



Sorbonne Université

**Réalisation d'une architecture
hétérogène sur FPGA
avec le projet PYNQ**

Projet SAR

Encadrant :
Damien Fruleux

Auteurs :
Harena Rakotondratsima
Florent Bordas
Myriam Mabrouki

Numéros d'étudiant :
28707195
21308825
28710344

Mai 2024

Table des matières

1	Introduction	2
2	Définitions	3
2.1	FPGA	3
2.2	CPU	4
2.3	Systèmes d'exploitation embarqués	4
3	Prise en main d'un système d'exploitation embarqué	8
3.1	Génération d'une distribution Linux sur un softcore	8
3.2	Distribution PYNQ	10
3.3	Interfaçage du système d'exploitation PYNQ et du FPGA	11
4	Architecture hétérogène : une accélération matérielle	12
4.1	Algorithme réalisé dans le <i>Processing System</i>	13
4.2	Algorithme réalisé dans le <i>Programmable Logic</i>	14
5	IP personnalisé : seuillage d'image en niveau de gris	17
5.1	Réalisation de l'IP	17
5.2	Interfaçage avec les ports GPIO	21
6	Ouverture : seuillage d'un flux vidéo externe	23
7	Conclusion	27

1 Introduction

Les systèmes embarqués sont des systèmes dédiés à une fonction précise. La plupart du temps, ils sont soumis à des contraintes (taille, poids, consommation d'énergie...). Par ailleurs, bon nombre d'entre eux sont des systèmes critiques temps-réel. Cela signifie que leurs tâches sont soumises à des échéances temporelles qu'elles se doivent de respecter au risques de failles importantes. Dans ce contexte, l'utilisation d'architecture hétérogène est de plus en plus fréquente.

Pour bien comprendre de quoi il s'agit, définissons ce qu'est une architecture hétérogène. Une architecture hétérogène, chez Xilinx (AMD), consiste à interconnecter une partie logicielle, que l'on appelle *Processing System* (PS), avec une partie matérielle reprogrammable que l'on appelle *Programmable Logical* (PL) [10]. On utilise des méthodes de *co-design* pour proposer des telles architectures.

Le PS et le PL sont tous les deux des circuits numériques intégrés. Un circuit numérique intégré est un composant électronique de petite taille, basé sur un semi-conducteur et enfermé dans un boîtier qui possède des contacts électriques le tout dans le but de réaliser des fonctions électroniques plus ou moins complexes [10]. La différence entre ce qu'on appelle PS et PL est que, après leur fabrication, le PS, ou CPU, est conçu pour être programmé, tandis que le PL, ou FPGA, est conçu pour être reconfiguré [10]. Ainsi, dans un CPU on a un jeu d'instructions fixe qui peut être utilisé pour réaliser n'importe quel algorithme tandis que dans un FPGA on va réaliser une architecture matérielle spécifique à une application.

Si dans un FPGA, on a une architecture matérielle plus adaptée à une application spécifique, il faut noter que le temps de développement est bien plus long que celui d'un CPU qui sera utilisé dans beaucoup plus de types d'applications. Il est alors intéressant d'utiliser un FPGA pour certaines applications spécifiques qui nécessitent un temps de traitement plus long. En particulier, on a souvent recourt au FPGA afin de paralléliser plusieurs tâches. Il peut se comporter comme un GPU, un CPU dédié au calcul graphique, dans les architectures modernes, mais, à la différence d'un GPU, on peut reconfigurer le FPGA pour se spécialiser dans une autre application. Le GPU, à l'inverse, restera spécifique au traitement graphique et ses dérivés et ne pourra être reconfiguré pour d'autres usages.

Maintenant que nous avons compris le concept d'une architecture hétérogène, présentons le projet avec lequel nous allons travailler. Le projet PYNQ est un projet open-source qui simplifie l'utilisation des systèmes Zynq via l'usage du langage Python [5]. Les systèmes Zynq sont des systèmes sur puce (SoC), produit par Xilinx (AMD), qui intègrent processeurs et FPGA au sein d'un même circuit. Grâce à ce projet, il est alors possible faire de nombreux *overlays*, c'est-à-dire des bibliothèques matérielles de circuits logiques reconfigurable que l'on peut appeler telle une bibliothèque logicielle. Ainsi, les *overlays* peuvent être utilisés par des concepteurs de systèmes embarqués dans leurs applications sans que ces derniers n'aient besoin de les concevoir eux-mêmes, et facilite le test de plusieurs architectures.

Notre objectif dans ce projet est d'utiliser ce projet PYNQ pour exploiter une architecture hétérogène et ainsi accélérer une application. On pourra alors mesurer ce facteur d'accélération. Notre étude de cas portera sur du traitement de signal.

Tout d'abord, nous allons commencer par définir des termes essentiels – tels que FPGA, CPU ou système d'exploitation embarqué – pour comprendre le projet. Ensuite, nous verrons la réalisations d'un système d'exploitation embarqué. Nous continuons par voir l'accélération matérielle qu'il est possible d'obtenir depuis le PL avec un circuit prédéfini. Après cela, nous enchaînerons en voyant toujours l'accélération matérielle mais cette fois-ci depuis un circuit que nous avons nous-mêmes défini. Le circuit en question réalise le seuillage d'une image. Enfin, nous verrons ce que nous aurions aimé faire pour aller plus loin dans ce projet si nous avions davantage de temps. En l'occurrence, il aurait été question de développer notre circuit non plus pour une image, mais pour un flux vidéo.

2 Définitions

2.1 FPGA

Si l'on sait désormais qu'un FPGA est conçu pour être reconfiguré dans le but d'avoir une architecture matérielle plus adaptée à une application spécifique, il reste néanmoins essentiel de savoir comment il fonctionne réellement. Expliquons donc ce qu'il en est.

Un FPGA est un circuit composé de cellules, d'entrées/sorties et de logique de routage. Détaillons chacun des trois composants.

Une cellule est composée d'un *Look-Up Table* (LUT) et de bascules. Une bascule est un circuit électronique de base, composé de deux inverseurs, permettant de stocker une donnée sur un bit. De ce fait, les bascules nous permettent de mémoriser des données. Une bascule correspond à un point de mémoire. Un LUT, quant à lui, est composé d'un multiplexeur et de plusieurs points de mémoire. Il sert à réaliser la table de vérité d'une fonction booléenne spécifique. Un multiplexeur est un circuit électronique de base permettant de choisir une entrée parmi plusieurs pour la transmettre en sortie. Grâce aux points de mémoire et au multiplexeur on peut alors réaliser la table de vérité puisqu'on précalcule et on stocke toutes les valeurs de la fonction booléenne. Les entrées reçues dans le LUT servent alors à sélectionner, grâce au multiplexeur, les valeurs stockées dans nos points de mémoire que l'on transmet en sortie.

Une entrée/sortie peut représenter une interface entre le *Processing System* et un périphérique. Un périphérique est un dispositif externe au PS qui reçoit et/ou envoie des données depuis/vers ce dernier.

Enfin, la logique de routage désigne la façon dont sont interconnectées les cellules.

Maintenant, voyons comment nous allons procéder tout au long du projet pour reconfigurer notre FPGA. Pour y parvenir, nous utiliserons Vivado. Vivado est un logiciel permettant notamment la description matérielle de composants électroniques via des langages de description matérielle (HDL). Dans ce projet, c'est ce logiciel que nous allons utiliser pour réaliser des *Intellectual Properties* qui feront partie de nos *overlays*. Un *Intellectual Property* (IP) est un circuit prédéfini. Il est destiné à être ajouté comme partie d'un système. Tout au long de ce projet, nous concevrons des IPs réalisant des fonctions précises. Une fois que notre architecture matérielle est prête, Vivado nous permet de générer une séquence de bits, aussi appelée *bitstream*, contenant toutes les informations de l'architecture conçue et qui permettra la reconfiguration du FPGA. Pour chaque projet Vivado qu'on crée, on choisit le SoC pour

laquel il va générer le *bitstream*, et lors de la création des blocs design, on aura la possibilité d'utiliser les IPs qui sont fournis avec la carte choisit.

Lorsque nous reconfigurons notre FPGA, nous avons alors des fonctions booléennes dans les LUT spécifiquement adaptées à l'application que nous souhaitons réaliser. De fait, c'est cette architecture matérielle qui nous donnera les meilleures performances pour une application donnée. Toutefois, bien que l'architecture définie soit la plus performante pour une application donnée, il va sans dire que nous ne pouvons pas réaliser toutes les architectures possibles et inimaginables. En effet, en plus du temps de développement, nous sommes limités par le nombre de cellules et d'interconnexions que possède le FPGA.

Ainsi, nous savons maintenant comment fonctionne un FPGA et comment nous allons le reconfigurer.

2.2 CPU

Le processeur, ou CPU, de l'acronyme anglo-saxon *Central Processing Unit*, désigne l'unité de traitement ou microprocesseur principal d'un ordinateur [8]. Celui-ci a pour vocation à faire tourner n'importe quel algorithme en exécutant une suite d'instructions.

Nous pouvons classer les processeurs selon deux familles : les processeurs *hard core* et les processeurs *soft core*. Les premiers sont identiques aux processeurs traditionnels et donc gravés dans le même circuit intégré que le FPGA. Les seconds, quant à eux, sont fournis sous la forme d'une description de langage matériel, et donc à implémenter dans le FPGA. Si les processeurs *soft core* permettent une paramétrisation du processeur assez large (fréquence, instantiation d'un grand nombre de coeurs, ajout d'instruction personnalisées...), ils restent tout de même limités par l'architecture interne du FPGA à cause, notamment, du nombre de cellules ou des ressources de routage. Par ailleurs, leur temps de développement peut être beaucoup plus important que pour les processeurs *hard core* dont l'architecture est figée.

2.3 Systèmes d'exploitation embarqués

Certains systèmes embarqués n'ont pas besoin d'un système d'exploitation, car ils sont dédiés à une tâche bien précise, tandis que d'autres ne peuvent pas se le permettre en terme de ressources. Par exemple, certains n'ont pas de *Memory Management Unit* (MMU). Le MMU est un composant matériel qui gère l'accès mémoire, notamment la translation d'adresse virtuelle en adresse physique. D'autres ont un trop faible quantité de RAM, comme c'est le cas avec le microcontrôleur LPC 1768 qui a un processeur ARM Cortex-M3 mais seulement 64KB de RAM et 512KB de mémoire flash. Ces systèmes sont dits *bare metal* car ils exécutent des instructions sans passer par l'intermédiaire d'un système d'exploitation.

Cependant, avec la croissance et le développement des systèmes embarqués, ces derniers tendent à être de plus en plus multitâches et multiprocesseurs et donc à réaliser des tâches de plus en plus complexes. Par conséquent, on retrouve les mêmes problématiques de gestion des CPUs et de la mémoire que sur un processeur traditionnel. Il devient alors rapidement plus difficile de se limiter au *bare metal*, ce qui nous oblige à manipuler manuellement le matériel et les périphériques sans pouvoir accéder à des bibliothèques avancées.

Viennent alors les systèmes d'exploitation embarqués, des versions plus épurées des systèmes d'exploitation de base GNU/Linux, qui conservent uniquement les fonctionnalités essentielles à l'application.

Nous allons passer en revue des composants, fonctionnalités et outils essentiels pour les systèmes d'exploitation GNU/Linux, plus précisément pour une cible embarquée. Il est toujours important de rappeler qu'un système d'exploitation est composé d'un noyau, la librairie système, de différents services utilisateurs et du *bootloader*.

Cross-Compilation

Un noyau est programmé comme tout autre logiciel et nécessite d'être compilé pour pouvoir être exécuté. Malheureusement, la compilation du noyau sur une cible embarquée n'est pas toujours possible pour plusieurs raisons : manque de stockage ou de mémoire, puissance de CPU insuffisante, ou encore nécessité d'installer toute la chaîne d'outils de compilation, ce qui peut être une surcharge en ressources. On introduit alors le *Cross-Compilation*. Son but est très simple : compiler un code et produire le binaire qui est spécifique à l'architecture cible (dans la plupart des cas, une architecture ARM pour les systèmes embarqués).

Pour le réaliser, il existe deux solutions possibles : le faire manuellement, mais cela reste très fastidieux et nécessite une maîtrise complète de l'environnement ; ou bien utiliser une *toolchain* telle que BuildRoot, YoctoProject ou PetaLinux (spécifique aux circuits Xilinx (AMD)). Ces *toolchains* nous assistent en pré-paramétrant différentes options disponibles, mais sans nous contraindre puisque nous conservons la possibilité d'apporter nos propres modifications.

initramfs

Avant de parler de l'*initramfs*, nous devons parler de *ramfs* et *tmpfs*. *ramfs* est un *filesystem* qui va monter le système de cache de Linux ; il sera de taille variable. Les utilisateurs en mode *root* pourront créer des fichiers directement dans les caches. Cependant, les pages de ces derniers auront toujours le *flag dirty* car ils n'ont pas de stockage sur lequel synchroniser leur contenu. Ainsi, ils ne sont jamais évincés du cache. Ce système sature les caches. Pour y faire face, *tmpfs* a été mis en place. Celui-ci se comporte comme *ramfs* à la différence que *tmpfs* a une limite de taille, synchronise ses données des pages sur le swap, ce qui permet donc de les évincer du cache. De plus, un utilisateur normal peut y écrire ou créer un fichier. Lors du *boot time*, si une archive (.cpio) *initramfs* est présente dans l'image noyau, spécifié par : `CONFIG_INITRAMFS_SOURCE`, le noyau l'extrait sur *tmpfs* ou *ramfs* et le monte comme *rootfs* permanent ou temporaire.

La commande :

```
echo init | cpio -o -H newc | gzip > initramfs_test.cpio.gz
```

permet de créer une archive *initramfs initramfs_test*, et cette archive devra contenir au moins un script *init*.

Root Filesystem

Les *Filesystems* sont utilisés pour organiser des données dans des dossiers et fichiers d'un périphérique de stockage, un stockage sur le réseau ou bien la RAM (*ramfs*, *tmpfs*).

Le *Root Filesystem* est le premier système de fichiers que le noyau va monter lors de son initialisation ; il est la racine (/) de l'arborescence du système de fichiers. Habituellement, pour monter une partition du disque, une clé USB sur un dossier, on utilise la commande `mount`, qui fait partie de la bibliothèque `libc`, elle-même incluse dans le *Root Filesystem*. Ainsi, on ne peut pas l'utiliser pour monter le *Root Filesystem* lui-même. Le montage de ce système de fichiers se fait alors automatiquement par le noyau lors de son démarrage, et on peut spécifier quel stockage (disque dur, clé USB, card SD, réseau via le protocole NFS, *ramfs*...) on veut monté en tant que *root filesystem* par défaut avec la commande `root=` dans le menu de configuration du *Root Filesystem* ou bien dans le menu du *bootloader* pour spécifier un espace temporaire.

Pour accélérer le démarage, on a tendance à monter le *Root Filesystem* sur un filesystem en RAM : **initramfs**, car il est déjà présent en mémoire lors du boot time. Cela peut également être utile si l'on veut monter le *Root Filesystem* sur un périphérique mais que le pilote de ce périphérique ne fait pas partie de l'image du noyau. Le noyau va d'abord monter le *Root Filesystem* sur un *initramfs*, exécute *init*. Le processus *init* de PID=1 pourra ensuite, charger les pilotes de périphériques et monter le *rootfs* final sur le périphérique souhaité. Le fait de ne pas inclure les pilotes de périphériques dans le noyau vise à limiter la taille de cette dernière. En effet, il est de plus en plus courant de configurer un noyau sur mesure, répondant à des besoins bien spécifiques, de le tester et de l'installer sur plusieurs systèmes identiques. Il est donc important d'avoir une taille raisonnable pour faciliter le partage.

Device Tree

Le *device tree* est une structure de données qui permet de décrire le matériel et son intégration [18].

Il existe deux types de périphériques, les périphériques découvrables et les périphériques non découvrables par le BIOS/UEFI ou le noyau. Les périphériques non-découvrables ont besoin d'une description pour pouvoir être utilisés. Beaucoup d'architectures embarquées ont du matériel non-découvrable : série Ethernet, I2C... Décrire le matériel en un langage de description matériel et l'enregistrer dans le *device tree* est un moyen d'y parvenir. On peut aussi le décrire dans les tables ACPI du BIOS (pour les architectures x86) ou en un code C et qu'on insère dans le noyau.

La description du matériel se fait dans un fichier .dts

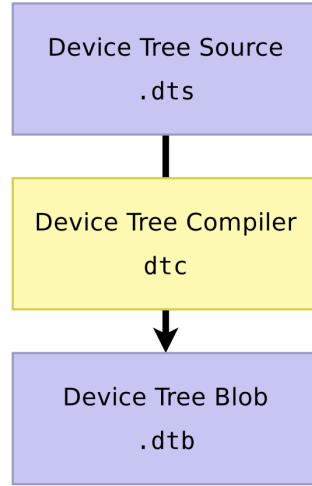


FIGURE 1 – chaîne de création du fichier .dtb
[6]

Le fichier .dtb sera transmis au noyau au boot time.

Il existe un *device tree* différent pour chaque architecture supportée par le noyau, disponible dans arch/<architecure>/boot/dts/<boards>.dtb, boards étant un matériel.[10]

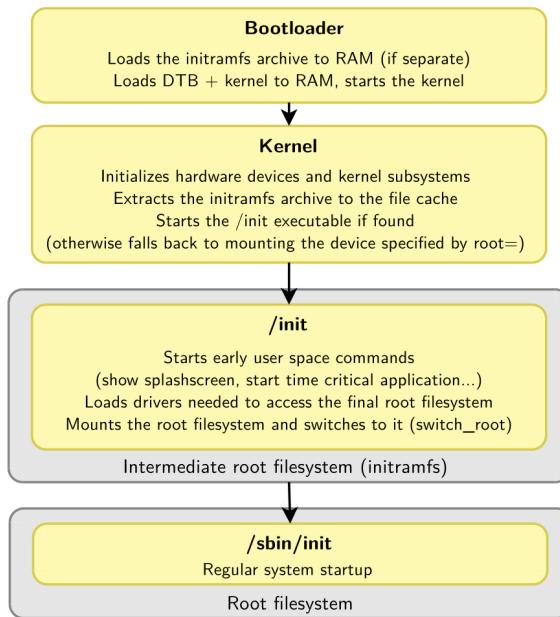


FIGURE 2 – Schéma de démarrage à partir du *bootloader*[6]

3 Prise en main d'un système d'exploitation embarqué

3.1 Génération d'une distribution Linux sur un softcore

Afin de bien comprendre le fonctionnement d'un OS embarqué sur SoC Xilinx (AMD), nous avons commencé par générer une distribution Linux pour le softcore MicroBlaze d'AMD avec les périphériques qui lui sont associés (voir design de l'architecture sur la figure 3), d'une carte Basys3 [4], contenant un FPGA, en utilisant la toolchain *PetaLinux*. Nous utilisons cette carte pour la prise en main puisque c'est la même carte qu'on utilise dans les différentes UEs de FPGA de Licence et de Master. Le MicroBlaze est un microprocesseur RISC (Reduce Instruction Set Computer : architecture de microprocesseur qui utilise des jeux d'instructions très légers et très optimisés) dont l'architecture a été décrite dans un langage de description matérielle (ici Verilog).

On voit arriver une vague de softcore et hardcore open source avec un jeu d'instruction RISC-V.[9]

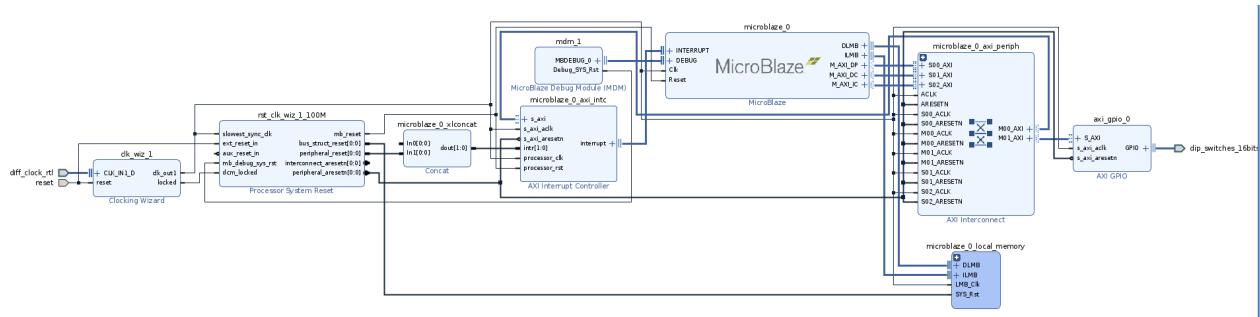


FIGURE 3 – Design Microblaze avec un GPIO



FIGURE 4 – Carte Basys3 [16]

Commande pour générer la distribution :
Création du projet :

```
petalinux-create -type project -template microblaze -name <project-name>
```

Importer la description matérielle, fichier généré depuis Vivado, dans le projet :

```
petalinux-config -get-hw-description file.sdk -oldconfig
```

Pour configurer le noyau, choisir les drivers à compiler statiquement dans le noyau (on a un noyau monolithique modulaire), ajouter un initramfs :

```
petalinux-config -c kernel
```

Petalinux va proposer les drivers à compiler statiquement dans le noyau en fonction du type de microprocesseur qu'on a spécifié et les périphériques qu'il contient grâce au fichier file.sdk

Pour configurer le rootfs :

```
petalinux-config -c rootfs
```

Petalinux va automatiquement générer un *device tree* en fonction du fichier file.sdk qu'on lui a fourni. Mais on peut encore ajouter la description d'un matériel en l'ajoutant dans :

```
project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi
```

Pour générer la distribution/créer le projet :

```
petalinux-build
```

On se connecte en micro-usb sur la carte et on lance un *serial* terminal sur la carte avec putty :

```
sudo putty
```

On reconfigure le FPGA avec le bitstream du projet, notamment pour reconfigurer le FPGA avec le *softcore* :

```
petalinux-boot -jtag -fpga
```

Charger le noyau dans la mémoire et la lancer :

```
petalinux-boot -jtag -kernel
```

Nous venons donc de générer une distribution GNU/Linux embarqué pour ce circuit. Si nous souhaitons modifier notre circuit, nous sommes obligés de générer un autre système d'exploitation adapté au nouveau design. Une limitation majeure de ce processus est donc que chaque distribution est spécifique à un circuit donné. On ne peut pas modifier le circuit tout en conservant la même distribution. Or, l'un des avantages d'un FPGA est de permettre le test de divers circuits, ce qui serait une perte de temps considérable si nous devions régénérer une distribution à chaque modification du circuit.

3.2 Distribution PYNQ

Par la suite, nous allons nous baser sur le projet PYNQ (PYthon on zyNQ) avec la carte PYNQ-Z2, comme on le voit dans la figure 5, qui intègre un processeur multicoeur ARM Cortex_A9 et un FPGA en un seul circuit intégré. Il s'agit, comme vu avant, d'une architecture hétérogène. Le projet PYNQ a sa propre distribution de système d'exploitation Linux nommé PYNQ, qu'il est possible de télécharger depuis un répertoire git [20]. Cette distribution comprend les éléments de base d'un système d'exploitation, des pilotes de périphériques de base (pilote DMA, pilote GPIO, pilote carte réseau,...) mais aussi une bibliothèque `pynq`. Celle-ci permet de charger dans la mémoire des applications, gérer et contrôler les *overlays* (chargement, accès à un IP), de gérer l'allocation mémoire et de gérer le PL (reconfiguration du FPGA avec le *bitstream*, `file.bit`). Elle permet donc de charger/décharger dynamiquement un circuit, donc sans changer de système d'exploitation. Ce qui permet de faciliter le test de plusieurs architectures car il suffit de charger *l'overlay* du design qui va entraîner la reconfiguration du PL avec *l'overlay*, en utilisant la méthode :

```
overlay(<chemin vers l'overlay/file.bit>)
```

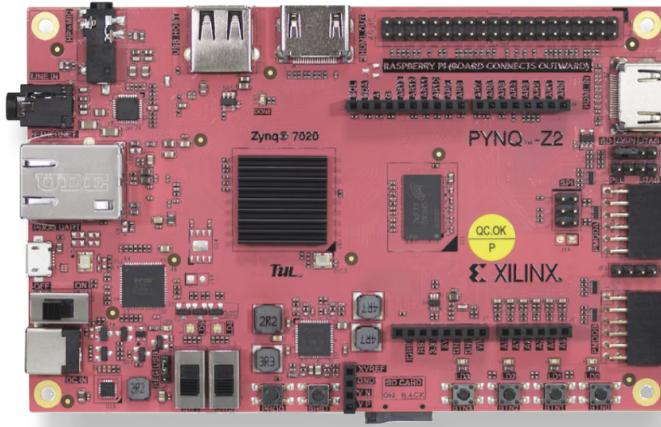


FIGURE 5 – PYNQ Z2 Xilinx [1]

L'avantage de cette distribution clés en main est qu'elle facilite grandement l'exploitation du CPU et du FPGA en offrant des APIs de très haut niveau. On peut travailler avec le projet sur n'importe quelle distribution (MacOs, Linux, Windows). La distribution PYNQ nous offre la possibilité d'interagir graphiquement avec la carte via des *Notebooks Jupyter*. Pour cela, nous devons nous connecter à la PYNQ Z2 via Ethernet, puis accéder à la carte via un navigateur en utilisant l'adresse web : <http://192.168.2.99> (par défaut).

3.3 Interfaçage du système d'exploitation PYNQ et du FPGA

Les IPs peuvent être vu comme des périphériques du processeurs et nécessitent donc un driver pour être interfacé correctement.

Dans la suite de notre projet, nous passerons par un *Direct Memory Access* (DMA) pour communiquer avec notre IP, car il permet de transférer des données depuis la mémoire RAM vers un périphérique externe de stockage de données, ou inversement, sans l'intervention du CPU. Cela allège le CPU et accélère le transfert de données. On parle d'un accès direct car le *Processing System* n'intervient aucunement dans ce processus de transfert, sauf pour démarrer et stopper celui-ci. Le *driver* de cet IP est aussi fournit par Pynq, nous n'avons donc pas besoin de le développer.

Le DMA est aussi présent parmi les nombreux IPs proposés sur Vivado. Il faudra donc l'insérer dans le design de notre architecture si on veut l'utiliser. Le logiciel Vivado permet aussi d'apporter une configuration du PS (activation de port, gestion de la taille des canaux, gestion des horloges...) en instanciant son IP dans le design, et sera configuré en même temps que le reste du FPGA.

4 Architecture hétérogène : une accélération matérielle

Notre objectif a d'abord été de proposer une accélération d'un algorithme écrit en Python grâce au système Zynq, avec le projet PYNQ.

Pour ce faire, nous allons concevoir dans le PS (CPU), comme effectué dans [14], un algorithme de filtre *Finite impulse response* (FIR), mesurer son temps d'exécution. Ensuite, nous allons concevoir un *overlay* avec un IP réalisant le filtre FIR sur Vivado comme dans [21] et tester cet IP contenu dans l'*overlay* sur python et mesurer son temps d'exécution. Enfin, nous pourrons comparer les résultats de cet algorithme dans le PS et dans le PL.

Expliquons ce qu'est un filtre FIR. Un filtre est une *biporte linéaire destiné à transmettre les composantes spectrales de la grandeur d'entrée selon une loi spécifiée, en général en vue de laisser passer les composantes dans certaines bandes de fréquences et à les affaiblir dans d'autres bandes* [7]. Plus simplement, il s'agit d'un circuit électronique qui atténue certaines composantes fréquentielles d'un signal et en laisse passer d'autres. Un filtre FIR est un filtre à temps discret, c'est-à-dire que sa réponse est uniquement basée sur un nombre fini de valeurs du signal d'entrée [4]. Nous pouvons retrouver une représentation schématique à la figure 6.

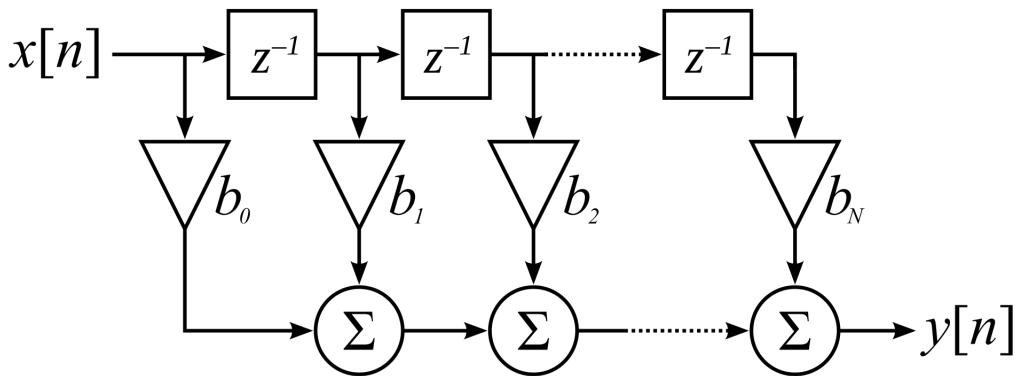


FIGURE 6 – Réalisation directe d'un filtre à réponse impulsionnelle finie d'ordre N [4]

4.1 Algorithme réalisé dans le *Processing System*

Dans un premier temps, nous réalisons un signal contenant du bruit, tel que dans la figure 7, afin d'y appliquer le filtre FIR dessus.

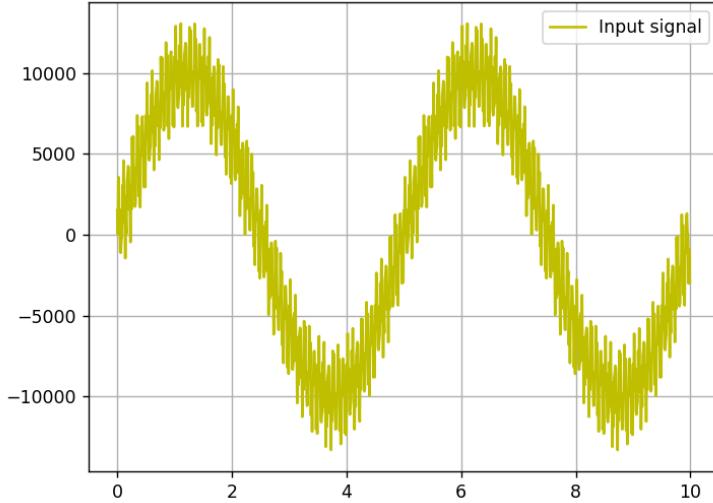


FIGURE 7 – Signal bruité

Ensuite nous utilisons la fonction `lfilter` de la bibliothèque SciPy en python à laquelle nous ajoutons des coefficients pour simuler un filtre FIR. Ces coefficients sont calculés en fonction de notre bande passante et de notre bande d'arrêt grâce à cette ressource [11]. Nous obtenons le filtre suivant, présenté ci-dessus dans la figure 8.

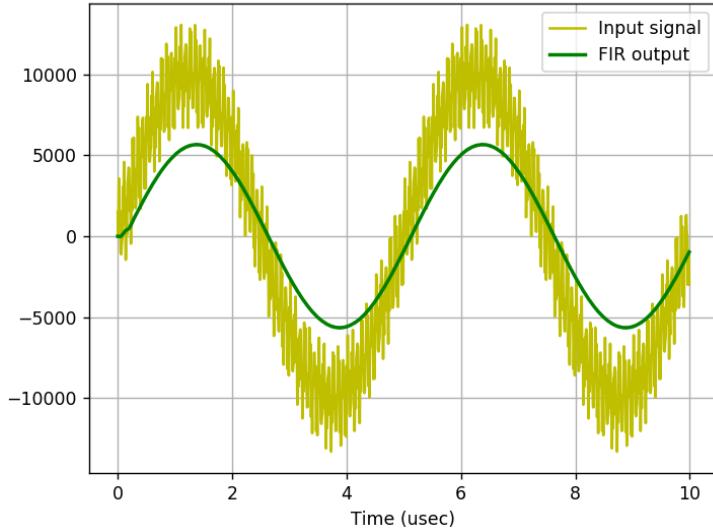


FIGURE 8 – Signal bruité auquel on a appliqué le filtre FIR

Enfin, nous mesurons le temps d'exécution de cette fonction en faisant une moyenne sur 100 exécutions tel qu'on le voit dans la figure 10. Nous obtenons un temps d'exécution de 83,7 ms en moyenne.

4.2 Algorithme réalisé dans le *Programmable Logic*

Dans un second temps, nous concevons un *overlay* sur Vivado dans lequel se trouve l'IP réalisant le filtre FIR. Pour y parvenir, nous ajoutons d'abord dans notre *overlay* les IPs (*Processing System*, IP filtre FIR et DMA). Ensuite, nous devons configurer notre IP FIR. Il s'agit d'un IP déjà préexistant sur Vivado auquel il suffit simplement d'ajouter des coefficients (ce sont les mêmes coefficients que ceux donnés à la fonction `lfiter` sur python précédemment). De plus, nous reconfigurons l'horloge de sorte à avoir un échantillon du signal d'entrée par période d'horloge (la fréquence d'horloge est de 100 MHz).

Après cela, nous configurons notre DMA. Pour cela nous maximisons la taille (23 bits) du registre qui indique la taille du tampon. Enfin, nous connectons les ports

`S_AXIS_S2MM` et `M_AXIS_MM2S` du DMA respectivement aux ports `M_AXIS_DATA` et `S_AXIS_DATA` de l'IP FIR pour relier les deux composants.

Pour finir nous activons dans notre *Processing System* un port de haute performance vers le DDR pour que le DMA puisse y accéder. Le DDR, ou mémoire *Double Data Rate*, est un type de mémoire qui permet de transférer des données à la fois sur le front montant et sur le front descendant des impulsions d'horloge [3]. Ainsi, cela permet de doubler la vitesse d'accès à la mémoire par rapport à une mémoire RAM classique.

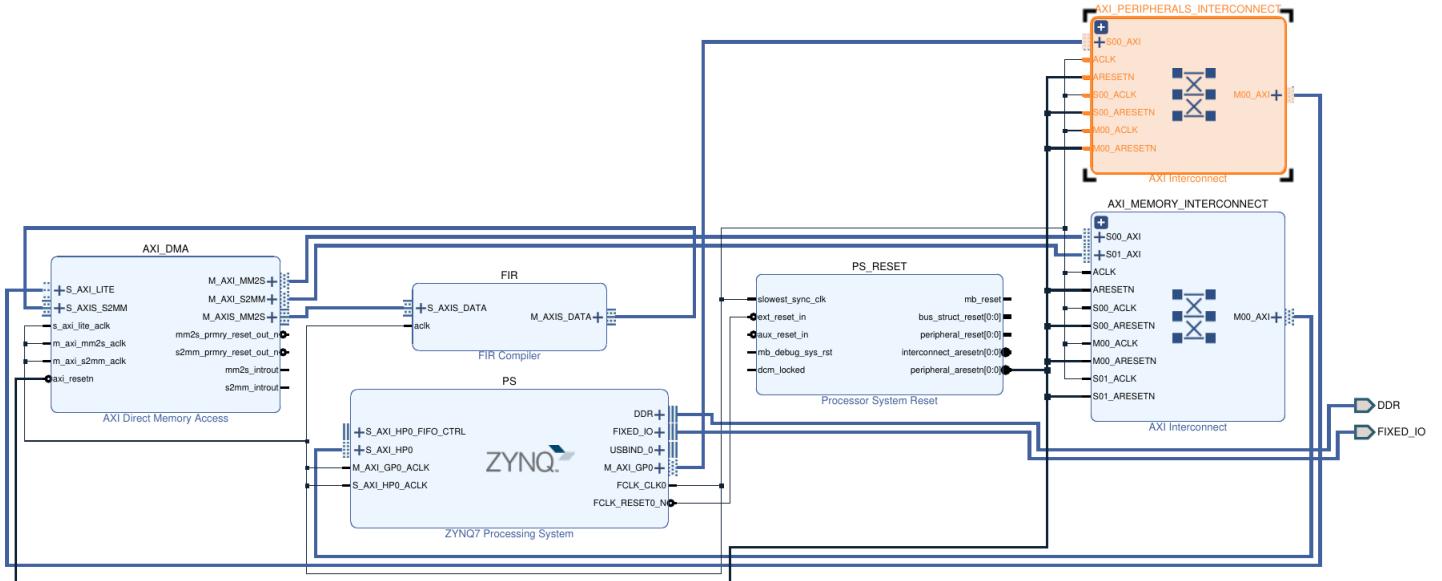


FIGURE 9 – *Block design* de l'*overlay* contenant le filtre FIR

Nous pouvons voir dans la figure 9 que nous obtenons des blocs supplémentaires dans notre design qui sont `PS_RESET`, `AXI_MEMORY_INTERCONNECT` et `AXI_PERIPHERALS_INTERCONNECT`. Le premier sert à fournir un système de réinitialisation asynchrone tandis que les deux autres servent à relier certains composants entre eux ainsi qu'à proposer du multiplexage entrée/sortie.

Enfin, nous exportons notre *Block Design*, c'est-à-dire le *design* de notre *overlay* avec tous les circuits nécessaires, pour pouvoir l'utiliser sur python et mesurer son temps d'exécution.

Nous obtenons alors le même filtre qu'obtenu dans la figure 8. Quant au temps d'exécution, celui-ci vaut en moyenne 2,3 ms sur un total de 100 exécutions tel qu'on le voit dans la figure 10.

	Filtre logiciel	Filtre matériel
Temps d'exécution moyen (en ms)	83,7	2,3

TABLE 1 – Comparaison des temps d'exécution en ms du filtre FIR

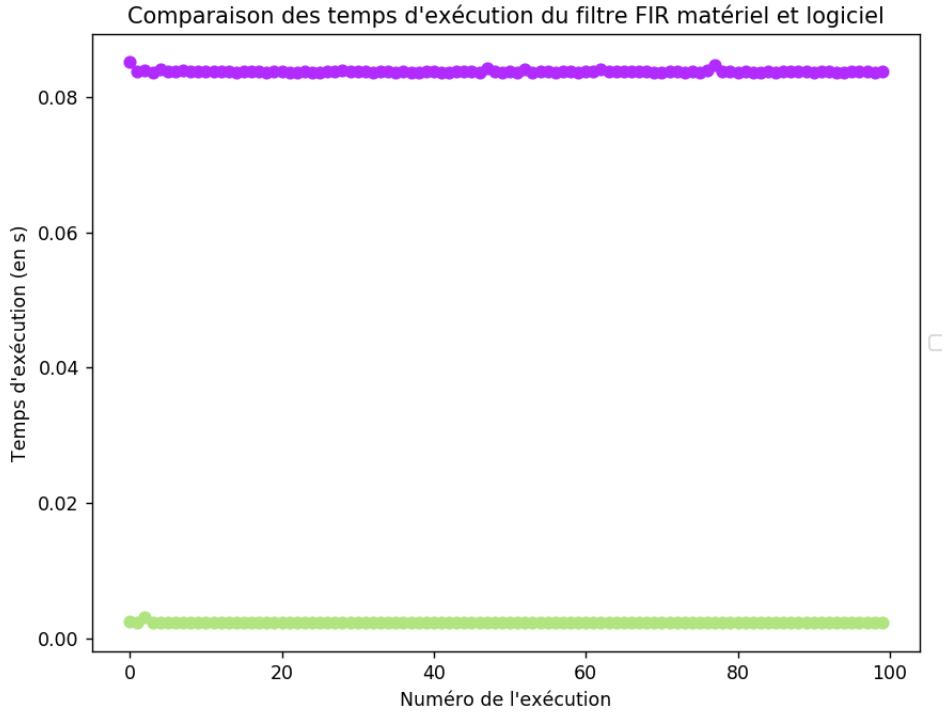


FIGURE 10 – Comparaison des temps d'exécution en ms du filtre FIR

Nous pouvons déjà voir la différence de temps d'exécution entre les deux filtres dans le tableau 1 et dans la figure 10.

$$\frac{83,7}{2,3} \approx 36 \quad (1)$$

Nous obtenons alors un facteur d'accélération de 36 en utilisant une architecture matérielle dédiée plutôt qu'en réalisant un algorithme sur un processeur classique. Ce résultat est particulièrement considérable. En effet, nous n'avons pas seulement divisé par deux le temps d'exécution mais bien par 36. Par exemple, cela signifie que si l'algorithme prenait 1 minute pour se terminer, avec l'architecture matérielle dédiée il prendrait moins de 2 secondes.

Pour comparer, dans le tutoriel que nous avons suivi [14], les valeurs obtenues sont celles présentées ci-dessous.

	Filtre logiciel	Filtre matériel
Temps d'exécution moyen (en ms)	106,6	3,7

TABLE 2 – Comparaison des temps d'exécution en ms du filtre FIR

Avec un facteur d'accélération de

$$\frac{106.6}{3.7} \approx 28.8 \approx 29. \quad (2)$$

Si nous avons obtenu des temps d'exécution plus faibles dans nos mesures ainsi qu'un meilleur facteur d'accélération que dans le tutoriel, cela peut s'expliquer par deux raisons. Tout d'abord, contrairement au tutoriel, nous avons réaliser plusieurs mesures, une centaine à chaque fois, pour obtenir un temps d'exécution moyen. De ce fait, il est possible que la valeur obtenue dans le tutoriel ne soit pas représentative du facteur d'accélération qu'il est possible d'obtenir. En outre, la carte utilisée dans [14] est la carte PYNQ-Z1 tandis que, pour notre part, nous utilisions la carte PYNQ-Z2. Ainsi, il est possible que notre carte soit plus performante que l'autre.

Finalement, nous pouvons constater une gain de temps considérable grâce à une architecture matérielle dédiée et ainsi conclure que le *Programmable Logic* nous permet d'obtenir des performances bien plus importantes que le PS, tout du moins dans cette application.

5 IP personnalisé : seuillage d'image en niveau de gris

5.1 Réalisation de l'IP

Par la suite, nous avons décidé de développer un IP permettant de réaliser un seuillage sur une image en niveau de gris. Nous faisons le choix de ne traiter qu'une image plutôt qu'un flux vidéo pour éviter d'utiliser des IPs HDMI. Cela facilite autant le code Python que le design à mettre en place. En outre, cela nous permet de nous focaliser sur la communication entre le PS, le DMA et notre IP, ainsi que sur le bon fonctionnement de cette dernière. Nous choisissons un algorithme simple de seuillage car nous disposons d'un petit FPGA, qui est limité en nombre de LUT et ne supporte pas des algorithmes qui demandent plus de ressources comme le filtre de *Sobel*[2] par exemple. Le bloc design est basé sur un design déjà existant que l'on peut voir dans la figure 11, mais nous avons modifié **Custom AXI-Streaming IP** pour ajouter un traitement au flux de pixels, tel qu'on peut le voir dans la figure 14. On la renommera l'**IP Threshold**. Il faudra par la suite configurer le *Processing System* et l'AXI DMA pour pouvoir les faire communiquer. A noter que les configurations ont été effectuées sur le logiciel Vivado. L'**IP interconnect** se charge de l'interconnexion des canaux.

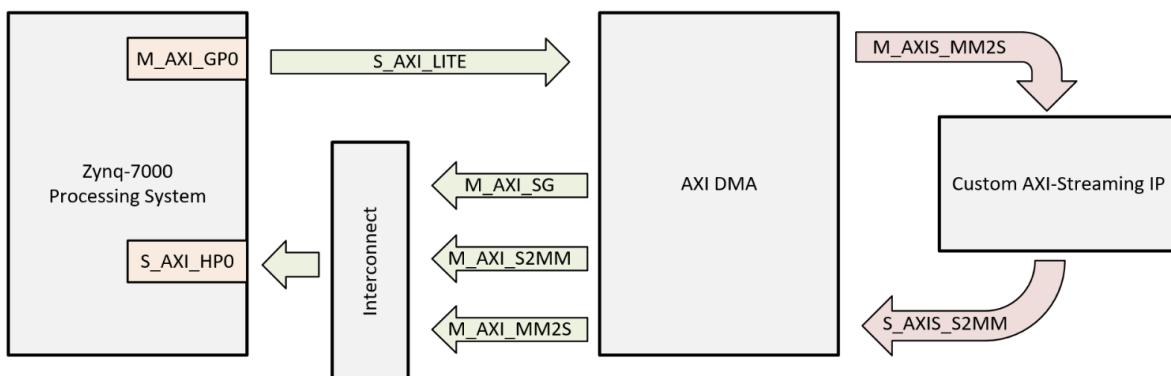


FIGURE 11 – Bloc diagram du design [13]

- Configuration du Processing System (figure 12) :
 - 1 port **S_AXI_LITE** : pour se connecter au **S_AXI_LITE** du AXI DMA
 - Permet d'envoyer des instructions pour configurer, arrêter ou démarrer l'AXI DMA
 - 2 high performance ports
 - 1 port pour l'écriture du AXI DMA dans la mémoire, connecté au port **M_AXI_S2MM** du AXI DMA
 - 1 port pour la lecture du AXI DMA depuis la mémoire, connecté au port **M_AXI_MM2S** du AXI DMA
- Configuration du AXI DMA (figure 12 et 13) :
 - *Buffer length register* : taille maximale d'un paquet transféré en une seule transaction effectuée par l'AXI DMA : $2^{26} = 67$ Mibits.

- *Address width* : taille des adresses des mots de la mémoire du Processing System : 32 bits pour la pynq Z2.
- Configuration des canaux d’écriture et de lecture :
 - *Memory mapped data width* : Taille d’un mot que l’AXI DMA lira ou écrira vers la mémoire via les high performance ports, qui eux sont configurés à 64 bits/mots.
 - *Stream data width* : 32 bits, taille d’un mot que l’AXI DMA lira (resp. écrira) depuis (resp. vers) le matériel.

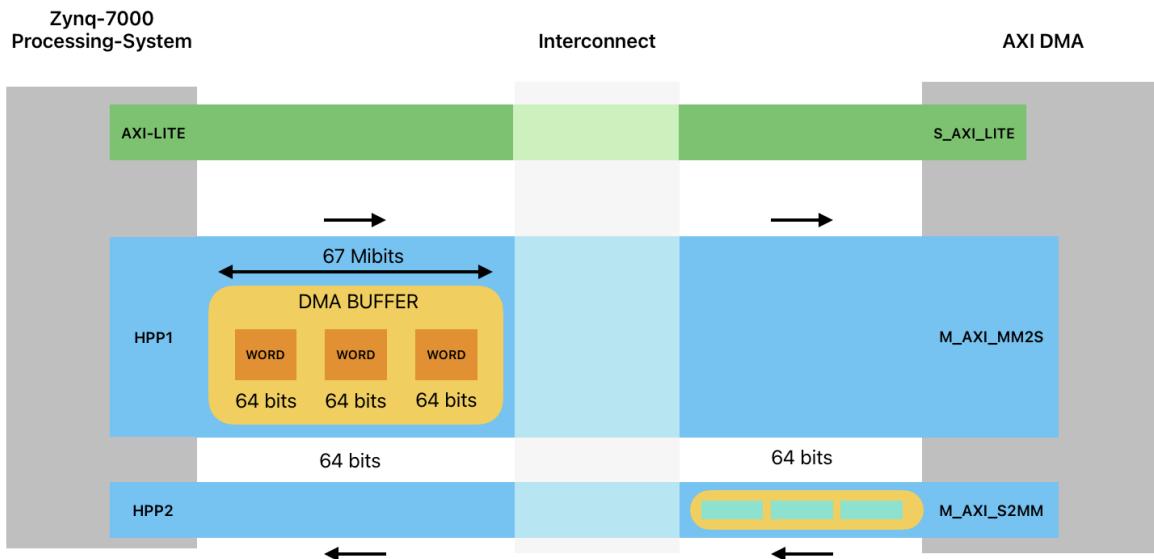


FIGURE 12 – Connexion PS - AXI DMA
HPP : High Performance Port

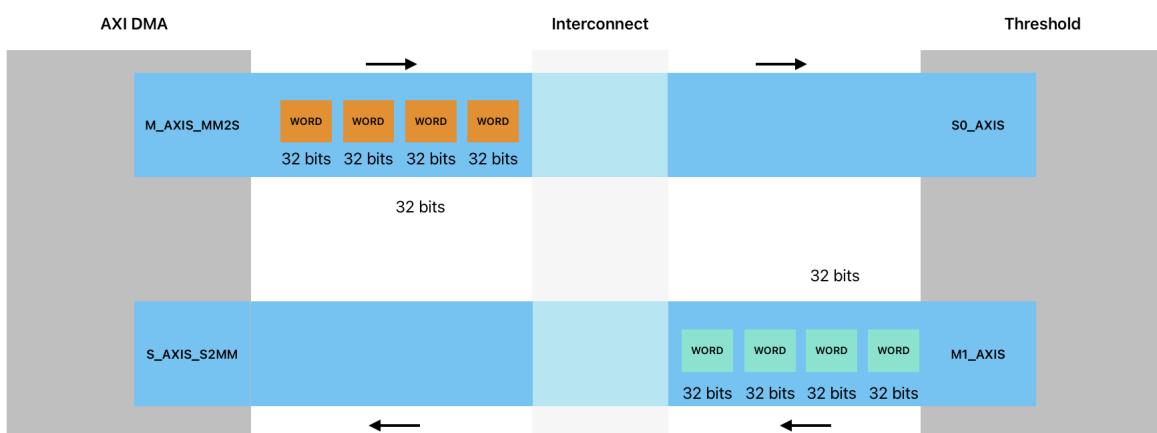


FIGURE 13 – Connexion AXI DMA - Threshold

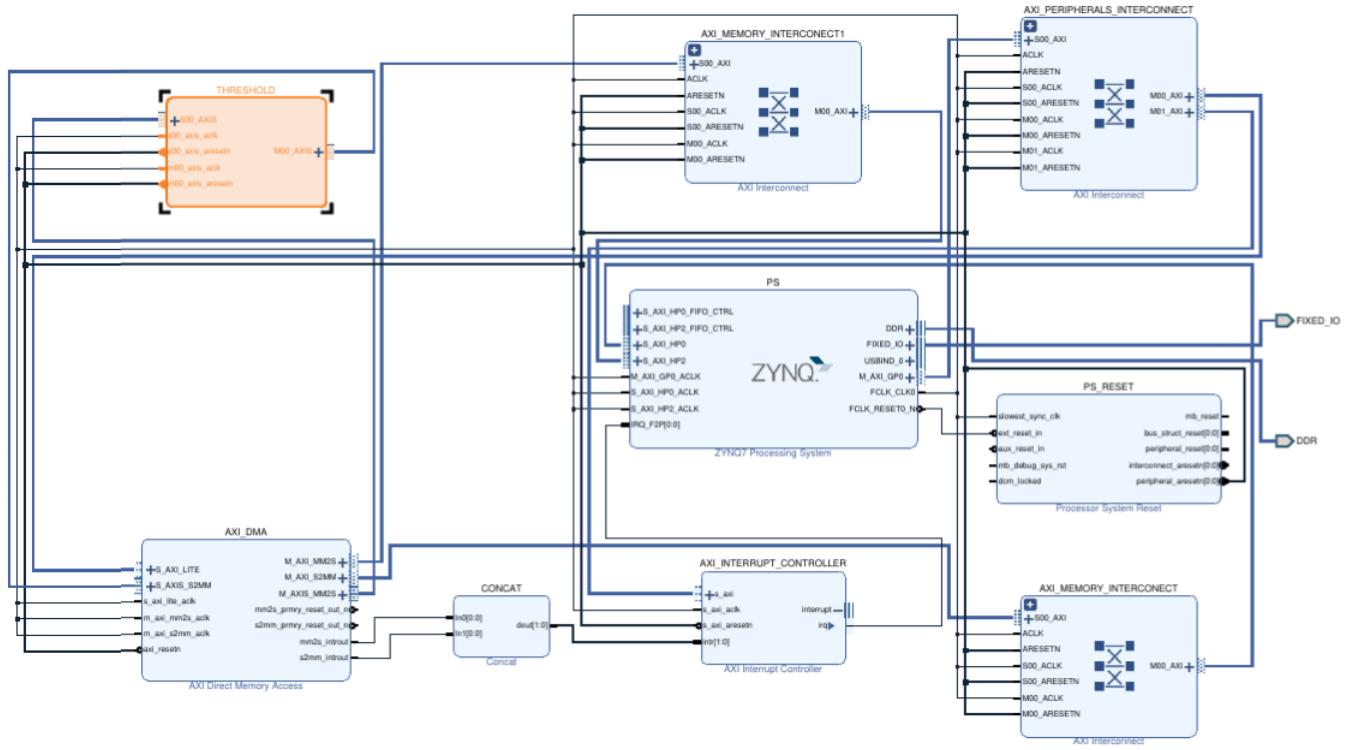


FIGURE 14 – Block design de l’*overlay* contenant l’IP Threshold

On pourra modéliser l’IP **Threshold** par une fonction pour comprendre son fonctionnement interne :

Algorithme 1 : Threshold

Entrées : mot_in : entier 32 bits
Sorties : mot_out : entier 32 bits

```

1 si mot_in < 128 alors
2   mot_out ← 255;
3 sinon
4   mot_out ← 0;
```

La communication avec cette IP va se faire à travers l’AXI DMA en utilisant les primitives définies dans la bibliothèque PYNQ.

Tout d’abord on instancie l’overlay :

```
Overlay = overlay(chemin_vers_overlay_du_design/file.bit)
```

Ensuite, on récupère l’objet qui permet de manipuler le DMA :

```
dma = Overlay.dma
```

Nous aurons à disposition les méthodes suivantes pour interagir avec le DMA :

- `dma.sendchannel.start()` : démarre les canaux d’émission vers l’IP **Threshold** de l’AXI DMA : M_AXI_MM2S et M_AXIS_MM2S

- `dma.recvchannel.start()` : démarre les canaux de réception depuis l'IP **Threshold** de l'AXI DMA : **M_AXI_S2MM** et **S_AXIS_S2MM**
- `dma.sendchannel.transfer(data_in)` : initie le transfert de données `data_in` vers l'IP **Threshold**.
- `dma.recvchannel.transfer(data_out)` : initie le transfert de données et les stocke dans `data_out` depuis l'IP **Threshold**

Les données `data_in` et `data_out` ne doivent pas dépasser la longueur du registre de la mémoire tampon : *buffer length register*. Si leur taille dépasse cette limite, il est nécessaire de répéter l'opération de réception et d'envoi de données jusqu'à avoir la totalité des données.

La taille en pixel de l'image de notre test est de :

$$320 \cdot 240 = 76800 \text{ pixels} \quad (3)$$

En sachant qu'un pixel sera encodé sur 32 bits, la taille en bit de l'image :

$$76800 \text{ pixels} \cdot 32 \text{ bits} = 2 \text{ Mbits} \quad (4)$$

L'image pourra donc être transférer vers l'IP en une seule transaction DMA.

Dans notre cas, `data_in` serait les pixels de l'image qu'on veut appliquer un seuil, des données qui ont une valeur comprises entre 0 et 255 et `data_out` les pixels résultants, des données qui ont une valeur soit 0 (noir) soit 255 (blanc).

On a testé le seuillage sur la photo de la figure 15 en n'utilisant que la PS (seuillage des pixels directement en python) et en passant par notre IP. Nous avons bien le même résultat, tel qu'on peut le voir sur la figure 16, pour une valeur de seuil à 128.



FIGURE 15 – Image originale



FIGURE 16 – Image résultat

On observe dans l'image résultat que les pixels ayant une couleur claire, c'est-à-dire ceux dépassant la valeur de 128, sont rendus blancs, tandis que les pixels ayant une couleur sombre, valeur inférieure à 128, sont rendus noirs. L'image résultat correspond bien à l'image attendue. Ainsi, on en conclut que notre IP de seuil fonctionne.

Maintenant, comparons dans le tableau 3 les temps d'exécution entre le PS et le PL pour voir le gain que notre IP nous a permis d'obtenir.

	Seuillage logiciel	Seuillage matériel
Temps d'exécution moyen (en s)	1.24	0.09

TABLE 3 – Comparaison des temps d'exécution en s du Seuillage

Nous obtenons alors un facteur d'accélération de

$$\frac{1.24}{0.09} \approx 13,8. \quad (5)$$

Pour un même résultat, on s'attendait bien que passer par le PL aurait été plus rapide pour le seuillage que passer par le PS car il est possible qu'une tâche plus prioritaire interrompe le traitement des pixels dans le PS. Cela contraindrait le PS à traiter cette tâche avant de pouvoir poursuivre le traitement des pixels. On a aussi le fait que le CPU effectue beaucoup plus d'instruction que le PL pour traiter les pixels, car le PL est conçu spécialement pour le seuillage de pixel et le PS est une architecture beaucoup plus générale. En passant par le PL, le PS a également été soulagé, ce qui a permis au CPU de traiter d'autres tâches en attendant que le PL effectue le traitement des pixels.

Cependant, notre design (figure 14) présente une limitation par rapport au traitement des pixels sur le PS. En effet, la valeur de seuil est fixée à 128 et il n'est pas possible de la modifier depuis le PL sans modifier l'*overlay* sur Vivado et donc de régénérer un nouveau *bitstream*, contrairement à ce qui est possible avec le traitement fait sur le PS où cette valeur peut être aisément ajustée.

Une amélioration serait de passer par le bus `axi_lite` pour configurer le seuil, mais ceci demande pas mal de travail d'implémentation sur Vivado et de programmation en python ensuite. Une autre limite de cette conception est l'impossibilité de désactiver le seuillage d'image. Toutefois, dans la section suivante, nous présentons une solution qui utilise un GPIO pour activer ou désactiver le seuillage.

5.2 Interfaçage avec les ports GPIO

Une fois l'IP réalisé, nous avons souhaité utiliser les *General Purpose Input/Output* (GPIO) afin de pouvoir modifier certains paramètres de notre IP. Le GPIO représente un ensemble de ports destinés à contrôler les entrées et les sorties. L'interfaçage, quant à lui, désigne la *jonction entre deux matériels ou logiciels leur permettant d'échanger des informations par l'adoption de règles communes physiques ou logiques* [19]. Le but étant de pouvoir activer ou désactiver le traitement de seuil avec l'interrupteur. Ceci permet simplement d'interagir avec le *Programmable Logic*, en plus du traditionnel *Processing System*. Cette méthode n'est pas forcément la plus commune, mais elle nous semblait intéressante. En effet, elle nous permet une interaction avec notre IP non seulement depuis un code python mais aussi avec le GPIO de notre carte.

Pour ce faire, nous modifions l'IP précédemment décrit pour lui ajouter une entrée sur un bit que nous appelons `sw`. Ensuite, nous ajoutons un fichier de contraintes, c'est-à-dire un fichier dans lequel les ports physiques d'une carte sont mis en correspondance avec certains

signaux de nos modules. Dans ce fichier de contraintes, nous assignons l'interrupteur 0 de notre carte à notre entrée **sw**. Enfin, nous lisons le signal **sw** dont la valeur nous sera donnée selon la position de l'interrupteur 0. S'il vaut 0, nous réalisons notre traitement classique tandis que s'il vaut 1 le traitement est désactivé comme on peut le voir dans l'algorithme ci-dessous.

Algorithme 2 : Threshold_GPIO

Entrées : mot_in : entier 32 bits

Sorties : mot_out : entier 32 bits

```

1 si sw = 0 alors
2   si mot_in < 128 alors
3     | mot_out ← 0;
4   sinon
5     | mot_out ← 255;
6 sinon
7   | mot_out ← mot_in;
```

Nous observons que si l'interrupteur est baissé, nous obtenons une image seuillée et, donc nous sommes passés par la première condition, tandis que s'il est levé, aucun traitement n'est réalisé, et donc nous sommes passés par la seconde condition. Ainsi, nous pouvons conclure que notre IP fonctionne.

Comparons maintenant les temps d'exécution dans l'IP en fonction de si l'on réalise (interrupteur baissé) ou non (interrupteur levé) le traitement.

	Interrupteur levé (pas de traitement)	Interrupteur baissé (traitement)
Temps d'exécution moyen (en s)	0.0900	0.0901

TABLE 4 – Comparaison des temps d'exécution en s en fonction de la valeur de **sw**

On constate sur le tableau 4 que le temps d'exécution est quasiment identique puisqu'on a seulement 0.1ms de différence. Cela n'est pas surprenant car la différence entre les deux traitements (seuillage ou non) est faible.

6 Ouverture : seuillage d'un flux vidéo externe

Dans la suite de ce projet, nous aurions aimé réaliser notre seuillage sur un flux vidéo externe en temps réel. Pour cela, nous voulions utiliser notre IP connecté aux ports HDMI IN et HDMI OUT de la carte PYNQ-Z2 avec un *Video DMA* (VDMA). Le VDMA permet la même chose qu'un DMA mais avec de la synchronisation d'image et plus de canaux pour s'adapter aux transferts vidéos.

Bien que nous n'y sommes pas encore parvenus, nous avons eu néanmoins un début d'idée en prenant un *block design* comprenant tous les IP dont nous avons besoin [17] et en y ajoutant notre IP. De plus, nous connectons notre IP au port HDMI de sortie comme montré dans les *block designs* ci-dessous.

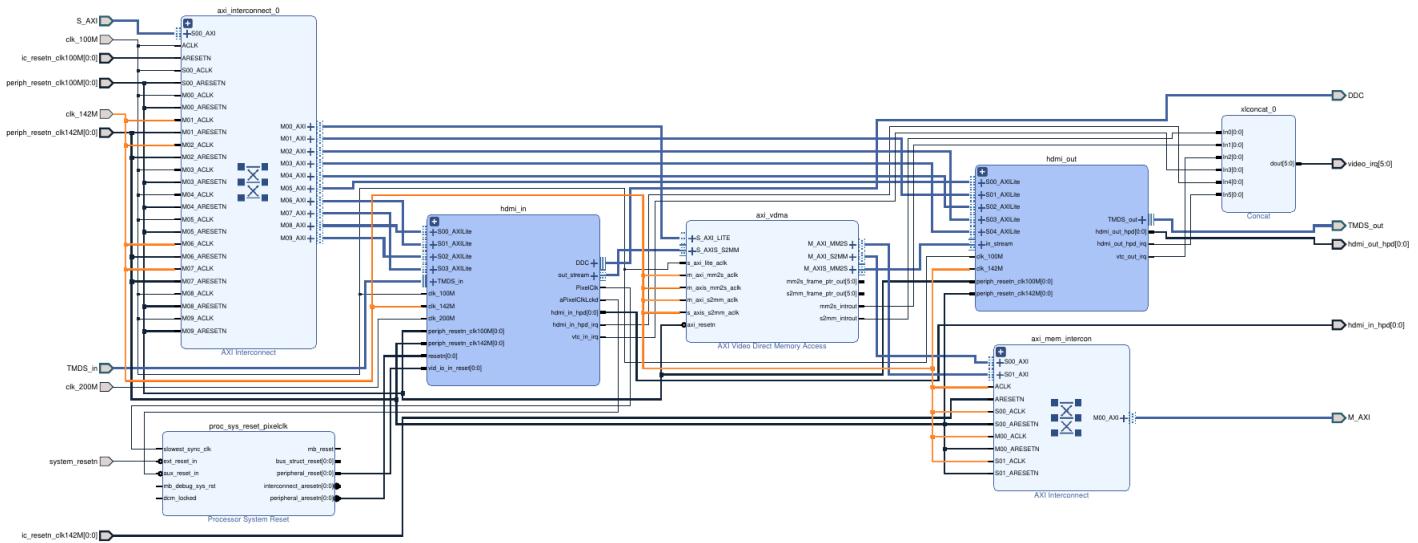


FIGURE 17 – *Block design* de l'*overlay* contenant le VDMA

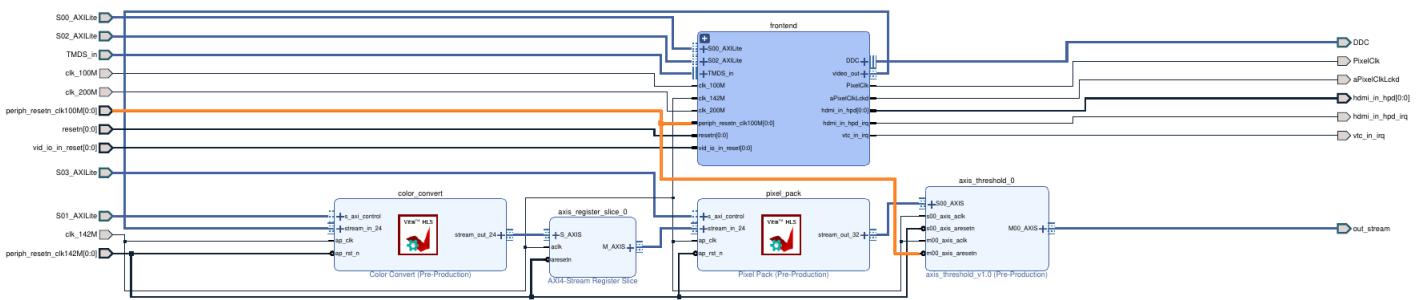


FIGURE 18 – *Block design* contenu dans HMDI_IN

De cette manière nous passons entièrement par le PL en nous passant complètement du PS afin d'avoir de meilleures performances. En effet, le processus sera plus rapide et nous aurons davantage d'images par seconde en sortie. Nous avons mesuré que le traitement d'une image via notre IP customisé était de 0,1 seconde en utilisant la DMA. Si on suppose qu'avec la VDMA, le temps de traitement pour une profondeur de 4 *frames*, comme on le voit dans la figure 19, est de 0,1 seconde également, on pourrait atteindre un débit de 40 images par seconde.

AXI Video Direct Memory Access (6.3)

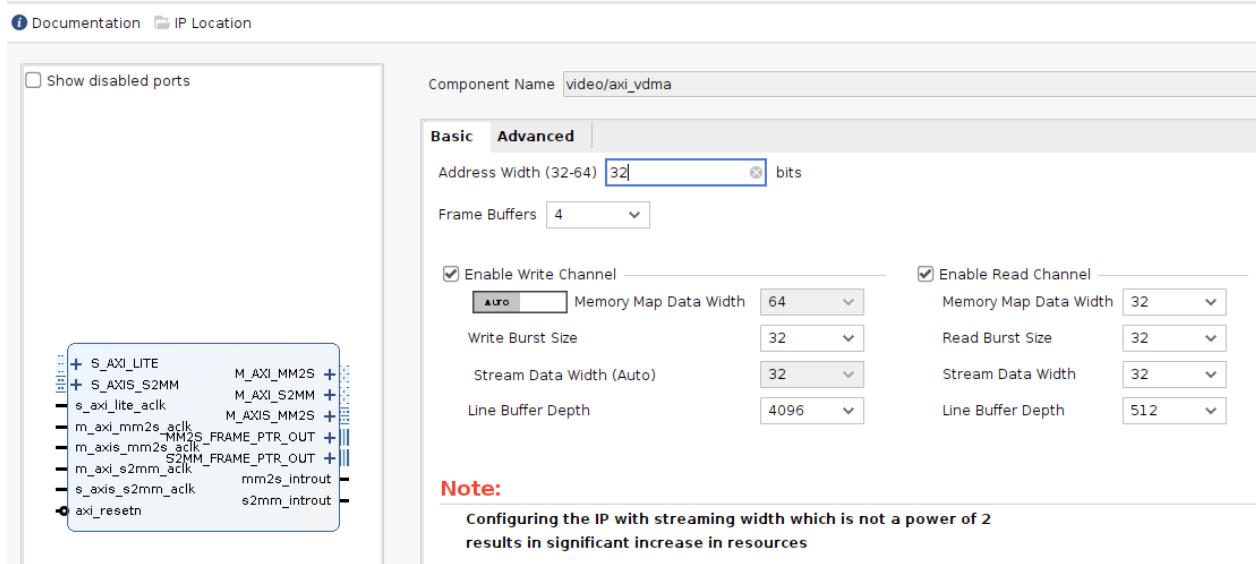


FIGURE 19 – Fenêtre de paramétrage de l'IP VDMA sur Vivado

Cependant, une limite à cela est le temps de développement du processus. En effet, si nous devions procéder de la sorte pour des projets de plus grande envergure, le temps de conception de nos *overlays* deviendrait bien trop important pour atteindre les objectifs fixés. Développer des projets de grande ampleur sur FPGA est donc à réserver pour des applications qui nécessitent cet investissement en temps. Il peut s'agir d'applications dans les domaines militaires ou médical par exemple. Par ailleurs, le temps de génération du *bitstream* prendrait lui aussi énormément du temps puisque nous en venons à utiliser quasiment la totalité des cellules du FPGA.

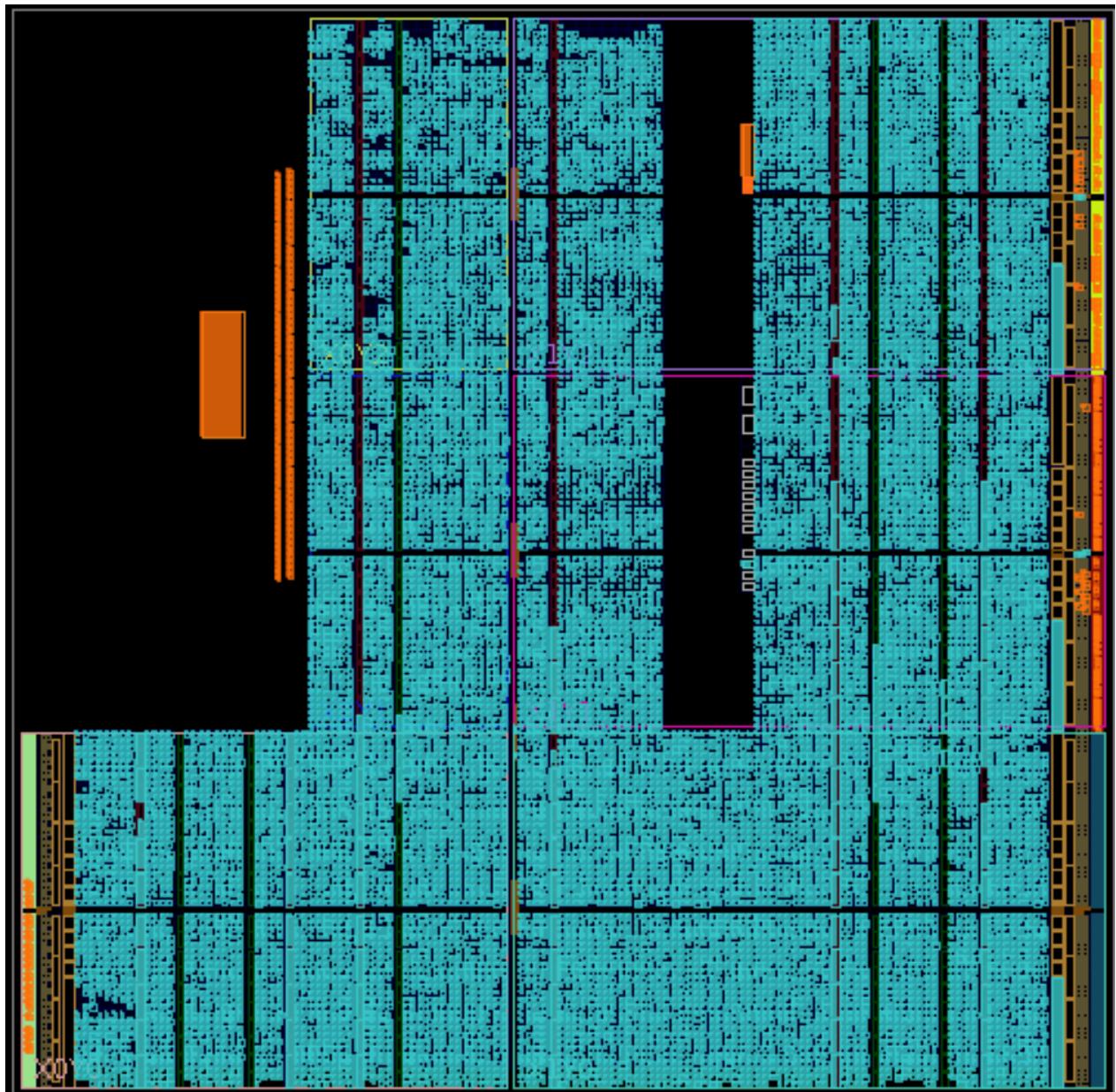


FIGURE 20 – Implémentation du design sur la PYNQ-Z2

Comme on peut le voir sur la figure 20, les parties en bleu sont les cellules utilisées. Ainsi, on voit qu'on est proche de limite du nombre de cellules.

Resource	Utilization	Available	Utilization %
LUT	35876	53200	67.44
LUTRAM	1624	17400	9.33
FF	50551	106400	47.51
BRAM	77	140	55.00
DSP	18	220	8.18
IO	119	125	95.20
MMC	3	4	75.00

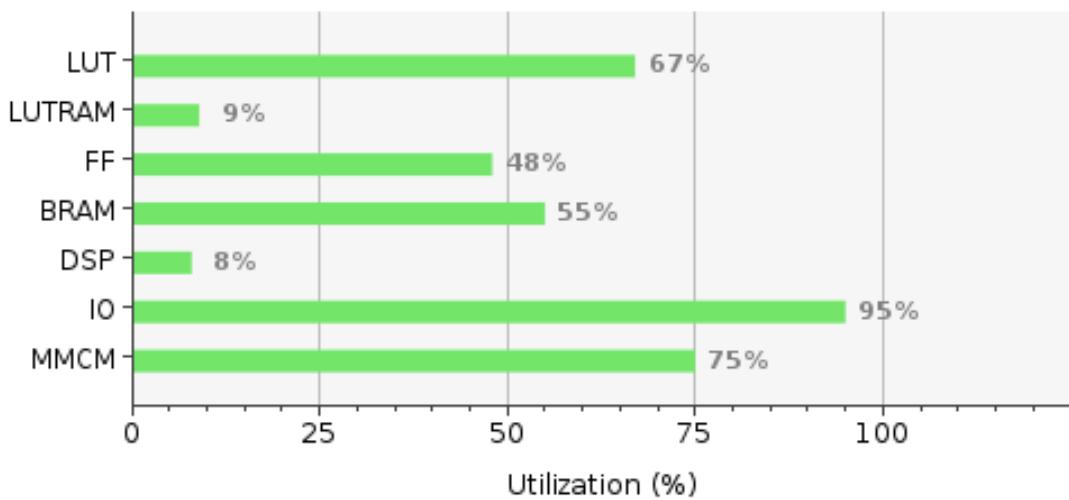


FIGURE 21 – Pourcentages d'utilisation des ressources du FPGA (LUT, DSP...)

Dans le graphique de la figure 21 on peut observer qu'on utilise 67.44% des LUT disponibles du FPGA, ce qui est assez conséquent. Cependant, vu qu'on ne réalise pas de calcul intensif mais seulement un filtrage au niveau d'une DMA on utilise très peu de DSP.

Notre design sur Vivado est accepté par l'outil de synthèse et nous pouvons générer un *bitstream* pour charger notre *overlay* personnalisé sur la carte qui permet d'utiliser les ports HDMI. Néanmoins, nous faisons face à une erreur probable dans notre design qui empêche le flux vidéo de l'entrée HDMI d'arriver au port HDMI de sortie. Au départ, notre objectif était d'obtenir un filtre seuil sur le flux vidéo, dont la valeur pourrait être modifiée par les interrupteurs et boutons sur la carte pour réaliser un changement au niveau matériel.

7 Conclusion

Pour résumer les étapes de notre travail sur l'accélération matérielle sur FPGA, nous avons tout d'abord utilisé des outils d'implantations de systèmes d'exploitations au lieu de passer par une programmation dite *bare metal*. Pour réaliser cela, nous avons utilisé Petalinux qui permet de faire une compilation croisée du noyau pour le *flasher* directement sur une carte FPGA. Il existe de nombreux problèmes d'incompatibilités entre le système d'exploitation utilisé, la version de Petalinux et la version de Vivado, ce qui a rendu cette étape de notre travail assez complexe, nous sommes donc passés par l'utilisation d'une machine virtuelle pour respecter les spécifications de système d'exploitation.

C'est ainsi que dans la suite de notre projet l'utilisation de Pynq s'est révélée beaucoup plus simple. En effet, le système d'exploitation nous offre la possibilité de faire de la reconfiguration de l'*overlay* en modifiant dynamiquement le *bitstream* [12] nous autorisant à basculer rapidement d'un *design* à un autre. Néanmoins, l'atout majeur est de pouvoir interagir avec le matériel depuis les *Notebooks Jupyter* en utilisant des fonctions python de haut niveau. Par conséquent, cela nous permet de réaliser des fonctions assez complexes tout en maximisant les performances de celle-ci grâce au *design* que l'on a conçu. Le tout se fait par ailleurs dans un intervalle de temps très réduit, une fois le circuit réalisé.

Le but final est de pouvoir réaliser une accélération matérielle en combinant tous ces éléments. Dans le cas d'un FPGA, nous faisons l'accélération matérielle en contrôlant le *Processing System* avec le *Programmable Logic*, pour après comparer les performances en regardant le PS isolé et l'association PS-PL [15]. Dans ce projet, nous nous sommes d'abord familiarisé avec l'API PYNNQ en créant de petits algorithmes utilisant le design de base du FPGA PYNNQ-Z2, pour ensuite mettre en place une accélération matérielle sur un filtre FIR. Par ailleurs, nous avons généré des *overlays* contenant des IPs que nous avons customisés. En particulier, nous avons conçu un IP permettant de réaliser un filtre seuil sur une image et nous avons réussi à l'interfacer aux ports GPIO.

Toutefois, si nous avions plus de temps nous aurions aimé réaliser ce seuil via un flux vidéo au lieu d'une image fixe.

Bien qu'il existe quelques difficultés dans l'utilisation de la plateforme Pynq (lenteur de l'interprète python, mise à jour du dépôt *git* peu fréquente causant des difficultés d'utilisation...), elle reste un outil qui permet une grande souplesse pour le prototypage une fois qu'on a générée nos architectures. Cependant, le manque de maintenance du dépôt *git* laisse à penser, selon nous, qu'il y a un manque d'engouement de la part de la communauté ce qui s'avère dommage puisqu'il y a beaucoup de potentiel. Selon nous, on peut considérer la carte PYNNQ-Z2 comme une bonne option dans une optique pédagogique mais trop faible pour une utilisation professionnelle. En effet, au cours de projet nous avons par exemple envisagé d'implémenter un filtre de Sobel. Cependant, nous n'y sommes pas parvenus à cause du faible nombre de cellules présentes dans le FPGA. Toutefois, avec des FPGA plus conséquents, nous pensons que cette plate-forme peut se révéler très bénéfique en terme de productivité. En effet, il permettrait de réaliser rapidement des projets de grande envergure puisqu'une partie des ingénieurs ayant un profil d'électronicien pourrait s'occuper de réaliser des *overlays* tandis que l'autre (les ingénieurs ayant un profil d'informaticien) pourrait s'occuper de les tester très facilement sans avoir besoin de connaissances en langage de description matérielle.

Bibliographie

- [1] <https://www.amd.com/fr/corporate/university-program/aup-boards/pynq-z2.html>.
- [2] Filtre de Sobel — Wikipédia — fr.wikipedia.org. https://fr.wikipedia.org/wiki/Filtre_de_Sobel.
- [3] Ddr sdram, 2023. URL : https://en.wikipedia.org/wiki/DDR_SDRAM.
- [4] Finite impulse response, 2024. URL : https://en.wikipedia.org/wiki/Finite_impulse_response.
- [5] AMD. What is pynq? URL : <http://www.pynq.io/>.
- [6] Bootlin. Embedded Linux system development training. URL : <https://bootlin.com/doc/training/embedded-linux/embedded-linux-slides.pdf>.
- [7] International Electrotechnical Commission, 2001. URL : <https://www.electropedia.org/iev/iev.nsf/display?openform&ievref=151-13-55>.
- [8] Futura DSciences. Cpu, 2023. URL : <https://www.futura-sciences.com/tech/definitions/informatique-cpu-5741/>.
- [9] Inc. Efinix. RISC-V and FPGAs — efinixinc.com. <https://www.efinixinc.com/blog/riscv-and-fpgas.html>.
- [10] Damien Fruleux. Présentation de pynq et de son exploitation dans une expérience de cryptographie quantique.
- [11] Peter Isza. The free online fir filter design tool, 2011. URL : <http://t-filter.engineerjs.com/>.
- [12] Benedikt Janßen, Pascal Zimprich, and Michael Hübner. A dynamic partial reconfigurable overlay concept for pynq. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017. doi:10.23919/FPL.2017.8056786.
- [13] Jeff Johnson. Creating a custom axi-streaming ip in vivado, 2017. URL : <https://www.fpgadeveloper.com/2017/11/creating-a-custom-axi-streaming-ip-in-vivado.html>.
- [14] Jeff Johnson. How to accelerate a python function with pynq, 2017. URL : <https://www.fpgadeveloper.com/2018/03/how-to-accelerate-a-python-function-with-pynq.html/>.
- [15] Vineeth C Johnson, Jyoti Bali, Shilpa Tanvashi, and C B Kolanur. Fpga-based hardware acceleration using pynq-z2. In *2023 Second International Conference on Electrical, Electronics, Information and Communication Technologies (ICEEICT)*, pages 1–4, 2023. doi:10.1109/ICEEICT56924.2023.10157764.

- [16] Sam K. Basys 3 - Digilent Reference — digilent.com. <https://digilent.com/reference/programmable-logic/basys-3/start>.
- [17] Cathal McCabe. Tutorial : Rebuilding the pynq base overlay (pynq v2.6), 2012. URL : <https://discuss.pynq.io/t/tutorial-rebuilding-the-pynq-base-overlaypynq-v2-6/1993>.
- [18] François MOCQ. Un point sur le device tree, 2015. URL : <https://www.framboise314.fr/un-point-sur-le-device-tree/>.
- [19] Centre national de ressources textuelles et lexicales, 2012. URL : <https://www.cnrtl.fr/definition/interfa%C3%A7age>.
- [20] Graham Schelle. Pynq release. URL : <https://github.com/Xilinx/PYNQ/releases>.
- [21] Adam Taylor. Understanding fir filters with pynq, 2024. URL : <https://www.hackster.io/adam-taylor/understanding-fir-filters-with-pynq-33395e>.