM : 5-9

On remarque qu'il n'y a de différences que sur les bords des images. Ceci est dû au fait que les deux structures traitent ces pixels spéciaux soit par une compensation grâce à un étalement du pixel lui même, soit un arrondi vers 0.

p	p	p
p	p	
p		

(a) Compensation de valeur - ARM+FPGA

0	0	0
0	p	
0		

(b) Arrondi vers 0 - ARM

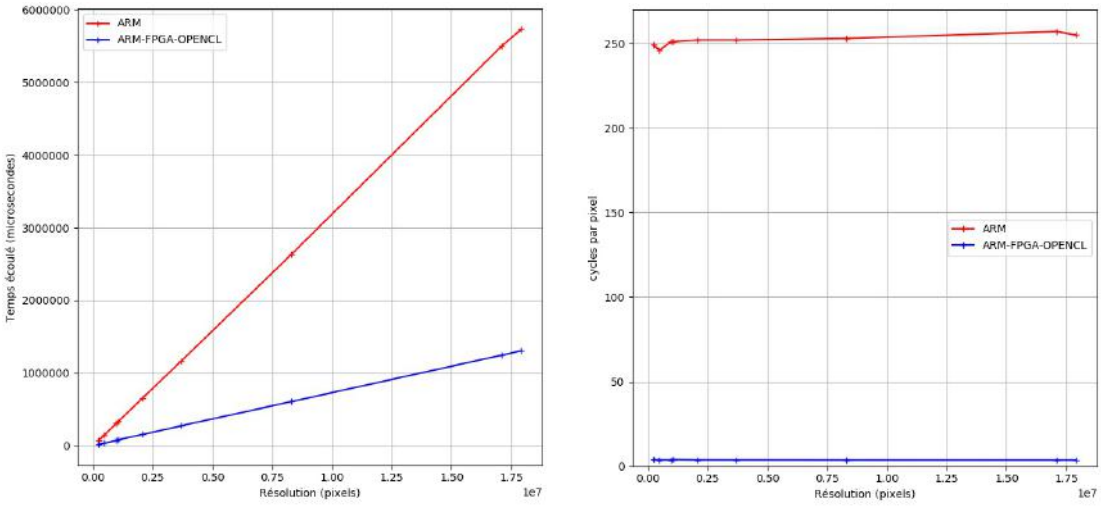
FIGURE 3.5 – Traitement du pixel du bord en (0,0)

Pour les autres régions, il n'y a pas de différence visuelle. Il faut alors examiner les performances de chacune des architectures.

3.4 Analyse des performances

3.4.1 Mesures temporelles

Dans un premier temps, on reprend les mêmes critères de mesure vu en 2.3. Bien-sûr, le *FPGA* est cadencé à 50 MHz :



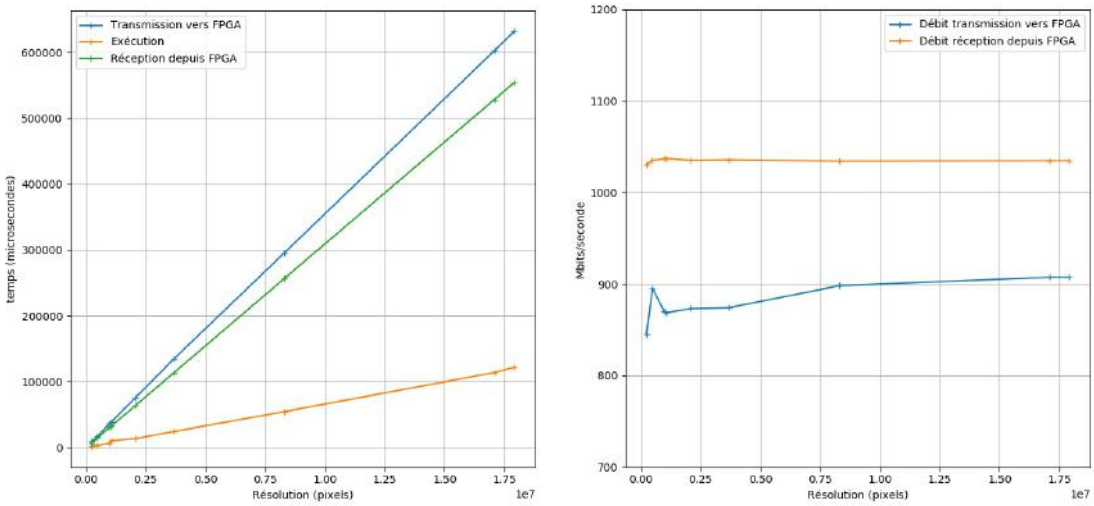
(a) Évolution du temps de traitement (b) Mesures du nombre cycles par pixel

FIGURE 3.6 – Évolution des performances en fonction du nombre de pixels traités

On observe de bien meilleures performances. Notamment, le *cpp* qui est très faible pour le traitement *ARM-FPGA* ($\approx$ 3 cycles/pixel) alors que le *cpp ARM* est autour de 250 cycles/pixel.

3.4.2 Mesures des débits

C'est résultat sont prometteurs mais insuffisants. Il faut aussi examiner les temps d'échange entre *CPU* et *FPGA* et les débits associés :



(a) Temps de communication ARM ↔ FPGA

(b) Débits d'échange ARM ↔ FPGA

FIGURE 3.7 – Évolution des échanges ARM ↔ FPGA en fonction du nombre de pixels

On remarque que les temps de transmission et de réception sont beaucoup plus importants que le temps du traitement lui même. Ceci peut indiquer qu'il y a un encombrement au niveau du bus.

On a priori aucune idée sur l'intégrité de la transmission mais on peut vérifier qu'elle est correcte si le résultat est identique à celui fait par *CPU*.

On remarque aussi que le débit $FPGA \rightarrow CPU$ est plus important que le débit $CPU \rightarrow FPGA$. C'est lié au fait que le *FPGA* n'a que ses données à traiter et envoyer. Alors que le *CPU* prend en charge le traitement, l'envoi, l'*OS* et toutes les processus utilisés tel que l'*UDHCP*.

3.4.3 Mesures structurelles

L'implémentation de chaque architecture associée à chaque résolution induit une différence en ressource utilisée sur le *FPGA*. Le traitement étant le même, on n'a aucun changement sur le nombre de registres, ni la logique utilisée comme montré en Figure 3.8.

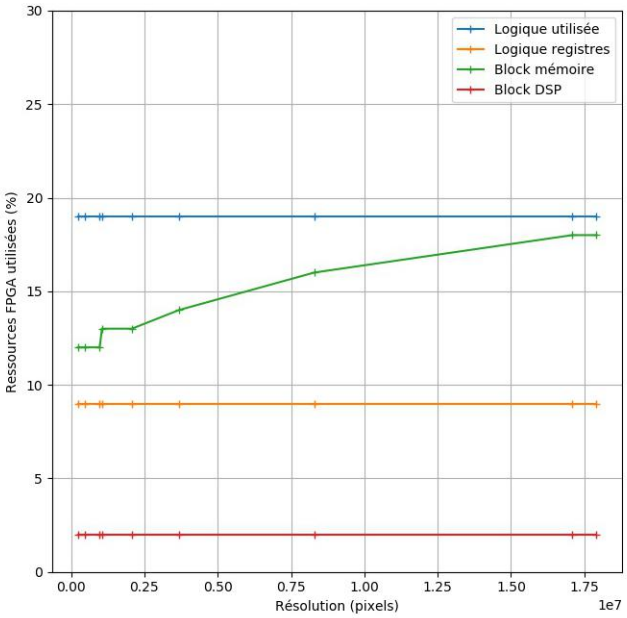


FIGURE 3.8 – Évolution des ressources *FPGA* utilisé en fonction du nombre de pixels

Par contre, on remarque que le nombre de blocs mémoire utilisés augmente linéairement avec le nombre de pixels à traiter. Donc il y a bien une limite matérielle à la dimension de l'image que l'architecture *CPU+FPGA* peut traiter.

Cette limite peut être déterminée de plusieurs manières : connaissance exacte du fonctionnement du *BSP* [7], test empirique, examen des différentes structures mémoire de la carte...

Par exemple, on peut tester plusieurs résolutions en fonction de la taille non-compressée de l'image et voir si le noyau peu être compilé pour cette résolution. Ici, on obtient ce résultat sur ces mêmes images *.ppm* :

Largeur	Hauteur	Taille (Mo)	Acceptée
640	360	0.67	oui
800	600	1.4	oui
1280	760	2.8	oui
1366	768	3.0	oui
1920	1080	6.0	oui
2560	1440	10.8	oui
3840	2160	24.3	oui
6000	2848	50.0	oui
5184	3456	52.4	oui
8272	6200	150.2	non
10000	7000	210	non

TABLE 3.1 – Résolutions et tailles des images testées pour le traitement

On a alors une indication sur une certaine capacité mémoire autour de 64 Mo. Cette quantité correspond à la *SDRAM* de 64 Mo du coté du *FPGA* [6].

Conclusion

L'usage du *CPU+FPGA* améliore considérablement les performances comparé au *CPU* seul. Mais ce n'est pas sans inconvénients. Il est impossible de traiter différentes résolutions sans refaire la synthèse sur *FPGA* en reprogrammant le noyau, là où le *CPU* peut traiter dynamiquement plusieurs résolutions. L'usage des deux noyaux implique aussi une communication entre les deux et il y a donc un risque d'encombrement du bus de communication. Il faut noter aussi que l'usage de ressources du *FPGA* pour le traitement d'image limite les ressources restantes pour les d'autres applications sur ce même *FPGA*. Dans le prochain chapitre, on traitera de la vidéo sur ces deux mêmes architectures.

Chapitre 4

Traitement vidéo

Introduction

Une fois le traitement d'image effectuée, on passe au traitement vidéo. Dans ce chapitre, on applique l'opérateur Sobel sur une vidéo sur chacune des architectures vues précédemment.

4.1 Procédure de traitement

4.1.1 Principe d'une vidéo

Une vidéo consiste en une séquence d'images où chacune est appelée *frame*. Le "mouvement" est réalisé par l'émission rapide de ses frames. On dit alors que l'images est cadencée à un certain de nombre de *fps (frames per second)*. Tous les frames sont composés du même nombre de pixel (soit la même résolution). Donc afin d'exécuter l'opérateur de Sobel sur une vidéo, il faut extraire chaque frame, le traiter puis l'enregistrer.

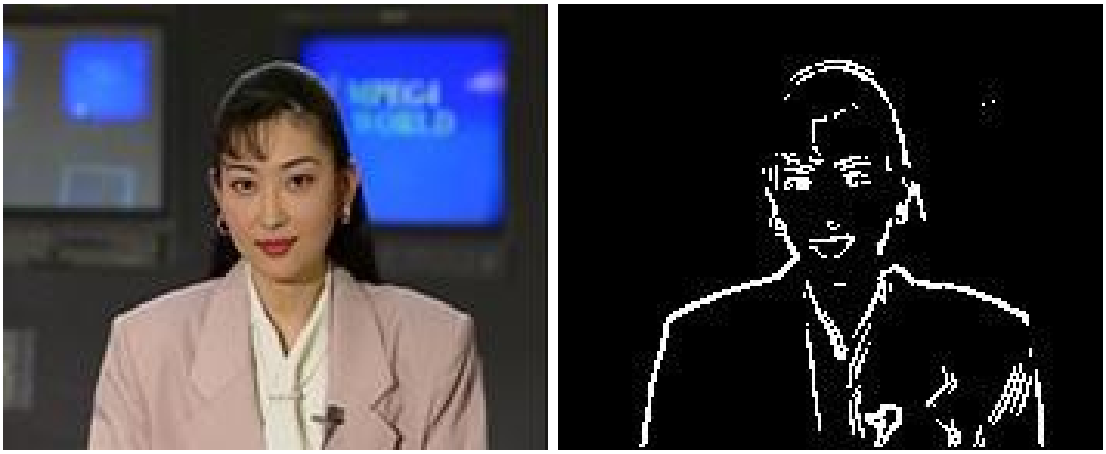
4.1.2 Format YUV

Ici, les fichiers vidéos à traiter sont au format *YUV*[9]. Y représente la composante de luminance (intensité du niveau gris). U et V représentent la valeur de la chrominance associée au pixel. Les données d'images YUV sont stockées dans un tableau qui contient les composantes Y, U et V à tour de rôle où chaque élément est codé sur 8 bits (type *unsigned char*).

Pour l'encodage au format 4 : 2 : 0, on échantillonne toutes les composantes Y (luminance) mais une ligne et une colonne sur deux des composantes U et V (chrominances). Pour l'opérateur Sobel, on doit prendre en compte uniquement la composante Y (luminance).

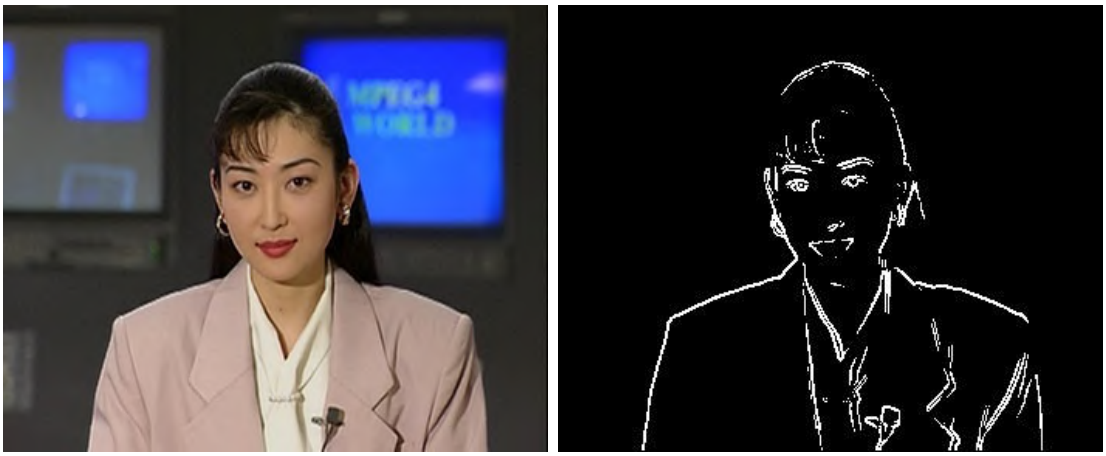
4.2 Résultats visuels

Comme vu précédemment, le résultat entre le traitement sur les deux architectures est identique.



(a) Résolution 176×144 (b) Vidéo traitée

FIGURE 4.1 – Images des vidéos traitées : 1



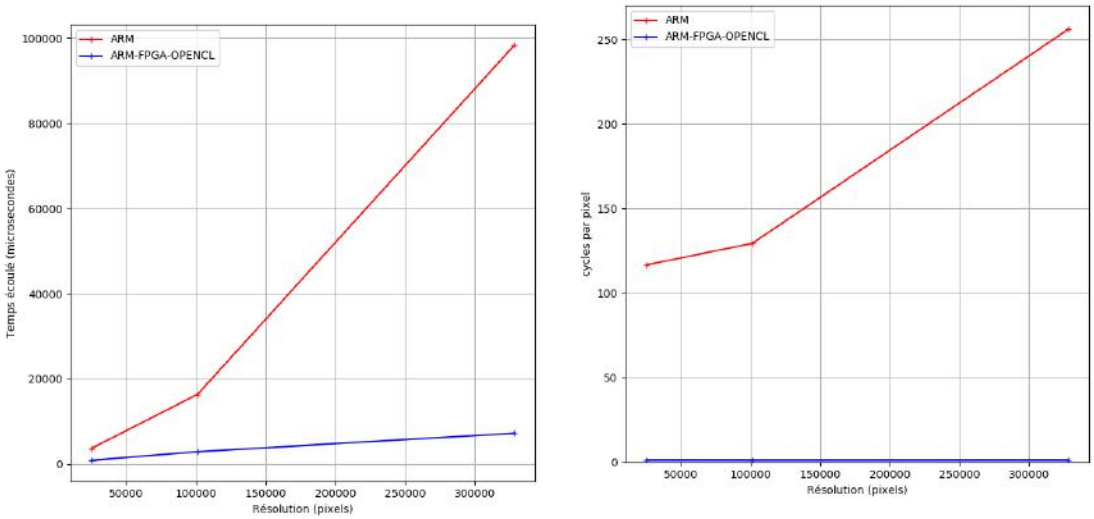
(a) Résolution 352×288 (b) image traitée

(c) Résolution 640×480 (d) image traitée

FIGURE 4.2 – Images des vidéos traitées : 2 - 3

4.3 Performances

On mesure les performances en se basant sur les mêmes critères :



(a) Évolution du temps de traitement (b) Évolution du Cpp

FIGURE 4.3 – Évolution des performances fonction du nombre de pixels

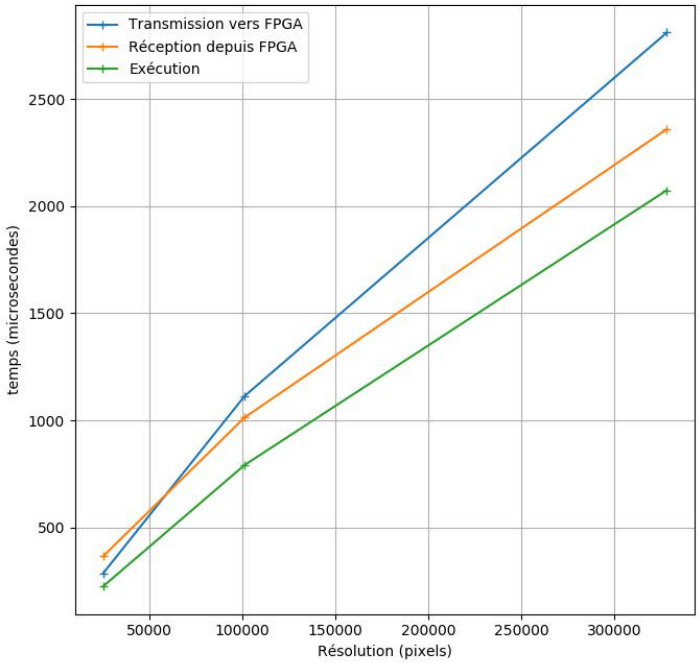


FIGURE 4.4 – Évolution des temps en fonction du nombre de pixels

On retrouve les même résultats que pour des images uniques. Ceci explique bien le mécanisme de succession d'images pendant la lecture d'une vidéo. On peut alors constaté une certaine latence entrée l'entrée du flux d'images/caméra et la sortie/l'affichage. cette latence doit être inférieur au temps nécessaire pour la capture et l'envoi d'un image depuis la caméra :

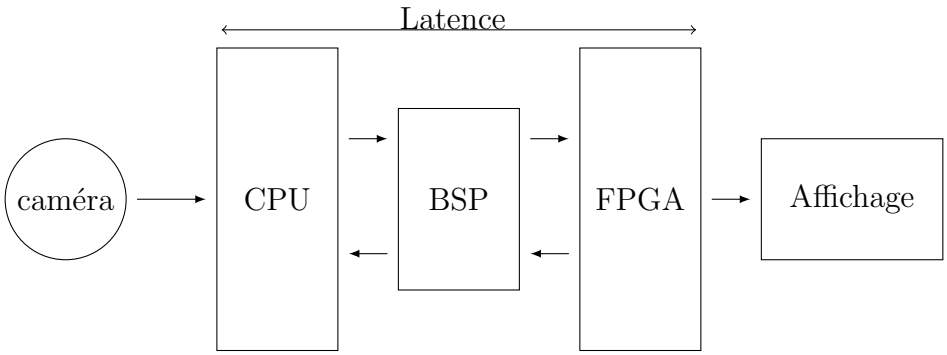


FIGURE 4.5 – Schéma d'acquisition - traitement - affichage de vidéo

La caméra émettra un certain nombre d'images à la second (*fps*). La latence reste relativement acceptable mais très fortement corrélé à la résolution de la vidéo. Elle dépend beaucoup de l'application pour laquelle l'architecture est déployée. Il faut donc que le système ait une latence telle que :

$$Latence < \frac{1}{fps}$$

Si on prend une vidéo cadencée à 30 *fps*, la résolution maximale acceptable est d'environ 30 ms.

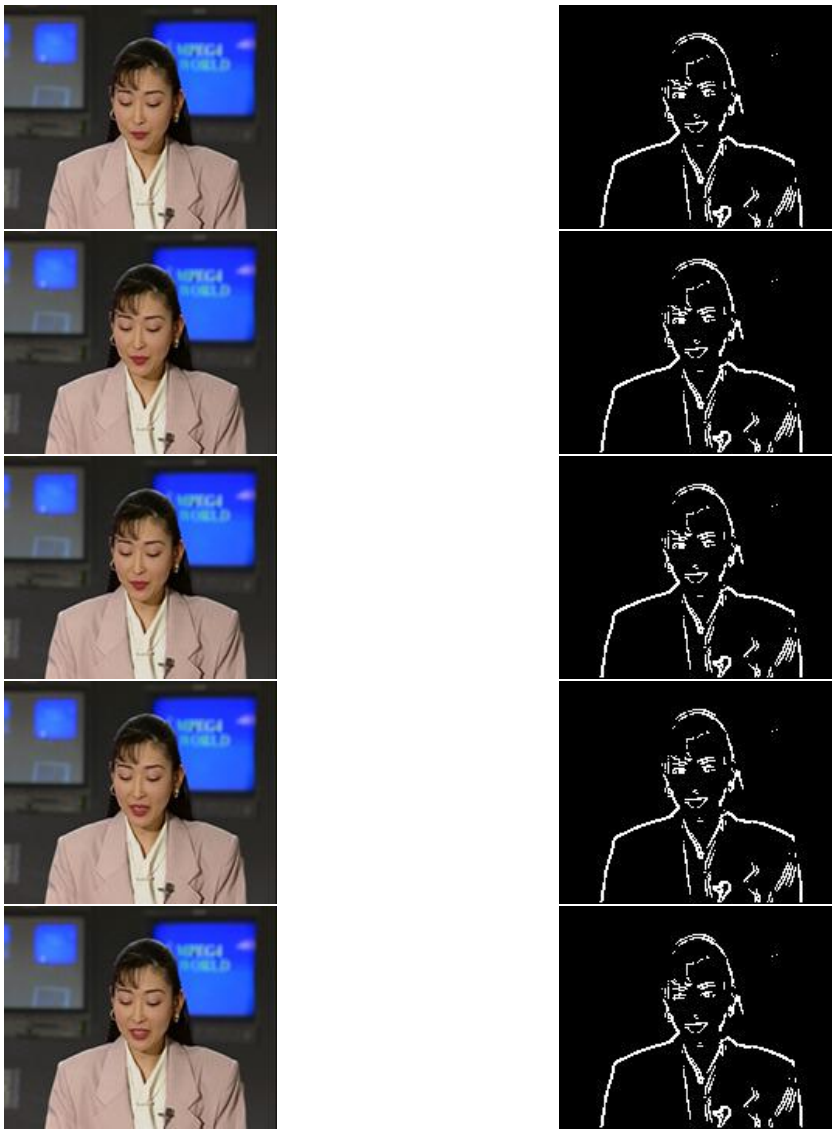


FIGURE 4.6 – Séquence d'images de la vidéo.

Conclusion

Il n'y a à priori pas de différence entre le traitement d'une image et une séquence d'image. Mais, là où le traitement d'image peut parfois être retardé, le traitement de vidéo est plus restreint au niveau des temps critique. Si par exemple, on considère un algorithme de détection d'objet par analyse d'images pour piloter une voiture, une latence de l'ordre de la seconde est inacceptable dans ce cas vu la vitesse possible du véhicule.

Chapitre 5

Traitements parallèles

5.1 Étude de la parallélisation

5.1.1 Possibilité

Dans le chapitre 2, on a implémenté l'algorithme désiré pour le *CPU* qui exécute les instructions de manière séquentiel, tant dit qu'en 3.2, la structure synthétisée était parallélisée. Idéalement, on pourrait paralléliser le calcul du *CPU* sur plusieurs cœurs où chacun travaillerait sur une sous-image.

5.1.2 Critères de parallélisation

Cet algorithme est bien parallélisable car il peut appartenir à la famille *Devide and Conquer*[1] car :

- Il peut être décomposé en segments discrets pour une exécution simultanée.
- Les différents segments discrets peuvent être exécutés ç tout moment.
- Le temps d'utiliser plusieurs ressources de calcul doit être inférieur à celui d'utiliser une seule ressource.
- Il est exécutable sur le système qui est multi-processeurs.

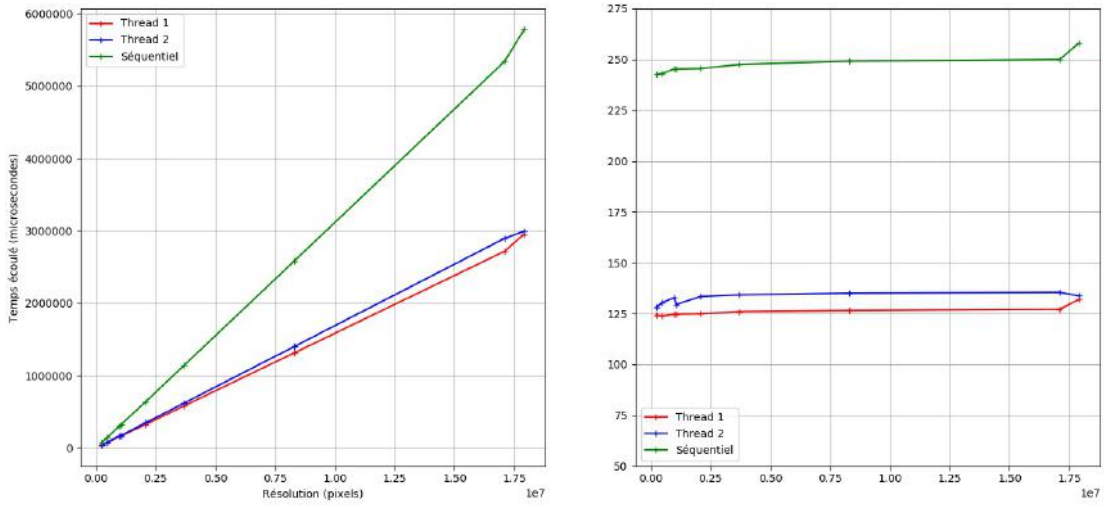
5.2 Mise en place de la parallélisation

La parallélisation consiste en trois grande étapes fondamentales [1] :

1. Partitionnement Il faut réécrire l'algorithme séquentiel en 2 parties de taille égale. On attribut donc à chacune des deux threads une moitié de l'image. On lance l'exécution.
2. Communication Ici il n'y a aucun besoin de communiquer entrée les *threads* car leurs taches respectives ne se chevauchent pas. les *threads* sont donc encapsulées.
3. Synchronisation En fin de calcul, les deux *threads* se rejoignent pour reformer l'image dans sa totalité. Dans ce cas, il n'y a pas aucune dépendance de données car chaque *thread* à un accès mémoire sur une partie restreinte de l'image.

5.3 Résultats

En compilant le code pour l'exécution de deux *threads* en parallèle sur chaque cœur, on obtient les résultats en figure 5.1. On a bien un temps d'exécution et un *cpp* deux fois moins important sur chaque *threads* par rapport à l'exécution séquentielle. Même si en général, les programmes possèdent une partie qui ne peut pas être accélérée [10]. On remarque aussi que les performances des deux *threads* ne sont pas identiques. Ceci est expliqué par un effet de non équilibrage des charges mais aussi il se peut qu'un *thread* a eu plus de défauts de cache qu'un autre. Certains pics sur le graphique sont expliqués par le fait que le *CPU* gère aussi l'OS et les autres taches en même temps que l'exécution du programme.



(a) Évolution du temps de traitement (b) Évolution du Cpp

FIGURE 5.1 – Évolution des performances fonction du nombre de pixels

Idéalement, il faudrait mettre en place une architecture qui puisse exploiter les deux cœurs *CPU* en même temps que deux noyaux *FPGA*.

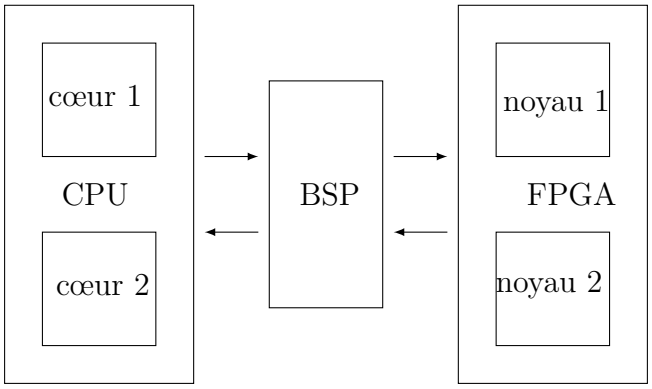


FIGURE 5.2 – Schéma de traitement : 2 cœurs - 2 noyaux

Cependant, un problème persiste. On ne peut envoyer que une seule donnée en même temps sur les bus pour les échanges $CPU \rightleftarrows FPGA$. Il faut alors séquencer et l'envoi et la réception et en même temps traiter les images pendant l'envoi.

1. Envoi : *cœur* 1 → *noyeau* 1.
2. Traitement sous image 1 + Envoi : *cœur* 2 → *noyeau* 2.
3. Traitement sous image 2 + Envoi : *noyau* 1 → *cœur* 1.
4. Envoi : *noyau* 2 → *cœur* 2.

On utilise la même voie pour envoyer/recevoir plusieurs données de plusieurs sources/destinations. La voie est alors multiplexée.

Conclusion

La parallélisation a pu être effectuée avec succès. Les résultats sont meilleurs que ce que la théorie a prédit à cause de l'encapsulation correcte des *threads*. On peut déjà envisager une architecture 2 cœurs *CPU* - 2 noyaux *FPGA*.

Conclusion générale et perspectives

Conclusion

Durant ce projet, nous avons abordé des notions concernant les architectures hétérogènes et les cœurs qui peuvent être employés dans ces architectures afin d'améliorer les performances d'un algorithme de traitement d'images.

Nous avons observé que le traitement sur *CPU+FPGA* était bien plus performant qu'un traitement *CPU* seul. Ce même constat a pu être entendu sur le traitement vidéo qui lui doit satisfaire une contrainte supplémentaire : avoir une latence sous un certain seuil fixer par le nombre d'image à la secondes (*fps*) que la caméra peut capter.

Par ailleurs, nous avons mis en place un traitement parallèle sur *CPU* et avons synthétisé une structure *FPGA* parallélisée et en pipeline grâce à *AOC*.

Il est important de noter que, durant la conception des différentes architectures, il y a plusieurs compromis : la différence en fréquence notable des différents cœurs/noyaux (jusqu'à 10 fois plus), les différentes mémoires et leurs gestions/accès, les différents schémas et possibilités de parallélisation pour *CPU* et/ou *FPGA*,...

Les architectures hétérogènes sont alors des conceptions plus évoluées et plus complexes qui doivent être adaptées aux contraintes imposées par l'application ciblée. Ces architectures sont de plus en plus présentes et cette tendance ne semble pas régresser.
Ce projet nous a permis de découvrir les aspects importants entourant ces architectures et d'avoir une meilleure compréhension des mécanismes d'échange et de communication entre les différents types de noyaux.

Perspectives

Nous prévoyons comme perspectives :

- Implémentation d'une liaison *VGA* du sur *FPGA* pour l'affichage des images.
- Implémentation d'une interface *USB* sur *ARM* pour la lecture d'un flux de caméra *USB*.
- Mise en place de plusieurs noyaux parallèles sur *FPGA*.
- Étude du fonctionnement du *BSP* de la carte.
- Comparaison entre une architecture *CPU+FPGA* et *CPU+GPU*.

Bibliographie

[1] DOU, GHAOUI, ZHANG. Évaluation d'une architecture SoC (ARM-FPGA) pour des applications de traitement d'images - Compte rendu bibliographique, Université Paris Sud, 2019.

[2] Jef Poskanzer. Ppm standard. http://netpbm.sourceforge.net/doc/ppm.html.

[3] Opencv dev team. OpenCV documentation. https://docs.opencv.org/2.4/doc/tutorials/introduction/load_save_image/load_save_image.html.

[4] Irwin Sobel. An Isotropic 3x3 Image Gradient Operator.

[5] Michael J Flynn. *Computer architecture : pipelined and parallel processor design.* MA : Jones and Bartlett.

[6] Intel/Altera. DE1-SoC Board manual. http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/DE1-SoC_User_manual.pdf.

[7] Terasic. BSP (Board Support Package) for Altera SDK OpenCL 14.0. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836&PartNo=4.

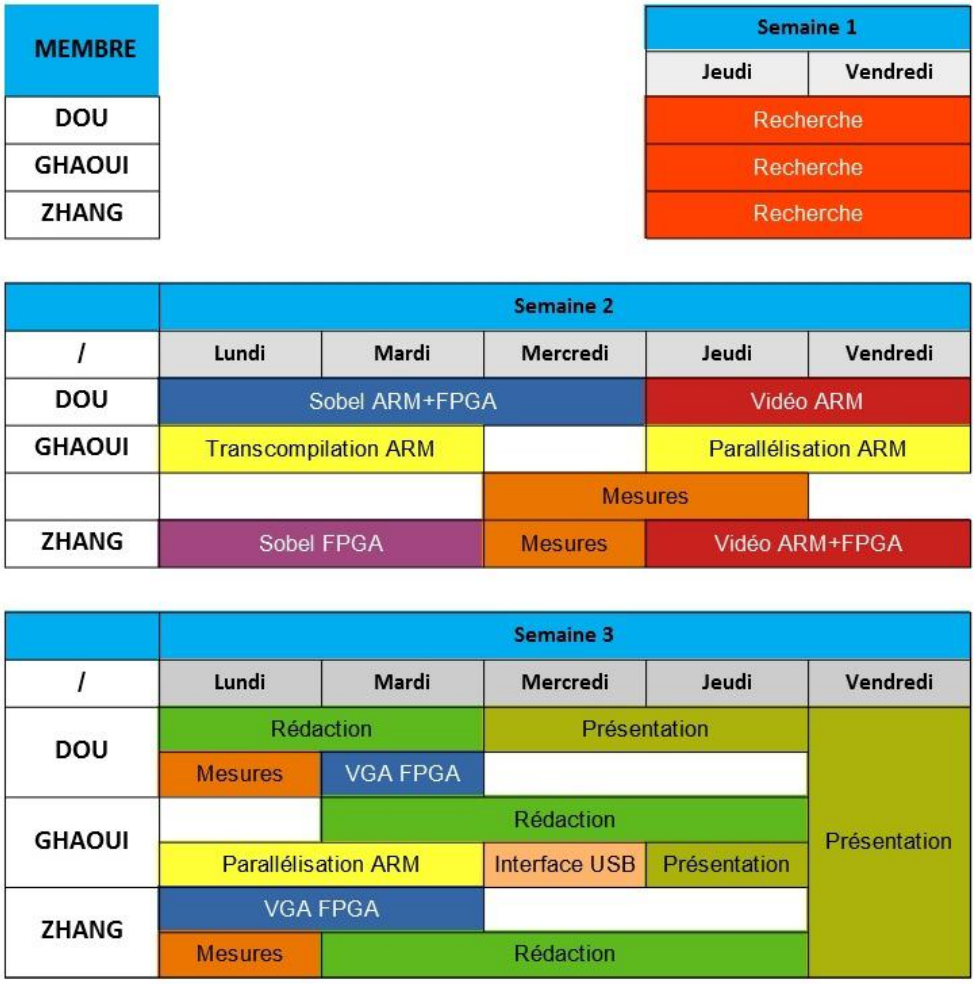
[8] Intel®. Intel® FPGA SDK for OpenCL™ Pro Edition Programming Guide . https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

[9] Dave Wilson. Yuv formats. http://www.fourcc.org/yuv.php.

[10] Joel F. Klein. Amdahl's law. https://commons.wikimedia.org/wiki/Category:Amdahl's_law.

Annexe

Diagramme de Gantt du déroulement du projet



Les codes sources ayant servis à la conception de l'évaluation des performances sont sur : `https://github.com/anisghaoui/SOBEL-DE1-SOC`.