Rapport de projet : Concours RISC-V

Référent :
LEROUX Camille

Etudiants :
MOLINARI Caio
AUGERAT Alice
LETEMPLE Maxime
BOURDIN Bryan

Année universitaire 2023-2024
ENSEIRB-MATMECA : Filière Électronique SE 3A

Sommaire

1 Introduction

1.1 Contexte

L'objectif primaire de ce projet était de changer l'architecture du processeur RISC-V CVA6 de Thalès afin d'optimiser son architecture pour des applications d'intelligence artificielle. La première étape était la recherche bibliographique du sujet et l'étude du processeur de base fourni par Thalès, bien que la configuration de l'environnement de compilation et exécution du projet. L'équipe a consacré 4 séances à cette partie du travail, à la fin desquelles nous nous sommes confrontés avec certains problèmes d'ordre pratique, notamment le manque de la carte utilisé par le projet CVA6 (la Zybo Z7-20 de chez Digilent, qui contient un circuit FPGA Zynq 7000 de chez Xilinx) et surtout le fait que le dépôt GitHub de Thalès avec les codes sources n'avait pas été mis à jour jusqu'à ce moment. De ce fait, nous en avons discuté avec notre tuteur qui nous a proposé de changer le sujet de notre projet à cause des problématiques mentionnées ci-dessous.

L'idée était dorénavant de reprendre en main un projet déjà existant et développé en partie par notre tuteur : le processeur Plasma. Il s'agit d'un processeur avec des instructions RISC 32 bits-softcore qui tourne à une fréquence de 25 MHz. L'ajout de l'interface avec plusieurs PMODs a été réalisé par une équipe de l'option SE, il y a quelques années, permettant l'interaction avec du matériel en «haut-niveau »(en langage C). Le nouvel objectif était, d'abord, de prendre en main le processeur et son environnement de développement en étudiant l'architecture, les interfaces PMODs et les applications déjà conçues auparavant, ensuite développer une application d'intelligence artificielle en C pour classifier des images du dataset MNIST (image 1) et finalement, si le temps permet, de rajouter une instruction personnalisée ou changer l'architecture pour optimiser l'application fonctionnelle.

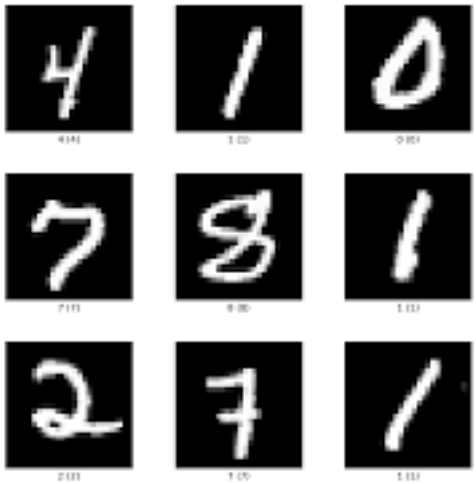

FIGURE 1 – Exemple de données du dataset MNIST

2 Projet Plasma

2.1 Introduction

Le CVA6 étant un processeur fournit par thalès et necessitant de la place, il est finalement decidé d'utiliser le plasama_pmod fournit par Camille Leroux c'est un processeur assez simple auquel on peut rajouter facilement des Pmod dont les modules de bases sont présentés comme dans l'image 2.

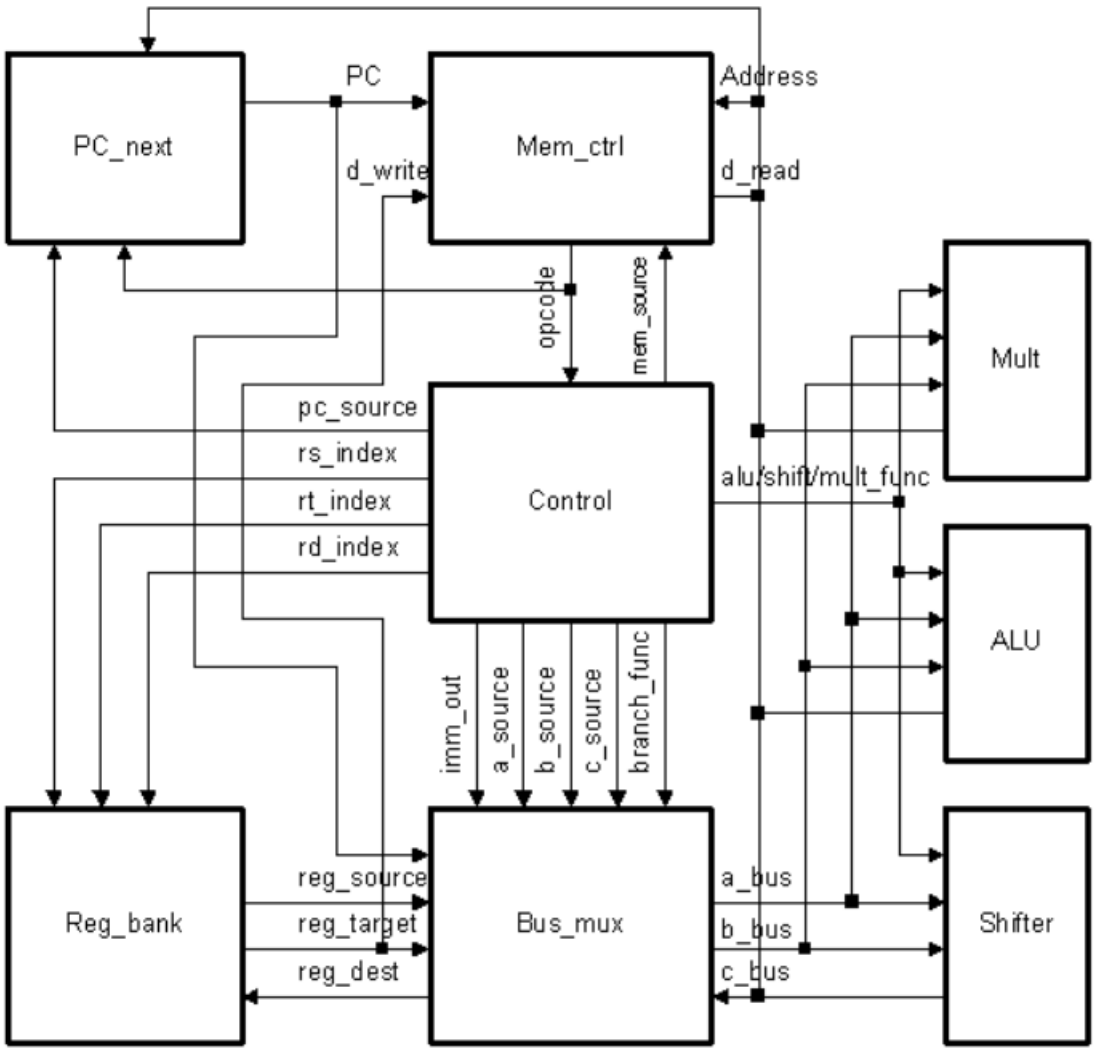

FIGURE 2 – Schéma du Plasma PMOD

2.2 Configuration de l'environnement de développement

Avant de démarrer le projet, la première étape est de configurer l'environnement de compilation et exécution. Le *build system* du projet Plasma est entièrement basé sur le système d'exploitation Linux, ce qui a mené certains membres de l'équipe qui sont utilisateurs Windows sur leur machine personnel à configurer la *Windows Subsystem for Linux* (WSL). Il aurait été possible de faire un *dual-boot* d'un système Linux complet ou d'utiliser une machine virtuelle pour effectuer le développement, cependant, la WSL est une solution élégante qui est maintenue est améliorée par Microsoft, de sorte que c'est une technologie intéressante à maîtriser, d'où le choix de l'équipe.

Tout d'abord, l'installation de la WSL est réalisé à travers commandes sur le terminal Windows, il faut faire attention à la version de la WSL qui doit être la deuxième, la première est toujours supportée par Microsoft, mais elle ne reçoit pas les dernières mises à jour telles que la deuxième. Une fois installé, il est intéressant de changer l'emplacement de stockage des fichiers de la WSL si on a deux dispositifs de stockage dans la même machine (un SSD pour le système d'exploitation et l'autre pour les fichiers). Cela est dû au fait que Windows installera la WSL dans le SSD, qui, en général, ne possède pas beaucoup de capacité mémoire sauf pour l'OS lui-même. –

Par la suite, il faut configurer et installer le logiciel Vivado, essentiel pour le développement du projet. Malheureusement, la WSL n'est pas encore supportée par Xilinx de façon «native », de sorte qu'il n'y a pas de procédure d'installation propre à elle. Pour résoudre ce problème, l'installation de l'outil comme si notre système était un Linux conventionnel a été suivi. Pour ce faire, le fichier .tar qui contient le logiciel a été installé et décompressé. Ensuite, le logiciel a été configuré de façon *headless*, c'est-à-dire, sans utiliser une GUI (*Graphical User Interface).* Une fois l'installation complétée, il a suffi de rajouter le chemin d'accès au binaire de Vivado à la variable globale PATH de la WSL.

La dernière configuration cruciale qui doit être réalisée est l'ajout de l'accès aux ports USB. Par défaut, cette utilité n'est pas disponible dans la WSL, il faut, donc, installer un logiciel développé par Microsoft qui permet de rattacher des ports USB de la machine à la WSL. Cette étape est importante lors du téléversement de programmes dans le processeur Plasma et lors de l'envoi et réception de données (qui peut être utilisé en utilisant minicom ou Putty, par exemple). L'installation de cet outil rend possible l'utilisation d'un terminal Windows et un terminal Linux simultanément, cf figure 3.

Les détails concernant cette partie tels que les commandes et des liens vers des tutoriels ont été compilés sous forme d'un fichier de documentation.

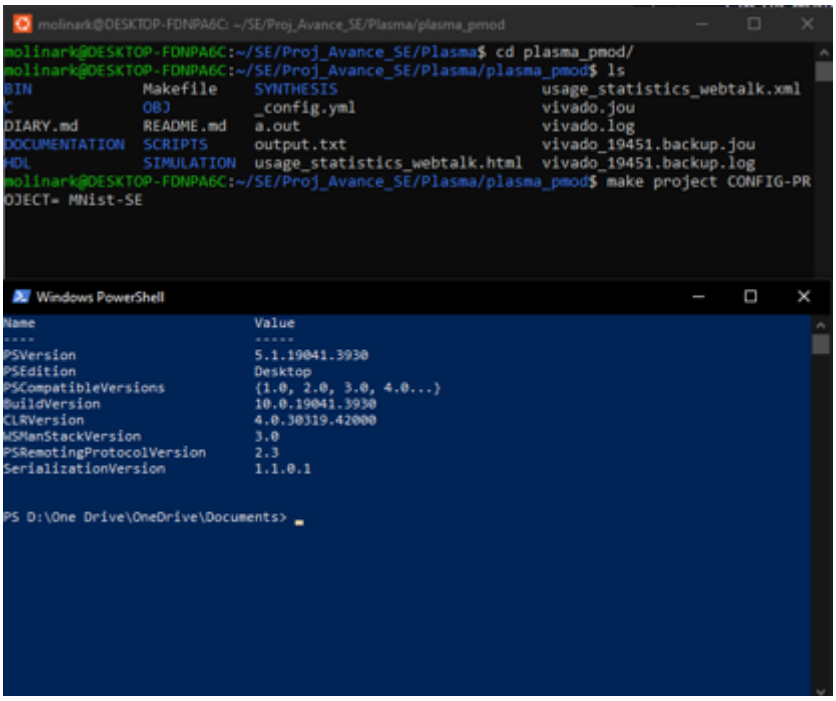

FIGURE 3 – Utilisation de la WSL 2

2.3 Création du modèle

Il s'agit de créer un modèle de réseau de neurones capable de reconnaître un chiffre sur une image (les images du dataset MNIST de la figure 1).

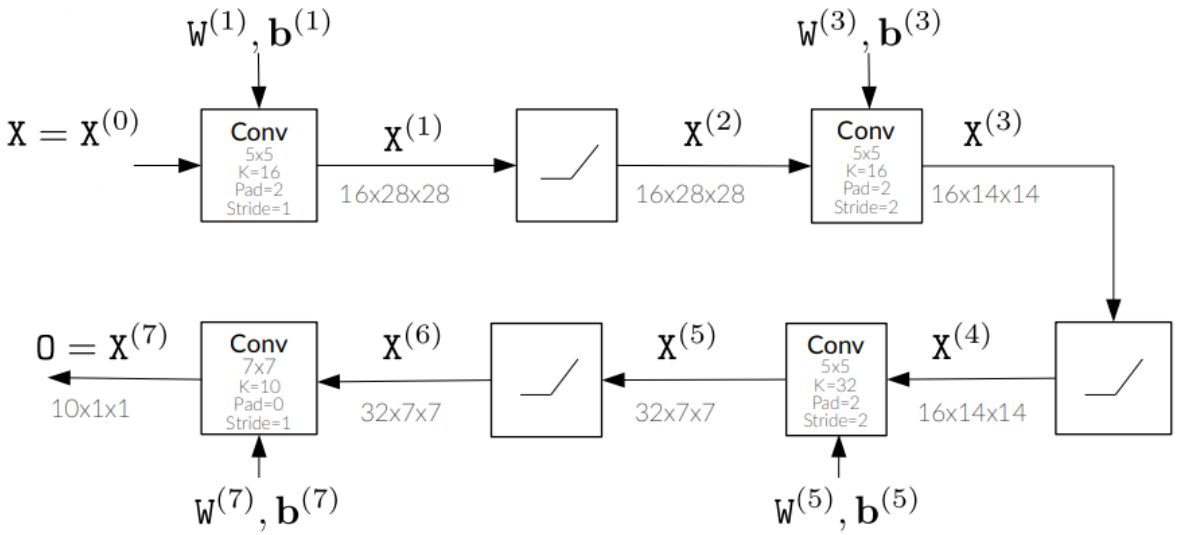

FIGURE 4 – Structure du réseau de neurones choisis, issu du cours de Guillaume Bourmaud

2.3.1 Première conversion du python au C

La première idée était de reprendre un modèle de réseau récupéré dans un cours de M.Bourmaud, professeur en IA, qui est écrit en python, et de le traduire en C en remplaçant les fonctions python par leur équivalent en C. Le modèle est constitué de 4 couches comprenant pour chacune une convolution puis une ReLU. La taille des convolutions varie, afin de permettre une bonne détection des formes de chaque chiffre (voir *figure 3*). La dernière couche agit comme une FC pour récupérer un tableau de 10 valeurs comprenant les scores pour chacun des chiffres possibles.
Le dernier calcul à faire serait de normaliser chacun des poids à l'aide de la fonction *softmax*, mais celle-ci requiert l'utilisation de fonctions mathématiques non implémentées, à savoir le logarithme népérien et l'exponentielle. Ce n'est pas spécialement important, car ce calcul se contente de conserver la relation d'ordre et ne sert qu'à afficher un score avec des pourcentages. Le programme ici se contente de trier les valeurs par ordre de leur score.

2.3.2 Adaptation d'un réseau à une utilisation spécifique

Finalement, l'option de récupérer un réseau de neurone déjà fonctionnel et de l'adapter aux critères choisis était la solution la plus optimale en temps et en productivité. Le code récupéré est celui de ce github qui est déjà fonctionnel avec MNIST et qui utilise des types double. Le code est modifié pour des types float et simplifié pour être adapté à l'implémentation sur la carte. Le réseau est entrainé sur le dataset d'entraînement qui fait 60000 données et qui donne un taux de réussite à 97%, dénotant un entrainement réussi.

Les poids d'entraînement sont sauvegardés dans un fichier *weights.bin*, puis la partie entraînement du code est supprimé puisqu'elle n'est plus utile pour l'utilisation souhaitée.

2.3.3 Quantification des poids du réseau

Avec les poids qui ont été chargés, nous faisons une analyse pour les adapter à l'optimisation du processeur.
En réduisant les poids donnés à 5 valeurs de poids possibles, nous perdons environ 5% de précision. Les valeurs de ces poids (-0.5, -0.25, 0, 0.25, 0.5) sont choisies de façon à faire un décalage de bits plutôt qu'une multiplication, ce qui fait gagner de la place et du temps.

2.4 Implémentation sous contraintes du modèle

Il s'agit d'adapter le modèle du réseau à une implémentation sur carte qui nécessite des nombres à virgules fixes.

2.4.1 Conversion en virgule fixe

Après avoir créé un code fonctionnel en C d'un réseau de neurones qui ne nécessite pas d'entraînement (les poids ont été calculés au préalable et doivent être envoyé sur la carte) il s'agit d'adapter ce modèle à des nombres à virgule fixe sur 32 bits. Après analyse de la précision,

12 bits suffisent pour la partie décimale pour satisfaire les critères du réseau. Les fonctions qui permettent la transition des virgules flottantes vers les virgules fixes sont créées et listées dans le fichier *fixed.h*. Celles-ci ne sont en réalité pas des fonctions, mais des macros (une liste de define) qui remplace dans le code les virgules flottantes (décalage de la virgule, opération simple et conversion float vers int) ce qui explique pourquoi un simple fichier.h est suffisant.

Il suffit ensuite de remplacer les type float ou les opérations par les *define* adéquate.

2.5 Envoi des données

Dans cette sous-partie, nous nous focalisons sur la transmission des données d'intérêt sur la carte. Ces données sont les images et les filtres contenant les poids de notre réseau de neurones.

Pour cette partie, nous avons décidé d'utiliser un code en Python, permettant d'utiliser aisément des bibliothèques et fonctions utiles associées.

2.5.1 Envoi des images

Afin de procéder à l'envoi des images, nous ajoutons tout d'abord des arguments à l'appel de notre programme Python afin de fournir le nombre d'images envoyées et la dimension de celles-ci :

```
parser = argparse.ArgumentParser()
parser.add_argument("-n", "--number", type=int)
parser.add_argument("-he", "--height", type=int)
parser.add_argument("-w", "--width", type=int)
args = parser.parse_args()

images_number = args.number
```

Ensuite, l'objectif est de récupérer aléatoirement dans le dataset MNIST les images. Pour ce faire, nous utilisons la bibliothèque **os** de Python :

```
list_dataset = os.listdir('mnist_images/train')
data_name = []
for data in (list_dataset) :
    data_name.append(data)

image_line = list()

image_line.append(images_number)

for i in range(images_number):
    data_chosen = data_name[randrange(len(data_name))-1]
    FILEPATH_TARGET = f"mnist_images/train/{data_chosen}"

    image = Image.open(FILEPATH_TARGET)
```

Nous appliquons dorénavant, pour chaque image, les traitements nécessaires de redimensionnement et de conversion au format approprié PGM :

```
image_resized = image.resize((IMG_WIDTH,IMG_HEIGHT),
resample=Image.BILINEAR)

OUTPUT_FILEPATH = "MNIST_data.pgm"

image_resized.save(OUTPUT_FILEPATH)


# Opening and treating image #
image = cv2.imread("MNIST_data.pgm")
# Remember, opencv by default reads images in BGR rather than RGB
# So we fix that by the following
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

image = SCALING_FACTOR*image

#https://www.statology.org/numpy-array-to-int/
image = (np.rint(image)).astype(int)
```

Il suffit dorénavant d'ordonner les pixels RGB dans le vecteur contenant les données pour l'envoi vers la carte :

```
# reordering RGB pixels
for i in range(IMG_HEIGHT):
    for j in range(IMG_WIDTH):
        image_line.append(image[i,j,0]) # R
        image_line.append(image[i,j,1]) # G
        image_line.append(image[i,j,2]) # B

# serializing data
image_line = bytearray(image_line)
```

L'envoi des données se fait via l'UART et l'appel **ser.write()** qui envoie les données octet par octet :

```
# sending out image #
ser = serial.Serial(port="/dev/ttyUSB1", baudrate=115200,
bytesize=serial.EIGHTBITS, parity=serial.PARITY_NONE,
stopbits=serial.STOPBITS_ONE)
ser.write(image_line)
```

2.5.2 Envoi des filtres et biais

Une fois les images transmises à la carte, il nous reste plus qu'à envoyer les poids de nos neurones, donc les filtres et biais.
Une fois l'entraînement du réseau effectué, les poids sont enregistrés dans un fichier binaire **.bin**.

Pour envoyer ces données via la commande **ser.write()**, nous devons définir que nous récupérons un octet après l'autre avec l'appel **f.read(1)**.
Étant donné que nous devons transmettre des entiers, nous effectuons la conversion en précisant que les premiers bits récupérés sont les bits de poids fort.

La récupération des entiers se fait donc via le code suivant :

```
weight_line = list()

# Open a binary file
with open("weights_fixed.bin", "rb") as f:

        fixed_t = f.read(1)
        fixed_t_int = int.from_bytes(fixed_t, "big")
        weight_line.append(fixed_t_int)
        while fixed_t:
            fixed_t = f.read(1)

            # converting to int
            # byteorder is big where MSB is at start
            fixed_t_int = int.from_bytes(fixed_t, "big")
            weight_line.append(fixed_t_int)
```

Il suffit par la suite d'envoyer ces données avec la commande **ser.write()**.

2.6 Récupération des données

Une fois les données transmises, il nous reste à procéder à la récupération sur carte.
Les codes suivants sont donc écrits en C.

2.6.1 Récupération des images

Nous allons tout d'abord procéder à la récupération des images MNIST sur la carte.
Pour ce faire, nous décidons d'initialiser les images comme étant des structures contenant un tableau de pixels à deux dimensions.

Afin de récupérer plusieurs images, nous créons un tableau de structure contenant le nombre d'images envoyées :

```
typedef struct Image {
int pixels[30][30];
}Image;

Image images[6];
```

NOTE : L'initialisation des paramètres, comme ici le tableau de structures, doit se faire globalement. La carte ne permet ni l'allocation dynamique, ni d'être assuré de retrouver les bonnes adresses mémoires lors d'utilisation de variables locales dans d'autres fonctions.

Par la suite, il suffit de fournir le nombre d'images arrivant dans l'UART et de stocker chaque pixel dans le tableau de l'image associé :

```
int images_number = wait_data();

MemoryWrite(SEVEN_SEGMENT_REG, 1);
for (int image = 0; image < images_number; image++)
{
    for (int py = 0; py < HEIGHT; py++)
    {
        for (int px = 0; px < WIDTH; px++) {
            r = wait_data();
            g = wait_data();
            b = wait_data();
            pixel = (r + g + b) / 3;

            images[image].pixels[py + 1][px + 1] = pixel;
        }
    }
}
```

NOTE : Les indices fournis lors de la récupération des pixels (à savoir, py + 1 et px + 1) permettent de prendre en compte le zéro padding qui nous sera utile lors des tests de produits de convolution dans la section dédiée (2.7).

2.6.2 Récupération des filtres

Enfin, il est nécessaire de récupérer les poids des neurones afin de les stocker dans les filtres. Pour ce faire, nous décidons d'initialiser les filtres en tant que tableaux à une dimension afin de simplifier la récupération.

Il est nécessaire ici de prendre en compte le fait que les valeurs contenues dans les filtres sont codées sur 24 bits, alors que l'appel **wait_data()** renvoie un entier sur 8 bits.
Étant donné que les octets arrivent dans l'ordonnancement poids forts puis poids faibles, il suffit de procéder à un décalage sur 23, 15 puis 7 bits à gauche pour les trois premiers octets pour enfin effectuer la somme des quatre données.
Nous avons ainsi la récupération suivante pour l'un des filtres :

```
for(int coeff_num=0; coeff_num < 16*1*5*5; coeff_num++){

    int coeff1 = wait_data();
    coeff1 = coeff1 << 23;
    int coeff2 = wait_data();
    coeff2 = coeff2 << 15;
    int coeff3 = wait_data();
    coeff3 = coeff3 << 7;
    int coeff4 = wait_data();
    coeff4 = coeff4 << 0;


    weights1[coeff_num] = coeff1 + coeff2 + coeff3 + coeff4;
}
```

2.7 Premiers tests sur l'application du produit de convolution

A ce stade, nous devions tester la possibilité d'appliquer des produits de convolution sur nos images envoyées sur carte.

Le test a été effectué avec un filtre de dimensions 3x3, qui est un filtre de détection de bordures vertical :

```
signed char filtre[3][3] = { {1, 0, -1}, {2, 0, -2}, {1, 0, -1} };
```

Nous appelons ensuite pour chaque image l'opération de convolution suivante :

```
void convolution(Image* data, Filtre* filtre) {

    for (int i = 1; i < HEIGHT-1; i++) {
        for (int j = 1; j < WIDTH-1; j++) {
            new_image[i][j] =
                data->pixels[i][j] * filtre->coeff[1][1] +
                data->pixels[i - 1][j - 1] * filtre->coeff[0][0] +
                data->pixels[i + 1][j + 1] * filtre->coeff[2][2] +
                data->pixels[i - 1][j + 1] * filtre->coeff[0][2] +
                data->pixels[i + 1][j - 1] * filtre->coeff[2][0] +
                data->pixels[i][j - 1] * filtre->coeff[1][0] +
                data->pixels[i][j + 1] * filtre->coeff[1][2];
        }
    }

    for (int i = 1; i < HEIGHT - 1; i++) {
        for (int j = 1; j < WIDTH - 1; j++) {
            data->pixels[i][j] = new_image[i][j];
        }
```

```
    }
}
```

Enfin, nous décidons, pour la visualisation, d'utiliser le bouton de la carte afin de passer à l'affichage de l'image suivante :

```
int i = 0;
int remaining_images = images_number;

while (i < images_number) {
if (MemoryRead(BUTTONS_CHANGE) == 1
                && MemoryRead(BUTTONS_VALUES)==1)
{
sleep(1000);
MemoryWrite(SEVEN_SEGMENT_RST, 1); // reset the 7 segment controler
MemoryWrite(SEVEN_SEGMENT_REG, remaining_images);
for (int py = 0; py < HEIGHT; py++)
{
for (int px = 0; px < WIDTH; px++) {
pixel = images[i].pixels[py][px];
pixel = ((pixel >> 3) << 11) | ((pixel >> 2) << 5) | (pixel >> 3);
printPixel(py, px, pixel);
}
}
i = i + 1;
remaining_images = remaining_images - 1;
}
}
MemoryWrite(SEVEN_SEGMENT_REG, remaining_images);
```

L'afficheur 7 segments nous permet d'afficher le nombre d'images qu'il nous reste à visualiser, et donc, à valider le chargement de toutes nos images.

Un résultat est fourni ci-dessous, montrant le stockage de 5 images dans la carte :

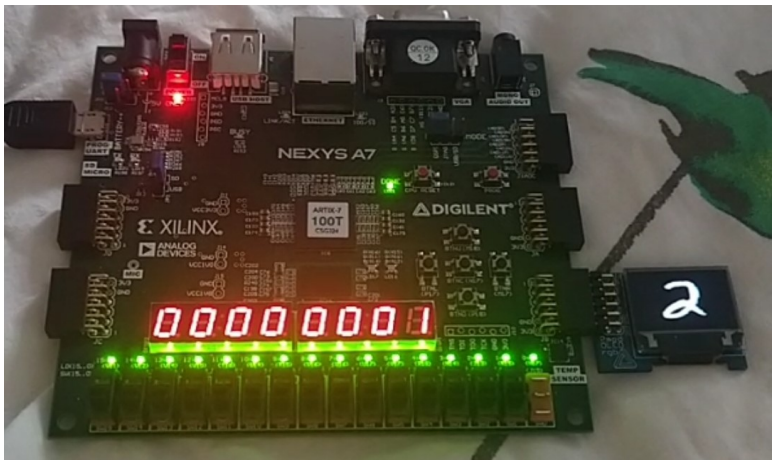

FIGURE 5 – Stockage de 5 images dans la carte

2.8 Implémentation du réseau sur la carte

Le programme du réseau de neurones fonctionne sur l'ordinateur hôte, mais il ne fonctionne pas encore sur la carte.
Cependant, la compilation fonctionne et, à priori, l'espace mémoire utilisé est suffisant. Il reste encore quelques étapes de débogage à effectuer.

2.9 Améliorations possibles

Le programme actuel comporte les poids modifiés, mais il reste à implémenter le décalage binaire à proprement parler.

Actuellement, le programme se contente toujours de réaliser des multiplications sur les nombres à virgule fixe (qui de son point de vue ne sont que des integers). Il reste donc à implémenter dans un premier temps une structure conditionnelle dans le code, à la place de la multiplication. Ceci va probablement grandement améliorer la vitesse du programme, car une multiplication d'entiers prend environ 30 cycles, alors qu'un décalage binaire n'en prend qu'un seul.

Une autre amélioration serait d'implémenter matériellement la convolution (qui contient les multiplications) pour pouvoir paralléliser au maximum les calculs. Il faut donc réfléchir à une autre manière de stocker et envoyer les poids à l'architecture dédiée.

3 Bibliographie

Plasma :

— Camille Leroux. Github - Plasma PMOD.
https://github.com/madellimac/plasma_pmod, 2019.

CVA6 :

— OpenHW GROUP. Github - CVA6.
https://github.com/openhwgroup/cva6, 2018.

IA :

— Guillaume Bourmaud. Teaching.
https://gbourmaud.github.io/teaching/, 2024.

— Euske. Github - Convolutional Neural Network in C (for educational purposes).
https://github.com/euske/nn1, 2021.