Implémentation d'un réseau de neurones sur FPGA

Paul Luperini
Lucas Mahieu
Hugues de Valon

encadré par
Frédéric PETROT
et
Adrien PROST-BOUCLE

dans le cadre du Projet SLE 3A Ensimag 2016/2017

2 février 2017

Table des matières

Remerciements

Nous remercions chaleureusement toutes les personnes qui nous ont aidés durant ce projet et notamment nos tuteurs Frédéric Pétrot, pour le sujet et son soutien, et Adrien Prost-Boucle, pour son encadrement, sa grande disponibilité et ses nombreux conseils durant le projet.

Ce travail n'aurait pu être mené à bien sans la disponibilité et l'aide que nous a témoigné Robin Rolland au CIME Nanotech, pour paramétrer l'environnement *Vivado*, l'accès au matériel et ses conseils lorsque nous rencontrions des problèmes sur *Vivado*.

1 Introduction

De nos jours, les réseaux de neurones artificiels reprennent de plus en plus d'importance car la puissance de calcul disponible permet d'obtenir des résultats satisfaisants en temps raisonnable. Le traitement d'images, la reconnaissance vocale ou les traitements lexicaux sont des applications qui pourraient être intégrées dans des systèmes embarqués. Pour ce type d'application, il est possible d'implémenter sur des CPU ou GPU des algorithmes neuronaux, mais la consommation et la vitesse de traitement deviendraient vite limitants.

Créer un composant électronique (ASIC ou une IP FPGA dans un premier temps) implémentant un réseau de neurones réduirait drastiquement sa consommation et améliorerait la vitesse du réseau par rapport à un CPU. De plus, étant donné que l'IP serait spécialisée pour cette application et paramétrable dynamiquement, cela lui permettrait de s'adapter à de multiples applications en ayant des performances élevées.

Le projet "Réseau de neurones sur FPGA" s'inscrit dans ce cadre. Sous le tutorat de Frédéric Pétrot et Adrien Prost-Boucle, nous devons créer un tel composant, et tester ses performances en vue de le comparer à des systèmes existants tels que la puce *SpiNNaker* [1] ou *TrueNorth* [2] d'IBM ou encore des systèmes en développement tels que les réseaux de neurones ternaires du laboratoire TIMA. Une fois implémenté et validé, nous utiliserons une carte FPGA ZedBoard pour tester notre composant sur une application classique de reconnaissance de chiffres manuscrits, en utilisant la base de données MNIST [3].

2 Cahier des charges détaillé

2.1 Présentation générale du problème

2.1.1 Projet

Le but du sujet est de réaliser un réseau de neurones sur FPGA. Un réseau de neurones est un algorithme schématiquement inspiré du fonctionnement des neurones biologiques. On représente un réseau de neurones par un certain nombre de niveaux de plusieurs neurones. Un neurone est représenté par un noeud qui reçoit des données de la part des neurones du niveau précédent et diffuse sa valeur de sortie aux neurones du niveau suivant. Les opérations effectuées sur les données par chaque neurone sont assez simples, ce sont des multiplications à accumulations : $\sum_{i=0}^{n} w_i * d_i$, avec n données entrantes dans le réseau, w_i le poids pour l'entrée i et d_i la donnée venant du neurone i du niveau précédent. Le nombre de niveaux et le nombre de neurones sont des paramètres qui peuvent être dimensionnés en fonction de l'application voulue. Sur la figure 1 page 6 vous trouverez un schéma représentant un exemple de réseau de neurones à 3 niveaux.

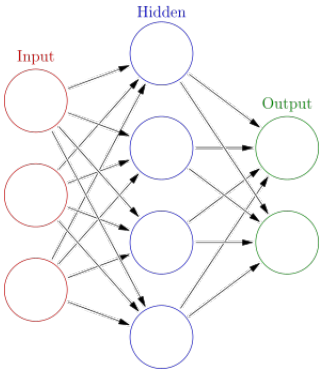

FIGURE 1 – Schéma d'un réseau de neurones à 1 niveau "caché" de 4 neurones avec 3 entrées et 2 sorties

2.1.2 Finalités

Ce réseau de neurones sur FPGA a pour but d'établir une référence pour pouvoir comparer un réseau de neurones ternaire en cours de développement au laboratoire TIMA à un réseau de neurones plus classique, tous deux réalisés sur le même matériel.

Un objectif secondaire serait de comparer le composant réalisé à d'autres produits similaires tels que la puce neuromorphique *SpiNNaker* [1] de l'université de Manchester, ou bien la puce *TrueNorth* [2] d'IBM.

2.1.3 Contexte

Le projet sera réalisé au CIME Nanotech, à Grenoble.

Quatre heures par semaine sont allouées, dans notre emploi du temps, à la réalisation de ce projet. Cependant, il est nécessaire de travailler en dehors des horaires prévus pour terminer le projet.

Les tests sur carte FPGA seront réalisés dans un premier temps sur une carte Zybo, puis une fois la fonctionnalité du composant validée, les tests se poursuivront sur carte ZedBoard.

2.1.4 Énoncé du besoin

Les finalités du produit pour le futur utilisateur telles que prévues par le demandeur sont :

- il faut concevoir un logiciel de référence. Celui ci doit respecter les caractéristiques du réseau de neurones demandé, et produire un résultat qui servira de référence à l'IP conçue ;
- le composant matériel doit pouvoir être générique, c'est-à-dire qu'il doit être possible de changer le nombre de neurones, les coefficients de chaque neurone ou encore la taille des données d'entrées, sans la moindre adaptation de code (simplement en changeant les variables voulues dans un fichier) ;
- le composant matériel doit produire le résultat attendu, c'est-à-dire qu'il doit produire un résultat identique au programme de référence ;
- le produit étant fortement technique et devant être utilisé par des personnes n'ayant pas conçu cette IP, une documentation détaillée doit être fournie.

2.2 Expression fonctionnelle du besoin

2.2.1 Fonctions de service et de contrainte

Fonctions de service principales

Le produit doit calculer le résultat d'une donnée d'entrée soumise à un réseau de neurones, paramétré selon les coefficients précédemment donnés au composant. Dans le cadre de ce projet, les données d'entrée seront les images issues de la base de données MNIST [3] composée d'image de caractères manuscrits. Ainsi le but premier du réseau sera de trouver quel caractère lui a été donné en entrée.

Fonctions de service complémentaires

Pour une question d'évolution et de réutilisation du projet, le produit rendu doit être reconfigurable très rapidement. En modifiant seulement quelques variables, le produit doit être capable de s'adapter à d'autres applications que celle prévue pour ce projet.

Contraintes

L'architecture globale du composant et ses interfaces (bus, interconnexions, ...) sont déterminées par le laboratoire TIMA.

Le composant devra utiliser les cellules DSP du FPGA pour réaliser la multiplication à accumulation d'un neurone. De plus, il devra utiliser des BRAM pour stocker les coefficients nécessaires aux calculs.

Décomposition en modules, sous-ensembles

L'IP conçue s'intègre dans un environnement complexe. Dans une même puce, se trouvent un FPGA, deux coeurs ARM et de nombreuses autres IP telles qu'un DMA, de la mémoire et un bus permettant de faire communiquer toutes ces IP entre elles. Le réseau de neurones est donc implanté dans le FPGA, et est composé dans l'ordre de propagation des données de :

- une FIFO permettant de récupérer les données envoyées par le micro-processeur via le bus ;
- un premier niveau de plus de cents neurones ;
- chaque neurone produit un résultat stocké dans une deuxième FIFO ;
- les données produites par le premier niveau de neurones passent par une fonction non linéaire ;
- ces données ainsi traitées sont envoyées par une FIFO au deuxième et dernier étage de neurones comptant 10 neurones (1 par chiffre à prédire) ;
- les résultats finaux sont ainsi renvoyés vers le micro-processeur via le bus AXI de la puce.

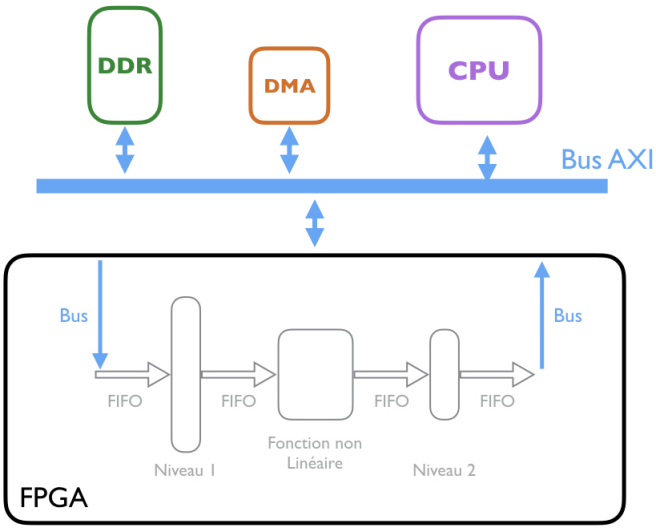

FIGURE 2 – Schéma haut niveau de l'architecture du réseau de neurones et de son environnement dans le FPGA Xilinx

2.2.2 Critères d'appréciation

Les critères permettant de mesurer la qualité du composant produit sont :
- correction du composant : le résultat doit être celui attendu ;
- taux d'utilisation des cellules du FPGA ;
- performances du composant (fréquence, nombre de cycles pour calculer le résultat ...) ;
- généricité du composant.

2.2.3 Niveaux des critères d'appréciation

Niveaux dont l'obtention est imposée

Il est nécessaire que le composant satisfasse les niveaux de critères suivants :
- correction : le résultat doit être correct ;
- taux d'utilisation des cellules du FPGA : un neurone doit utiliser une cellule DSP.

Niveaux souhaités mais révisables

Il est souhaitable que le composant satisfasse les niveaux de critères suivants :
- performances : Le résultat d'un calcul du composant doit être plus rapide que son équivalent réalisé sur un processeur classique ;
- généricité : Les fichiers HDL du composant doivent pouvoir être modifié de façon mineure pour changer les paramètres du réseau de neurones.

3 Étapes de conception

Nous allons présenter dans cette partie les principales étapes de conception de notre réseau de neurones en détaillant tous les blocs réalisés et les choix faits à propos de leur architecture.

3.1 Programme de référence

Le programme de référence implémente le même réseau de neurones que notre IP mais directement en C. On peut ainsi l'executer sur un PC ou un micro-processeur et comparer les résultats avec l'implémentation réelle sur FPGA. On comparera la valeur des résultats de sortie et le temps d'exécution entre logiciel et matériel.

3.1.1 L'algorithme

```
for f ← 0 to frames do
    out1[neurons] ← 0 ;
    out2[10] ← 0 ;
    for r ← 0 to rows do
        for c ← 0 to columns do
            for n ← 0 to neurons do
                out1[n] ← out1[n] + get_pixel(f, r, j) × w1[n][r][c] ;
            end
        end
    end
    for n ← 0 to neurons do
        out1[n] ← cut(out1[n]) ;
        out1[n] ← out1[n] + b1[n] ;
        if out1[n] < 0 then
            out1[n] ← 0 ;
        end
        out1[n] ← cut(out1[n]) ;
    end
    for n ← 0 to neurons do
        for i ← 0 to 9 do
            out2[i] ← out2[i] + out1[n] × w2[i][n] ;
        end
    end
    for i ← 0 to 9 do
        out2[n] ← cut(out2[n]) ;
        out2[n] ← out2[n] + b2[n] ;
        out2[n] ← cut(out2[n]) ;
    end
    assert(max(out2) == get_label(f)) ;
end
```

Algorithm 1 : Boucle de calcul principal du réseau de neurone logiciel

L'algorithme 1 page 9 implémente un réseau de neurones à deux étages. Chaque étage réalise des multiplications et accumulations pour chaque pixel de chaque image avec des poids, spécifiques à chaque neurone. Après chaque étage, le résultat de chaque neurone est additionné avec une constante.

3.1.2 Les données à traiter dans le réseau

La configuration actuelle de ce réseau de neurone est adaptée pour la reconnaissance de chiffres manuscrits de la base de données MNIST [3]. Une image en entrée (28×28 pixels) correspond à un chiffre manuscrit de 0 à 9. Le but de ce réseau de neurone est de reconnaître et d'identifier le chiffre d'une image. Pour une image donnée, le chiffre identifié par le réseau correspond au numéro du neurone de sortie qui a la plus grande valeur.
Dans ce modèle logiciel, les pixels sont encodés sur un octet, les poids et les constantes sont sur 2 octets. Afin que dans le pire cas, aucune information ne soit perdue lors de l'accumulation aux différents étages, on réalise cette opération sur 8 octets. Cependant, afin que le logiciel de référence reproduise le même comportement que notre IP, on ne peut faire transiter entre deux étages uniquement des données sur 4 octets. C'est le rôle de la fonction `cut` qui permet de passer de 64 à 32 bits tout en conservant le signe. Notre IP garde uniquement les 32 bits de poids faible de ce mot de 64 bits, cependant des essais en software pour 100 et 200 neurones ont permis de découvrir des meilleurs masques qui augmentaient le taux de succès jusqu'à $0,5\%$ (table 1 page 10).

Masque	Taux de succès (100 neurones)	Taux de succès (200 neurones)
`0x00000000FFFFFFFF`	93,8	94,4
`0x00000007FFFFFFF8`	93,9	94,5
`0x0000000FFFFFFFF0`	93,9	94,7
`0x0000001FFFFFFFE0`	94,3	94,8
`0x0000003FFFFFFFC0`	79,4	89,4
`0x000000FFFFFFFF00`	8,5	8,5

TABLE 1 – Essais de différents masques et leur taux d'erreur

En choisissant un autre masque que `0x00000000FFFFFFFF` il est nécessaire d'effectuer un décalage à droite ensuite.

3.1.3 Mise en forme des données MNIST

Les poids des neurones et les constantes des étages de recodage pour l'application MNIST [3], que nous utilisons dans le logiciel de référence et dans notre IP nous ont été fournis au début du projet sous forme de fichier texte où sont présentes toutes les données, séparées par des virgules, et où chaque ligne correspond à un neurone. Un script Python nous a permis d'analyser ces fichiers pour récupérer les données utiles et de les replacer dans un unique fichier créé `net.c` avec les définitions de 4 tableaux d'entiers. Ce fichier est inclus et compilé dans le logiciel de référence et pour le logiciel embarqué ; ainsi les poids et les constantes sont facilement accessibles depuis le logiciel.
Concernant les images, nous les avons récupérées à partir du site officiel MNIST (`yann.lecun.com/exdb/mnist/`) sous forme de fichiers binaires où 1000 images de taille 28×28 pixels sont représentées comme des tableaux d'octets en C. Un autre fichier binaire nous permet de connaître le résultat, c'est-à-dire le chiffre, de chacune de ces images afin de comparer le résultat de notre réseau de neurones avec le résultat attendu. Les fonctions `get_pixel` et `get_label` permettent d'obtenir facilement ces informations depuis le logiciel.

3.1.4 Les résultats

Les résultats du logiciel de référence en sortie de chacun des niveaux nous sont utiles pour les comparer avec ceux de notre IP et s'assurer que les calculs sont justes pour la classification des images MNIST [3]. On s'attend à trouver un même taux d'erreur (93.80%) car les configurations sont identiques mais un calcul beaucoup plus rapide de toutes les images.

3.2 IP du réseau de neurones

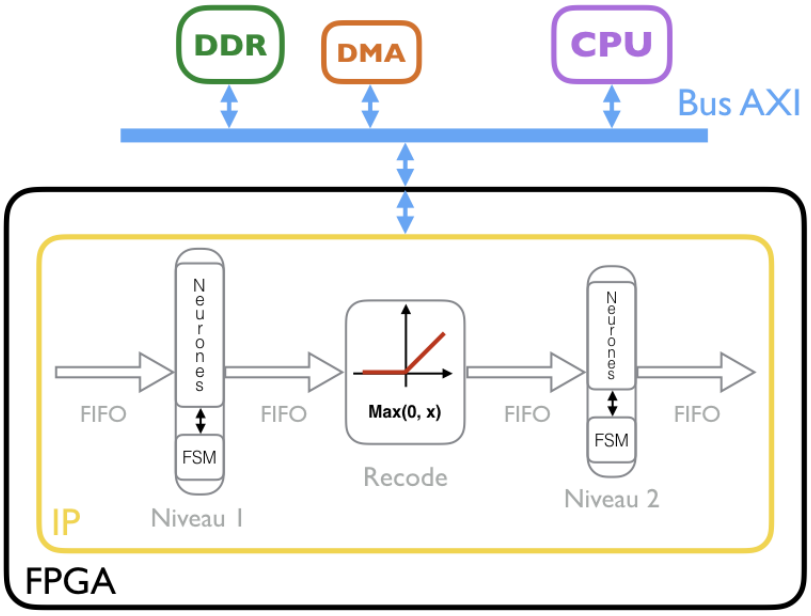

FIGURE 3 – Schéma de l'IP réseau de neurones dans son environnement

3.2.1 Le neurone

Le neurone est l'élément de base de notre réseau. C'est donc la première chose que nous avons implémenté. Pour faire cela, nous avons pensé le neurone en terme de fonctionnalités uniquement. Nous allons donc décrire ici la façon dont nous avons conçu un neurone. Un neurone a deux modes de fonctionnement. Le premier est le mode de chargement des poids en mémoire, le second est le mode de calcul.

Le mode de chargement des poids

Pour permettre aux neurones d'accomplir leur rôle principal, il faut que chaque neurone ait accès à ses poids. Étant donné que chaque neurone doit disposer de ses propres poids (identiques pour une application donnée et donc pour une grandes quantité d'images) il est donc préférable que chaque neurone les mémorise. Afin d'accomplir cela, les neurones reçoivent une adresse (numero du poids à mémoriser) ainsi qu'un poids. Alors le neurone peut mémoriser le poids présent sur son entrée dans sa BRAM à l'adresse reçue à condition que ce neurone possède le *token* de configuration et qu'il reçoive le signal lui indiquant qu'un poids est présent sur son entrée.
Le token de configuration est passé de neurone en neurone sur ordre du signal de contrôle pour qu'ils se configurent l'un après l'autre. Vous trouverez une illustration de ce fonctionnement en figure 4 page 12.

Le mode de calcul

Le mode de calcul est le mode principal d'un neurone. C'est dans ce mode que le neurone reçoit des données et qu'il effectue une succession de multiplications à accumulations entre la valeur du pixel reçu et le poids correspondant à l'adresse qui lui est donnée. L'opération

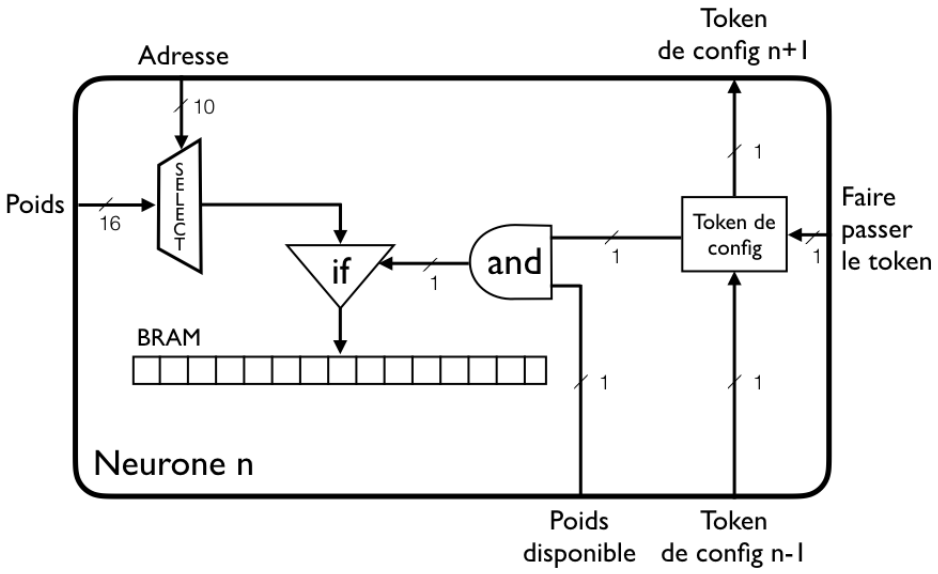

FIGURE 4 – Schéma fonctionnel de l'implémentation du neurone en mode de configuration

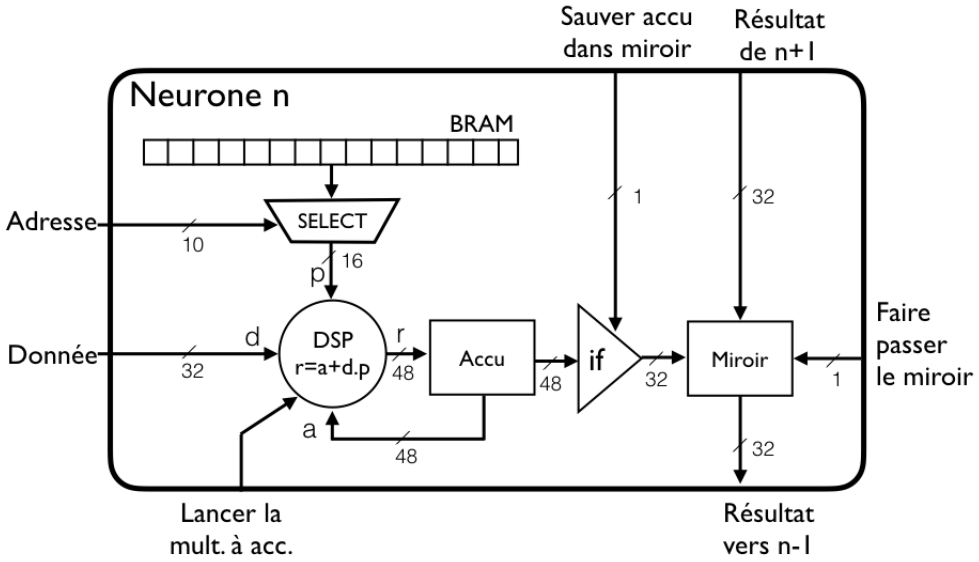

FIGURE 5 – Schéma fonctionnel de l'implémentation du neurone en mode de calcul

clé d'un neurone peut être écrite comme :

$$\sum_{adr=0}^{n} p_{adr} * d$$

avec n données entrantes dans le réseau, p_{adr} le poids à l'adresse adr et d la donnée entrante.

En effectuant cette opération sur chaque pixel d'une image, un neurone du premier niveau du réseau est capable de produire un résultat par image. Les neurones des niveaux suivant ne reçoivent pas des pixels mais les résultats du niveaux précédent et opère de la même façon

pour enfin produire chacun un résultat (voir figure 5 page 12).
Notez qu'entre deux niveaux les résultats passent par l'étage de recodage (cf. Partie 3.2.4 "L'étage de recodage").

Une fois que chaque neurone a produit son résultat, ils vont en faire une copie dans un registre appelé *miroir* et remettre le registre d'accumulation à 0. Ainsi, le calcul de l'image suivante pourra être commencé alors que le résultat de l'image précédente sera évacué vers la FIFO de sortie suivante. Pour ce faire, à la réception du signal `sauver_accu_dans_miroir`, chaque neurone va faire passer la valeur de son registre miroir au neurone suivant et récupérer la valeur du neurone précédent. Cela permet de ne connecter qu'un neurone à la FIFO et de faire évacuer les données les unes après les autres dans la FIFO (voir figure 5 page 12).

3.2.2 La machine à états (FSM) du niveau de neurones

La machine à états ou FSM (Finite State Machine) contrôle l'ensemble des neurones d'un niveau de neurones. Elle a été conçue de façon à contrôler un nombre de neurones indéfini : qu'il y ait 1 ou 100000 neurones, la machine à états reste la même.

Les signaux de contrôle de la FSM sont reliés à tous les neurones.

Pour des raisons de simplicité le contrôle des neurones est séparé en deux machines à états qui communiquent entre elles par le biais d'un drapeau.

La machine à états de contrôle des neurones

L'ensemble de cette machine à états est visible dans la figure 6 page 14.

Dans le mode accumulation, il n'y a pas besoin de distinguer chaque neurone, on se contente donc d'envoyer les signaux de contrôle à tous les neurones. Dans le mode chargement des poids de neurones, il faut être capable de commander un neurone à la fois. Pour cela, la FSM génère un token de configuration au début de la phase de chargement des poids et le donne au premier neurone. Sans ce token, le neurone est inactif dans cette phase. Une fois que la machine à états a fini de charger les poids d'un neurone, elle déclenche un décalage du token de configuration entre les neurones. Une fois le dernier neurone atteint, le décalage du token le renvoie sur la FSM qui sait donc qu'elle a fini de configurer l'ensemble des neurones.

Voici de courtes explications pour les différents états :

— `RESET_STATE` : état de reset ;
— `FSM_MODE` : choix du mode de fonctionnement du niveau de neurones, soit chargement des poids, soit accumulation ;
— `MODE_ACC` : on est en mode accumulation, on le signale aux neurones, et on attend qu'ils aient pris en compte le changement ;
— `WAIT_1D` : on attend que la première donnée soit présente dans la FIFO d'entrée pour commencer l'accumulation ;
— `SEND_DATA` : accumulation des neurones. Si jamais on a atteint la fin d'une image on le notifie à la machine à états de la chaîne de miroir, sinon on attend une nouvelle donnée ;
— `WAIT_DATA` : on attend une nouvelle donnée ;
— `END_ACC` : on a fini l'accumulation sur une image, on retourne dans l'état de choix de mode de la FSM ;
— `MODE_WEIGHT` : on est en mode chargement des poids, on le signale aux neurones, et on attend qu'ils aient pris en compte le changement ;
— `NOTIFY_1N` : on passe le token de configuration au premier neurone et on attend qu'il nous confirme sa réception ;
— `SEND_WEIGHT` : chargement du poids dans la mémoire du neurone. Si jamais on a atteint la fin d'un neurone, on décale le token de configuration, sinon on attend un nouveau poids ;
— `WAIT_WEIGHT` : on attend un nouveau poids ;
— `SHIFT_NOTIFY` : on décale le token de configuration ;

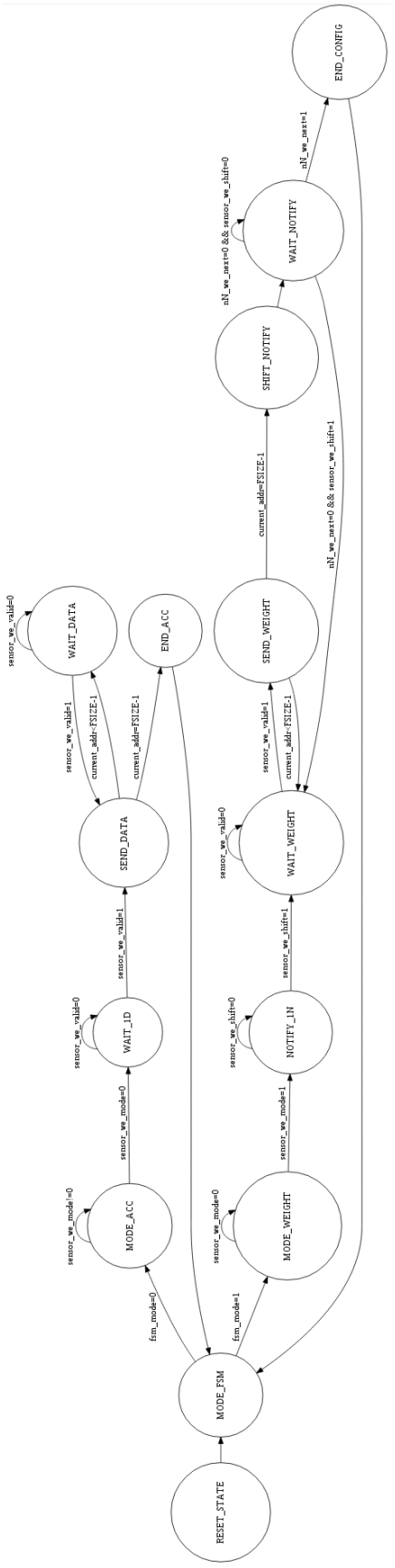

FIGURE 6 – Machine à états de contrôle des neurones

— `WAIT_NOTIFY` : on attend que les neurones aient bien décalé le token de configuration, si jamais la FSM le reçoit cela signifie que l'on a configuré tous les neurones, dans ce cas on passe en fin de configuration, sinon on continue de configurer les neurones suivants ;
— `END_CONFIG` : on a fini le chargement des poids, on retourne dans l'état de choix de mode de la FSM.

La machine à états de la chaîne de miroirs

L'ensemble de cette machine à états est visible dans la figure 7 page 15. Elle est chargée de décaler les miroirs de la chaîne de miroirs des neurones pour les transmettre à la FIFO de sortie.

FIGURE 7 – Machine à états de la chaîne de miroir

Voici de courtes explications pour les différents états :
— `SHIFT_MODE` : état d'attente, si la FSM de contrôle des neurones passe le drapeau de communication à 1, il faut commencer à décaler la chaîne de miroir ;
— `SHIFT_CPY` : copie de la valeur des registres d'accumulation de chaque neurone dans leur miroir respectif ;
— `WAIT_SHIFT_CPY` : on attend que les neurones aient copiés leur registre dans le miroir ;
— `SHIFT` : on décale les miroirs de chaque neurone dans le neurone suivant uniquement s'il y a suffisamment de place dans la FIFO de sortie. On prend en compte le temps de propagation des signaux de contrôle pour qu'il n'y ait pas de données perdues à cause d'un manque de place dans la FIFO de sortie.

3.2.3 Le niveau de neurones

Le niveau de neurones est l'entité de notre IP qui regroupe un certain nombre de neurones (100 ou 200 pour le premier niveau, 10 pour le deuxième) et une machine à états afin de les contrôler.
Afin de pouvoir relier les signaux de contrôle de la machine à états à plus de 100 ou 200 neurones, de simples fils auraient posés problème. En effet, pour deux neurones très éloignés dans le FPGA, les temps de propagation des données envoyées au même moment de la FSM n'auraient pas été identiques et cela aurait créé des problèmes de synchronisation. Pour cela, nous avons utilisé des buffers de distribution, déjà disponible, pour distribuer les signaux de contrôle de la FSM à tous les neurones ou les données d'entrée aux neurones. Un buffer de distribution est constitué d'un arbre de fils avec la racine connectée à un port de sortie de la FSM et les feuilles connectées à tous les neurones. Le nombre de fils fils de chaque noeud est un paramètre du buffer (2 dans la figure 8 page 16). L'avantage d'utiliser un buffer de distribution est qu'il certifie que le passage d'un niveau de l'arbre prend un cycle. On est alors assuré que la donnée arrivera en même temps à tous les neurones.
Certains signaux se connectent uniquement au premier ou au dernier des neurones et d'autres entre deux neurones consécutifs, ces connexions sont alors réalisées avec de simples fils. Pour gérer ces connexions on instancie tous les neurones grâce à la boucle VHDL :
`gen_neu : for i in 0 to NBNEU-1 generate`
où chacun des ports des neurones est relié à une case d'un tableau de signaux pour faciliter leur utilisation.

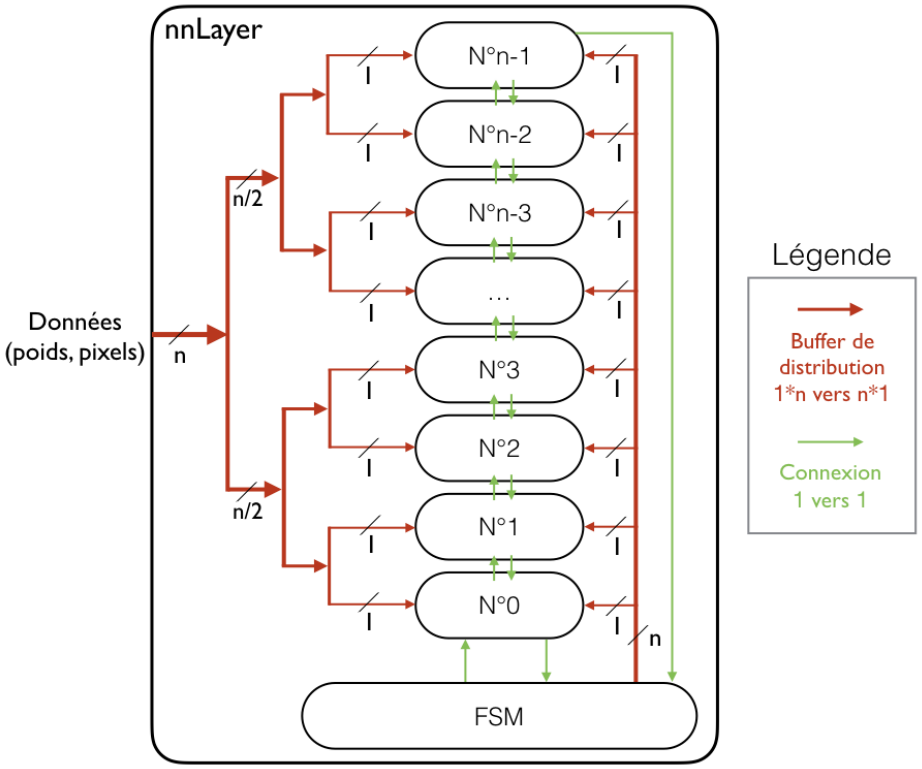

FIGURE 8 – Schéma d'un niveau de recode montrant les différents type d'interconnexion entre les différents éléments le constituant

3.2.4 L'étage de recodage

Situé après le niveau de neurones, l'étage de recodage possède deux fonctions : ajouter une constante aux valeurs sortants du niveau de neurones (chaque constante est propre à un neurone) et supprimer les valeurs négatives après ajout de constante.

L'étage est contrôlé par une machine à états visible sur la figure 9 page 17.

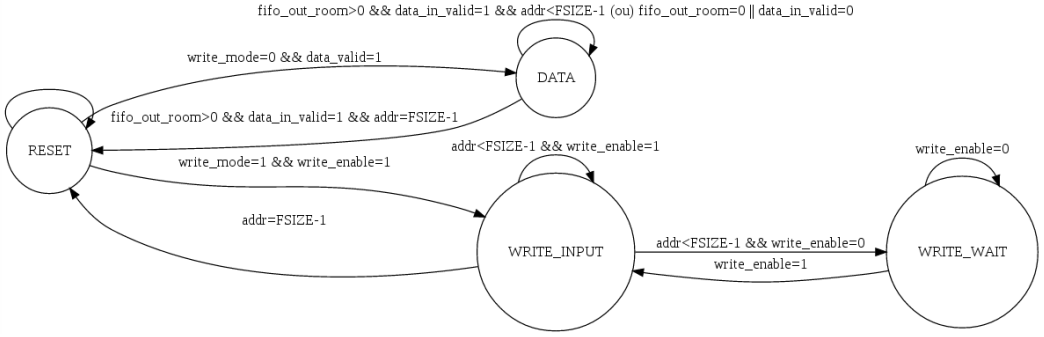

FIGURE 9 – Machine à états de l'étage de recodage

Voici de courtes explications pour les différents états :

— `RESET` : état de choix de mode et d'attente de données présentes. Si le mode est le traitement classique et qu'il y a une donnée disponible dans la FIFO d'entrée, on passe dans l'état de traitement des données. Si il s'agit du mode de chargement des poids et qu'il y a un poids disponible dans la FIFO d'entrée, on passe dans l'état de chargement des poids ;

— `WRITE_INPUT` : on écrit le poids dans la mémoire, si on a atteint la fin de la mémoire, on repasse dans l'état `RESET`, s'il y a une donnée disponible ensuite on continue d'écrire les poids dans la mémoire, sinon on attend qu'il y ait un poids disponible ;

— `WRITE_WAIT` : on attend qu'il y ait un poids disponible ;

— `DATA` : on effectue le traitement de la donnée en entrée seulement si : elle est disponible, il y a de place dans la FIFO de sortie et on n'a pas atteint la fin de la totalité des données sortant d'un décalage complet de la chaîne de miroirs. Le traitement consiste à ajouter une constante, et supprimer la valeur (envoyer 0 en sortie) si la valeur après ajout de constante est négative, sinon on envoie la valeur avec l'ajout de constante. S'il n'y a pas de place en sortie ou pas de donnée en entrée, on attend que les deux conditions soient valides. Si on a atteint la fin d'un décalage complet de la chaîne de miroirs, on repasse dans l'état de `RESET`.

3.2.5 Le pipeline complet

Une fois chaque élément ci-dessus créé, il a fallu les mettre ensemble pour accomplir le fonctionnement global du réseau. Il est important de réfléchir à la façon dont ces différents éléments échangent leurs données. En effet, les différents organes du réseau ne doivent pas calculer la même quantité de données, donc le temps de traitement d'une image n'est pas identique dans tout le réseau. C'est pourquoi, entre chaque élément du réseau (entrée, niveau de neurone 1, recodage, niveau 2, sortie) nous avons utilisé des FIFOs. Les FIFOs utilisées nous ont été fournies, elles ont une entrée et une sortie sur 32 bits, 64 cases mémoire de 32 bits, 4 signaux de contrôle et 2 signaux donnant le nombre de places disponibles et occupées. Deux des signaux de contrôle permettent à la FIFO d'avertir l'élément suivant qu'une donnée peut être sortie de la FIFO et à l'élément précédent qu'une donnée peut être écrite. Les deux autres signaux de contrôle permettent aux éléments d'avertir la FIFO qu'une donnée est lue

ou écrite (c'est un signal d'acquittement). L'utilisation des FIFOs du réseau est différente dans les deux modes du réseau.

Le pipeline en mode chargement des poids Lorsque le réseau de neurones est en mode configuration, chaque élément va récupérer les constantes dans la toute première FIFO. En effet, le DMA va faire un transfert de donnée de la DDR vers la FIFO d'entrée via le bus AXI. En connectant correctement les signaux de contrôle de la première FIFO aux différents éléments du réseau, ils pourront récupérer les données dont ils ont besoin pour le mode de calcul chacun leur tour. (voir. figure 10 page 18)

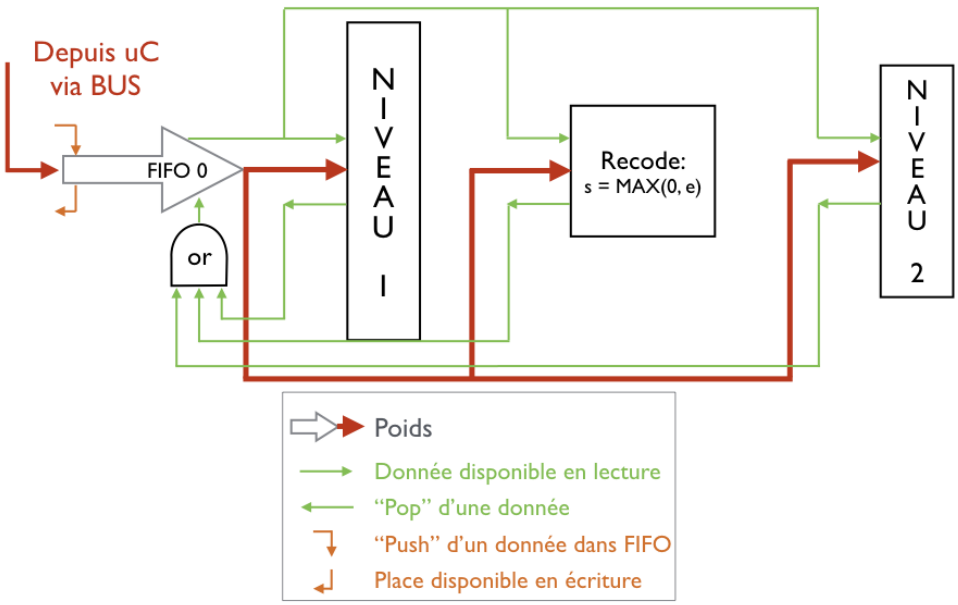

FIGURE 10 – Pipeline du réseau en mode chargement des poids pour illustrer l'utilisation de la FIFO 0

Le pipeline en mode calcul Lorsque le réseau de neurones est en mode de calcul, les images arrivent pixel par pixel dans la première FIFO. Chaque neurone produit 1 donnée qu'il envoie une à une dans la FIFO n°2, le recodage traite alors ces données une à une et enfin l'étage 2 reçoit ces résultats en lisant dans la FIFO n°3. La dernière FIFO permet de stocker les données de sortie du réseau afin de les envoyer (par paquets de 16) au micro-processeur. (voir figure 11 page 19).

3.3 Logiciel de commande

Le logiciel de commande est le logiciel embarqué sur la carte, qui s'exécute sur processeur ARM. Celui ci permet de configurer notre réseau de neurones avec les poids et les constantes choisies, de lancer les calculs avec les valeurs voulues et de récupérer les résultats. Il s'écrit en C et s'interface avec l'utilisateur grâce à la liaison série (composant UART). Le squelette de base était déjà disponible avec toutes les fonctions nécessaires, le code était écrit pour la configuration d'un niveau de neurone, l'envoi d'images simples et la récupération des résultats. Les données concernant les poids, les constantes, les images et les paramètres (taille d'une image, nombre de neurones à chaque niveau, nombre d'images) étaient toutes notées dans un fichier (`dataset.h`). Nous avons gardé ce fonctionnement pour la majeure

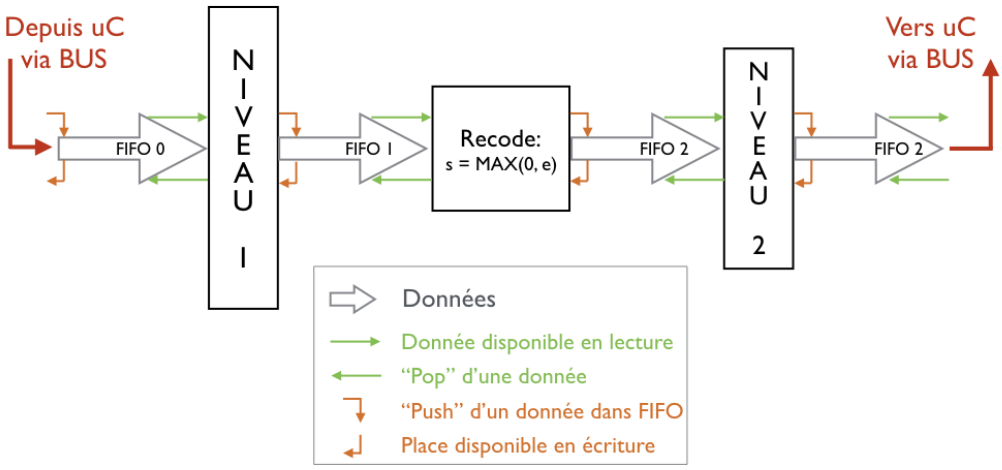

FIGURE 11 – Pipeline du réseau en mode de calcul pour illustrer l'utilisation des FIFO dans le réseau

partie de notre projet car il nous a permis de tester facilement notre IP pour des images de petites tailles, des niveaux de neurones avec très peu de neurones (par exemple 4 dans le premier niveau et 3 dans le deuxième) et de configurer facilement les pixels des images et les poids choisis pour connaître les résultats attendus et comparer avec ceux de notre IP. Nous avons cependant complété ce code afin qu'il puisse gérer notre IP complète avec deux niveaux de neurones et un étage de recodage. Par la suite, nous avons repris les mêmes données MNIST [3] que pour le logiciel de référence afin de tester l'application réelle.

3.3.1 Moyens de communication avec l'IP

Pour pouvoir échanger des données entre le logiciel embarqué et notre IP, 16 registres de 32 bits peuvent être lus et écrits des deux côtés de notre système. Par exemple les 4 premiers bits du registre 3 contiennent la configuration actuelle de notre IP : mode envoi des images, mode configuration (c'est-à-dire chargement des poids) du niveau 1, 2 ou de l'étage de recodage. Cette configuration est transmise aux blocs concernés afin qu'ils changent leur comportement et attendent les données utiles en entrée. Un autre bit de ce registre permet d'envoyer le signal `RESET` à toute notre IP. Les registres sont accédés avec un offset à partir d'une adresse de base sur le bus AXI. Des fonctions utilitaires déjà présentes permettent de faciliter les accès grâce à des masques et des opérations binaires avec les bons registres.
Pour envoyer ou recevoir de plus grosses quantités de données comme l'envoi des poids ou des frames, ou récolter les résultats de notre réseau de neurones, on utilise la fonction burst de notre bus AXI. On peut alors transférer des données de façon rapide entre la DDR où le logiciel alloue de l'espace pour stocker poids, frames et résultats et les FIFOs d'entrée et de sortie pour communiquer avec notre réseau de neurones.
A chaque fois que l'on souhaite faire un transfert, les mêmes opérations sont faites :

- si c'est une écriture, les données à écrire sont présentes, comme variables globales, dans des tableaux multi-dimensions dans un fichier C ;
- on alloue un tableau d'entiers (signé pour les poids, non signé pour les frames) de 32 bits de manière dynamique. La taille du tableau alloué dépend des données considérées mais on alloue 32 bits (1 case du tableau) par donnée (1 poids, 1 constante ou 1 pixel d'une frame) afin de faciliter leur traitement par l'IP, les FIFOs ayant des cases de même taille ;

— afin de préparer ce tableau aux bursts, qui sont des transferts par bloc de 16 mots de 32 bits, soit 16 cases du tableau vers une FIFO, on doit aligner son adresse sur des frontières de 16×4 octets. On ajoute alors 16×4 comme supplément à la taille nécessaire du tableau et on incrémente l'adresse renvoyée par `malloc` jusqu'à trouver une frontière de 16×4 octets, cad une adresse terminant par `0x00`, `0x40`, `0x80` ou `0xC0`. Grâce au supplément dans la taille ajouté, on est sûr d'avoir suffisament de place pour écrire les données même après avoir incrémenté l'adresse pour l'arrondir ;
— on écrit les données dans le tableau alloué à partir des variables globales déjà présentes ;
— on passe l'adresse de ce tableau à l'IP par un registre ;
— on indique dans un autre registre le nombre de bursts nécessaires pour transférer tout le tableau vers la FIFO. Si le tableau n'a pas un nombre de case qui est multiple de 16, on fait un burst supplémentaire pour inclure le reste. Il faudra alors envoyer à l'IP un signal `RESET` entre deux écritures afin de vider les FIFOs de leurs données résiduelles invalides.

3.3.2 Débogage

Pour déboguer le système global, notre IP avec le logiciel embarqué, nous avons utilisé différentes manières.
Pour des problèmes uniquement liés au logiciel (mauvaise taille, mauvais indices des tableaux, ...), nous avons utilisé le composant UART avec la fonction Xilinx `printf` pour envoyer par le lien série des informations sur un terminal distant. En passant par la suite à la carte ZedBoard, nous avons pu faire fonctionner la fonction Debug sur SDK Xilinx pour pouvoir avancer pas à pas dans le code et inspecter l'état des variables.
Les problèmes liés à notre IP sont plus difficiles à déboguer car nous ne pouvons pas simuler le système complet à cause de licences manquantes. Pour repérer certains problèmes comme des boucles d'attente infinies, des données de notre IP sont écrites dans des registres comme les signaux `READY` et `ACK` de chacunes des FIFOs et les `COUNT` pour ces FIFOs. Des fonctions utilitaires permettent de lire facilement ces informations à partir des registres concernés et de nous renseigner plus précisément sur l'étage bloquant de notre système.
Cependant ces méthodes peuvent nous renseigner uniquement sur l'étage qui fait bloquer tout le système et non pas sur la validité des données produites en sortie de chaque étage. Une fonctionnalité pour sortir des données de chacune des FIFOs intermédiaires a alors été implémentée du côté hardware, active à la lecture d'un certain registre, qui n'était jusque là pas utilisé. La gestion côté software est correctement effectuée pour attendre qu'il y des données dans les FIFOs avant de les sortir. Ce mécanisme nous a permis d'isoler les différents bugs de notre IP à un étage et de pouvoir les corriger individuellement.

3.4 Caractéristiques finales du système implanté

Pour étudier les caractéristiques finales de notre système, il est important de définir les caractéristiques théoriques atteignables.

3.4.1 Caractéristiques théoriques

On suppose qu'il n'y a pas de blocage de données en sortie de notre composant et que des données sont toujours disponibles en entrée.

Ainsi, il faut $2 \times FSIZE$ cycles (où $FSIZE$ est la taille d'une image) pour accumuler les données dans le premier niveau de neurones. Puis pour accumuler dans le deuxième niveau, il faut $2 \times N_1$ cycles (où N_1 est le nombre de neurones dans le premier niveau de neurones).

De plus, pour décaler les données du premier niveau jusqu'au deuxième niveau, il faut N_1 cycles pour décaler les données dans la FIFO, puis trois cycles pour que la première donnée traverse les FIFOs et l'étage de recodage, les données qui suivront n'auront pas de latence. Ainsi entre le premier niveau de neurones et le deuxième, il y a $N_1 + 3$ cycles. Mais le deuxième niveau de neurones va être bloquant (car $2 \times N_1 > N_1 + 3$ pour $N_1 > 2$). Donc

pour que les données soient décalées du premier niveau de neurones et accumulées par le deuxième, il faut $2 \times N_1$ cycles.

Après le deuxième niveau de neurones, il faut N_2 cycles (où N_2 est le nombre de neurones dans le deuxième niveau) pour décaler les données.

Ainsi le nombre de cycles requis pour traiter une image est :

$$N_{calcul} = N_1 \times 2 + FSIZE \times 2 + N_2 \tag{1}$$

Donc le temps de calcul d'une image est :

$$T_{calcul\_optimal} = \frac{N_{calcul}}{f_{réseau\_de\_neurones}} \tag{2}$$

Néanmoins, si nous avions optimisé la machine à états de contrôle des neurones pour qu'elle prenne les données en entrée en un cycle au lieu de 2, il est possible d'obtenir :

$$N_{calcul\_optimal} = N_1 + FSIZE + N_2 \tag{3}$$

Et

$$T_{calcul\_optimal} = \frac{N_{calcul}}{f_{réseau\_de\_neurones}} \tag{4}$$

Malheureusement, nous avons manqué de temps pour optimiser cette machine à états et valider que l'IP fonctionnait encore après optimisation.

3.4.2 Implantation sur carte ZedBoard

Avec l'outil *Xilinx Vivado*, nous avons implanté sur la carte ZedBoard notre IP. La ZedBoard dispose de 220 cellules DSP (utilisées pour l'accumulation dans les neurones).

Après implantation de l'IP pour 100 neurones dans le premier niveau et 10 dans le deuxième, à une fréquence de 115 MHz, nous obtenons, les caractéristiques suivantes :

Catégorie	Utilisées	Total	Taux d'utilisation
Cellules LUTs	4197	53200	7.89%
Mémoires LUTs	282	17400	1.62%
Mémoires BRAMs	55.5	140	39.64%
Cellules DSPs	110	220	50%

3.4.3 Évaluation des performances

Il s'agit maintenant de comparer les performances (en théorie et en pratique) aux performances du logiciel de référence fonctionnant sur le processeur embarqué.

Pour $FSIZE = 784$, $N_1 = 100$ et $N_2 = 10$, les performances sont disponibles dans le tableau 2 page 21. Dans ce cas-là, le facteur d'accélération vaut $t_{logiciel}/t_{IP(pratique)} = 457$.

Entité	Fréquence (MHz)	Taux de réussite pour 1000 images (MNIST)	Temps de calcul pour 1000 images (secondes)	Nombre d'images par secondes
Logiciel de référence	667	93,80%	8,67455	115,28
IP (Théorie)	115	93,80%	0,0154608	64679,4
IP (Pratique)	115	93,80%	0,0189669	52726,3

TABLE 2 – Performances pour $FSIZE = 784$, $N_1 = 100$ et $N_2 = 10$

4 Validation

La validation est une partie importante de notre projet. En effet, elle permet d'assurer le bon fonctionnement de notre IP.

Pour valider notre IP, nous avons du avoir recours à deux modes de validation : la validation en simulation et la validation sur carte. En effet, le simulateur ne permet pas de simuler notre IP complète avec le bus AXI et le processeur ARM, il faut donc avoir recours à une validation sur carte Zybo et ZedBoard.

4.1 Validation en simulation

La validation en simulation permet de vérifier le comportement de chacun des composants de notre IP, individuellement et en association avec d'autres composants.

Les simulations ont été réalisées à l'aide du simulateur de *Xilinx Vivado*.

4.1.1 Le neurone

Pour tester le neurone, il faut que le test bench simule l'environnement dans lequel le neurone doit évoluer. C'est-à-dire, les interactions avec les autres neurones, la machine à états et les FIFO de données en entrée et en sortie. Ainsi dans le test bench du neurone, nous avons vérifié que le neurone :

- puisse écrire dans sa BRAM lorsqu'il est en mode chargement des poids ;
- puisse lire les coefficients en BRAM quand il est en mode accumulation ;
- réagisse bien à la demande du passage de token ;
- effectue les bons calculs ;
- copie bien leur registre d'accumulation dans le registre miroir ;
- évacue bien la donnée du miroir par la chaine de miroir ;
- interagisse bien avec les signaux de contrôle des FIFO en lecture et en écriture.

4.1.2 La machine à états

La machine à états contrôle l'ensemble des neurones d'un niveau de neurones. Ce test bench permet de vérifier le comportement précis de la machine à états du niveau de neurones.

Durant ce test bench, nous avons vérifié que la machine à états :

- soit dans l'état attendu tout au long du test bench ;
- contrôle correctement un neurone, à partir du fonctionnement détaillé dans les spécifications ;
- passe en mode chargement des poids ou accumulation et ait le comportement attendu dans chacun des cas ;
- effectue un décalage de la chaîne de miroir à la fin d'une accumulation.

4.1.3 Le niveau de neurones

Le niveau de neurones est composé d'une machine à états et de neurones. Ce test bench permet de s'assurer que la machine à états et les neurones interagissent correctement ensemble et que l'on tient bien compte des buffers de distribution des signaux.

Durant ce test bench, nous avons vérifié que la machine à états :

- contrôle correctement plusieurs neurones, en tenant compte des buffers de distribution des signaux qui ajoutent un délai dans la communication ;
- passe en mode chargement des poids ou accumulation et ait le comportement attendu dans chacun des cas ;
- effectue un décalage de la chaîne de miroir à la fin d'une accumulation.

Nous avons vérifié que les neurones :

— chargent correctement les poids en mode chargement des poids.
— en mode accumulation, accumulent correctement les données et décalent le résultat dans la chaîne de miroir à la fin de l'accumulation.
— se passent correctement les données dans leurs miroirs lors de la phase de décalage des miroirs, en mode accumulation.

4.1.4 Le recodage

L'étage de recodage permet d'ajouter une constante aux valeurs en sortie du niveau de neurones et de supprimer les valeurs négatives.

Durant ce test bench, nous avons vérifié que l'étage de recodage :
— charge ses poids en mode chargement des poids ;
— en mode accumulation, ajoute une constante propre à chaque neurone à la donnée d'entrée, vérifie que le résultat soit positif, le transmet à la FIFO suivante dans ce cas, ou transmet 0 dans le cas contraire.

4.1.5 Interaction FIFO-niveau de neurones

Ce test bench permet de s'assurer que la FIFO d'entrée d'un niveau de neurones soit bien contrôlée par le niveau de neurones correspondant.

Durant ce test bench, nous avons vérifié que le niveau de neurones :
— contrôle correctement la FIFO d'entrée, c'est-à-dire qu'il prend les données quand elles sont disponibles, et ce pour les deux modes (accumulation et chargement des poids) ;
— contrôle correctement la FIFO de sortie, c'est-à-dire qu'il envoie des données à l'intérieur uniquement s'il y a suffisamment de place dans le mode accumulation. Dans le mode chargement des poids, il ne doit pas y avoir d'envoi de données dans cette FIFO.

4.1.6 Interaction FIFO-recodage-FIFO

Ce test bench permet de vérifier qu'un étage de recodage contrôle bien ses FIFOs d'entrée et de sortie.

Durant ce test bench, nous avons vérifié que l'étage de recodage :
— contrôle correctement la FIFO d'entrée, c'est-à-dire qu'il prend les données quand elles sont disponibles, et ce pour les deux modes (accumulation et chargement des poids). Dans le mode accumulation, il ne doit prendre des données que s'il y a de la place dans la FIFO de sortie ;
— contrôle correctement la FIFO de sortie, c'est-à-dire qu'il envoie des données à l'intérieur uniquement s'il y a suffisamment de place dans le mode accumulation. Dans le mode chargement des poids, il ne doit pas y avoir d'envoi de données dans cette FIFO.

4.1.7 Interaction FIFO-niveau de neurones-FIFO-recodage

Ce test bench vérifie que le pipeline FIFO-niveau de neurones-FIFO-recodage fonctionne correctement.

Durant ce test bench, nous avons vérifié que l'ensemble des composants :
— charge ses poids en mode chargement des poids, chacun leur tour. De plus, ils ne doivent pas se comporter de façon incorrecte quand ils ne sont ni en train de charger les poids ni en train d'accumuler ;

— effectue l'opération d'accumulation en mode accumulation, prenne les données dans sa FIFO d'entrée et transmet les données en sortie uniquement s'il y a de la place.

4.1.8 Conclusion

Avec l'ensemble des tests en simulation, nous avons couvert tout le pipeline de notre IP. En effet, chaque composant a été testé individuellement et nous avons testé les interactions entre FIFO-niveau de neurones-FIFO, FIFO-recode-FIFO. Nous avons donc vérifié que le pipeline complet FIFO-niveau de neurones-FIFO-recodage-FIFO-niveau de neurones-FIFO fonctionne correctement.

Ces tests nous permettent de couvrir l'ensemble de interactions entre composants de notre IP.

4.2 Validation sur carte

Suite aux tests réalisés en simulation et à l'impossibilité de simuler l'ensemble de la carte, nous avons dû réalisé des tests sur cartes (Zybo puis ZedBoard).

Ces tests nous permettent de tester le fonctionnement conjoint du logiciel, du bus AXI et de notre IP. Cependant, ces tests sont long (il faut générer le bitstream à charger dans la carte à chaque fois que l'on veut modifier le test ou en effectuer un autre). De plus, il est impossible d'observer l'ensemble du fonctionnement de notre IP une fois chargée dans le FPGA. Nous avons dû avoir recours à des registres non utilisés pour récupérer des données et contrôler plus finement notre IP. Le contrôle et le débogage du logiciel se fait via `GDB`.

4.2.1 Tests des données envoyées

Pour vérifier que notre logiciel envoie des données correctes sur le bus à notre IP, via le DMA, il faut regarder les données en entrée de notre IP. Pour cela, nous avons légèrement modifié le code VHDL afin de bloquer les données dans la première FIFO tant qu'il n'y a pas de lecture dans un certain registre. Ainsi la lecture dans ce registre provoque la sortie de la première donnée de la FIFO. En sortant les données une à une, nous pouvons vérifier que le logiciel envoie les bonnes données.

4.2.2 Tests des données en sortie de premier niveau de neurones

Pour vérifier que le premier niveau de neurones est correctement configuré et renvoie les données attendues en sortie, nous avons recours au même procédé que dans la partie précédente, mais en sortie de la FIFO entre le premier niveau de neurones et l'étage de recodage. Ainsi, nous pouvons observer les données en sortie du premier niveau de neurones. En envoyant des images complètement vides(pixels à 0) sauf pour un pixel à 1, nous pouvons récupérer l'ensemble de la configuration du premier niveau et vérifier qu'elle correspond à ce que nous attendons.

4.2.3 Tests des données en sortie de l'étage de recodage

Pour vérifier que l'étage de recodage est correctement configuré et renvoie les données attendues en sortie, nous avons recours au même procédé que dans les tests sur carte précédents. Nous récupérons les données en sortie de l'étage de recodage via la FIFO qui est connectée à sa sortie. De plus, en modifiant le VHDL, il est possible de charger des données dans la FIFO d'entrée de l'étage de recodage de façon manuelle, en écrivant des données dans un certain registre. Grâce à cela, nous pouvons vérifier que l'étage est correctement configuré et fonctionne comme attendu.

4.2.4 Tests des données en sortie de deuxième niveau de neurones

Pour vérifier que le dernier étage de neurones est correctement configuré et renvoie les données attendues en sortie, on procède de la même manière que pour l'étage de recodage. On vérifie les données en sortie du niveau via un registre, et au besoin on contrôle son entrée via un autre registre. Grâce à cela, nous pouvons vérifier que l'étage est correctement configuré et fonctionne comme attendu.

4.2.5 Conclusion

Grâce aux tests réalisés sur carte, nous pouvons terminer de valider notre IP. En effet, après avoir vérifié le comportement de chaque composant en simulation, nous pouvons voir comment cela se passe sur carte avec le logiciel. Certains problèmes logiciels comme matériels sont apparus lors de cette phase, et ont été assez difficiles à analyser correctement à cause du manque d'information par rapport à une simulation classique sur ordinateur.

5 Manuel utilisateur

Le réseau de neurones est un composant qui communique avec le logiciel via le bus AXI.

5.1 Registres

Le composant réseau de neurones est composé de 16 registres de 32 bits, dont l'utilisation est détaillé dans le tableau 3 page 27.

Numéro	Read/Write	Taille	Utilisation (bits)
0	Read/Write partiel	32 bits	— 15-0 : Nombre de données par image — 31-16 : Nombre maximum de données par image (Read only)
1	Read/Write partiel	32 bits	— 15-0 : Nombre de neurones dans le premier étage — 31-16 : Nombre maximum de neurones dans le premier étage (Read only)
2	Read/Write partiel	32 bits	— 15-0 : Nombre de neurones dans le second étage — 31-16 : Nombre maximum de neurones dans le second étage (Read only)
3	Read/Write partiel	32 bits	Configuration du mode de l'IP : — 3-0 : — `0000` : inactif — `0001` : Calcul du réseau de neurones — `0010` : Chargement des poids pour le premier niveau de neurones — `0100` : Chargement des poids pour l'étage de recodage — `1000` : Chargement des poids pour le second niveau de neurones — 08 : Reset — 09 : État occupé du maître AXI
4	Inutilisé	32 bits	Inutilisé
5	Inutilisé	32 bits	Inutilisé
6	Read/Write	32 bits	— 31-00 : Nombre de valeurs de sorties du composant à écrire dans la mémoire
7	Inutilisé	32 bits	Inutilisé
8	Inutilisé	32 bits	Inutilisé
9	Inutilisé	32 bits	Inutilisé
10	Read/Write	32 bits	— 31-00 : Adresse de la mémoire où lire les poids ou données

11	Read/Write	32 bits	— 31-00 : Adresse de la mémoire où écrire les données de sortie
12	Read/Write	32 bits	— 31-00 : Nombre de bursts pour la lecture dans la mémoire. La lecture commence quand ce registre est écrit.
13	Read/Write	32 bits	— 31-00 : Nombre de bursts pour l'écriture dans la mémoire. L'écriture commence quand ce registre est écrit.
14	Read Only	32 bits	— 07-00 : Nombre de données dans la FIFO entre le premier niveau de neurones et l'étage de recodage. — 15-08 : Nombre de données dans la FIFO entre l'étage de recodage et le second niveau de neurones. — 23-16 : Nombre de données dans la FIFO après le second niveau de neurones.
15	Read Only	32 bits	— 31-16 : FIFO Ready/Ack signals, in et out : 12 signals

TABLE 3 : Registres du composant FPGA réseau de neurone

5.2 Implantation sur FPGA avec Vivado

5.2.1 Ajout de l'IP dans un projet

Pour utiliser notre IP, il faut l'ajouter dans un projet *Xilinx Vivado*.

Pour cela, il faut tout d'abord posséder l'IP dans un dossier. Pour notre projet, il s'agissait du dossier `ip_repo/` qui contenait le dossier `my_axi_full_master/` (qui contenait lui-même notre IP). Vous pouvez tout simplement copier ce dernier dossier dans votre propre dossier contenant vos IPs.

Ensuite, il faut ouvrir le projet et lancer "*Tools -> create and package IP*". Enfin, sélectionner "*package a specific directory*" puis choisir "`<dossier_contenant_les_IPs>/my_axi_full_master`". Vérifier que le périphérique ajouté est bien un "*AXI peripheral*" avant de confirmer.

Après cela, il faut rajouter des blocs de façons à obtenir un *bloc design* ressemblant à la figure 12 page 28 tout du moins en ce qui concerne le bloc *myaxifullmaster_0* (IP réseau de neurones).

Pour modifier les paramètres et générer le bitstream, il faut se référer à la partie 5.2.2 page 28.

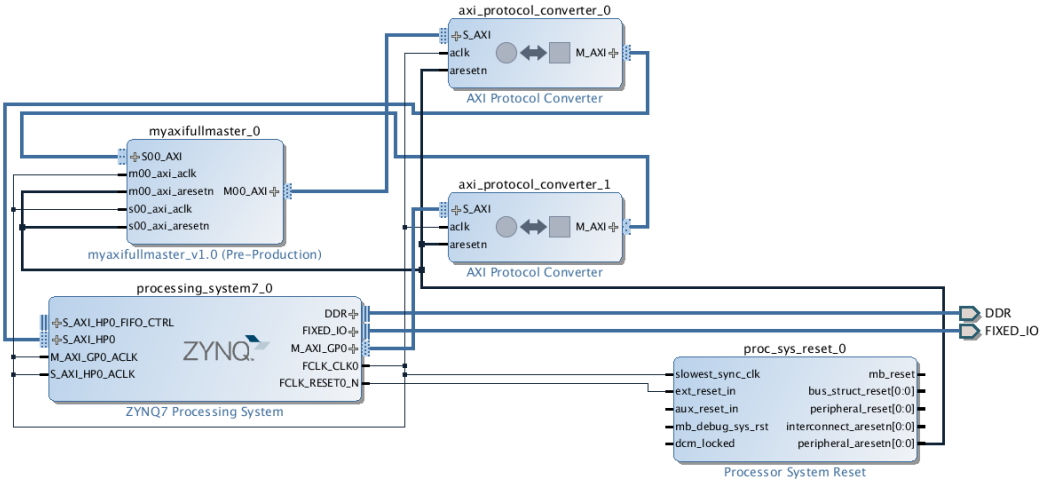

FIGURE 12 – *Bloc design* d'un projet utilisant l'IP réseau de neurones

5.2.2 Modification des paramètres du réseau de neurones

Pour modifier les paramètres du réseau de neurones (nombre de neurones de chaque niveau et taille des images), il faut préciser à *Vivado* que les fichiers sources de l'IP ont changés. Pour cela, il faut :

1. modifier dans le fichier `myaxifullmaster_v1_0_S00_AXI.vhd` (situé dans le sous-dossier `hdl/` de `<dossier_contenant_les_IPs>/my_axi_full_master`), les paramètres :
   — `LAYER1_FSIZE` pour changer la taille des images en entrée,
   — `LAYER1_NBNEU` pour changer le nombre de neurones dans le premier niveau,
   — `LAYER2_NBNEU` pour changer le nombre de neurones dans le deuxième niveau ;
2. sélectionner l'IP dans la partie "*Project Manager*" et avec un clic-droit sélectionner l'option "*Re-package IP*". Dans la partie "*Review and package*", cliquer sur "*Re-package IP*" afin de confirmer les modifications des fichiers sources de l'IP et retourner sur la fenêtre principale de *Vivado*.

Une fois cela fait, il faut générer le nouveau bitstream en cliquant sur "*Generate Bitstream*". Après génération, il faut faire "*File-> export Hardware (cocher include bitstream) -> Ok*". Enfin pour lancer *Xilinx SDK* afin de programmer le FPGA et lancer le logiciel embarqué, il faut faire "*File-> Launch SDK*".

5.3 Utilisation du réseau de neurones

Une fois que l'IP du réseau de neurones a été intégrée à *Vivado*, que le bitstream a été généré, exporté et que le SDK est lancé, l'utilisateur peut utiliser des fonctions simples incluses dans le projet pour lancer le réseau de neurones. Il peut soit utiliser les données MNIST pour les poids et frames qui sont déjà dans le projet (fichiers `net.c` et `frame.c`) soit utiliser ses propres données à rentrer dans le fichier `dataset.h`.
Une fonction permet de lancer le processus de calcul sur l'IP et de récupérer les résultats, pour chaque frame envoyée, des neurones de sortie. Une fonction identique permet de faire cette opération seulement en logiciel afin de, plus tard, comparer performances logicielles et matérielles. D'après le prototype

```
void nn_hardware(uint8_t *frames, uint32_t *results,
                 int16_t *weights_level1,
```

```
int16_t *weights_level2,
int16_t *constants_recode_level1,
int16_t *constants_recode_level2);
```

on envoie à la fonction l'adresse de la première case des tableaux qui contiennent les images, les poids, les constantes et du tableau qui contient les résultats du calcul. La fonction

```
void classify(uint32_t *results, uint32_t *classification);
```

classe ces résultats et place dans `classification` pour chaque image le neurone séléctionné (pour MNIST c'est le chiffre prédit).
Une fois que le code est écrit, l'utilisateur peut le compiler sous le SDK et lancer tout le programme sur la carte (`Run`) ou l'exécuter pas à pas (`Debug`). Les fonctions utilisées affichent du texte grâce au lien série avec le composant UART.

6 Planning et répartition des tâches

Pour travailler de façon efficace, il est essentiel d'établir les dépendances entre les tâches et de les répartir entre les membres du groupe.

6.1 Planning

Sur la figure 13 page 30, les tâches ont été réparties temporellement sur la période du projet. Le planning réel est quelque peu différent du planning prévisionnel. Il est visible sur la figure 14 page 30.

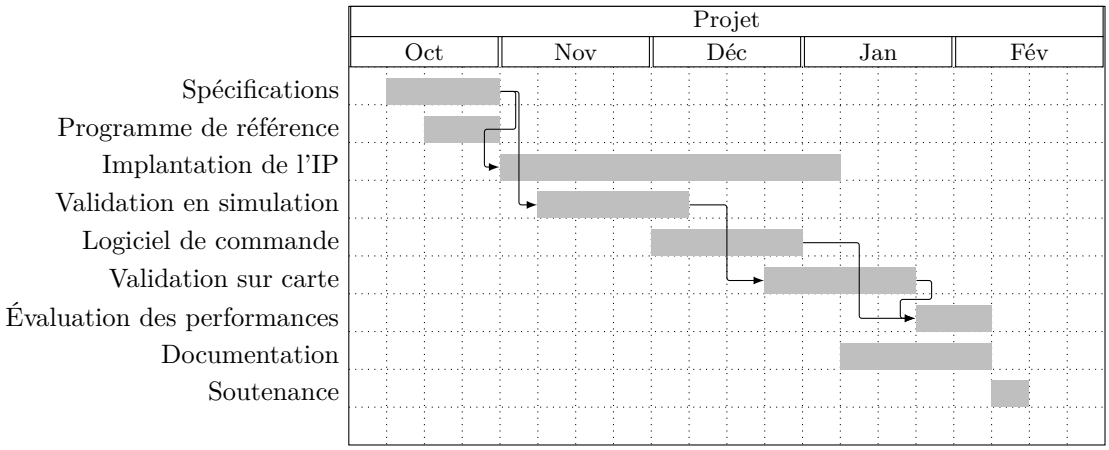

FIGURE 13 – Planning prévisionnel

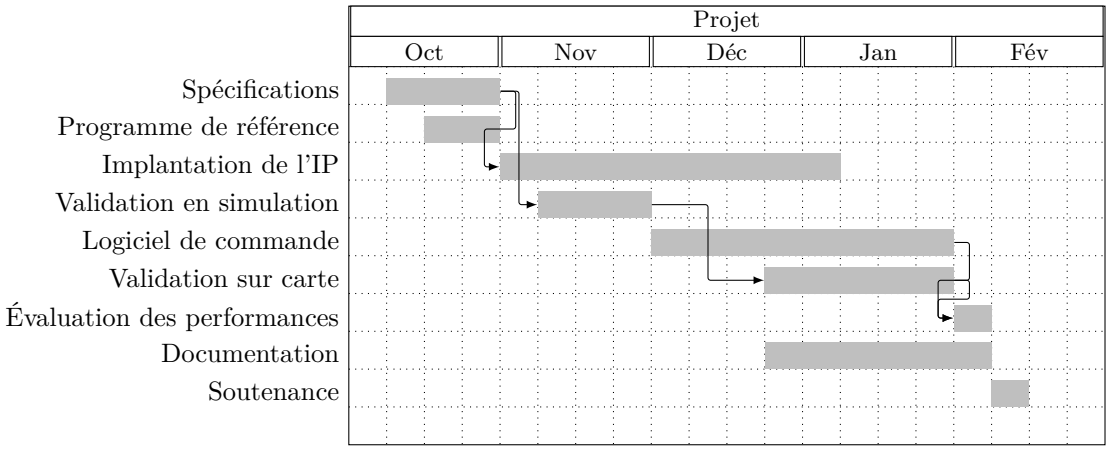

FIGURE 14 – Planning réel

Voici la description des différentes tâches :

— **Spécifications** : écrire les spécifications du réseau de neurones sur FPGA, définir son architecture ;
— **Programme de référence** : écriture du programme en C permettant de comparer les résultats de notre IP à une version logicielle. Il servira aussi lors de l'évaluation de performances pour mesurer le facteur d'accélération de notre IP ;
— **Implantation de l'IP** : écriture en VHDL des différents composants de l'IP ;
— **Validation en simulation** : Test benches pour vérifier le comportement de nos composants, leurs interactions et le fonctionnement global de l'IP ;

— **Validation sur carte** : Tests sur carte permettant de vérifier le comportement de notre IP sur carte Zybo, puis ZedBoard ;
— **Logiciel de commande** : écriture du logiciel permettant de contrôler l'IP ;
— **Evaluation des performances** : comparaison de performances de notre IP par rapport au logiciel de référence fonctionnant sur la même carte ;
— **Documentation** : écriture de la documentation comprenant le cahier des charges, le planning, le manuel utilisateur, les étapes de conception, la validation et la présentation de la démonstration ;
— **Soutenance** : Soutenance finale incluant une démonstration du travail effectué.

6.2 Répartition entre les membres du groupe

La plupart des tâches étant parallélisables, nous avons donc pu les répartir entre les membres du groupe.

Voici les occupations principales de chaque personne :
— Hugues : écriture du programme de référence et du logiciel de commande, implantation de l'IP, documentation ;
— Lucas : spécifications, implantation de l'IP, validation sur carte, documentation ;
— Paul : spécifications, implantation de l'IP, validation en simulation, documentation.

Cependant, certains problèmes majeurs dans l'implantation de l'IP et les essais sur carte sont devenus bloquants et ont fortement réduit la parallélisation des tâches en obligeant tous les membres du groupe à chercher une solution aux problèmes rencontrés.

6.3 Problèmes rencontrés

Parmi les quelques problèmes rencontrés, l'utilisation de *Vivado* nous a fait perdre beaucoup de temps. En effet, bien que ce logiciel de conception soit puissant et complet, il a certains comportements assez étranges. Nous citerons par exemple, la démarche à mettre en oeuvre pour que le logiciel prenne effectivement en compte les modifications des fichiers sources (près de 10 clics et environ 2 minutes). Un autre exemple, qui nous a fait perdre près d'une demi-journée, est le fait que le logiciel décide (sans prévenir) qu'il génère le bitstream dans un nouveau fichier sans prévenir le SDK qui, lui, envoie toujours l'ancien bitstream. Un dernier cas nous a posé problème, ainsi qu'à nos tuteurs, et le passage du projet d'un carte à l'autre. Pour réussir à faire cela, nous avons du recréer le projet en partant de zéro.

Outre ces nombreux problèmes dûs aux outils de conception, nous nous sommes confrontés à un autre problème inattendu. En effet, nous avions prévu de passer beaucoup moins de temps sur la partie de validation sur carte et plus de temps sur la validation en simulation. Nous pensions que si l'IP fonctionnait en simulation, elle aurait le même comportement après synthèse et sur le FPGA. Or, à plusieurs endroits de notre IP, l'outil de synthèse instanciait des latchs à la place de registres. A cause de cela, le fonctionnement n'était pas celui attendu, et nous avons dû repasser du temps pour déboguer l'IP.

7 Présentation de la démonstration

Pour démontrer que notre réseau de neurone est capable de trouver le nombre écrit à la main dans une image, nous allons vous faire une démonstration de ses capacités ainsi qu'une démonstration de ses performances. La démonstration se décomposera donc en deux étapes.

7.1 Démonstration du fonctionnement

Création de l'image à analyser

L'image à analyser par notre réseau de neurones doit être créée. Ainsi, nous préparerons une image de 28×28 pixels dans laquelle nous dessinerons un chiffre de 0 à 9. Cette image de 784 pixels sera alors transformée en un tableau de 784 valeurs. Ce tableau, une fois exporté dans notre logiciel de commande de l'IP, pourra être envoyé comme données d'entrée à traiter par le réseau de neurones.

Analyse de l'image et prédiction du chiffre par le réseau de neurone

Le réseau de neurones va donc être utilisé exactement comme avec les données MNIST [3], mise à part que l'image source aura été produite par nos soins. À la sortie du réseau nous trouverons alors un vecteur de taille 10. La coordonnée du vecteur dont la valeur est la plus grande sera la valeur prédite par le réseau de neurone. Par exemple, si en sortie du réseau de neurone nous avons $\{991342, 13124, -104, 54, 567, -345789, 1086, 2, 0, -10\}$ alors le réseau de neurone prédit que le chiffre manuscrit présent sur l'image en entrée est un 0.

Affichage du résultat

Pour finir la démonstration, le programme affichera de façon plus conviviale la valeur que le réseau de neurone a prédite et l'image du chiffre dessinée donnée en entrée. De plus, nous afficherons le taux de réussite sur toutes les images essayées.

7.2 Démonstration des performances

Pour la démonstration des performances, nous tenterons de traiter 1000 images et de faire la comparaison en temps et en taux de réussite entre notre IP et le programme de référence purement software. A l'issue de ce test, nous afficherons les différentes grandeurs permettant la comparaison.

8 Conclusion

À la fin de ce projet, nous disposons d'une IP réseau de neurones très performante (environ 450 fois plus rapide qu'un logiciel implémentant le même réseau de neurones). Celle-ci est générique : il suffit de modifier quelques constantes dans un fichier VHDL et dans les fichiers sources du logiciel pour changer les paramètres du réseau de neurones (nombre de neurones de chaque niveau, taille des images d'entrée et nombre d'images). De même, il est configurable dynamiquement, sans avoir à générer à nouveau le bitstream : il est possible via des appels de fonctions de changer la configuration de chaque niveau de neurones et de l'étage de recodage.

Cependant, il est possible d'obtenir des performances encore meilleures (un facteur d'amélioration environ égal à 2) en optimisant la machine à états de contrôle de neurones. Malheureusement, nous n'avons pas eu le temps d'effectuer cette dernière et de la valider en simulation et sur carte dans le temps imparti. En effet, nous avons rencontré des problèmes innatendus qui nous ont fortement freiné dans notre progression. Néanmoins, nous avons réussi à remplir les objectifs du projet à temps en proposant une IP réseau de neurones générique, correcte, configurable dynamiquement et beaucoup plus performante que le logiciel.

L'IP développée durant ce projet pourra permettre au laboratoire TIMA de comparer les performances de leur réseau de neurones ternaires à un réseau de neurones plus classique sur un même matériel. De même, il serait intéressant de comparer les performances de notre réseau à des alternatives telles que *SpiNNaker* [1] de l'université de Manchester, ou bien *TrueNorth* [2] d'IBM.

Références

[1] Eustace Painkras, Luis A Plana, Jim Garside, Steve Temple, Francesco Galluppi, Cameron Patterson, David R Lester, Andrew D Brown, and Steve B Furber. Spinnaker : A 1-w 18-core system-on-chip for massively-parallel neural network simulation. *IEEE Journal of Solid-State Circuits*, 48(8) :1943–1953, 2013.

[2] Dharmendra S Modha. Introducing a brain-inspired computer. *IBM Research, accessed at www. research. ibm. com/articles/brain-chip. shtml*, 2014.

[3] Yann LeCun, Corinna Cortes, and Christopher JC Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available : http ://yann. lecun. com/exdb/mnist*, 2, 2010.

List of Symbols

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface : bus de donnée

BRAM Block Random Access Memory : mémoire spécifiques Xilinx pour les cellules FPGA

CIME Centre Interuniversitaire de MicroElectronique et nanotechnologies

CPU Central Processing Unit

DDR Double Data Rate : type de mémoire

DMA Direct Memory Access : composant permettant des transferts de mémoire direct sans passer par le CPU

DSP Digital Signal Processing : Composant électronique disponible dans le FPGA permettant de faire des multiplications et additions de façon optimisées

FIFO First In First Out : structure de données permettant de stocker des données et de les restituer dans l'ordre d'arrivée

FPGA Field-Programmable Gate Array : circuits intégrés re-programmables

FSM Finite State Machine : machine à états finis

GPU Graphics Processing Unit

HDL Hardware Description Language : description du comportement matériel attendu par un langage de description tel que VHDL ou Verilog

IP Intellectual Property : bloc hardware réutilisable ayant une fonctionnalité spécifique

LUT Look Up Table : table de correspondances pour programmer le FPGA

MNIST Mixed National Institute of Standards and Technology : Base de données de caractères manuscrits servant de tests à de nombreux algorithmes de reconnaissance d'écriture

PC Personnal Computer : ordinateur personnel

SDK Software Development Kit : outil de développement logiciel

TIMA Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés

UART Universal Asynchronous Receiver Transmitter : composant permettant de gérer la communication par liaision série entre la carte et l'ordinateur