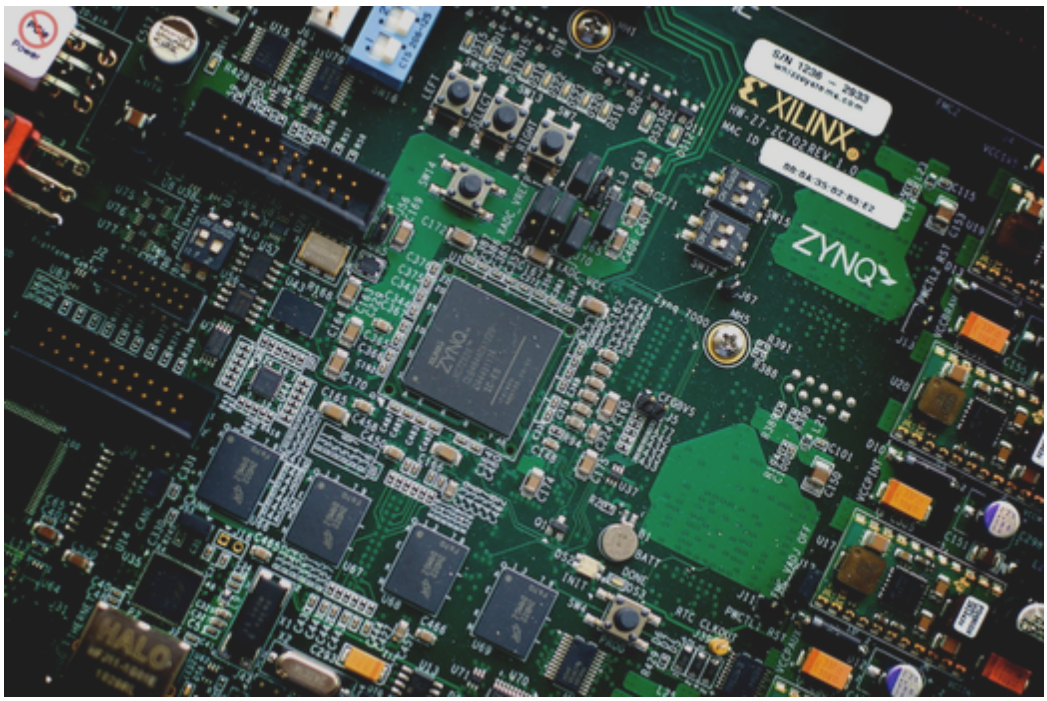
Rapport de TX

Accélération matérielle au moyen d'une architecture ARM+FPGA interconnectée



Par
Pierre AVITAL
Julien GRAVE

Sommaire

1. Introduction

Durant cette TX, il nous a été demandé de faire usage du System on Chip (SoC) Zynq-7000 de Xilinx, un système hybride intégrant un PS[1] ARM Cortex A9 ainsi qu'un circuit PL[2] FPGA, interconnectés de manière à ce que l'un puisse exploiter l'autre dans diverses applications.

Notre principal objectif était de nous familiariser avec la carte de manière à produire un document en permettant une utilisation plus aisée par de nouveaux utilisateurs, plus particulièrement concernant les capacités et modalités d'interactions entre le CPU et le FPGA.

Il nous a également été demandé d'étudier les capacités du FPGA à être programmé pour servir d'accélérateur matériel pour réduire la durée de calculs liés au traitement d'images, ainsi qu'alléger la charge de travail du processeur.

Nous nous sommes donc fixés comme objectifs d'implémenter quelques fonctions de traitement d'image de base, puis de les exécuter d'un côté de manière logicielle sur le CPU, et de l'autre de manière matérielle sur la partie FPGA, afin de comparer les performances de ces deux approches.

Plus particulièrement, nous avons commencé par implémenter le calcul de l'histogramme d'une image et le rehaussement de contraste, traitements usuellement préalables à tout autre traitement d'une image, avant d'implémenter une convolution 2D pour tester des traitements plus complexes.

1.1. Codes du rapport

Dans ce rapport, nous tacherons de détailler uniquement les parties de codes nous semblant pertinentes avec leur contexte, le reste des sources que nous avons utilisées étant disponible sur notre dépôt GitHub[3] et dont les liens complets et abrégés peuvent être trouvés en annexe B.

Aussi, tout nom de variable en verrouillage majuscule est un macro, défini dans un fichier dédié *define.h*, qui ne sera jamais détaillé dans le rapport. Les diverses acronymes employés au fil de ce rapport sont définis dans le glossaire, à la fin du rapport.

L'ensemble des bogues que nous avons rencontrées, ainsi que les solutions que nous y avons trouvées, peuvent être trouvés dans le chapitre débogage des erreurs. Ces erreurs étant communes, nous conseillons aux lecteurs de se référer à cette partie dès qu'une erreur apparaît dans leur implémentation.

[1] PS ou Processing System : cela désigne le système de traitement ARM Cortex A9. Nous nous référerons aussi à celui-ci sous les termes "CPU", "processeur" ou en reprenant des parties de son nom dans la suite de ce document pour abréger.
[2] PL ou Programmable Logic : PL est une abréviation qui désigne le circuit logique programmable FPGA.
[3] Adresse du dépôt : https://github.com/Darkin47/Zynq-TX-UTT

2. Configuration

2.1. Équipement

Pour réaliser ce projet, nous avons utilisé un ordinateur fonctionnant sous elementary OS version 0.3.2 (Freya), une distribution de Linux basée sur Ubuntu 14.04 (Trusty Tahr), mais mise à jour avec le kernel 4.4.0, supporté sur Ubuntu 16.04 (Xenial Xerus). Ainsi, il se peut que certaines parties de ce rapport, notamment concernant la configuration initiale, ne soient pas entièrement applicables à Windows et aux distributions de Linux très différentes d'Ubuntu.

2.2. Prérequis

Voici, au préalable, les installations qui sont nécessaires pour la réalisation du projet :

- Installer Vivado Design Suite dans sa version la plus récente.
- Installer OpenCV 2.4.x pour les simulations en C faites sur Vivado HLS.

2.3. Vivado Design Suite

2.3.1. Présentation de Vivado Design Suite

Vivado Design Suite est la suite de développement de Xilinx qui a pris la suite de ISE Web Design, qui est entré en "phase de maintien" en 2013 et n'a plus reçu de mises à jour depuis, pour la programmation des SoC Xilinx. Nous avons travaillé avec la version 2016.1.

Parmi les outils de cette suite, nous avons utilisé les logiciels suivants :

- Vivado est un environnement de développement permettant l'assemblage de blocs IP[4], qui pourront être implémentés sur le circuit FPGA. Il est notamment doté de nombreuses fonctions d'automatisation de la création des liens entre les broches de ceux-ci.
- Vivado HLS est un environnement de développement dédié à la programmation de blocs "IP", les composants pouvant être utilisés sur Vivado. Il supporte les langages C, C++ et SystemC. Nous utiliserons par ailleurs les termes "blocs IP" et "composants" de manière interchangeable dans ce rapport.
- Xilinx SDK (XSDK) est un environnement de développement voué à la programmation d'applications embarquées pour les microprocesseurs de Xilinx. À notre connaissance, cette programmation doit être faite en C/C++.

Nous recommandons fortement d'exécuter les logiciels Vivado, Vivado HLS et Xilinx SDK en mode "root" via la commande *sudo*. Pour faciliter cela, nous avons créé les alias suivants :

```
alias xsdk='sudo /opt/Xilinx/SDK/2016.1/bin/xsdk'
alias vivado='sudo /opt/Xilinx/Vivado/2016.1/bin/vivado'
alias vivado_hls='sudo /opt/Xilinx/Vivado_HLS/2016.1/bin/vivado_hls'
```

[4] Bloc IP : un bloc IP (Intellectual Property) est un composant codé en langage de description de matériel (Verilog, VHDL, …) qui décrit le comportement d'un circuit électronique numérique.

2.3.2. Connectiques et configuration des communications séries

Initialement, la carte Zynq est alimentée par le courant électrique via sa prise d'alimentation qui est fournie dans le Kit. Parmi la pléiade de connectiques mis à disposition de l'utilisateur, nous allons avoir besoin de relier notre ordinateur uniquement aux ports séries UART et JTAG, désignés respectivement en annexe A par les dénominations suivantes :

- USB-to-UART Bridge, USB Mini-B Connector.
- USB JTAG Module with integrated USB Mini-B Connector.

Le port UART va nous permettre de lire les communications sortantes de notre carte tandis que le port JTAG servira à implémenter notre assemblage, nos applications et à administrer notre carte FPGA.

La détection et l'utilisation des ports séries est presque entièrement transparente pour l'utilisateur sauf pour la lecture du port UART depuis le logiciel Xilinx SDK. Pour configurer celle-ci, voici la procédure à suivre :

1. Ouvrir un Terminal, entrer : *dmesg | grep cp210x*. Retenir à quel "ttyUSBx" est attaché le port série "cp210x".
2. Dans XSDK, afficher la console "SDK Terminal" accessible depuis le menu "Window->Show View->Other...".
3. Se connecter en cliquant sur l'icône vert "+" de la console "SDK Terminal" et puis en sélectionnant le port "ttyUSBx" de l'étape 1 et le Baud Rate "115200".

Cette configuration sera à effectuer chaque fois que vous lancerez le logiciel Xilinx SDK.

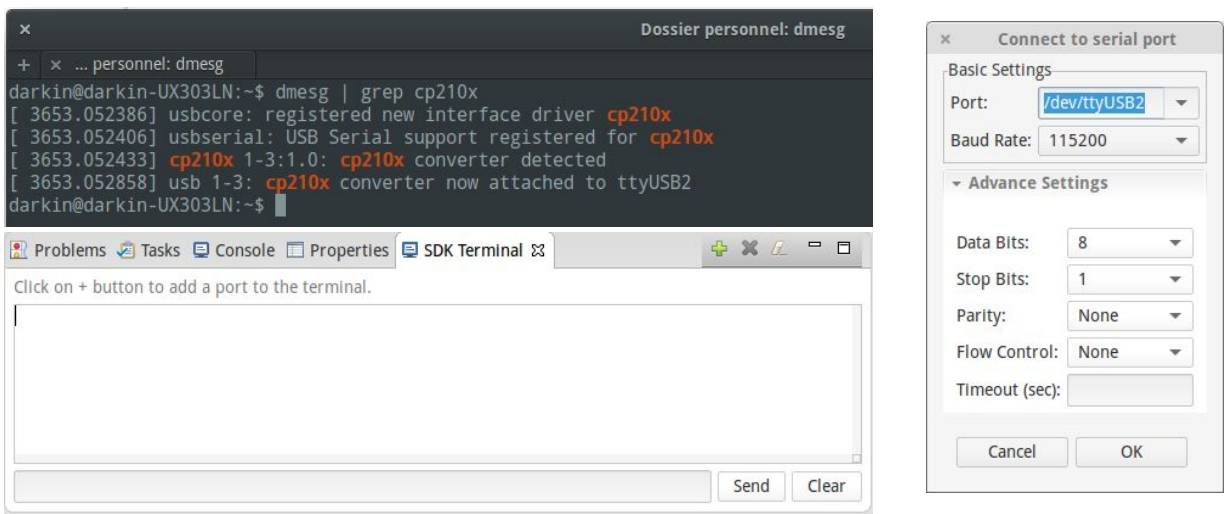


Figure 1 - Connexion au port série UART

Remarque : se référer au chapitre "Débogage des erreurs" en cas de soucis avec les communications.

3. Bases de la programmation pour Zynq

Dans cette partie, nous aborderons rapidement quelques concepts de base de la programmation pour Zynq, notamment la création de blocs IP personnalisés et leur mise en place sur la carte.

3.1. Vivado HLS et les blocs IP à partir du C/C++

Lors de cette TX, Vivado High Level Synthesis nous a permis de créer des composants IP à partir de fonctions C/C++, nous simplifiant la tâche par rapport à l'utilisation obligatoire de langages de description de matériel, tels que VHDL ou Verilog.

Cependant, cette utilisation de C/C++ ne se fait pas sans quelques particularités :

- La chose la plus importante à tenir en compte lors de la création d'un composant est que l'allocation dynamique est impossible dans ce contexte. Cela fait sens lorsqu'on tient en compte le fait que les composants sont finalement implémentés sous forme d'un ensemble de tables de correspondances et ont donc besoin de connaître à l'avance les espaces mémoires qu'ils devront visiter de manière à lire leurs paramètres d'entrée et écrire leurs sorties.
  Il reste cependant possible d'utiliser des formats de données dynamiques à travers les flux *hls_stream*. L'utilisation de ceux-ci doit cependant rester prudente, car nos expériences ont montré que la mémoire n'étant pas réellement allouée, des corruptions de mémoires et dépassements de tampons sont possibles.

- Les sorties sont par ailleurs impossibles à travers l'utilisation du *return* classique des fonctions C/C++. On remplace donc l'utilisation de fonctions typées par l'utilisation de fonctions *void* où les adresses où écrire les données de sorties sont passées en arguments sous forme de pointeurs.

- L'utilisation obligatoire de *pragmas*, à savoir des instructions spécifiques au compilateur pouvant être intégrées dans le code (par exemple, *#pragma once* permet de signaler au compilateur de Visual Studio de ne compiler qu'une seule fois un fichier le contenant). Dans Vivado HLS, les *pragmas* permettent principalement de signaler au compilateur qu'une variable a besoin d'être allouée dans un bus de communication de la manière suivante, de manière à ce que les variables puissent être consultées par le composant et par le CPU :
  #pragma HLS INTERFACE [bus] port=[variable]

- Il existe également des *pragmas HLS* visant à optimiser le composant pour la tâche qui lui sera attribuée, tels que *PIPELINE* qui signale au compilateur qu'une boucle peut être traitée de manière non-séquentielle, permettant de réduire les temps de calculs en lançant plusieurs itérations de la boucle en parallèle.

3.2. Intégration des composants avec Vivado

Vivado permet d'intégrer sur un schéma des blocs IP et de connecter leurs entrées et sorties afin de former un assemblage qui représente les parties PL et PS, et qui sera intégré à notre carte FPGA. En outre, Vivado offre la possibilité de paramétrer nos blocs

(fréquence, …) mais aussi d'éditer manuellement le code VHDL de ces derniers. Finalement, l'assemblage pourra être validé, simulé puis synthétisé sous forme d'un "Bitstream" qui sera implémenté sur la carte.

3.2.1. L'architecture ARM+FPGA interconnectée

Pour la réalisation de notre comparatif entre les performances du PS et de la PL, nous avons dû chaque fois assembler une architecture interconnectée au moyen de Vivado. Le principe d'une telle architecture est de pouvoir, d'un côté, exécuter notre application sur le PS, puis de l'autre, utiliser la PL pour accélérer "matériellement" certains traitements.

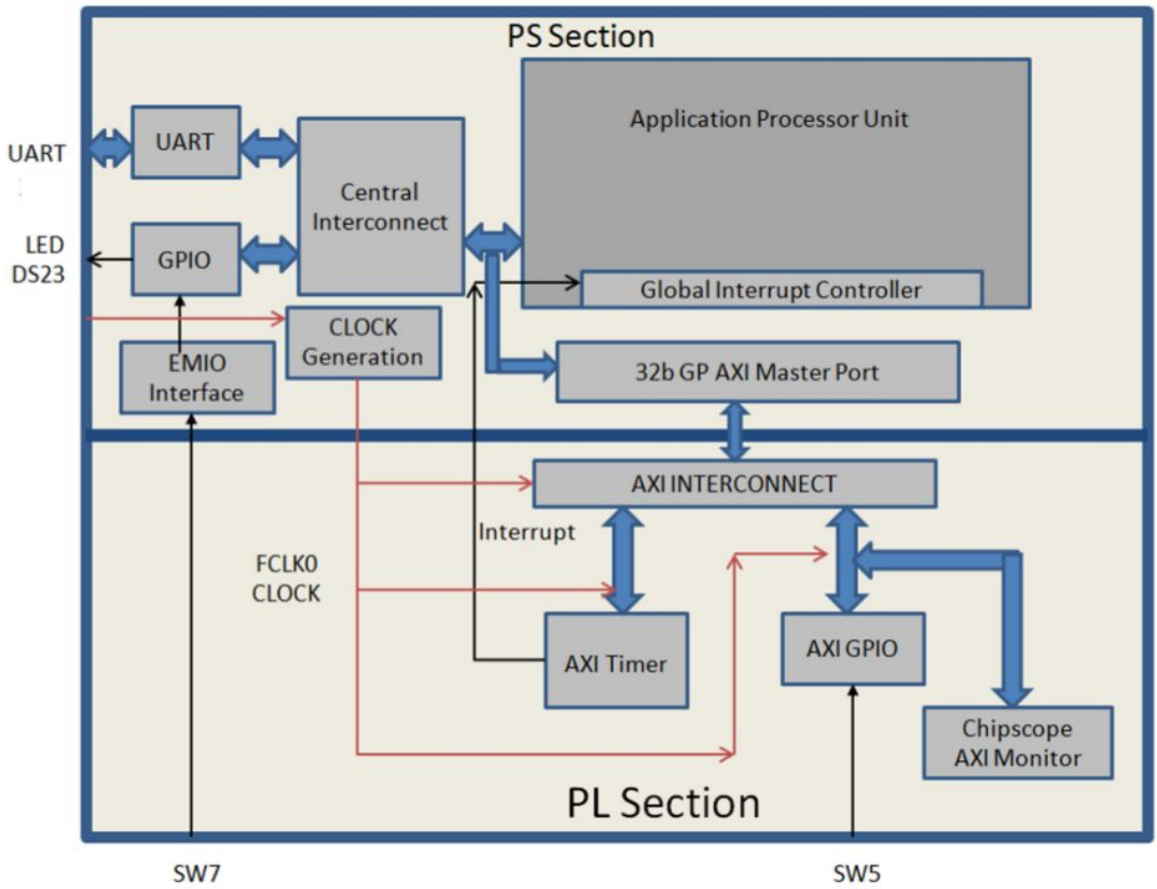


Figure 2 - Diagramme d'une architecture interconnectée

Pour interconnecter les sections PL et PS, il faut employer les périphériques AXI, tels que présentés sur le diagramme ci-dessus. Il existe plusieurs mises en place possibles des périphériques AXI, dépendant du type d'application à réaliser. Dans le cas du traitement d'image, nous allons réaliser les transferts par le biais du bloc AXI DMA, pour Direct Memory Acces, représenté par le périphériques "Central Interconnect" sur le diagramme. Son avantage est de transférer directement les données entre la PL et la mémoire DDR, et vice-versa, via le bus AXI *Interconnect*, ce qui correspond à l'usage haute performance requis pour un algorithme de traitement d'image produisant des flux continus de données.

Remarque : le bloc AXI VDMA, pour Video DMA, est, par exemple, le bloc requis pour les flux de données de type vidéo.

3.2.2. Assemblage basique d'une architecture interconnectée

Maintenant, pour illustrer, nous allons vous présenter un assemblage basique[5] d'une architecture interconnectée au moyen des périphériques AXI. Son comportement est simplement de boucler le flux de sortie du bloc AXI DMA à son flux d'entrée. Ainsi, nous allons effectuer une opération d'écriture puis de lecture vers et depuis la mémoire DDR, et on pourra alors observer que l'on récupère bien en lecture, le flux de données précédemment écrit. Il s'agit d'une reproduction de la fonction *memcpy()* du langage C.

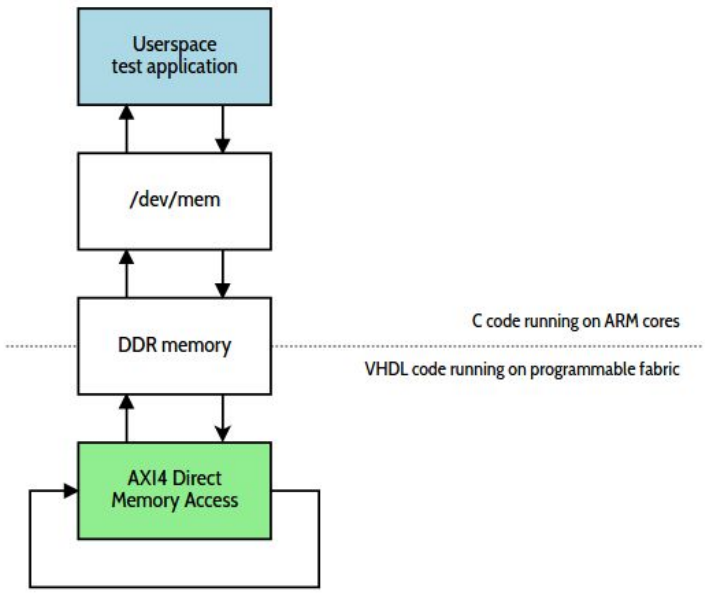


Figure 3 - Bouclage du flux de sortie au flux d'entrée du bloc AXI DMA

Voici, succinctement, les étapes de réalisation dans Vivado :

1. Créer un nouveau projet, de type RTL et avec pour carte cible la ZYNQ-7 ZC702.
2. Créer un "Block Design" et insérer le bloc IP ZYNQ7 puis cliquer sur "Run Block Automation". Paramétrer ZYNQ7 en ajoutant le port S_AXI_HP0, une horloge et un reset.
3. Insérer le bloc AXI *Direct Memory Access* puis cliquer sur "Run Connection Automation", une fois.
4. Terminer d'insérer les blocs AXI *Interconnect**2 et AXI *Protocol Converter*. Les relier (cf. figure 4).
5. Valider le design, générer son Bitstream et l'implémenter sur la carte.

[5] Cet assemblage est disponible en version complète et détaillée sur ce site : http://lauri.xn--vsandi-pxa.com/hdl/zynq/xilinx-dma.html

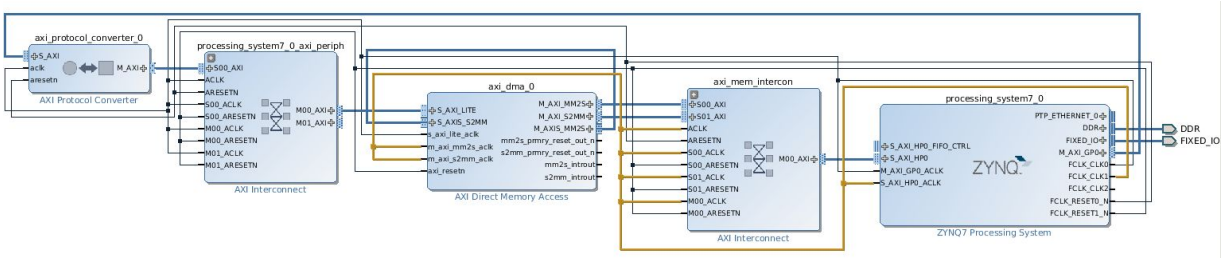


Figure 4 - Assemblage de blocs sur Vivado

Ainsi, nous avons réalisé un assemblage basique employant une architecture interconnectée. Nous verrons dans la partie suivante, comment utiliser XSDK pour implémenter une application sur une architecture créée et implémentée avec Vivado.

3.3. Implémentation d'une application avec Xilinx SDK

Xilinx SDK, troisième et dernier logiciel de la suite Vivado, permet de programmer une application en langage de haut niveau C/C++, qui sera compilée puis exécutée sur le processeur ARM de notre carte FPGA. Dans cette partie, nous allons présenter de manière générale comment programmer une application sur Xilinx SDK en employant l'architecture précédemment créée sur Vivado.

Avant de commencer, il faut impérativement lancer XSDK depuis la session Vivado où nous venons de concevoir notre assemblage, menu "File->Launch SDK". Remarque : il faudra d'abord avoir exporté l'Hardware, menu "File->Export->Export Hardware".

Pour appeler les fonctions implémentées sous forme de blocs IP et communiquer via la mémoire DDR, il faudra d'abord inclure la bibliothèque *xaxidma.h* et initialiser le bloc DMA, comme dans cet exemple :

```
#include "xaxidma.h"
XAxiDma axiDma; // Instantiate the AXI Direct Memory Access IP block
int initDMA() {
  XAxiDma_Config *cfgPtr;

  cfgPtr = XAxiDma_LookupConfig(XPAR_AXI_DMA_0_DEVICE_ID);
  XAxiDma_CfgInitialize(&axiDma, cfgPtr);

  // Disable interrupts
  XAxiDma_IntrDisable(&axiDma, XAXIDMA_IRQ_ALL_MASK, XAXIDMA_DEVICE_TO_DMA);
  XAxiDma_IntrDisable(&axiDma, XAXIDMA_IRQ_ALL_MASK, XAXIDMA_DMA_TO_DEVICE);

  return XST_SUCCESS;
}
```

Puis il faudra, pour chaque bloc créé avec Vivado HLS et intégré à notre assemblage, inclure leur bibliothèque et les initialiser, comme dans l'exemple suivant :

```
#include "xdoimgproc.h"
XDoimgproc doImgProc; // Instantiate our image processing block
int initDoImgProc() {
  int status;
  XDoimgproc_Config *doImgProc_cfg;

  doImgProc_cfg = XDoimgproc_LookupConfig(XPAR_DOIMGPROC_0_DEVICE_ID);
```

```
  if (!doImgProc_cfg) {
    printf("Error loading config for doHist_cfg\n");
  }

  status = XDoimgproc_CfgInitialize(&doImgProc, doImgProc_cfg);
  if (status != XST_SUCCESS) {
    printf("Error initializing for doHist\n");
  }

  return status;
}
```

Supposons maintenant, pour les besoins de notre exemple, que nous ayons intégré un bloc exécutant un produit de convolution, afin d'appliquer un filtre sur une image.

Pour initialiser les paramètres d'entrée de ce bloc, on utilisera les accesseurs mis à disposition dans la bibliothèque de notre bloc. Ensuite, on appel notre fonction puis on transfère le flux de pixel de notre image, et on patiente jusque la fin des transferts. Voici un extrait-exemple pour le bloc exécutant une convolution :

```
// Set the kernel and ask for a convolution
XDoimgproc_Write_kernel_Bytes(&doImgProc, 0, kernel, 9);
XDoimgproc_Start(&doImgProc);

axiTimer.startTimer(); // Start timer to evaluate processing time

// Do the DMA transfers to push and pull our image
Xil_DCacheFlushRange((u32)imgIn_HW, IMG_WIDTH * IMG_HEIGHT * sizeof(unsigned char));
Xil_DCacheFlushRange((u32)m_dma_buffer_RX,IMG_WIDTH * IMG_HEIGHT * sizeof(unsigned char));
XAxiDma_SimpleTransfer(&axiDma,
                       (u32)imgIn_HW,
                       IMG_WIDTH * IMG_HEIGHT * sizeof(unsigned char),
                       XAXIDMA_DMA_TO_DEVICE);
XAxiDma_SimpleTransfer(&axiDma,
                       (u32)m_dma_buffer_RX,
                       IMG_WIDTH * IMG_HEIGHT * sizeof(unsigned char),
                       XAXIDMA_DEVICE_TO_DMA);

// Wait for transfers to finish
while (XAxiDma_Busy(&axiDma, XAXIDMA_DMA_TO_DEVICE)) ;
while (XAxiDma_Busy(&axiDma, XAXIDMA_DEVICE_TO_DMA)) ;

// Invalidate the cache to avoid reading garbage
Xil_DCacheInvalidateRange((u32)m_dma_buffer_RX,
                          IMG_WIDTH * IMG_HEIGHT * sizeof(unsigned char));

axiTimer.stopTimer(); // Stop timer

double HW_elapsed = axiTimer.getElapsedTimerInSeconds();
printf("HW execution time: %f sec\n", HW_elapsed);
```

En résultat, on obtient la durée d'exécution du produit de convolution en *Hardware*[6]. Une donnée qui servira pour comparer avec le temps d'exécution de la partie *Software*[7].

[6] La partie Hardware désigne ce qui sera exécuté par la partie PL ou FPGA.
[7] La partie Software désigne ce qui sera exécuté par le PS ou processeur ARM.

Concernant la programmation d'instructions ne nécessitant pas l'utilisation d'accélération matériel via un composant du circuit, on pourra programmer celles-ci comme habituellement en C/C++.

Pour la compilation de notre code, on veillera à compiler notre code en mode *Release* pour optimiser le temps d'exécution de la partie *Software* : menu "Project->Build Configurations->Set Active->Default". Puis, recompiler tout, menu "Project->Build All".

Avant d'exécuter une application, il faut se connecter et lire le port série UART comme expliqué chapitre 2.3.2. Enfin exécuter l'application, menu "Run->Run As->Launch on Hardware (GDB)".

3.4. Récupération des données de sortie depuis la Zynq

La récupération des données depuis la carte se fait par le biais de la console XSDB, qui communique avec la carte par son port JTAG, en utilisant l'ensemble de commandes suivant :

```
xsdb% connect
xsdb% set logfile [open "fichier_de_log" "w"]
xsdb% puts $logfile [mrd -size b adresse_de_départ_DMA nombre_d'octets]
xsdb% close $logfile
```

Le fichier obtenu adopte la forme de *nombre_d'octets* lignes de la forme [*DMA_address: valeur_en_hexadécimal*].

Pour lire une image, nous avons utilisé cet ensemble de commandes, puis utilisé la commande `sed 's/^.*: *\([0-9A-Fa-f]*\).*$/\1/g' < in > out` et la commande `tr -d '\n' < in > out` pour supprimer les sauts de ligne, obtenant un fichier d'une ligne de valeurs hexadécimales sur 2 symboles. Ce fichier peut alors être transformé en image dans MATLAB avec la commande suivante :

```
f=fopen(fichier); image=uint8(reshape(fscanf(f,'%2x'),[width height])); fclose(f); clear f;
```

Cette méthode nous a permis d'accéder aux résultats de nos traitements d'images rapidement et avec très peu de programmation supplémentaire de la carte nécessaire, et nous l'avons donc utilisée systématiquement pour la récupération et vérification des résultats de nos tests sur la Zynq.

4. Traitement d'image

Tous les tests de cette partie ont été effectués avec l'horloge CPU réglée à 667MHz (sa fréquence maximale) et l'horloge de la PL à 150MHz (60% de sa fréquence maximale).

4.1. Histogramme

La réalisation d'un histogramme consiste simplement à compter les occurrences dans une image de toutes les valeurs possibles pour un pixel. Ce calcul permet de juger de l'exposition et du contraste d'une image, et permet par la suite des traitements de rehaussement de contraste.

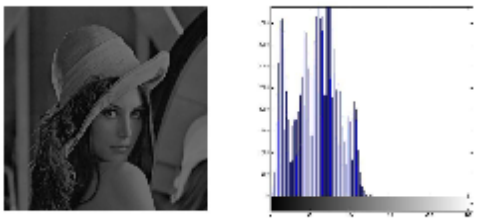


Image sous-exposée :
histogramme trop à gauche

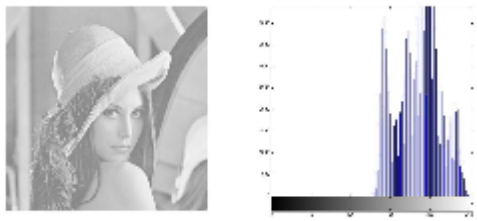


Image sur-exposée :
histogramme trop à droite

4.1.1. Implémentation

Comme décrit en §3.1, la différence la plus notable dans ce code par rapport à une implémentation normale d'un calcul d'histogramme est la présence de *pragmas* pour chaque argument.

core.cpp

```
#include "core.h"

void doHist(hls::stream<uint_8_side_channel>
          &inStream, int histo[PIXEL_RANGE])
{
#pragma HLS INTERFACE axis port = inStream
#pragma HLS INTERFACE s_axilite port = return
                            bundle = CTRL_BUS
#pragma HLS INTERFACE bram port = histo

  // Initialize histogram
  for (int idxHist = 0; idxHist < PIXEL_RANGE;
                                    idxHist++)
  {
    #pragma HLS PIPELINE
    histo[idxHist] = 0;
  }

  // Iterate over pixel stream of width*height
  for (int idxPixel = 0; idxPixel < (IMG_WIDTH
                    * IMG_HEIGHT); idxPixel++)
  {
    // Get pixel value
    uint_8_side_channel currPixelSideChannel =
                              inStream.read();

    // Increment corresponding histogram part
    histo[currPixelSideChannel.data] += 1;
  }
}
```

core.h

```
#include "define.h"

// Use the class stream
#include <hls_stream.h>

// Use the axi stream side-channel (TLAST, TKEEP,
                                        TUSR, TID)
#include <ap_axi_sdata.h>
typedef ap_axiu < 8, 2, 5, 6 > uint_8_side_channel;

void doHist(
    hls::stream <uint_8_side_channel>  &inStream,
    int  histo[PIXEL_RANGE]);
```

4.1.2. Résultats

Temps d'exécution sur le PS	2,84 ms
Temps d'exécution sur la PL	4,69 ms
Différence de performances	PL 65% plus lente

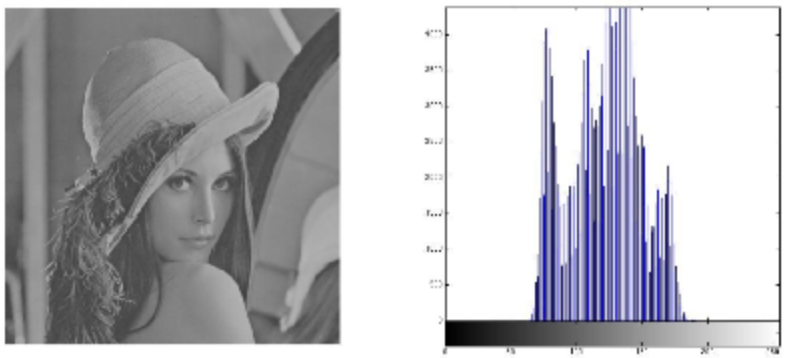
On observe ici que le PS reste largement plus rapide que la PL. Cette différence de performance peut se justifier par la cadence du processeur, largement supérieure à celle de la PL.

On observera cependant que le PS perdra grandement en efficacité pour le reste des traitements abordés, là où le FPGA retiendra des performances similaires tant que le même nombre de boucle est effectué.

On peut supposer que si le CPU a l'avantage ici, c'est parce que le PS comme la PL n'effectue qu'une opération par itération de la boucle, là où la PL effectuera ensuite N opération par itération, N étant le nombre d'instructions dans la boucle, alors que le PS devra traiter les instructions une à une.

4.2. Rehaussement de contraste

Il existe quelques manières de rehausser le contraste. Nous avons choisi d'implémenter un étirement de l'histogramme plutôt que l'égalisation de celui-ci pour des raisons de simplicité. Ci-dessous l'image d'origine ayant un contraste faible.



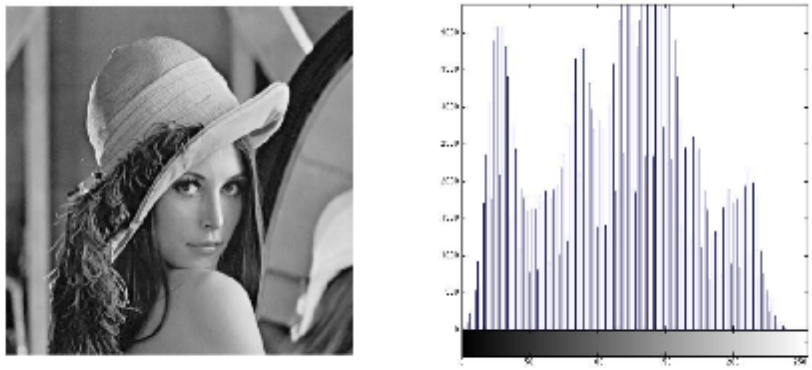
Histogramme compressé : contraste faible

4.2.1. Implémentation

L'implémentation de l'étirement d'histogramme d'une image pour bloc IP est très similaire à celle du calcul d'histogramme, nous ne la détaillerons donc pas ici. Vous trouverez le lien vers son code source en annexe B.

4.2.2. Résultats

Temps d'exécution sur le PS	14,16 ms
Temps d'exécution sur la PL	4,40 ms
Différence de performances	PL 68% plus rapide



Histogramme étiré : contraste plus élevé

Ici, on observe comme dit en §4.1.2 que la PL garde un temps d'exécution très similaire à celui de la réalisation d'histogramme pour une opération impliquant pourtant bien plus de calculs, d'où notre hypothèse que le temps de calcul pour la PL est essentiellement dépendant du nombre d'itérations de la boucle et non de son contenu. Ainsi, alors que le PS perd en performance, la PL maintient son temps de calcul et devient donc largement plus efficace.

4.3. Convolution 2D et détection de bordures

La convolution 2D est l'une des opérations les plus utilisées pour le traitement d'images.

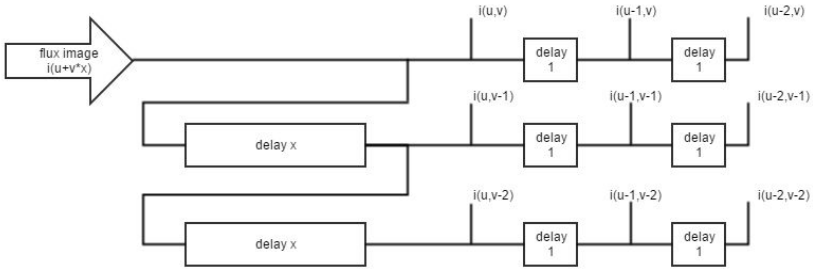
$$conv_{image,noyau}(i,j) = \sum_u \sum_v image(i-u, j-v) \times kernel(u,v)$$

Elle permet d'appliquer, à travers l'utilisation de divers noyaux (ou filtres), des effets sur une image tels qu'un flou ou, dans notre cas, une détection des bordures. Nous avons implémenté notre détection de bordures par l'utilisation de filtres de détection de bordure, avec un noyau 3x3 de centre 8 et de périphérie -1.

$$kernel = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

4.3.1. Implémentation

Les images étant représentées sous forme de flux unidimensionnels *hls_stream*, nous avons eu recours à des tampons *hls_LineBuffer* de hauteur *n* qui permettent de retenir les *n* dernières lignes de l'image lues dans le flux, *n* étant la hauteur du noyau. Ces lignes du *LineBuffer* peuvent être représentées schématiquement par un *delay* de *x* octets dans la lecture du flux, *x* étant la largeur de l'image en octets.



Le composant extrait de cette manière la fenêtre de l'image, qui peut alors être multipliée par le noyau puis sommée pour obtenir la valeur de sortie du pixel (u, v).

Une fois la valeur de la convolution pour le pixel (u, v) calculée, celle-ci est simplement ajoutée au flux de sortie du bloc IP.

4.3.2. Résultats

Temps d'exécution sur PS	37,22 ms
Temps d'exécution sur PL	17,54 ms
Différence de performances	PL 52% plus rapide

Similairement au cas de l'étirement d'histogramme, la PL maintient ici un fort avantage sur le PS. On peut donc affirmer que l'utilisation de FPGA pour le calcul de boucles comprenant un grand nombre d'instructions peut offrir un gain significatif de performance, doublant la vitesse de traitement malgré une fréquence d'horloge largement inférieure.

5. Analyse des performances

5.1. Les performances de l'accélération matérielle

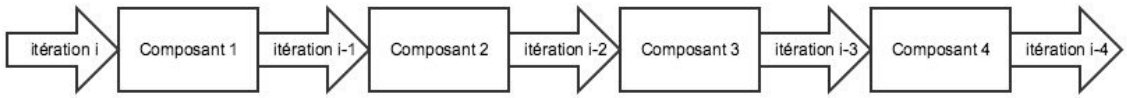
Lors de nos expérimentations avec la carte physique, nous avons observé de manière consistante que les temps de calcul sur la PL étaient largement plus rapides que ceux sur le PS dès que de grands groupes d'instructions en boucle étaient impliqués.

Si le résultat pour la réalisation de l'histogramme peut paraître décourageant, la PL restant derrière le CPU en terme de vitesse de traitement pure, la PL montre un gain significatif de performance pour des traitements plus complexes.

On peut ajouter que ces résultats ont été mesurés dans le cas d'un CPU investissant la totalité de ses ressources dans les calculs effectués, alors qu'il devrait effectuer d'autres processus dans la plupart des applications, ce qui réduirait son efficacité, accentuant les gains de performance offerts par un FPGA, même si la fréquence d'horloge de celui-ci devait être fortement inférieure à celle du processeur.

5.2. Le traitement en pipeline des instructions en boucle : la clé des performances du FPGA ?

Le traitement en pipeline des informations d'une boucle, mis en exergue par le pragma *HLS PIPELINE* décrit en §3.1, peut être vu comme la clé des performances du FPGA : le fonctionnement en pipeline implique que les données sont passées sous forme d'un flux continu à travers le tube d'instructions qu'est la boucle. Ainsi, en dehors des extrêmes, on peut donc considérer qu'au début de chaque itération i, les itérations i à i-n sont traitées simultanément, n étant le nombre de composants servant à représenter les instructions de la boucle.



En supposant que toute instruction prend un cycle d'horloge à compléter, tant pour le processeur que pour le FPGA par la représentation d'une instruction par un composant, et qu'une boucle a N itérations de n instructions, un FPGA finira donc d'itérer à travers une boucle en N+n cycles d'horloge alors qu'un CPU monocœur prendra N*n cycles.

5.3. Pistes à explorer : la transformée de Fourier

Par manque de temps, nous n'avons pas pu explorer les capacités et performances d'un FPGA pour des traitements tels que la transformée de Fourier. Cette dernière restant un des traitements de signal les plus communs, il pourrait être intéressant d'étudier les performances et coûts d'une accélération matérielle de ce calcul à travers un FPGA.

Il est possible que les gains obtenus par la parallélisation des opérations en boucle offrent un avantage significatif dans le cas de la FFT de Cooley-Tuckey. Reste cependant à voir si un composant IP peut ou non offrir de bonnes performances dans le cas d'un algorithme récursif.

6. Conclusion

À travers ce projet, nous avons découvert le SoC Zynq-7000, et avons eu l'occasion d'apprendre à configurer la suite Vivado, le groupe d'environnements de développements conçus par Xilinx pour programmer ce dernier. Nous avons pu acquérir une certaine aisance avec ce groupe d'outils, et avons appris à en gérer les particularités, comme vu dans les sections 3 et 7. Notamment, nous avons appris à connecter et assurer la communication entre les parties PS et PL du SoC, nous permettant de développer des applications hybride capable de prendre avantage des deux parties de ce matériel.

Par l'implémentation de quelques outils mathématiques de traitement d'image, nous avons pu non seulement montrer les capacités d'un FPGA pour le traitement de données, mais aussi, par l'analyse des temps de calcul et un raisonnement sur leur justification et sur certains des paramètres passés au compilateur, arriver à une conclusion sur ce que nous estimons être la cause principale de la différence de performances entre un FPGA et un CPU pour le traitement de données à travers une boucle.

Nous avons ainsi obtenu un ensemble de compétence relativement peu commun dans notre formation, et espérons avoir produit un document qui saura guider nos lecteurs pour la réalisation de leurs propres projets impliquant l'utilisation du SoC Zynq-7000.

Nous estimons cependant qu'il reste un aspect intéressant de ce SoC que nous n'avons pu explorer durant ce projet malgré nos premières ambitions : celui de l'ajout d'interfaces pour le processeur par leur implémentation en PL et l'utilisation des entrées/sorties de la carte qui nous a été fournie. En effet, nous pensons que par la programmation de circuits logiques et l'ajout de quelques composants électroniques, cette carte pourrait être un outil intéressant pour le prototypage d'interfaces telles qu'une carte son en facilitant l'itération sur les designs.

7. Débogage des erreurs

7.1. Vivado High Level Synthesis

Les simulations en C faites sur Vivado HLS des fonctions de traitement d'image peuvent échouer si l'installation de OpenCV n'est pas complète. Il faut notamment s'assurer de la présence des paquets-librairies suivants : *libjpeg62-dev*, *libtiff5-dev* et *libtiff3*.

libtiff3 devra aussi être téléchargé, compilé et installé manuellement, comme suit :

```
$ wget http://download.osgeo.org/libtiff/tiff-3.9.7.zip
$ unzip tiff-3.9.7.zip
$ cd tiff-3.9.7
$ ./configure && make
$ sudo cp libtiff/.libs/libtiff.so.3.9.7 /usr/lib/
$ ln -s /usr/lib/libtiff.so.3.9.7 /usr/lib/libtiff.so.3
```

Sources : forums.xilinx.com [1] [2]

7.2. Xilinx SDK

Lors des communications sur le port série UART, XSDK a planté à plusieurs reprises. Il s'est avéré que la version de la librairie *librxtxSerial.so* qui était fournie avec Vivado Design Suite n'était pas supportée par notre système, nous avons donc remplacé celle-ci avec celle de notre système, comme suit :

```
$ sudo apt-get install librxtx-java
$ sudo mv
/opt/Xilinx/SDK/2016.1/eclipse/lnx64.o/plugins/gnu.io.rxtx.linux.x86_64_2.1.7.3_v20071015/
os/linux/x86_64/librxtxSerial.so
/opt/Xilinx/SDK/2016.1/eclipse/lnx64.o/plugins/gnu.io.rxtx.linux.x86_64_2.1.7.3_v20071015/
os/linux/x86_64/librxtxSerial.so.ORIG
$ sudo ln -s /usr/lib/jni/librxtxSerial.so
/opt/Xilinx/SDK/2016.1/eclipse/lnx64.o/plugins/gnu.io.rxtx.linux.x86_64_2.1.7.3_v20071015/
os/linux/x86_64/librxtxSerial.so
```

Source : forums.xilinx.com

8. Liste des acronymes

ARM	Advanced RISC Machine
AXI	Advanced eXtensible Interface
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Array
HLS	High Level Synthesis
IP	Intellectual Property
JTAG	Joint Test Action Group
OS	Operating System
PS	Processing System
PL	Programmable Logic
SDK	Software Development Kit
SoC	System on Chip
UART	Universal Asynchronous Receiver/Transmitter
VHDL	VHSIC Hardware Description Language
XSDB	Xilinx System Debugger

9. Bibliographie

[1] About post. Dans : Lauri's blog [en ligne]. [Consulté le 22 Juin 2016]. Disponible à l'adresse : http://lauri.xn--vsandi-pxa.com/hdl/zynq/xilinx-dma.html

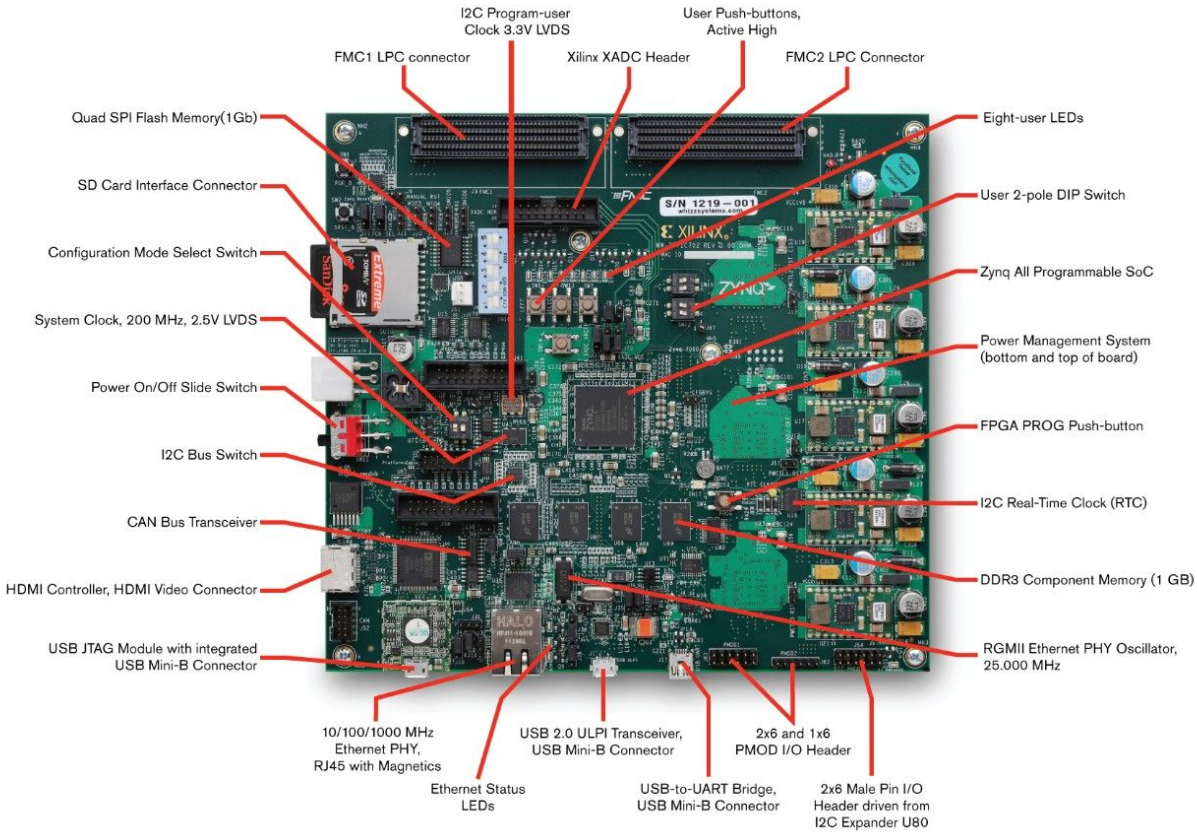
[2] zynq_cours_tp_vivado.pdf. Dans : Conservatoire National des Arts et Métiers [en ligne]. [Consulté le 22 Juin 2016]. Disponible à l'adresse : http://easytp.cnam.fr/alexandre/index_fichiers/support/

[3] Documentation Xilinx. Dans : Support Xilinx [en ligne]. [Consulté le 22 Juin 2016]. Disponible à l'adresse : http://www.xilinx.com/support.html#documentation

[4] http://www.youtube.com/channel/UC1ptV25-NEHRIEnM1kXMCrQ. The Development Channel. Dans : YouTube [en ligne]. YouTube. [Consulté le 20 Juin 2016]. Disponible à l'adresse : https://www.youtube.com/channel/uc1ptv25-nehrienm1kxmcrq

10. Annexes

Annexe A - Xilinx Zynq-7000 SoC ZC702



Annexe B - Sources du projet

- Page d'accueil du dépôt : https://github.com/Darkin47/Zynq-TX-UTT
- Code source bloc histogramme : https://github.com/Darkin47/Zynq-TX-UTT/tree/master/Vivado_HLS/image_histogram/src
- Code source bloc rehaussement contraste : https://github.com/Darkin47/Zynq-TX-UTT/tree/master/Vivado_HLS/image_contrast_adj/src
- Code source bloc convolution 2D : https://github.com/Darkin47/Zynq-TX-UTT/tree/master/Vivado_HLS/convolution_2D/src
- Code source application histogramme + rehaussement contraste : https://github.com/Darkin47/Zynq-TX-UTT/tree/master/Vivado/Hist_Stretch/Hist_Stretch.sdk/ContrastApp/src
- Code source application convolution 2D : https://github.com/Darkin47/Zynq-TX-UTT/tree/master/Vivado/image_conv_2D/image_conv_2D.sdk/testImgProc/src