12/07/2023

# Filtre de Sobel

## Rapport de validation

Atmani Hicham & Kamal Kherchouch
FORMATION SAFRAN CHEZ AJC

| Date | Version | Remarque | Auteur |
|---|---|---|---|
| **12/07/2023** | A0 | Rapport de validation en accord avec le plan de validation version A0 du 10/07/2023 | AHI & KKH |
| | | | |
| | | | |

# Table des matières

# 1. Entité et signaux interne de la solution proposée

## 1.1. Entité de la solution

Pour la réalisation de notre solution, nous devons réaliser l'architecture de notre boîte noire (système).


De ce fait, nous retrouverons :

- En entrée :
  - Resetn : signal de reset de notre système
  - Clk : signal d'horloge de notre système
  - Input_data : signal de données de l'image d'entrée (un pixel de 8 bits).
  - Input_data_valid : signal de validation de lecture du vecteur input_data.
- En sortie
  - Output_data : signal de données de l'image de sortie (un pixel de 8 bits).
  - Output_data_valid : signal de validation d'écriture du vecteur Output_data.

On ajoute également deux paramètres génériques :

- PX_SIZE : entier permettant de définir la taille d'un pixel
- Seuil_Sobel : valeur du seuil que nous utiliserons pour la comparaison lors de l'application du filtre de Sobel

On retrouve donc le code suivant :

```
entity VGA_filtre_Sobel is
    generic(
        PX_SIZE : integer := 8;         -- taille d'un pixel
        Seuil_Sobel : integer := 850    -- 850 pour le calcul ss racine et 600 pour la version avc racine
    );
    port(
        resetn  : in std_logic;
        clk     : in std_logic;
        input_data          : in std_logic_vector(PX_SIZE-1 downto 0);
        input_data_valid    : in std_logic;
        output_data         : out std_logic_vector(PX_SIZE-1 downto 0);
        output_data_valid   : out std_logic
    );
end VGA_filtre_Sobel;
```

## 1.2. Signaux internes utiles à la solution

Dans cette partie, nous allons lister les signaux internes que nous devrons utiliser lors de la mise en place de notre solution.
Il est à noter que nous avons opté pour un avancement process par process. C'est-à-dire que nous arriverons à notre solution finale après plusieurs étapes intermédiaires. De ce fait, tous les signaux ne seront pas forcément utilisés dans tous les process.

Les étapes de validation sont les suivantes :

- Process pour la validation du passe plat : On a une image en entrée, on veut la retrouver en sortie.
- Process pour la validation de la mise en place des registres et des FIFOs (Sliding Window): On a une image d'entrée où les pixels vont être stockés dans des registres et des variables, en sortie on souhaite retrouver notre image sans altération des pixels (désynchronisation) dû au temps de traitement induit par les FIFOs.
- Process pour la validation d'un filtre de Gauss : permettant de faire un petit clin d'œil au projet VGA précédent et valider la mise en place d'un filtre.
- Process pour la mise en place d'un filtre de Sobel en H (horizontal)
- Process pour la mise en place d'un filtre de Sobel en V (vertical)
- Process pour la validation du filtre de Sobel.

Les signaux utilisés seront donc les suivants :

- Reset : signal inverse du signal resetn permettant l'application du reset au FIFOs.
- Calcule_Gauss : signal permettant de faire le calcul de Gauss
- Calcule_Sobel_H_Nat : signal permettant de réaliser le calcul de Sobel en H
- Calcule_Sobel_V_Nat : signal permettant de réaliser le calcul de Sobel en V
- Calcule_Sobel_H_Nat : signal permettant de réaliser le calcul de Sobel en H
- Calcule_gradiant_Sobel_Nat : signal permettant de réaliser le calcul du gradiant
- Compteur_projet : compteur de coup d'horloge du projet
- Activation_ecriture_fifo_1 : activation ou désactivation de l'écriture de la fifo 1
- Activation_ecriture_fifo_2 : activation ou désactivation de l'écriture de la fifo 2
- Activation_lecture_fifo_1 : activation ou désactivation de la lecture de la fifo 1
- Activation_lecture_fifo_2 : activation ou désactivation de la lecture de la fifo 2
- Out_Registre_Fifo1 : signal de la sortie de la fifo 1
- Out_Registre_Fifo2 : signal de la sortie de la fifo 2
- Reg1 à Reg9 : signaux de registres
- K1_Gauss à K9_Gauss : kernel du filtre de Gauss
- K1_H_Nat à K9_H_Nat : kernel du filtre de Sobel en H
- K1_V_Nat à K9_V_Nat : kernel du filtre de Sobel en V

## 2. Test Bench de la solution proposée

Un test bench est à mettre en place afin de pouvoir tester nos différents process décrit dans la partie ci-dessus.
Le test bench aura pour objectif de :

- Générer un signal d'horloge cadencé à 100 MHz.
- L'ouverture d'un fichier d'entré (en lecture) au format texte (.txt) (conversion en texte de l'image).
- La lecture des valeurs des pixels présents dans le fichier texte.
- La fermeture du fichier d'entré.
- L'ouverture d'un fichier de sortie (en écriture) au format texte (.txt)
- L'écriture des valeurs de pixels de sortie.
- La fermeture du fichier de sortie.

Les paramètres suivants seront utilisés :

- IMAGE_WIDTH : valeur du nombre de pixel en horizontal (640)
- IMAGE_HEIGHT : valeur du nombre de pixel en vertical (480)
- Hp : demi période du signal d'horloge (5ms pour une fréquence de 100 MHz)
- Période : période du signal d'horloge (10ms ou 2*hp pour une fréquence de 100 MHz)
- Cpt_px_output : compteur de colonne pour l'écriture
- Cpt_line_output : compteur de ligne pour l'écriture
- Les signaux à affecter au projet à tester (signaux décris dans la partie 1)

L'ouverture d'un fichier se fait à l'aide de la fonction « fopen » et la fermeture se fait à l'aide de la fonction « fclose ».

```
input_file = $fopen("New_York.txt","r");
```

La fonction « fopen » est décrite de la manière suivante :

$fopen (« nom_du_fichier.txt », mode lecture ou écriture)

Lors de l'ouverture d'un fichier, on utilise la fonction « display » afin de notifier la bonne ouverture ou non de ce dernier.

```
if(input_file) $display("Input file opened\n");
else $display("Input file not opened\n");
```

La fonction « fscanf » permet de réaliser la lecture des données (pixel) présent dans le fichier de lecture.

Le test bench complet est à retrouver dans le fichier : ***tb_read_write_image.v***

# 3. Processus du passe plat

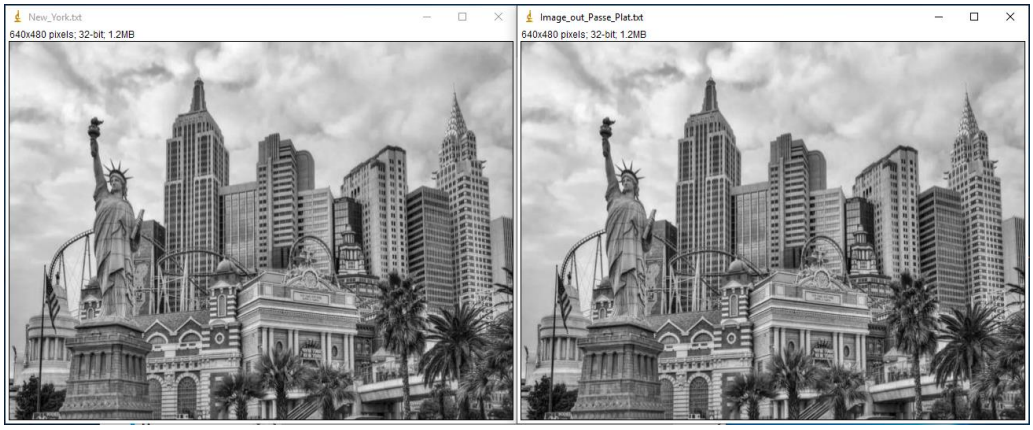
## 3.1. Code VHDL du passe plat

Le code VHDL du process passe plat est très simple à comprendre. Dès lors qu'un pixel est lu en provenance du fichier d'entré, ce dernier est ressorti dans le fichier de sortie à l'aide du signal d'autorisation d'écriture output_valid_data.

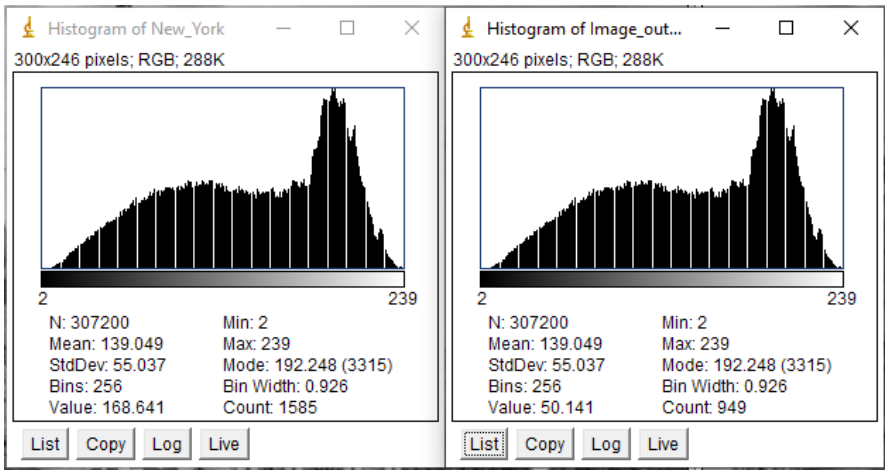
On obtient le code suivant :

```
process(clk, resetn)
begin
    if(resetn='1') then
        output_data <= (others => '0');
        output_data_valid <= '0';
    elsif(rising_edge(clk)) then
        if(input_data_valid = '1') then
           output_data <= input_data;
           output_data_valid <= '1';
        end if;
    end if;
end process;
```

## 3.2. Validation du passe plat




Le passe plat est validé car l'image de sortie (Image_Out_Passe_Plat.txt) est conforme à l'image d'entrée (New_York.txt). De plus, l'histogramme des deux images le confirme :



# 4. Processus du sliding window

## 4.1. Code VHDL du sliding window

La mise en place du sliding window consiste à utiliser les registres 1 à 9 et les FIFO 1 et 2.
Le principe du sliding window est le même que celui décrit dans le projet VGA.

La première étape consiste à activer ou non, l'écriture et la lecture des FIFOs.
L'activation de l'écriture de la première FIFO se fait après le remplissage des trois premiers registres (reg1, reg2 et reg3) et l'activation de la lecture de cette dernière se fait à la fin de la première ligne.
L'écriture et la lecture de la second FIFO se fait de la même façon avec les lignes deux et trois de l'image d'entrée.

Le code est le suivant :

```
if (Compteur_projet = 3) then
    activation_ecriture_fifo_1 <= '1';
elsif (Compteur_projet = 640) then
    activation_lecture_fifo_1 <= '1';
elsif (Compteur_projet = 643) then
    activation_ecriture_fifo_2 <= '1';
elsif (Compteur_projet = 1280 ) then
    activation_lecture_fifo_2 <= '1';
end if;
```

La seconde étape consiste à remplir les registres et les FIFOs afin de respecté la description proposée dans le plan de validation.
C'est-à-dire que le pixel d'entrée va :

- passer dans le registre 1 jusqu'au troisième
- Le registre 3 va envoyer le pixel dans la première FIFO
- La première FIFO va ressortir le pixel dans le registre 4
- Le registre 6 va envoyer le pixel dans la seconde FIFO
- La seconde FIFO va ressortir le pixel dans le registre 7
- Et le pixel sera oublié après le neuvième registre

Notre image de sortie sera prise à l'aide du centre de notre matrice, c'est-à-dire à l'aide du registre 5.

Le code est le suivant :

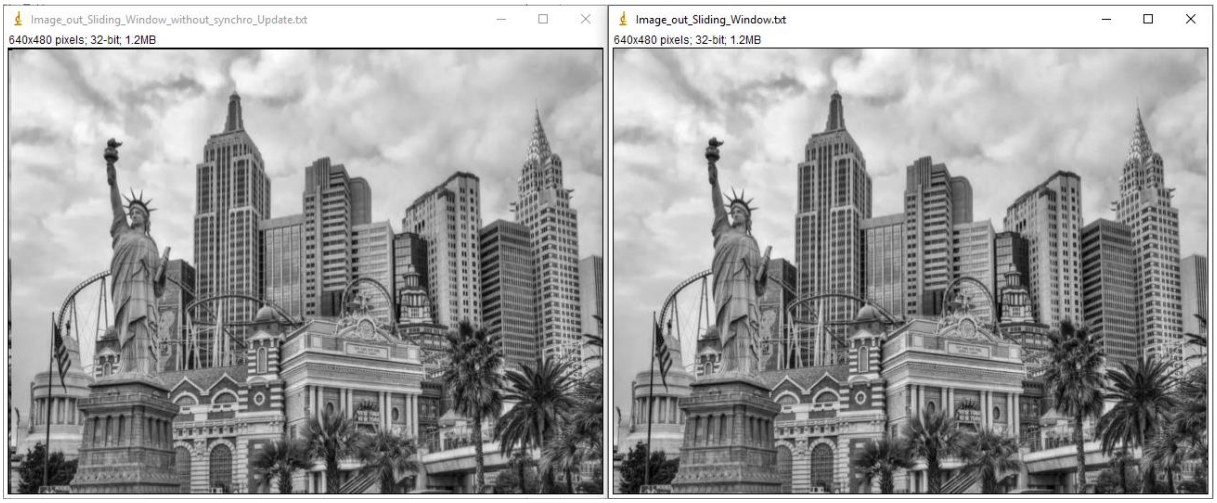
```
reg1 <= input_data;
reg2 <= reg1;
reg3 <= reg2;
reg4 <= Out_Registre_Fifo1;
reg5 <= reg4;
reg6 <= reg5;
reg7 <= Out_Registre_Fifo2;
reg8 <= reg7;
reg9 <= reg8;

output_data <= reg5;
output_data_valid <= '1';
```

Avec cette proposition de solution, le sliding window créé un décalage de l'image à cause de la latence dû à l'utilisation des FIFOs.
Afin de pallier ce problème, il faut activer l'écriture de la sortie après la réception de la seconde ligne.

On modifie donc le code en ajoutant une condition à la sortie :

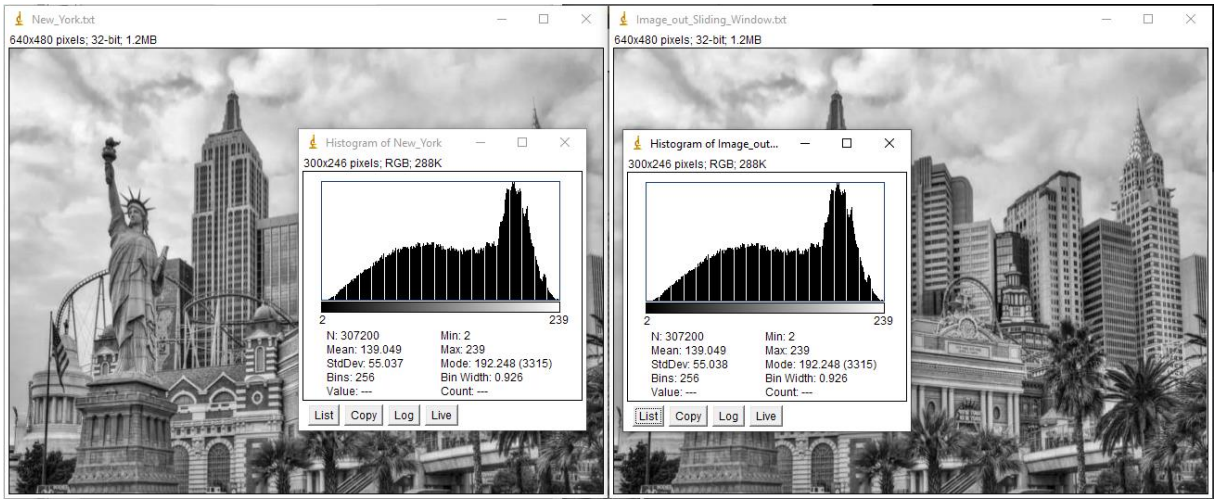
```
if(Compteur_projet > 643) then
    output_data <= reg5;
    output_data_valid <= '1';
else
    output_data_valid <= '0';
end if;
```

## 4.2. Validation du sliding window




On voit bien que si on ne prend pas en compte le temps de latence des FIFOs, l'image de sortie est décalée de 3 colonnes de pixels (photo de gauche)

Si on compare l'image d'entrée et l'image de sortie :




Les histogrammes montrent bien que l'image de sortie est conforme à l'image d'entrée.

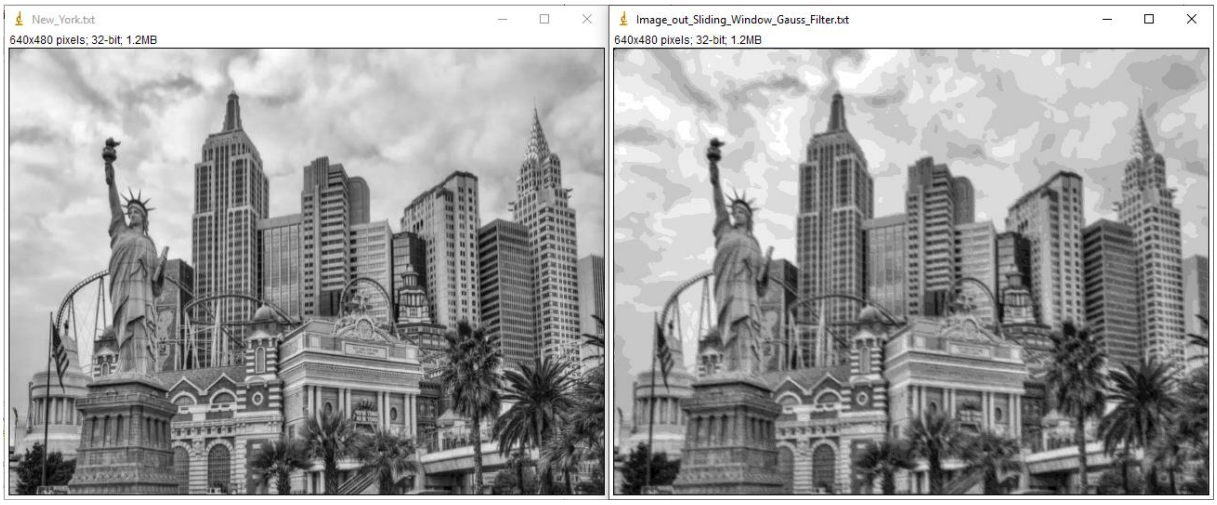
# 5. Processus du filtre de Gauss

## 5.1. Code VHDL du filtre de Gauss

On ajoute au code VHDL le calcul du filtre de gauss.

```
if(Compteur_projet > 643) then
    Calcule_Gauss <= (reg1*K1_Gauss) + (reg2*K2_Gauss) + (reg3*K3_Gauss) + (reg4*K4_Gauss) + (reg5*K5_Gauss) + (reg6*K6_Gauss) + (reg7*K7_Gauss) + (reg8*K8_Gauss) + (reg9*K9_Gauss);
    output_data <= Calcule_Gauss(15 downto 8);
    output_data_valid <= '1';
else
    output_data_valid <= '0';
end if;
```

## 5.2. Validation du filtre de Gauss




Le filtre de Gauss est validé car l'image de sortie est bien l'image d'entrée mais floutée.

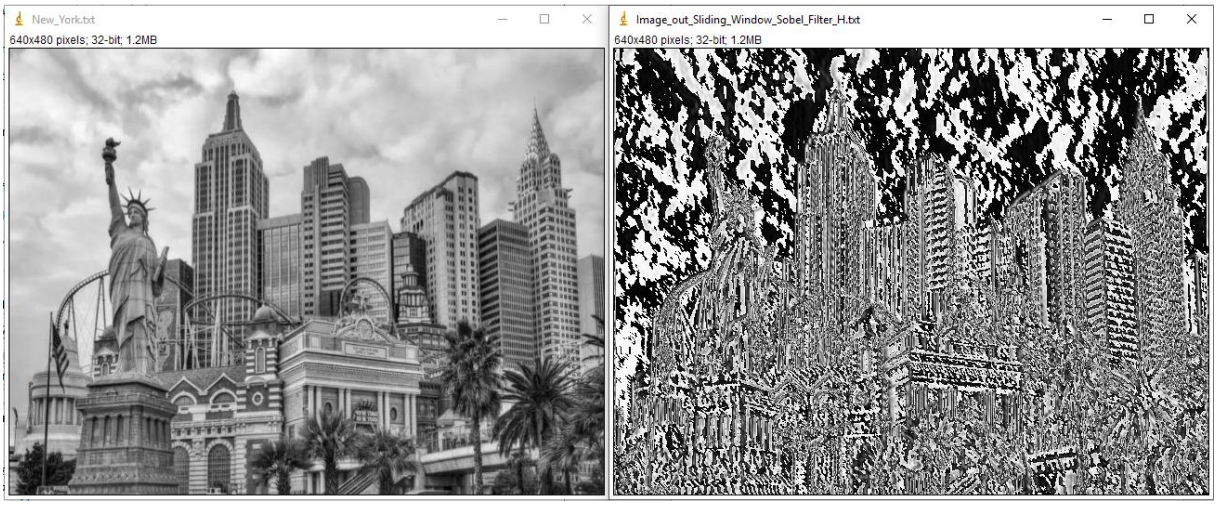
# 6. Processus du filtre de Sobel en Horizontal

## 6.1. Code VHDL

On reprend le code du processus qui gère le sliding window et on y ajoute le calcul du filtre de Sobel en position horizontale. Puis on affiche le résultat du calcul en le convertissant en vecteur de 8 bits.

```
if(Compteur_projet > 643) then
    Calcule_Sobel_H_Nat <= to_integer(signed(reg1))*K1_H_Nat + to_integer(signed(reg2))*K2_H_Nat + to_integer(signed(reg3))*K3_H_Nat +
    output_data <= std_logic_vector(to_unsigned(integer(Calcule_Sobel_H_Nat), 8));
    output_data_valid <= '1';
else
    output_data_valid <= '0';
end if;
```

## 6.2. Validation du filtre de Sobel en Horizontal




Avec ce filtre on récupère bien les positions horizontales de notre image d'entrée.
A ce stade du projet on ne peut rien dire de plus hormis que la formule fonctionne.

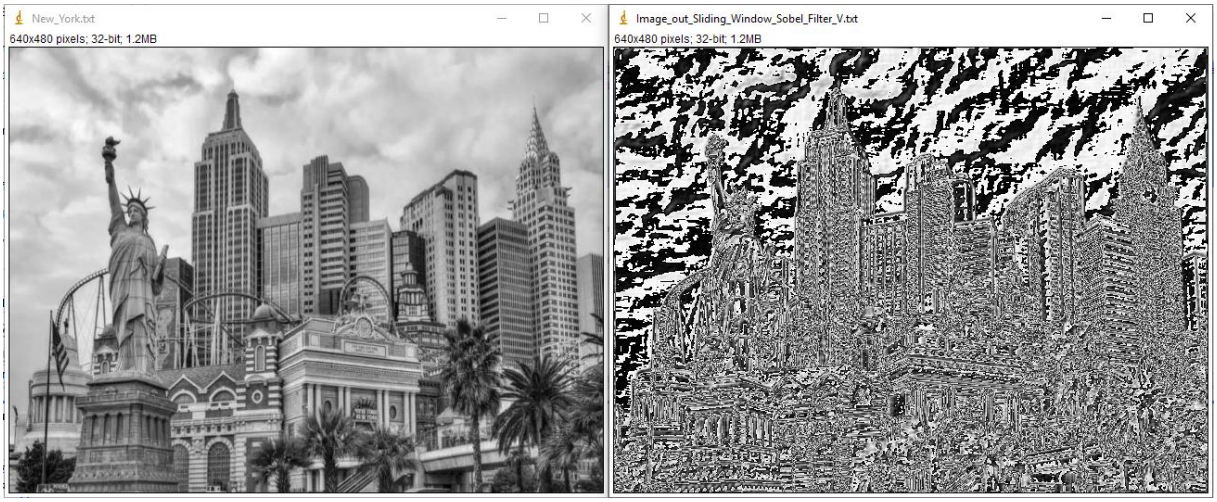
# 7. Processus du filtre de Sobel en Vertical

## 7.1. Code VHDL

On reprend le code du processus qui gère le sliding window et on y ajoute le calcul du filtre de Sobel en position verticale. Puis on affiche le résultat du calcul en le convertissant en vecteur de 8 bits.

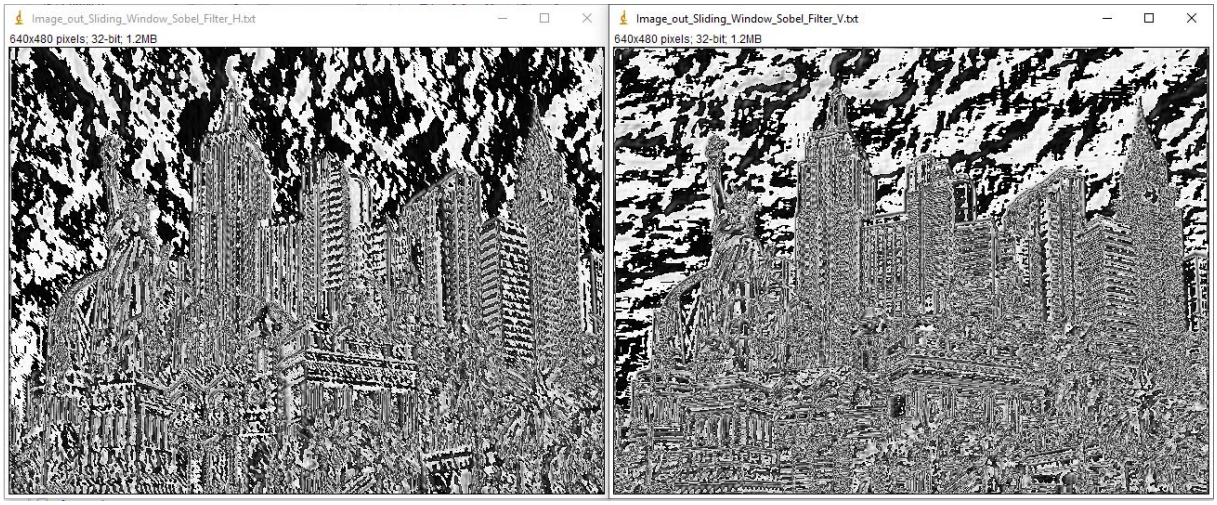
```
if(Compteur_projet > 643) then
    Calcule_Sobel_V_Nat <= to_integer(signed(reg1))*K1_V_Nat + to_integer(signed(reg2))*K2_V_Nat + to_integer(signed(reg3))*K3_V_Nat +
    output_data <= std_logic_vector(to_unsigned(integer(Calcule_Sobel_V_Nat), 8));
    --output_data <= std_logic_vector(to_signed((to_integer(signed(Calcule_Sobel_V))/4),8));
    output_data_valid <= '1';
else
    output_data_valid <= '0';
end if;
```

## 7.2. Validation du filtre de Sobel en vertical




Avec ce filtre on récupère bien les positions verticales de notre image d'entrée.
A ce stade du projet on ne peut rien dire de plus hormis que la formule fonctionne.

Si on compare le filtre en horizontal et en vertical, on peut voir une différence sur les points de détection. On le voit bien sur la phase des nuages, par exemple.



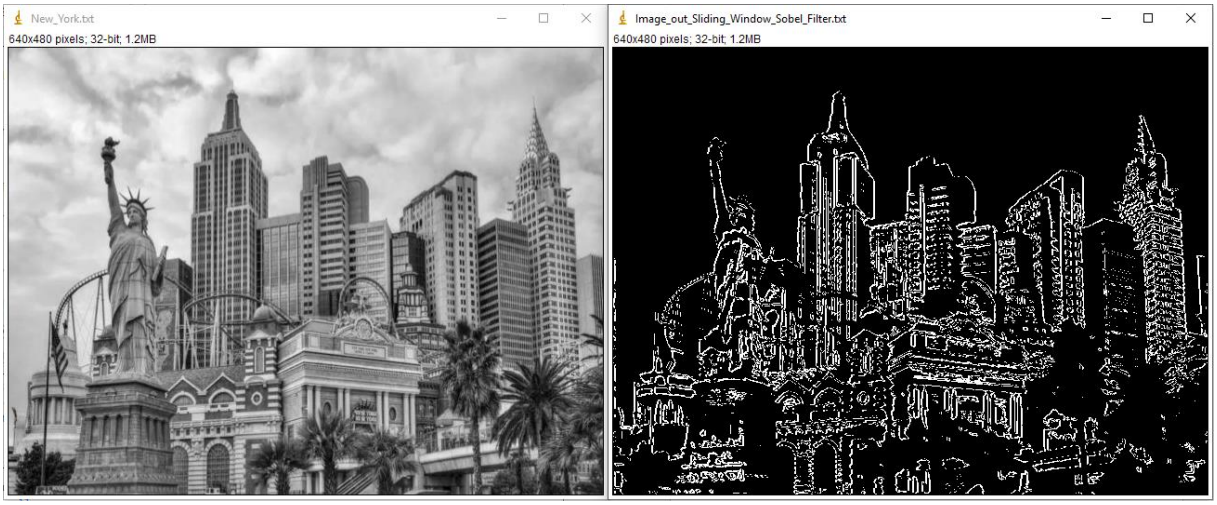
# 8. Processus du filtre de Sobel

## 8.1. Code VHDL

Dans le process de la mise en place du filtre de Gauss, on reprend les calculs en H et en V et on y ajoute la gestion du seuil.
La gestion du seuil se fait, soit à l'aide du calcul de la valeur absolue des résultats der H et V, soit en calculant la vraie formule avec la racine carré pour déterminer la valeur du gradient.

```
if(Calcule_Gradiant_Sobel_Nat > Seuil_Sobel) then
     output_data <= (others => '1');
else
     output_data <= (others => '0');
     --output_data <= reg5;
end if;
output_data_valid <= '1';
```

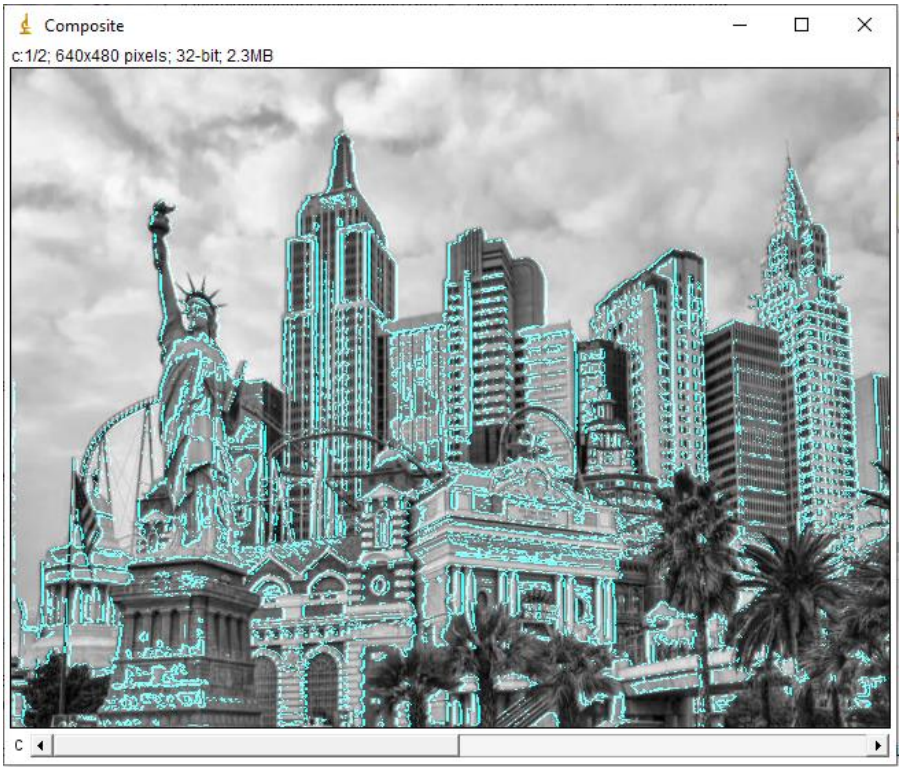
## 8.2. Validation du filtre de Sobel

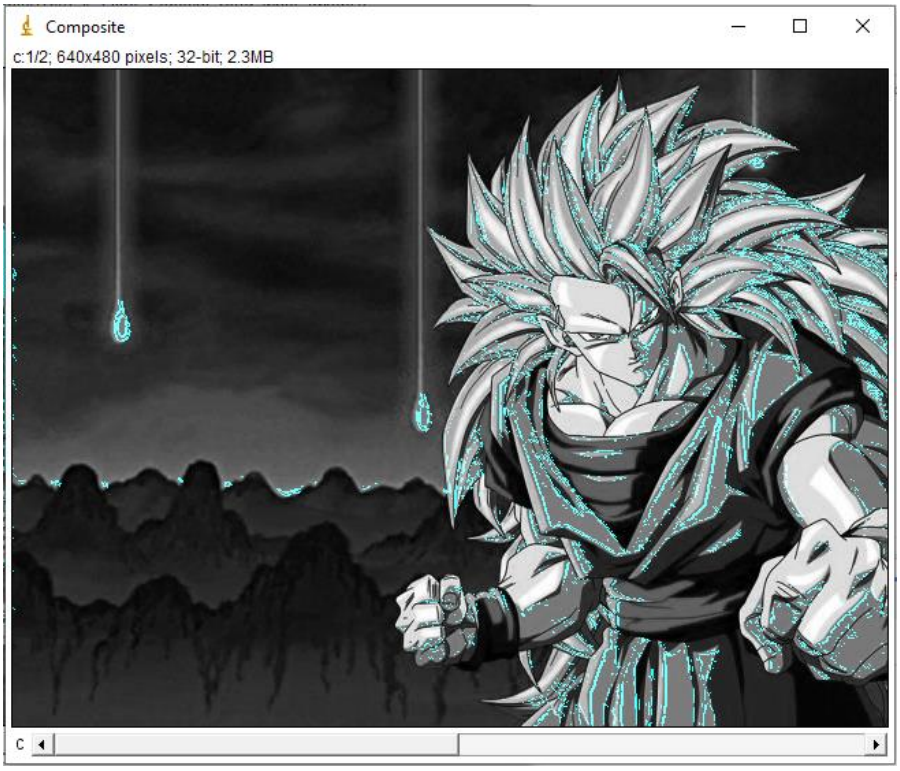



Ici on voit bien que le filtre de Sobel réalise bien la détection de contour dans notre image.

## 9. Utilisation du seuil

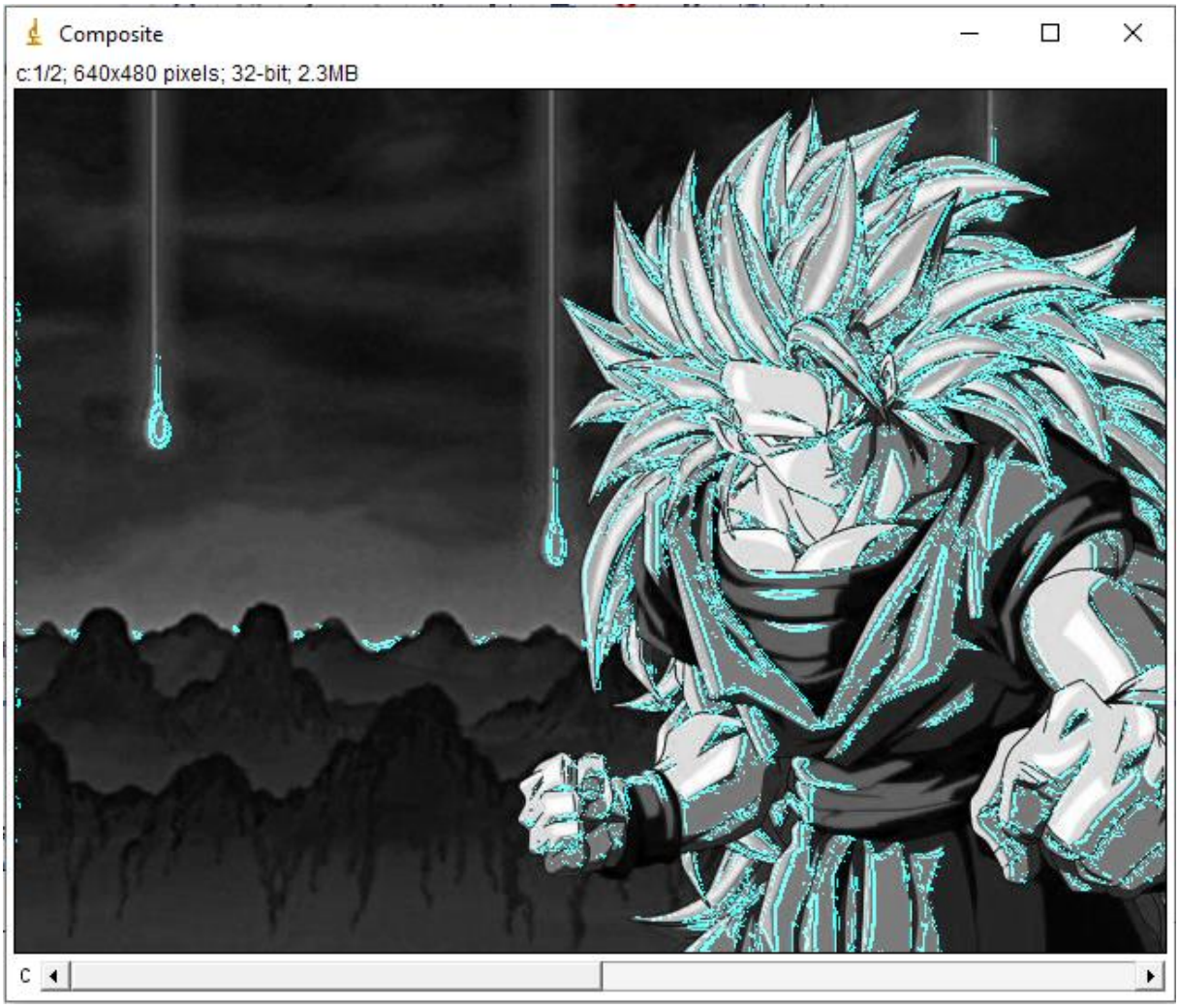
La détermination du seuil se fait en fonction de notre image d'entrée.
Par exemple pour un même seuil, deux images ne vont pas donner le même niveau de détection.



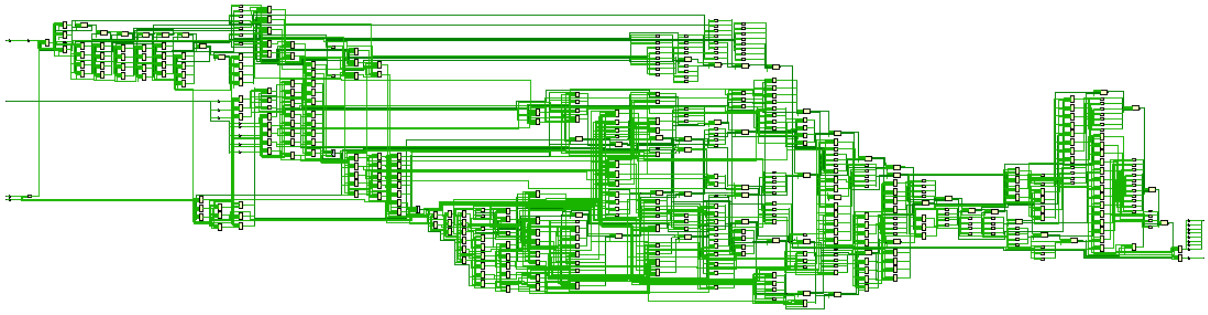

Sur les deux images précédente, le seuil est définit à 850. Si on veut accentuer la détection de la deuxième image, il faut descendre le niveau du seuil.

Voici un exemple avec un seuil à 650.



# 10. Synthèse de la mise en place du filtre de Sobel

Une fois la simulation du projet terminée, on réalise la synthèse de cette dernière pour avoir une estimation des ressources utilisées.



Dans notre architecture, on voit bien qu'on a 4 entrées et 2 sorties.

```
---------------------------------------------------------------------------------
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
      2 Input   10 Bit       Adders := 2
+---XORs :
      2 Input      1 Bit         XORs := 40
+---Registers :
                  10 Bit    Registers := 4
                   1 Bit    Registers := 4
+---Muxes :
      2 Input    1 Bit        Muxes := 1
---------------------------------------------------------------------------------
Finished RTL Component Statistics
---------------------------------------------------------------------------------
```

```
Report Cell Usage:
+------+---------+------+
|      |Cell     |Count |
+------+---------+------+
|1     |LUT1     |     5|
|2     |LUT2     |     6|
|3     |LUT3     |     5|
|4     |LUT4     |    24|
|5     |LUT5     |     4|
|6     |LUT6     |     6|
|7     |MUXCY    |    20|
|8     |RAMB18E1 |     1|
|9     |FDRE     |    40|
|10    |FDSE     |     4|
+------+---------+------+
```

Ici, on peut voir que notre architecture est composée d'additionneur, de portes logiques (XOR) de registres et de multiplexeurs.

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0,205 ns | Worst Hold Slack (WHS): | 0,091 ns | Worst Pulse Width Slack (WPWS): | 3,500 ns |
| Total Negative Slack (TNS): | 0,000 ns | Total Hold Slack (THS): | 0,000 ns | Total Pulse Width Negative Slack (TPWS): | 0,000 ns |
| Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 | Number of Failing Endpoints: | 0 |
| Total Number of Endpoints: | 451 | Total Number of Endpoints: | 451 | Total Number of Endpoints: | 261 |

**All user specified timing constraints are met.**

On peut voir que notre architecture ne possède pas de slack (malgré le nombre de cellule qu'utilise notre solution) donc on aura un fonctionnement réel correct.