Rapport Projet - Détection de points d'intérêt sur FPGA

Samory DIABY
Cédrine SOCQUET

Table des matières

1 Plan de validation **2**
1.1 Analyses . . . 2
1.2 Démonstrations . . . 2

2 Lecture d'image à partir d'un fichier texte **3**
2.1 Principe . . . 3
2.2 Résultats de simulations - Analyse . . . 3
2.3 Résultats finaux - Démonstration . . . 4

3 Fenêtre glissante **5**
3.1 Principe . . . 5
3.1.1 Schéma RTL . . . 8
3.2 Résultats de simulations - Analyse . . . 9
3.3 Résultats finaux - Démonstration . . . 12

4 Filtres de Sobel et détection de points d'intérêt **13**
4.1 Principe . . . 13
4.1.1 Sobel x (horizontal ?) . . . 14
4.1.2 Sobel Y (vertical ?) . . . 14
4.1.3 Détection de point d'intérêt . . . 15
4.1.4 Schéma RTL . . . 15
4.2 Résultats de simulations - Analyse . . . 16
4.2.1 Analyses de timing . . . 17
4.3 Résultats finaux - Démonstration . . . 19

1 Plan de validation

Voici le plan de validation que nous avons mis en place pour réaliser le projet de détection de points d'intérêts.

1.1 Analyses

— Vérification de l'obtention de l'image en sortie à partir du module de lecture / écriture d'images.
— Vérification de l'obtention de l'image en sortie après ajout d'un module calculant une fenêtre glissante.
— Vérification du résultat des filtres de Sobel (horizontal et vertical).
— Vérification du résultat de la détection de points d'intérêts.

1.2 Démonstrations

— Comparaison entre l'image d'entrée lue et réécrite par le design et l'image d'entrée ouverte via imageJ.
— Comparaison entre l'image d'entrée lue et réécrite par le design après un passage dans la fenêtre glissante.
— Comparaison entre les filtres de sobel (horizontal et vertical) en sortie de notre design et ceux obtenus via imageJ.
— Comparaison entre la détection de point d'intérêts en sortie de notre design et celle obtenue via imageJ.

2 Lecture d'image à partir d'un fichier texte

2.1 Principe

La première étape de ce projet avait pour but de vérifier que le module de lecture d'image à partir d'un fichier texte fonctionnait correctement. Pour ce faire nous avons intégrer le module dans notre projet et avons ce dernier pour que le module puisse accéder aux différents emplacements de fichiers depuis la simulation.

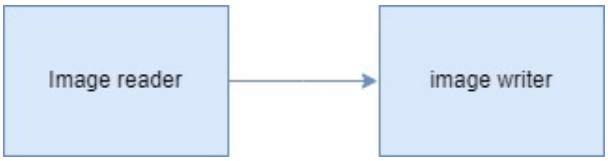

Figure 1 – Synoptique de la partie lecture/écriture

Les modules **image_reader** et **image_writer** s'occupent respectivement de lire une image depuis un fichier texte et d'écrire l'image en sortie du design dans un fichier texte. Ces deux modules ne peuvent lire et écrire depuis / dans des fichiers textes uniquement pendant la simulation du design dans vivado, car notre design n'est pas conçu pour utiliser la RAM présente sur la carte coraZ7, ce qui nous empêche de stocker une image sur la carte pour une future lecture et également de stocker les futurs résultats de traitements des images. La simulation nous permet de nous passer de cette étape de gestion de fichier sur la carte, mais nous empêche par la même occasion de tester ce projet sur la carte.

2.2 Résultats de simulations - Analyse

Nous allons maintenant observer le résultat de simulation des modules de lecture et écriture d'images.

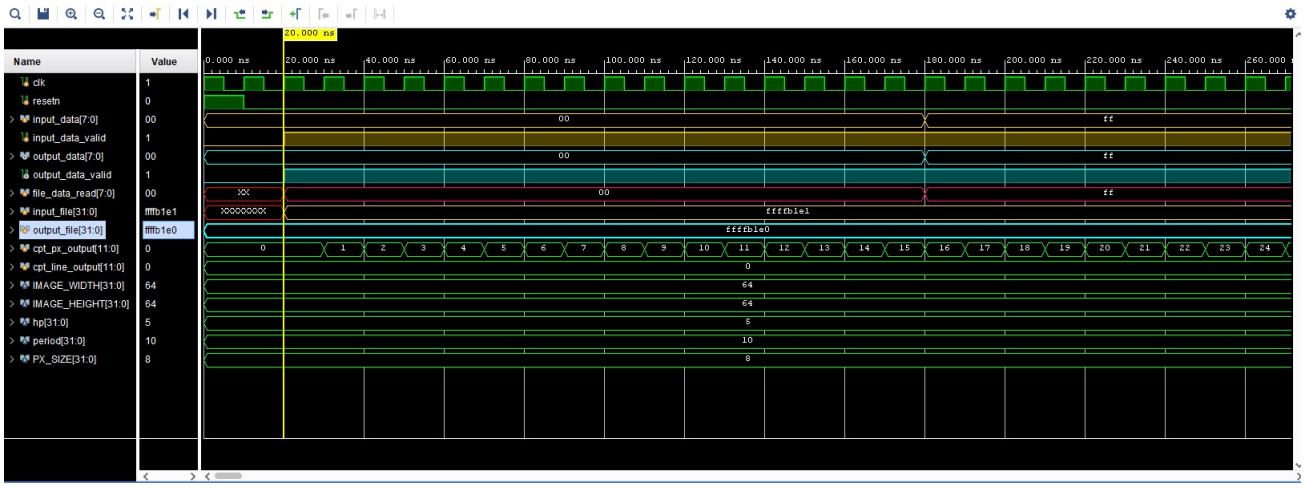

Figure 2 – Chronogramme de la lecture/écriture d'images

On peut voir sur le chronogramme que les données d'entrées (**input_data_valid**) sont valides à partir de 20 ms, ce qui correspond au moment où le fichier d'entrée est ouvert, ce que l'on remarque avec la mise à jour de l'adresse permettant d'écrire dans le fichier (**input_file**). Une fois le fichier ouvert, la lecture commence, on écrit chaque pixel lu dans le fichier de sortie, et on indique que la donnée de sortie est valide avec le signal **output_data_valid**.

2.3 Résultats finaux - Démonstration

Nous allons maintenant observer les images obtenues à partir des modules de lecture et écriture d'images en utilisant le logiciel **imageJ** qui nous permet de visualiser des images stockées sous forme de fichier texte. L'image ci-dessous sera l'image que nous utiliserons en entrée et que nous chercherons à obtenir en sortie.

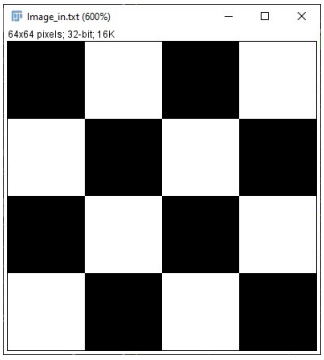

Figure 3 – Image d'entrée

L'image ci-dessous est l'image de sortie que nous obtenons après écriture des pixels lu depuis le fichier texte d'entrée dans notre fichier texte de sortie. On ne remarque aucune différence à l'œil nu avec l'image d'entrée.

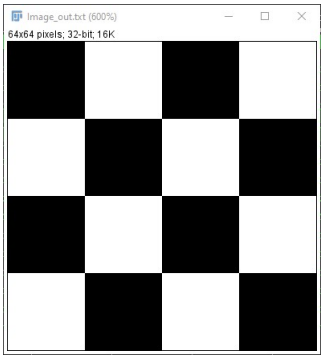

Figure 4 – Image de sortie

Pour vérifier qu'il n'y a vraiment aucune différence entre les deux images, nous allons les soustraire et vérifier que le résultat nous donne une image complètement noire, ce qui signifiera que les images étaient identiques.

L'image ci-dessus représente la soustraction de l'image de sortie à l'image d'entrée, et on s'aperçoit que le résultat donne l'image noire que nous attendions. Nos deux images sont

Figure 5 – Différences entre les deux images

donc identiques et nous pouvons commencer à travailler sur la prochaine partie de notre design.

3 Fenêtre glissante

Nous allons maintenant nous occuper de la partie concernant la *fenêtre glissante*, qui nous sert à mettre toute la logique en place pour la partie de calculs de convolutions.

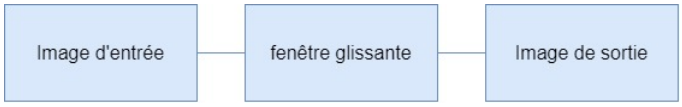

Figure 6 – Synoptique de la partie "fenêtre glissante"

3.1 Principe

L'objectif de cette partie est de permettre aux futurs blocs de calculs de convolutions d'avoir assez de données pour effectuer leurs calculs. Pour ce faire, ils ont besoin des trois pixels voisins au premier pixel de l'image, qui est le pixel en haut à gauche de l'image, comme indiqué sur la figure ci-dessous.

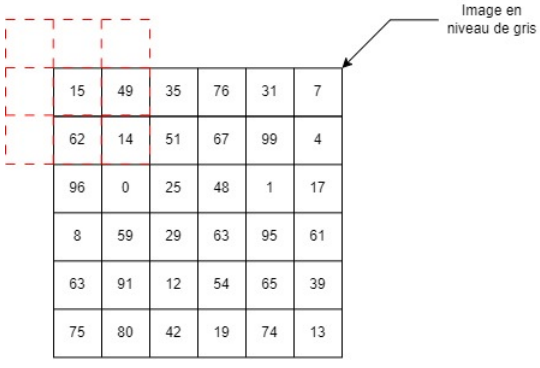

Figure 7 – Données nécessaire pour la convolution (3 x 3)

Le carré en pointillé rouge représente le noyau de convolution que nous allons utiliser pour effectuer nos calculs sur l'image, et que nous allons détailler dans la section suivante. Comme expliqué plus haut, nous aurons besoin des trois pixels voisins au pixel de valeur "15", qui sont les pixels de valeur "49", "62" et "14". Etant donné que nous obtenons les pixels un à un depuis le fichier texte, nous avons besoin de garder en mémoire les pixels précédemment lus de sorte à ce que nous ayons assez de pixels pour faire nos calculs. Pour cela, nous utilisons neuf registres qui nous servent à stocker les pixels qui seront "dans le noyau de convolution" (carré en pointillé rouge), ainsi que 2 files (FIFO) qui nous serviront de "tampon" de lignes de l'image.

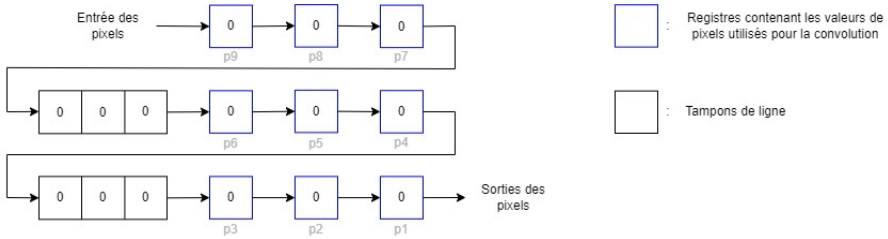

Figure 8 – fonctionnement de la fenêtre glissante sur FPGA

Le schéma ci-dessus illustre les explications précédentes. Nous avons nos neufs registres représentés par les carrés bleus qui servent à obtenir la **fenêtre glissante** (carré en pointillé rouge du schéma figure 7) ainsi que les deux files représentées par les rectangles noirs qui servent de tampon de lignes pour stocker les deux lignes les plus récentes lues dans le fichier contenant l'image. Nous avons besoin de stocker uniquement ces deux lignes, car les registres **p9** à **p7** stockent les trois pixels les plus récents. Les deux lignes stockées nous permettent de toujours avoir accès aux pixels qui seront sur le chemin de la fenêtre glissante, qui se déplace de gauche à droite sur l'image. Pour stocker tous ces pixels, on commence par "pousser" le premier pixel dans le registre **p9**, et on utilise un système de "registres à décalages" pour continuer de pousser ce pixel jusqu'au registre **p1**. A chaque fois qu'un pixel change de registre, un nouveau entre par le registre **p9** et est poussé de la même manière jusqu'au registre **p1**.

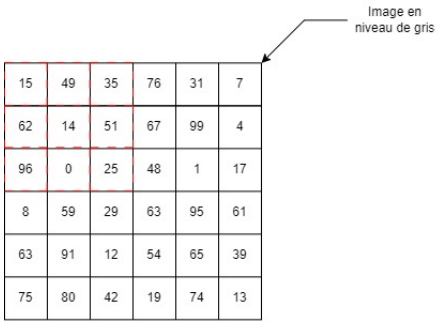

Figure 9 – Calcul de convolution 2

Sur l'image ci-dessus, nous souhaitons appliquer notre filtre de Sobel sur le pixel au centre du noyau de convolution (carré en pointillé rouge) de valeur **14**. Pour cela nous aurons besoin des huit pixels voisins qui sont les pixels de valeurs : **15**, **49**, **35**, **62**, **51**, **96**, **0** et **25**. Les pixels "96", "0" et "25" sont les trois pixels lus les plus récent et sont stockés dans le registres "p9", "p8" et "p7", tandis que les autres pixels sont ceux des lignes précédentes qui seront poussés sur les registres **p6**, **p5** et **p4** pour l'avant dernière ligne la plus récente, et les registres **p3**, **p2** et **p1** pour la ligne la plus ancienne stockée. Le système de décalage des pixels dans les registres nous assure donc que l'on aura correctement chaque pixel requis pour effectuer le calcul de convolution pour le pixel se trouvant au centre du noyau de convolution, et c'est ce système que nous allons chercher à valider au travers des sous sections suivantes, car s'il ne fonctionne pas correctement, tous nos calculs seront faussés par la suite.

3.1.1 Schéma RTL

Voici le schéma RTL que nous avons réalisé pour la partie "fenêtre glissante". On remarque que les compteurs **x** et **y** sont sur 32 bits, car nous utilisons le type "**integer**" du VHDL qui utilise par défaut 32 bits pour stocker des entiers.

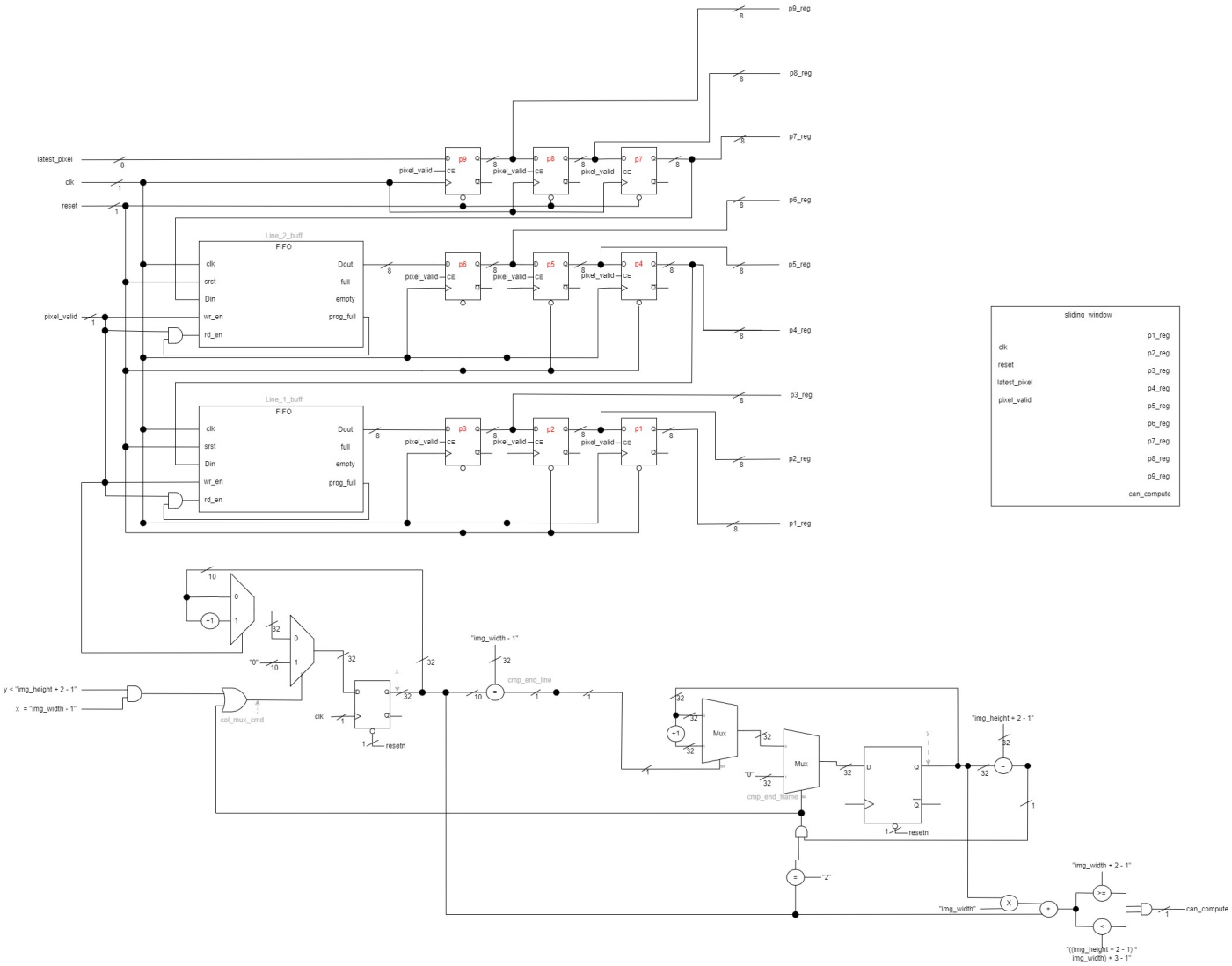

Figure 10 – Schéma RTL de la fenêtre glissante

3.2 Résultats de simulations - Analyse

Nous allons à présent nous intéresser aux résultats obtenus pour la simulation du système de fenêtre glissante présenté dans la sous-section précédente, l'objectif étant de vérifier que nous aurons le résultat attendu sur l'image de sortie, c'est-à-dire obtenir une image de sortie équivalente à l'image d'entrée. Pour la simulation nous avons utilisé une image représentant un damier composé de carré noir et blanc, chacun mesurant 16 pixels de large pour une dimension totale de l'image de 64 x 64 pixels. Nous avons également modifié l'image pour y inclure des valeurs spéciales que nous avons placé sur les bords gauche et droit de l'image ainsi que sur le bord droit du premier carré du damier. Voici donc l'image utilisée :

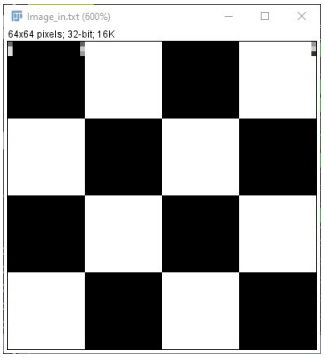

Figure 11 – Image d'entrée pour la simulation

Observons à présent le chronogramme ci-dessous. On observe qu'après la période d'attente de 20 ns, on commence à lire dans l'image d'entrée, et qu'à chaque front montant suivant de l'horloge **clk** la valeur lue dans le fichier texte contenant l'image est poussé dans un nouveau registre en commençant par le registre **p9** pour ensuite aller jusque dans la première file. Le premier pixel de notre image est une valeur spéciale "**6f**", que l'on a placé pour s'assurer que le décalage lié au calcul de convolution sera correct (Cf. figure **??**). Il faudra donc attendre que le file se remplisse pour que l'on retrouve notre valeur "6f" qui sera alors poussé dans le registre **p6**.

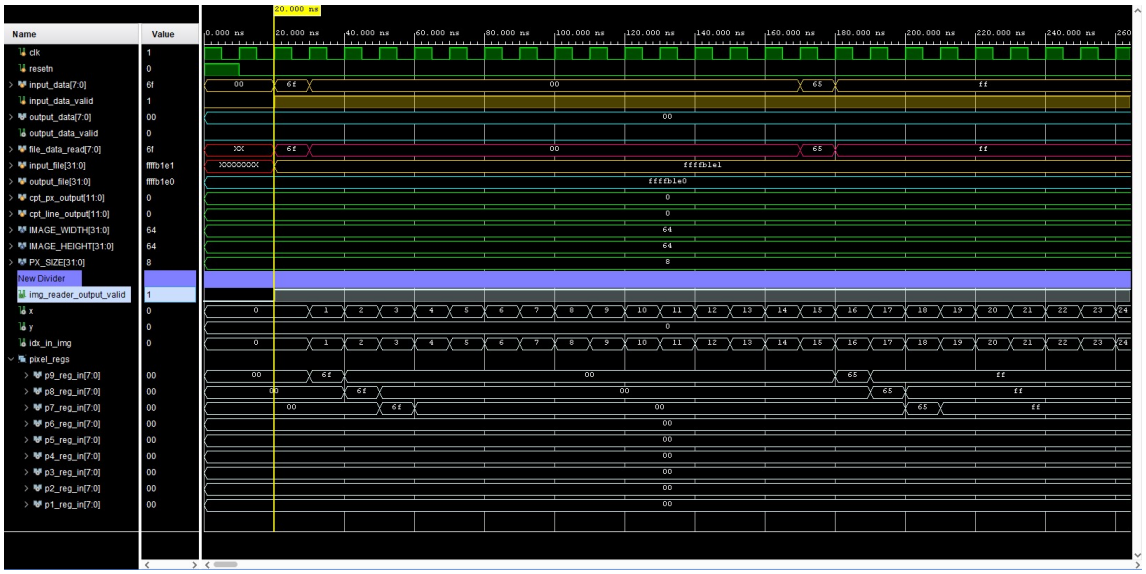

Figure 12 – Chronogramme de la partie fenêtre glissante

Sur le chronogramme ci-dessous, on retrouve notre valeur spéciale "**6f**" en sortie du registre **p5** lorsque "**L + 2**" pixels (avec **L** la largeur de l'image) ont été poussé dans notre système de registres à décalage. Ici la largeur de notre image est de 64 pixels et on observe que la répartition des pixels dans les registres **p9** à **p1** est correcte lorsque le 66^{me} pixel est poussé dans le système (le signal **idx_in_img** représente le pixel sur lequel on se trouve dans l'image moins un car on commence à compter à partir de "0"). On observe également qu'une nouvelle valeur spéciale qui est : **de**, a été poussé.

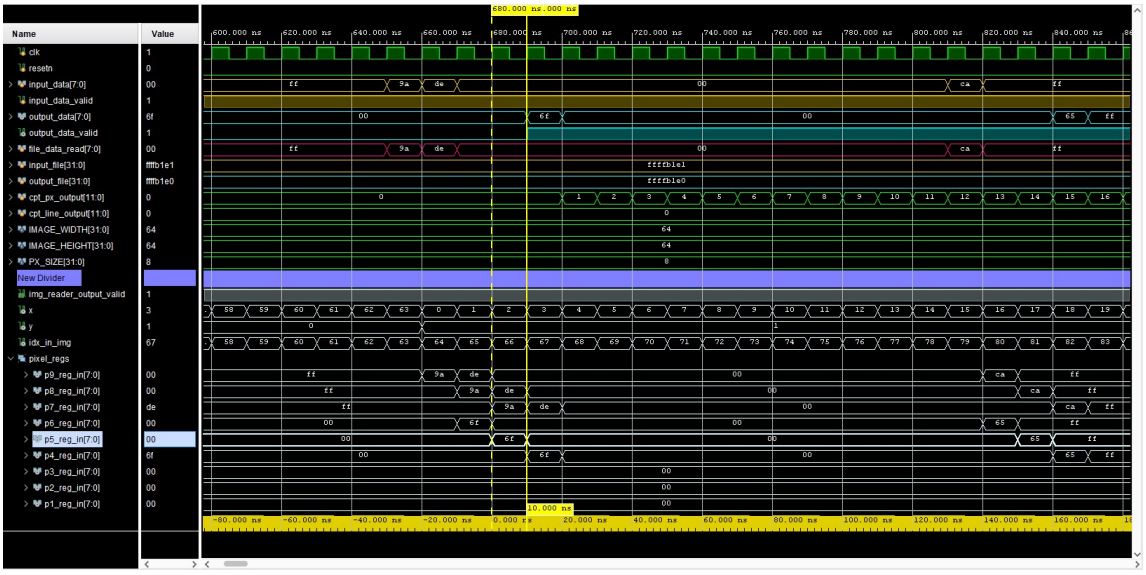

Figure 13 – Chronogramme de la partie fenêtre glissante - début d'envoi de pixels

On remarque donc que le calage est correcte à l'aide de ce schéma qui représente le noyau de convolution :

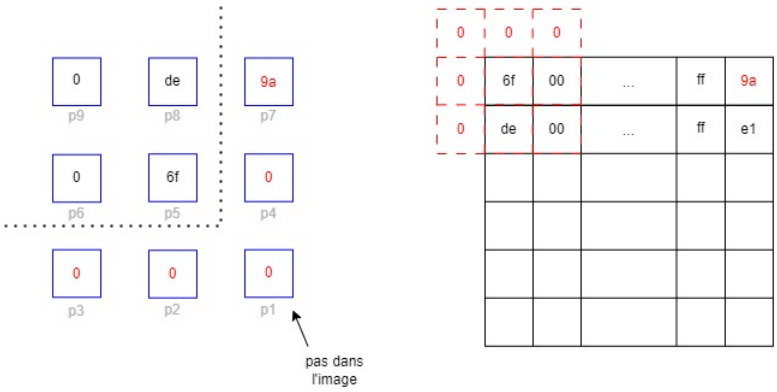

Figure 14 – schéma du calage des pixels dans la simulation

On peut voir sur la droite du schéma une reconstitution de la partie de l'image qui nous intéresse et sur la gauche la répartition des pixels dans les différents registres. On cherche ici à calculer la convolution sur le pixel de valeur hexadécimale "**6f**" qui se trouve dans le coin supérieur gauche de l'image. On remarque que le noyau de convolution "déborde" de l'image, et que les registres **p7**, **p4**, **p3**, **p2** et **p1** vont donc "fausser" notre calcul. On remarque également que notre système de registres à décalage et de files présenté plus haut amène dans le registre **p7** le pixel "**9a**" qui correspond au coin supérieur droit de l'image, cependant ce pixel n'est pas censé être présent lors du calcul pour le pixel "**6f**". Il y a plusieurs manières de contourner ce problème comme par exemple utiliser la valeur "**0**" pour chaque "pixel" du noyau de convolution qui ne serait pas dans l'image, cependant nous nous sommes contenter d'utiliser les pixels se trouvant dans les registres, même si ceux-ci vont "fausser" nos calculs sur les bords et coin de l'image.

3.3 Résultats finaux - Démonstration

Intéressons-nous maintenant au résultat obtenu après le passage de l'image dans la fenêtre glissante. Pour rappel, voici l'image d'entrée que nous avons utilisée avec les valeurs spéciales servant à s'assurer du calage correct des pixels.

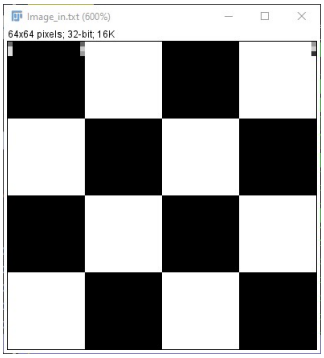

Figure 15 – Image d'entrée

L'image ci-dessous est le résultat obtenu après le passage de l'image ci-dessus dans la fenêtre glissante. On ne remarque pas de différence à l'œil nu entre ces deux images.

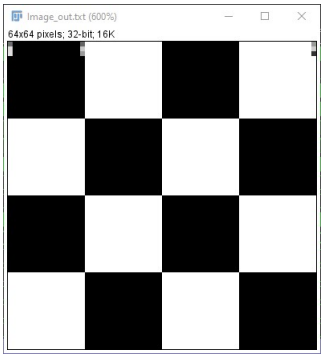

Figure 16 – Image de sortie

Pour nous assurer que ces deux images sont identiques, on soustrait l'image de sortie à l'image d'entrée en s'attendant à ce que l'image résultante soit complètement noire, ce que l'on observe sur l'image ci-dessous.

Figure 17 – Différences entre les deux images

4 Filtres de Sobel et détection de points d'intérêt

Dans cette section nous allons nous intéresser à la partie concernant la détection de points d'intérêt dans une image, expliquer l'intérêt des filtres de Sobel dans ce processus et également montrer l'intérêt de la section concernant le **fenêtre glissante**.

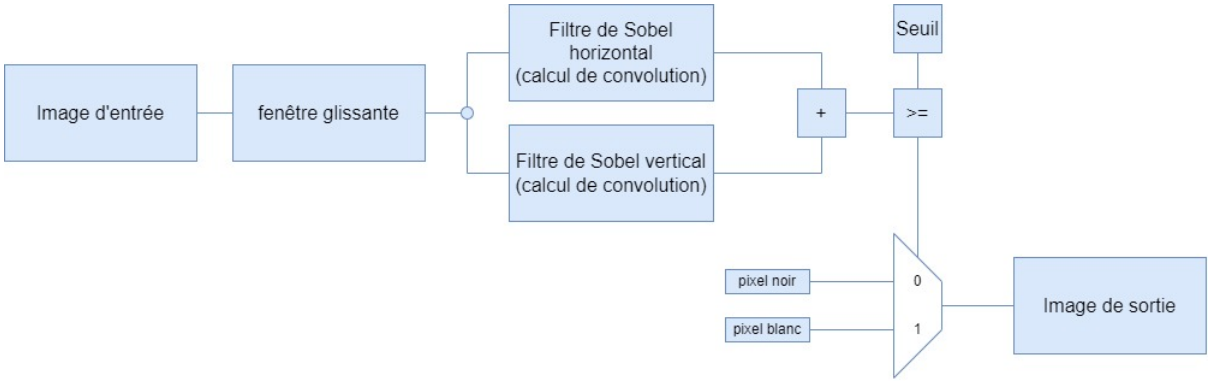

Figure 18 – Synoptique complet du projet

4.1 Principe

La détection de point d'intérêt consiste à trouver des éléments de l'image "facilement reconnaissable" par un algorithme dans le but d'effectuer d'autres actions avec ces éléments par la suite, comme par exemple reconstruire un objet en 3D à partir de plusieurs images prises dans des angles différents dans lesquelles on reconnaît nos points d'intérêts qui vont nous permettre d'obtenir les informations nécessaires à la reconstruction de l'objet.

Pour détecter ces points d'intérêt nous allons appliquer deux filtres sur notre image qui seront les filtres de Sobel horizontal et vertical, qui auront pour but de détecter les bords "verticaux" et "horizontaux" de notre image. Une fois ces bords obtenus, nous allons chercher les points d'intérêt en additionnant les résultats des filtres de Sobel horizontal et vertical, ce qui mettra en évidence les coins de chaque bord qui seront donc nos points d'intérêt.

Nous allons expliquer plus en détail le déroulement de ces différentes étapes en commençant par le filtre de Sobel horizontal.

4.1.1 Sobel x (horizontal ?)

Le filtre de Sobel horizontal a pour but de détecter les bords dit "verticaux" présents dans une image en niveau de gris. Pour détecter ces bords, on applique notre calcul de convolution sur chaque pixel de l'image avec la matrice suivante :

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Les bords verticaux sont détectés grâce aux contrastes entre les deux colonnes voisines au pixel central du noyau. Plus le contraste est élevé, plus la différence entre les colonnes de gauche et de droite du noyau est grande, ce qui indique que le pixel au centre du noyau de convolution se trouve sur un bord et le pixel résultant sera "clair" (proche de la valeur maximum d'un pixel en niveau de gris, soit **255**) pour l'indiquer. Au contraire, si les pixels des colonnes de gauche et de droite ont des valeurs similaires, on ne se trouve pas sur un bord et le pixel résultant sera "foncé" (proche de la valeur minimum d'un pixel en niveau de gris, soit **0**) pour l'indiquer.

4.1.2 Sobel Y (vertical ?)

Le filtre de Sobel vertical a pour but de détecter les bords dit "horizontaux" présents dans une image en niveau de gris. Pour détecter ces bords, on applique notre calcul de convolution sur chaque pixel de l'image avec la matrice suivante :

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Les bords horizontaux sont détectés de la même manière que les bords verticaux.

4.1.3 Détection de point d'intérêt

Comme expliqué au début de la section, la détection de points d'intérêt se fait en additionnant le résultat des filtres de Sobel horizontal et vertical. Le résultat nous permettra d'avoir une superposition des bords verticaux et horizontaux de l'image, ce qui mettra en évidence les différents coins associés à ces bords. Ces bords constitueront nos points d'intérêt, et nous utiliserons une valeur de seuil que nous pouvons ajuster pour filtrer certains points d'intérêt qui ne serait pas vraiment pertinent à conserver. Une fois le résultat filtré par le seuil, on utilise deux valeurs de pixel, noir ou blanc, pour obtenir un résultat visuel de la détection.

4.1.4 Schéma RTL

Voici le schéma RTL que nous avons réalisé pour la partie "filtre de Sobel". On remarque que les sorties **out_x** et **out_y** sont sur 32 bits, car nous utilisons le type "**integer**" du VHDL qui utilise par défaut 32 bits pour stocker des entiers.

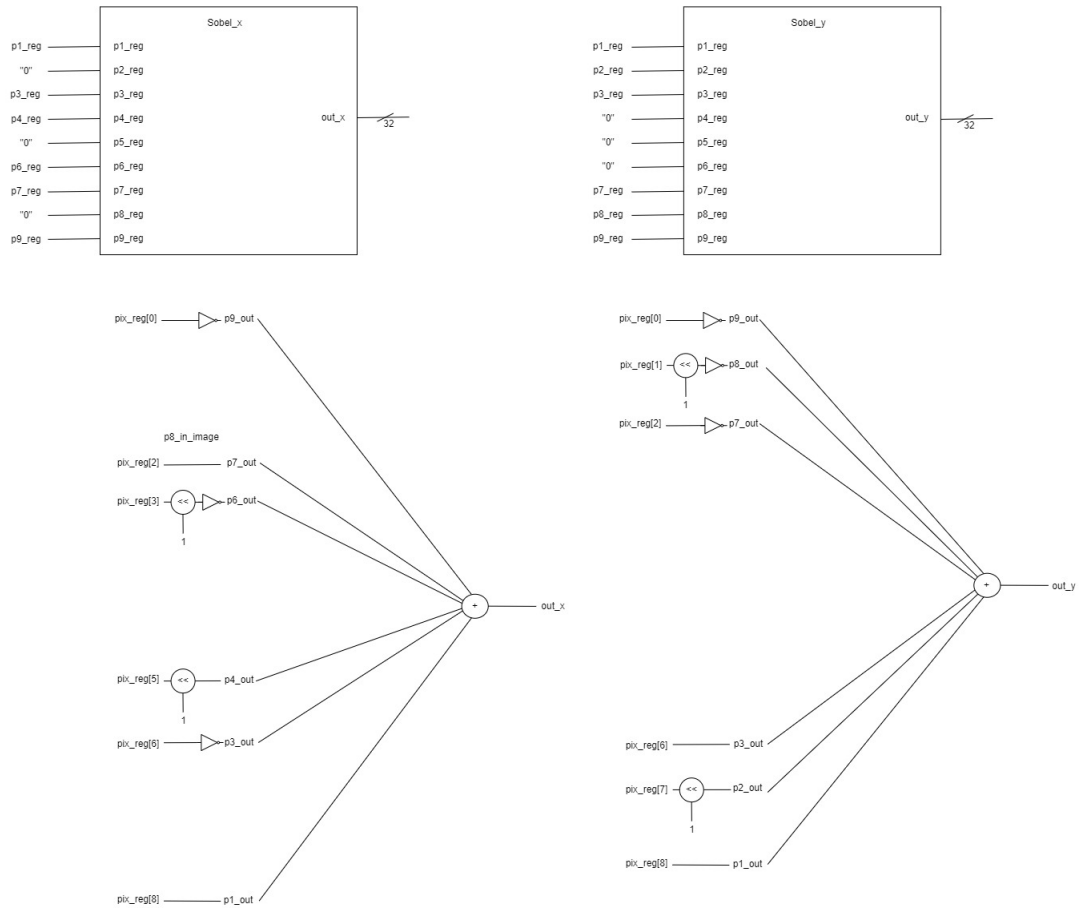

Figure 19 – Schéma RTL des filtres de Sobel x et y

Voici maintenant le schéma complet du design :

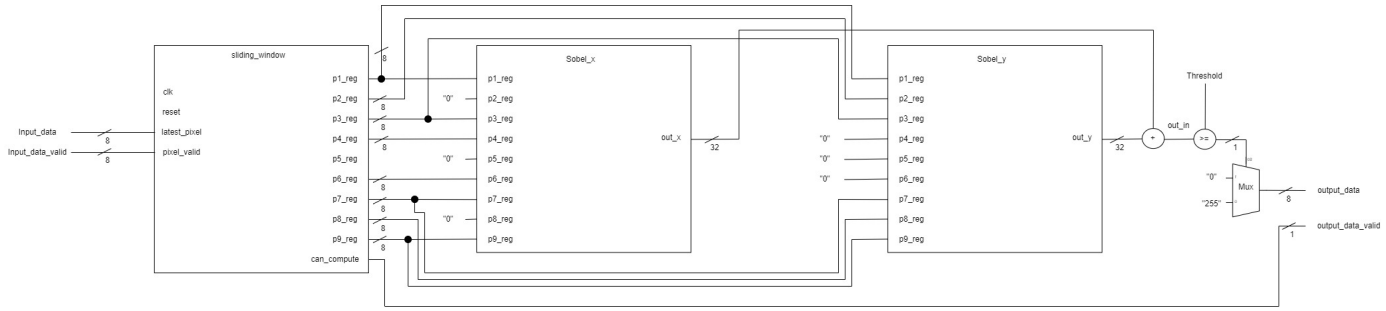

Figure 20 – Schéma RTL complet

4.2 Résultats de simulations - Analyse

Observons maintenant un résultat de simulation pour la partie de détection de point d'intérêt. Comme expliqué dans les sections précédentes, on cherche ici à trouver les points d'intérêt d'une image en lui appliquant les filtres de Sobel horizontaux et verticaux, puis en filtrant le résultat en fonction du niveau de détail que l'on souhaite obtenir. Retrouvons maintenant ces éléments dans le chronogramme suivant.

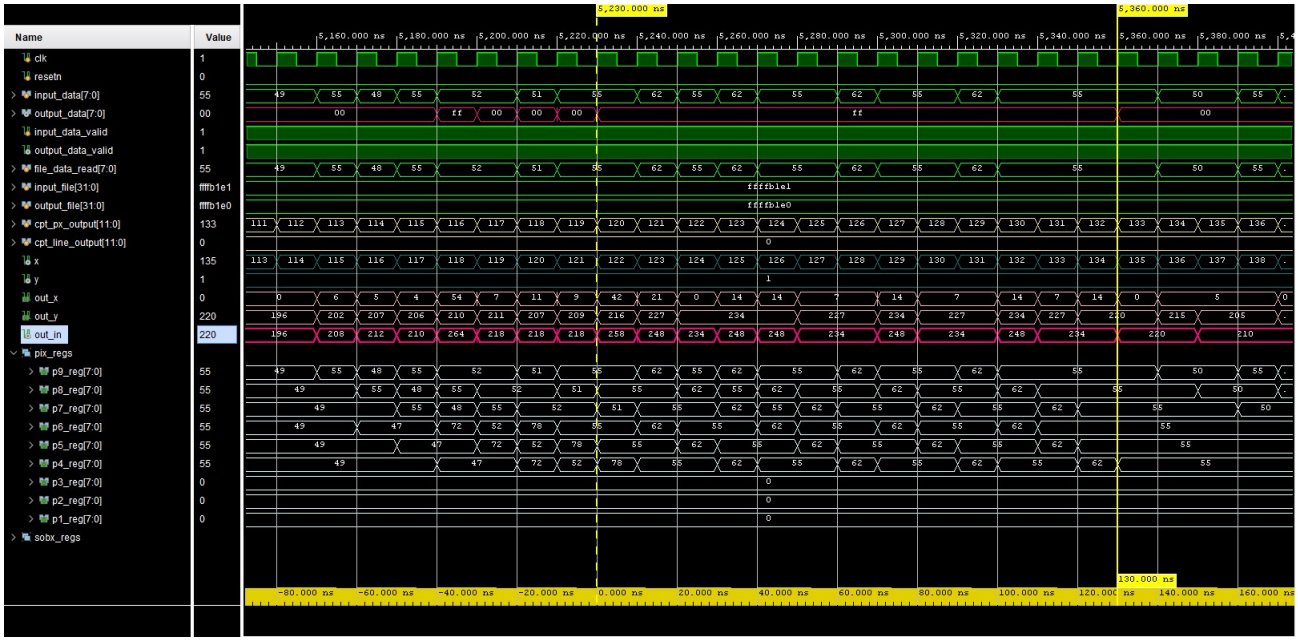

Figure 21 – Chronogramme de la détection de points d'intérêt

Concentrons nous donc sur les signaux "**out_in**" qui représente la somme des filtres de Sobel horizontal et vertical et "**output_data**" qui représente la sortie de notre design (un pixel noir ou blanc en fonction de la valeur de seuil choisie). On se place sur ce chronogramme à environ 5 µs, ce qui nous permet de repérer un cas de filtrage au niveau de la sortie "output_data". En effet, on remarque qu'à partir de 5 230 ns la sortie "output_data" passe d'un pixel blanc à un pixel noir, car "out_in" a dépassé la valeur de seuil qui dans notre cas est de **230**. "out_in" reste au-dessus de la valeur de seuil pendant 130 ns, puis on remarque que la sortie repasse à **0**. Nous pouvons donc nous dire que notre résultat de simulation est correct et nous allons maintenant le confirmer avec une démonstration.

4.2.1 Analyses de timing

Synthesis Report

Report Cell Usage :

Report Cell Usage:

	Cell	Count
1	fifo_generator	1
2	fifo_generator_0_	1
3	BUFG	1
4	CARRY4	72
5	LUT1	40
6	LUT2	83
7	LUT3	55
8	LUT4	37
9	LUT5	39
10	LUT6	97
11	FDCE	118
12	IBUF	11
13	OBUF	9

Figure 22 – Report Cell Usage

On retrouve dans le Report Cell Usage, les 2 fifo_generator utilisées comme lignes buffer pour le cœur de convolution de la fenêtre glissante.

On retrouve également les 11 entrées du schéma :
— clk
— resetn
— input_data (sur 8 bits)
— input_data_valid
Ainsi que les 9 sorties :
— output_data (sur 8 bits)
— output_data_valid
Par ailleurs on retrouve 118 registres FDCE, dont 107 pour la Sliding Window.

Detailed RTL Component Info :

```
Detailed RTL Component Info :
+---Adders :
       6 Input   32 Bit       Adders := 2
       2 Input   32 Bit       Adders := 1
       2 Input   31 Bit       Adders := 1
+---Registers :
                   8 Bit    Registers := 7
+---Multipliers :
                  1x32  Multipliers := 2
+---Muxes :
       2 Input   32 Bit        Muxes := 2
       2 Input    1 Bit        Muxes := 2
```

Figure 23 – Detailed RTL Component Info

Dans le Detailed RTL Component Info, on retrouve le nombre de registres ainsi que leur nombre de bits, mais aussi l'ensemble des multiplexeurs utilisés dans le schéma avec leur nombre de bits, ainsi que les multipliers et les adders qui servent à faire des opérations.

Timing Summary - Route Design

Design Timing Summary :

Design Timing Summary

WNS(ns)	TNS(ns)	TNS Failing Endpoints	TNS Total Endpoints	WHS(ns)	THS(ns)
2.374	0.000	0	438	0.061	0.000

Figure 24 – Design Timing Summary

On voit dans le rapport de timing qu'il n'y a pas de violations, car le Total Negative Slack (TNS) et Total Hold Slack (THS) sont tous les deux égale à 0. Le Worth Negative Slack (WNS) quant à lui est de 2.374 ns.

Max Delay Paths :

Le chemin critique (Max Delay Paths) prend sa source ici : «SLIDING_WINDOW_INST/y_reg[30]/C et sa destination ici : «SLIDING_WINDOW_INST/y_reg[29]/D».

```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack (MET) :             2.374ns  (required time - arrival time)
  Source:                 SLIDING_WINDOW_INST/y_reg[30]/C
                            (rising edge-triggered cell FDCE clocked by sys_clk_pin  {rise@0.000ns fal
  Destination:            SLIDING_WINDOW_INST/y_reg[29]/D
                            (rising edge-triggered cell FDCE clocked by sys_clk_pin  {rise@0.000ns fal
```

Figure 25 – Max Delay Paths

4.3 Résultats finaux - Démonstration

Observons maintenant les images associées au résultat de simulation de la section précédente. Voici tout d'abord l'image utilisée en entrée. On notera que cette image est en niveau gris car cela est nécessaire pour que la détection de points d'intérêt fonctionne correctement.

Figure 26 – Nouvelle image de référence pour la démonstration

Une fois que l'image est traitée par notre design, on obtient le résultat ci-dessous. On aperçoit distinctement en blanc une partie du contour du personnage présent sur l'image. Notre détection de points d'intérêt d'est donc déroulé correctement, car nous obtenons bien une suite de points en fonction des différences de contrastes (bords) de l'image.

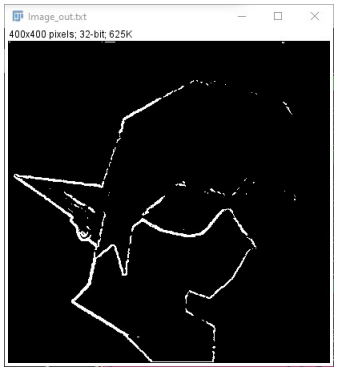

Figure 27 – Détection de contour sur la nouvelle image

La dernière image présentée ci-dessous montre quant à elle la superposition de l'image d'entrée avec l'image en sortie de détection, qui est ici affichée en jaune. On remarque que

la partie jaune qui correspond à nos points d'intérêt représente bien les contours de notre personnage.

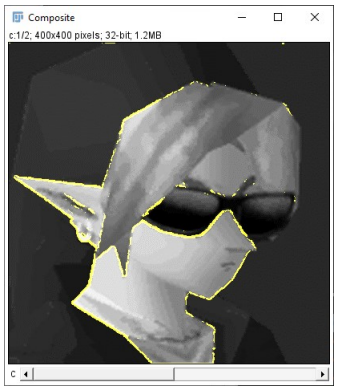

Figure 28 – Détection de contour + image de référence

Pour terminer cette section, nous allons appliquer notre détection sur une nouvelle image afin de nous rendre compte des différences de détection entre deux images.

Figure 29 – Détection de contour + image de référence