Guinebert Iban, Maillefert Antoine

Grenoble INP Phelma, 3ème année SEI SoC

Compte rendu du projet d'architecture
Convolutional Neural Network

2019-2020

Abstract

Ce document constitue un compte rendu de projet de 3ème année dans le cadre du cours "Adéquation algorithme et architecture" pour la filière SEI de Grenoble INP Phelma. Il est dédié à un exemple de Convolutional Neural Network (CNN) et à son implémentation sur FPGA Zybo avec également gestion d'un flux caméra et affichage sur écran VGA.

Table des matières

Table des figures

Liste des tableaux

Listings

Glossaire

Convolutional Neural Network (CNN) Réseau de neurones à base de calculs convolutifs . 1, 3–6, 8, 9, 11, 12

Floating Point Unit (FPU) . 10

High Level Synthesis (HLS) . 3, 10, 12

CIFAR10 Banque d'images labellisées cf [2]. 8, 9, 13

Références

[1] Algorithmic c datatypes. URL `https://hlslibs.org/#collapseACDatatypes`.

[2] A. Krizhevsky. Cifar-10 website. URL `https://www.cs.toronto.edu/~kriz/cifar.html`.

[3] Wikipédia. Convolutionnal neural network. URL `https://en.wikipedia.org/wiki/Convolutional_neural_network`.

Introduction

Contexte

Le développement et l'usage des réseaux de neurones et du "machine learning" s'est démocratisé ces dernières années, du fait de la grande amélioration de leurs performances, notamment pour ce qui est la reconnaissance d'image. Cependant, il s'avère que les réseaux "classiques" dis "perceptron" ne sont pas les plus efficaces pour le cas de la reconnaissance du sujet depuis une image, on utilise plutôt dans ce cas un CNN. En effet, à la place d'un perceptron multi-couches complètement connectées, où chaque neurone est connecté à tous les autres ; les CNN sont inspirés de la façon dont les animaux traitent les images qu'ils reçoivent, c'est à dire qu'une partie des neurones ne réagit qu'à une petite partie des pixels en entrée du réseau[3].

On dispose au début de ce projet d'un jeu de coefficients pour un CNN déjà entraîné ainsi que des exemples de projets catapult C et Vivado.

Objectifs

L'objectif principal de ce projet est la familiarisation avec le flot de développement hardware depuis un algorithme jusqu'à un bitstream pour FPGA en utilisant les outils de synthèse High Level Synthesis (HLS) de Catapult C et Vivado. Pour ce faire, on implémente un algorithme de Réseau de Neurones Convolutif sur une FPGA Zybo sous la forme d'un module matériel. Ce module doit être capable de reconnaître des images issues de la banque d'image CIFAR10 avec un taux de succès de l'ordre de 75%.

Déroulement du projet

Le projet se déroulera en plusieurs étapes, le plan de ce rapport reproduit le flot de développement :

- Développement Référence algorithmique sous la forme d'un programme Python
- Implémentation d'un premier module sur FPGA Zybo procédant à une convolution d'une image provenant d'une caméra et affichant le résultat sur un écran
- Implémentation d'une architecture hardware d'un CNN sur FPGA Zybo avec affichage du résultat sur un écran.

Première partie

Référence algorithmique

Avant de considérer l'architecture hardware sur FPGA, il est nécessaire d'effectuer la validation fonctionnelle de l'algorithme qui sera implémenté par la suite. Le système réalisé est un système de reconnaissance du sujet d'images basé sur la base de données CIFAR10.

1 Rappel de l'architecture du CNN de type CIFAR10

On dispose de la base de donnée CIFAR10 qui comprend plusieurs batches d'images 32x32 pixels RGB en format binaire. Certains, sont des batches d'entraînement et comprennent également un octet étiquette qui décrit le contenu de l'image. Le CNN choisi pour la suite se base donc sur le modèle du CIFAR10.

L'architecture se base sur une superposition de couches de traitements de différents types :

- convolution : effectue une convolution en 2 dimensions d'un kernel de poids sur une image d'entrée et ajoute un biais selon le canal de sortie
- Activation : introduit de la non-linéarité dans le traitement, ici la couche ReLU est directement intégrée à l'étape de convolution et filtre simplement les valeurs négatives pour ne garder que les valeurs positives issues de la convolution.
- Maxpooling : réduit la taille de l'image d'entrée en ne gardant que la valeur maximale parmi un groupe de valeur.
- Perceptron : effectue la multiplication d'un vecteur d'entrée avec une matrice de poids pour récupérer les coefficients finaux, auxquels sont sommés des biais.

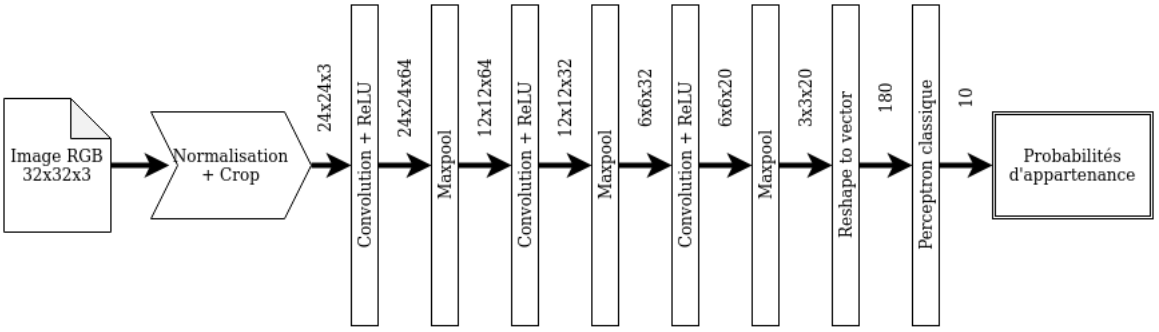


Figure 1 – Schéma de l'architecture du CNN de type CIFAR10

2 Implémentation en Python

Pour valider l'algorithme d'un point de vue fonctionnel, une première version de CNN est réalisé dans le langage python. L'objectif de cette étape du projet est l'établissement d'une plateforme en python qui puisse permettre de tester l'algorithme et vérifier qu'on obtient un taux de succès de reconnaissance d'image de l'ordre de 75% avec le jeu de coefficients fournis. Cette réalisation peut-être découpée en plusieurs étapes :

- gestion des entrées (images RGB 32x32) d'une part au format pgm/ppm et d'autre part directement depuis les fichiers binaires.
- implémentation des différents traitements ou couches du CNN sous la forme de classes pour faciliter le débogage et la construction d'architectures différentes
- implémentation de fonctions d'importation des jeux de coefficients sous forme de variables de type dictionnaires.

— test de chaque module de traitement séparément puis test global du fonctionnement du CNN sur un batch de plusieurs images.

2.1 Structure du code pour le CNN

Pour rendre le code plus modulable, on définit une classe "CNN" qui possède comme attribut un tableau de "layers" ordonnés dans l'ordre d'exécution. La classe layer quant à elle est la classe parente de toutes les couches vues dans la section 1.

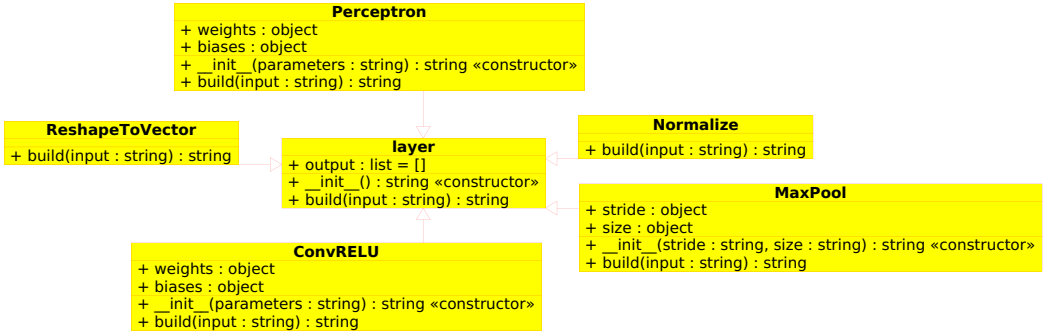


FIGURE 2 – Diagramme de classes des couches du CNN

Le diagramme figure 2 présente les prototypes de la classe "layer" et de ses classes filles. La routine de calcul de chaque couche est implémentée dans la fonction build de chaque classe layer. Elle prend en argument un tableau de données et retourne le tableau après traitement. La construction du modèle complet se fait ensuite simplement grâce à des fonctions de la classe CNN tel que présenté dans l'extrait du code CNN_upgrade.py (listing 1).

Listing 1 – Exemple de construction d'un CNN par ajout de layers

```
    def cifar10():
        import dicoCoeff
        d=dicoCoeff.DicoCoeff("CNN_coeff_3x3.txt")
        cnn=CNN(d.dico)
        cnn.addNormalizeLayer()
        cnn.addConvolLayer("conv1")
        cnn.addMaxPoolingLayer([3,3],[2,2])
        cnn.addConvolLayer("conv2")
        cnn.addMaxPoolingLayer([3,3],[2,2])
        cnn.addConvolLayer("conv3")
        cnn.addMaxPoolingLayer([3,3],[2,2])
        cnn.addReshapeToVectorLayer()
        cnn.addPerceptron("local3")
        return(cnn)
```

Pour lancer le traitement du modèle construit, il suffit d'utiliser la fonction run de la classe CNN qui va successivement appeler les routines de chaque layer dans l'ordre d'ajout au modèle (listing 2).

Listing 2 – Routine run de la classe CNN (extrait de CNN_upgrade.py)

```
class CNN:
  #[...]
  def run(self,inputPic):
      buf=inputPic
      write_pgm(inputPic,"input")
      if(len(inputPic)>0):
          for layer in self.layers:
              layer.build(buf)
              buf=layer.output
      return buf
```

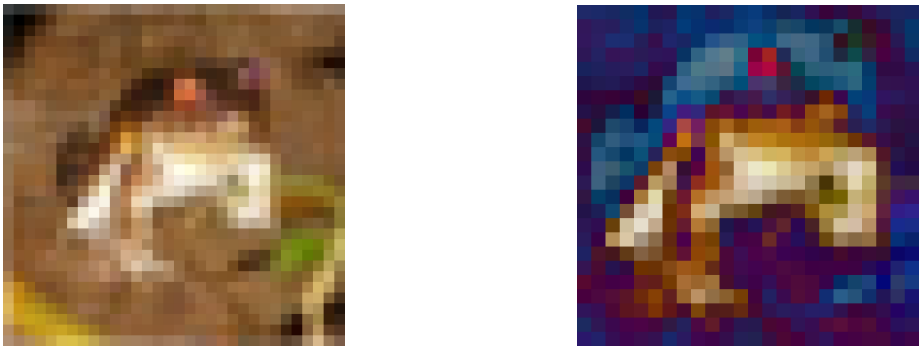
De manière générale, on travaille avec des numpy array de 3 ou 4 dimensions. Cela facilite le travail avec les images et notamment l'affichage des traces et le fait de pouvoir reformer la matrice grâce aux fonctions reshape et accéder rapidement aux tailles des matrices.

2.1.1 Étage de normalisation

La normalisation est une étape essentielle car elle permet de concentrer les valeurs des pixels des images autour de 0. Ce traitement permet d'uniformiser les images en entrée du CNN. L'opération de normalisation est formalisée selon les équations (1) et (2)

$$m_{norm}[i,j] = \frac{(m[i,j]-\mu)}{max(\sigma, \frac{1}{\sqrt{N}})} \tag{1}$$

$$\text{Avec} \quad \mu = \frac{1}{N}\sum m[i,j] \quad \text{et} \quad \sigma = \sqrt{\frac{1}{N}\sum (m[i,j]-\mu)^2} \tag{2}$$



(a) Avant normalisation

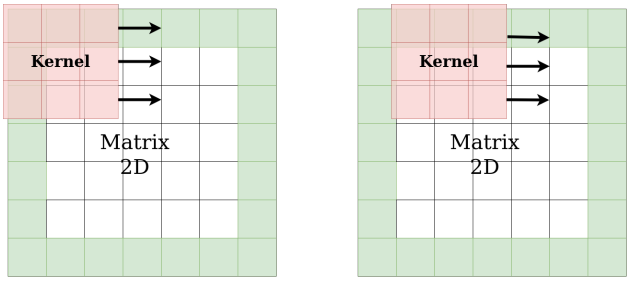
(b) Après normalisation

FIGURE 3 – Exemple de passage d'une image dans la couche de normalisation (valeurs des pixels x80 pour l'affichage de la sortie)

Ici, le calcul de μ et de σ se fait sur les trois composantes Rouge Vert et Bleu puis, on applique la formule (1) sur tous les pixels d'entrée. Après normalisation, on a $\forall(i,j), m[i,j] \in [-3,3]$, pour rendre l'image visible et pertinente, on affiche $\forall(i,j), m_{print}[i,j] = |80 * \lfloor m[i,j] \rfloor|$ (cf fig. 3b).

2.1.2 Étage de convolution et d'activation ReLU

L'étage de convolution est le coeur du CNN. Cette étape modélise le traitement de neurones spécialisées dans la reconnaissance visuelle telle que pouvant être trouvée chez les animaux. Chaque zone de l'image est traitée par des neurones différents, ayant donc des poids et des biais associés. La convolution utilisée dans le code python du fichier "CNN_upgrade.py" commence dans le coin



(a) Étape 1 (b) Étape 2

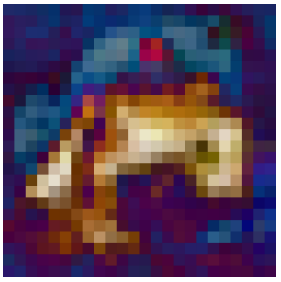
FIGURE 4 – Schéma de principe de la convolution en deux dimensions pour les deux premiers pixels d'un canal

supérieur gauche de l'image d'entrée. Dans ce cas, le kernel "déborde" à gauche et en haut (fig.4) et on considère alors que les valeurs des pixels autour de l'image d'entrée sont à 0. . L'équation 3 formalise cette opération.

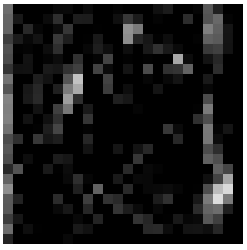
$$M_{out}[i,j,c] = \sum_{m,n,l} M_{in}[i+m, j+n, l] * Ker[m,n,l,c] \tag{3}$$

Listing 3 – Implémentation Python de la convolution + ReLU

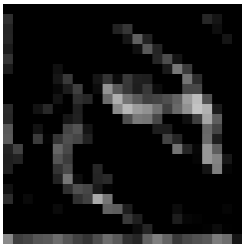
```
class ConvRELU(layer):
    def __init__(self,parameters):
        self.weights=parameters[0]
        self.biases=parameters[1]

    def build(self,input):
        height,width,canal=input.shape
        ker=self.weights
        bias=self.biases
        kHeight,kWidth,kChannelI,kChannelO=ker.shape
        SizeOut=height*width*kChannelO
        self.output=np.array([0 for t in range(SizeOut)],dtype=np.float64)
        self.output.reshape(height,width,kChannelO)
        for row in range(height):
            for col in range(width):
                for outChannel in range(kChannelO):
                    s=0
                    for kerH in range(-(kHeight//2),kHeight//2+1):
                        for kerW in range(-(kWidth//2),kWidth//2+1):
                            for inChannel in range(canal):
                                if (kerH+row < height and kerW+col < width):
                                    if( kerH+row>=0 and kerW+col>=0):
                                        M=input[row+kerH,col+kerW,inChannel]
                                        K=ker[kerH+1,kerW+1,inChannel,outChannel]
                                        s=s+M*K
                    s=s+bias[outChannel]
                    if (s<0):#activation ReLU
                        s=0
                    self.output[row,col,outChannel]=s
```



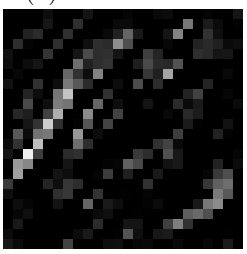
(a) Avant convolution



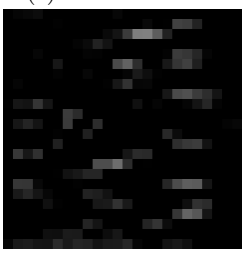
(b) Sortie Canal 2



(c) Sortie Canal 8



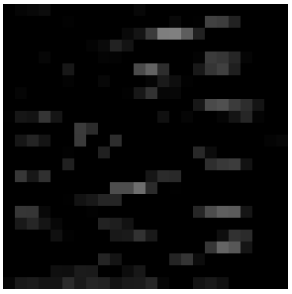
(d) Sortie Canal 13



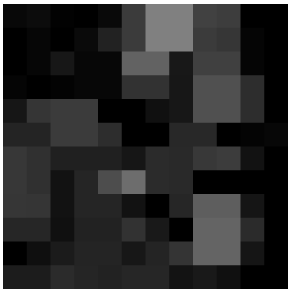
(e) Sortie Canal 62

FIGURE 5 – Visualisation du résultat de la première couche de convolution sur quelques canaux de l'image de sortie

2.1.3 Étage de Max pooling



(a) Canal 62 avant maxpool



(b) Canal 62 après maxpool

FIGURE 6 – Exemple de la sortie du maxpooling pour le canal 62 de l'image précédente

2.2 Importation du dictionnaire de coefficients

Le réseau CNN a déjà été entraîné au préalable sur la banque d'images CIFAR10. Les coefficients correspondant à l'architecture (fig.1), sont stockés dans un fichier texte. Pour utiliser ces coefficients dans le réseau implémenté en python, un programme est utilisé, il est contenu dans le fichier "dicoCoeff.py".

La classe DicoCoeff contient les fonctions nécessaires à la lecture des coefficients et la création d'un dictionnaire. Lorsqu'un objet DicoCoeff est instancié, on passe en argument le nom du fichier txt comme visible sur le listing 2 à la ligne 3, le chargement du dictionnaire dans le champ dico de DicoCoeff se fait automatiquement. De même, lorsqu'on instancie un objet CNN, on doit passer en argument un dictionnaire de coefficients. Ensuite, lorsqu'on ajoute des layer qui nécessite des coefficients, on passe la clé correspondante (par exemple : "conv1" ou "conv2", le programme récupère les weights et biases automatiquement).

2.3 Test de fonctionnement et débogage

Pour tester le modèle et les différents layer, plusieurs script python sont utilisés :

- testCifar10.py : lance le CNN sur plusieurs images depuis un fichier binaire CIFAR10 et affiche les résultat ainsi que le taux de succès
- testLayers.py : génère des images au format pgm pour visualiser les phases intermédiaires entre les couches du CNN

2.3.1 Validation fonctionnelle du CNN

Le programme testCIFAR10.py charge successivement les images depuis le fichier binaire data_batch_1.bin grâce aux fonctions du fichier "ioPGM.py" dans des numpy array, il lance ensuite le modèle du CNN et affiche les probabilités d'appartenances à une classe, si le résultat correspond au label et le taux de réussite sur l'ensemble des images traitées (figure 7).

```
Success
[ 4.56606569 12.82037991  0.81151779 -1.11039582 -1.3140827  -2.21411405
  1.82835475 -0.34294435  1.06980764  7.79839244]
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
Image n° 33 automobile
result =  automobile
Synthese : taux de réussite= 0.7878787878787878 ( 33  images treated,  26  success, 7 failures

Success
[-1.2863697  -1.12374892  3.96280517  9.12105166  5.87225282  4.85453478
  6.31205329  1.53487242 -0.71128399 -2.6067998 ]
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
Image n° 34 cat
result =  cat
Synthese : taux de réussite= 0.7941176470588235 ( 34  images treated,  27  success, 7 failures

Success
[ 2.92384527 -1.20342498  6.93499237  5.17116624 13.33286288  2.91118333
  4.43474096 -0.73780546 -1.89924555 -1.46299223]
['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
Image n° 35 deer
result =  deer
Synthese : taux de réussite= 0.8 ( 35  images treated,  28  success, 7 failures
```

FIGURE 7 – Capture d'écran de la sortie standard pendant l'exécution de "testCIFAR10.py"

Conclusion

La validation de l'algorithme n'a pas pu être réalisée dans un premier temps malgré débogage, en effet, le taux de succès stagnait aux environ de 25%. La résolution du problème n'est survenue qu'après la démonstration finale du projet. Cependant dans l'état actuel du projet, le programme python fonctionne et donne un taux de succès de l'ordre de 75% comme attendu.

Deuxième partie

Synthèse Hardware

1 Spécificités du code C pré-HLS

Pour générer une netlist avec Catapult, il suffit de disposer d'un code source écrit en C/C++ ou en System C pour réaliser une synthèse (HLS). Cependant, certaines précautions et ajustement sont nécessaires par rapport à un code à objectif purement logiciel. Les éléments développé dans cette section sont valables pour les tous les projets détaillés par la suite.

— les matrices de pixels sont des tableaux à une dimension
— tous les tableaux sont alloués de manière statique à l'aide de macros
— on utilise des données de type virgule fixe

2 Virgule fixe

Dans la version de référence en python, les données numériques sont des nombres flottants. Pour l'implémentation matériel cependant, on n'utilise pas de Floating Point Unit (FPU) et on va plutôt utiliser des nombres à virgules fixes plus adaptés aux calculs à virgule sur un système embarqué. Pour ce faire, on utilise la librairie ac_types [1] et plus particulièrement le type ac_fixed.

Plusieurs formats d'ac_fixed (cf table 1) sont utilisés dans ce projet, ces types sont définis dans le fichier "type.h" via des macros définies dans "macro_CNN.h". Les valeurs des pixels des images sont des entiers dans [0 ;255] donc on utilise un type ac_fixed de 8 bits entiers renommé CNN_IMAGE_TYPE. Pour ce qui est des données internes au modules de calculs, on utilise le type CNN_DATA_TYPE.

Type	Taille(bits)	Taille entière(bits)	Signé	Quantification
CNN_IMAGE_TYPE	8	8	non	arrondi
CNN_DATA_TYPE	20	10	oui	arrondi
CNN_MASK_TYPE	1	1	non	arrondi

Table 1 – Types utilisés dans le code C du CNN

3 Convolution seule

Avant de passer à l'implémentation du CNN complet, on implémente tout d'abord une simple convolution, qui effectue une détection de contour d'une image. Le module hardware sera intégré dans un système relié à une caméra et à un écran. La sortie ainsi que l'entrée sont deux tableaux de taille 320x240x8bits. La convolution est effectuée sur la totalité de l'image venant de la caméra soit 320x240 pixels en nuance de gris. Le module écrit ensuite dans un tableau de sortie.

Le code C pré-HLS consiste simplement en une fonction de convolution. Le kernel utilisé est le suivant :

$$kernel_{edgeDetector} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{4}$$

(a) Image d'entrée

(b) Image de sortie

FIGURE 8 – Exemple d'entrée et sortie du détecteur de contour (test C linux)

3.1 Description des fichiers

Pour implémenter le détecteur de crête, on a besoin d'une version simplifiée de l'algorithme et des fichiers utilisés pour l'implémentation du CNN, ainsi que des fichiers dédiés à l'utilisation de la virgule fixe.
On va d'abord écrire un fichier types.h définissant des nouveaux types utilisant le format ac_fixed, et qui seront utilisés dans l'intégralité des autres fichiers.
On écrit ensuite le fichier macros_CNN.h, permettant tout d'abord de paramétrer les types définis dans types.h, puis de définir des valeurs constantes réutilisables tout au long de l'algorithme.

On n'utilise ici que la convolution parmi toutes les fonctions développées pour le CNN. Ainsi, le fichier convolutionSimple.cpp est une version simplifiée de la convolution qu'on utilisera dans le CNN, sans utilisation de biais ni de choix multiples de macros. On se contente donc de transcrire l'algorithme d'une convolution. Le fichier convolutionSimple.h est le header de convolutionSimple.cpp, permettant notamment l'utilisation des types décrits au format virgule fixe.

De la même manière, le kernel utilisé pour cette convolution, et implémenté dans kernel.h, est beaucoup plus petit que celui utilisé pour le CNN, comme décrit plus haut, mais ce dernier sera écrit de la même manière.
Enfin, edgeDetector.cpp, et son header : edgeDetector.h, servira de fichier top, faisant le lien entre l'entrée et la sortie du module. Elle est écrite de manière à pouvoir être synthétisée correctement par CatapultC, et pouvoir permettre l'utilisation de la caméra en entrée et du VGA en sortie.

3.2 Synthèse du module détecteur d'arêtes

Une fois la validation fonctionnelle effectuée sur ubuntu, Catapult C est utilisé pour synthétiser le code en une netlist hardware sous la forme d'un fichier "ImgProcTest.edf". On utilise pour ce faire le script directive.tcl situé dans le dossier "code/CPP/PreHLS/EdgeDetector/", sur lequel on modifie juste les fichiers pris en compte pour la synthèse, et leur emplacement.
Dans un premier temps, on utilise ce fichier directive.tcl pour effectuer la synthèse HLS. Ce fichier décrit les étapes suivantes :

- choix des fichiers utiles à la synthèse
- création d'une hiérarchie de ces fichiers
- choix des librairies utiles
- Mapping
- création de l'architecture
- planification des tâches
- schématisation RTL

On retiendra parmi les paramètres que l'on applique à la synthèse (présents dans directives.tcl) le choix de ne pas unroll les différentes boucles, la taille des tableaux fixée à 1024 et la période fixée à 10ns.

CatapultC Une fois la synthèse HLS terminée, on génère une netlist au format edf, que l'on va pouvoir intégrer à un projet vivado pré-existant. Ce projet décrit le système d'acquisition et d'affichage vidéo, il est donc nécessaire de l'utiliser comme base pour la suite de la synthèse. C'est également pour pouvoir intégrer correctement la netlist à ce projet que des contraintes de nomenclatures ont été imposées dans l'écriture du code source (comme le nom ImgProcTest de la fonction main de l'ensemble du module).

Vivado On va maintenant utiliser Vivado dans le but de générer un fichier bitstream permettant de contrôler la sortie caméra. Après avoir remplacé la netlist déjà présente par celle générée avec CatapultC, on va pouvoir ouvrir le projet avec Vivado, et mettre ainsi à jour les IP pour intégrer notre module au projet. Il reste à effectuer la Synthèse et l'Implémentation du design, et enfin générer le fichier Bitstream.
On peut maintenant programmer la carte Zybo avec ce fichier bitstream, et ainsi configurer la caméra avec le filtre de détecter de crête.

Résultats Les résultats de cette programmation sont satisfaisants. Avec un délai moindre entre l'acquisition de l'image en entée par la caméra et l'affichage en sortie sur l'écran via un port VGA, on observe bien une image ne présentant que les contours des formes observées en entrée, comme décrit au début de cette partie. Ceux-ci, bien que manquant un peu de contraste, sont nets et témoignent du bon fonctionnement du module généré.
Ce contraste peut être amélioré par simple modification du kernel de convolution (suivi d'une nouvelle synthèse du module), avec des valeurs plus disparates, et donc sans modifications de l'algorithme en lui-même.
Outre le point précédent, le principal objectif d'amélioration de ce module serait la réduction du délai entre l'acquisition des images et l'affichage de celles-ci, et donc augmenter la vitesse de traitement des images par notre module.

L'implémentation d'une convolution sous la forme d'un détecteur de contours étant un succès, nous pouvons maintenant nous pencher sur l'intégration du CNN tout entier.

4 CNN complet

Une fois la convolution simple implantée sur FPGA Zybo et la confirmation du fonctionnement de l'edge detector faites, le prochain objectif est de réaliser le CNN complet en version matérielle.

Le CNN est entrainé exclusivement avec des image provenant de la banque de donnée CIFAR-10. On choisit donc, pour faciliter le test de l'implantation FPGA, d'utiliser une image pré-chargée au moment de la synthèse dans la mémoire et d'afficher le résultat du CNN sur un écran VGA.

4.1 Code source

La structure du code pré-HLS ne reprend pas celle de la référence python. Chaque layer du CNN est implémenté sous la forme d'une fonction décrite dans un couple (.cpp ;.h), le fichier CNN.cpp (listing 4) appelle ensuite successivement ces fonctions en passant en argument les tableaux d'entrée et de sortie. Pour éviter la multiplication des tableaux en RAM, on utilise deux tableaux mem1 et mem2 qui sont de la taille maximale qu'occupent un tableau en mémoire pendant le processus, c'est à dire lors de la sortie de la 1ère convolution. La normalisation ainsi que le découpage centré de l'image ont été fait avant grâce à un script Python.

Listing 4 – Contenu du fichier CNN.cpp

```
#include "CNN.h"
#include "imageNorm.h" //import static array with normalized picture
#include "image.h" //import static array with non normalized picture
#pragma hls_design top //tells Catapult that it is the top level module
/**
* @param img_in Input from the camera
* @param img_out Output to the screen
**/
void ImgProcTest(CNN_IMAGE_TYPE img_in[CNN_VGA_SIZE],
                 CNN_IMAGE_TYPE img_out[CNN_VGA_SIZE])
{
  // memories for intemediate results
  CNN_DATA_TYPE mem1[CNN_CONV1_OUT_SIZE];
  CNN_DATA_TYPE mem2[CNN_CONV1_OUT_SIZE];

  img_out[0]=img_in[0];//only serves to prevent Catapult to erase img_in from design

  convolutionReLU(imageNorm,mem2,conv1_weights,conv1_biases,1);
  maxpool(mem2,mem1,1);
  convolutionReLU(mem1,mem2,conv2_weights,conv2_biases,2);
  maxpool(mem2,mem1,2);
  convolutionReLU(mem1,mem2,conv3_weights,conv3_biases,3);
  maxpool(mem2,mem1,3);
  //no need for reshape
  perceptron(mem1,mem2,local3_weights,local3_biases);

  char label=0;
  CNN_DATA_TYPE max=-510;
  for (int j=0;j<10;j++) //seek for the max probability
  {
    if (mem2[j]>max)
    {
      max=mem2[j];
      label = j;
    }
  }
  display(label,imageIN,img_out);
}
```

On notera en particulier les évolutions suivantes par rapport au code source utilisé pour le détecteur de crête :

- CNN.h et CNN.cpp prennent le rôle de fichier top.
- 3 convolutions étant nécessaire au fonctionnement complet du CNN, la convolution telle que décrite dans la paire convolution.cpp/convolutionReLU.h permet de choisir les macros à utiliser en fonction de la place de la convolution dans le CNN. On ajoutera en plus l'action des biais en fin d'algorithme.
- Un fichier parameters.h remplace le fichier kernel.h. Il contient le tableau des biais ainsi que l'ensemble des kernels utilisés pour les convolutions.
- des fichiers maxpool.cpp/maxpool.h, perceptron.cpp/perceptron.h ont été ajoutées pour remplir leurs fonctions comme étages du CNN.
- les fichiers display.cpp/display.h gèrent désormais l'affichage
- un fichier masks.h décrit les différents masques, et macroMask.h définit des macros spécialement pour ces masques.

4.2 Affichage et overlay

Pour afficher la sortie du CNN, on crée des masques avec le nom des classes (frog, automobile, truck...) avec un logiciel de dessin puis avec un script python, on génère le fichier masks.h contenant dix array de CNN_MASK_TYPE. La fonction display combine ensuite imageIN avec le mask correspondant au label identifié dans img_out qui est envoyé sur la sortie.

4.3 Test sur cible linux

Pour vérifier le fonctionnement du CNN deux différents testbench en C++ sont utilisés. Le premier, "testbenchCNN.cpp" permet de tester le code du CNN tel qu'il sera utilisé dans la synthèse, avec l'image normalisé dans "imageNorm.h" et l'image d'entrée dans "image.h". Celui-ci ne permet pas d'afficher les valeurs intermédiaires, on utilise donc un deuxième testbench "testbenchBatch.cpp" qui permet de lancer le CNN sur tout un batch d'image CIFAR10 et génère les fichiers pgm intermédiaires.

(a) Grenouille (b) Automobile

FIGURE 9 – Fichiers pgm issus du test du code source C/C++ du CNN

```
2.8310546875 14.732421875 -.908203125 1.37109375 -.51171875 -2.40234375 2.5107421875 -.3818359375 2.505859375 6.98046875
Image 94 [automobile] / result : automobile : Success

1.759765625 -1.587890625 5.5361328125 5.537109375 6.51171875 1.783203125 12.6962890625 -.783203125 -.9716796875 -1.7490234375
Image 95 [frog] / result : frog : Success

4.2294921875 14.6640625 -.0791015625 -.755859375 -1.5234375 -2.931640625 2.7158203125 -3.0263671875 5.279296875 4.4404296875
Image 96 [automobile] / result : automobile : Success

2.1845703125 4.71484375 2.4208984375 1.1162109375 2.0673828125 1.1572265625 3.412109375 -.1962890625 .765625 1.4794921875
Image 97 [automobile] / result : automobile : Success

.994140625 .6533203125 2.9189453125 3.6982421875 7.466796875 3.4736328125 6.873046875 .9697265625 -.6943359375 1.0419921875
Image 98 [deer] / result : deer : Success

2.2880859375 14.6748046875 -1.2001953125 .650390625 -.8828125 -2.45703125 2.333984375 -1.857421875 2.982421875 6.9716796875
Image 99 [automobile] / result : automobile : Success
total success rate = 0.72
```

FIGURE 10 – Sortie standard pendant l'exécution de "CNN_Batch.exe -v 100"

4.4 Synthèse et implantation sur FPGA Zybo

Synthèse du module La synthèse du code est réalisée avec le logiciel "Catapult C", pour cela on utilise le script directive.tcl situé avec les fichiers sources dans le dossier "projet_CNN/code/CPP/PreHLS/CNN/".

Génération du bitstream On intègre la netlist au format edf dans le projet Vivado HDMI_PROC_TEST comme pour la convolution seule. Vivado procède à la synthèse du système complet puis à son implémentation sur la cible FPGA enfin il génère le bitstream de l'architecture matérielle complète. La figure 11 présente les principaux rapport de consommation, de timing et d'utilisation de la FPGA.

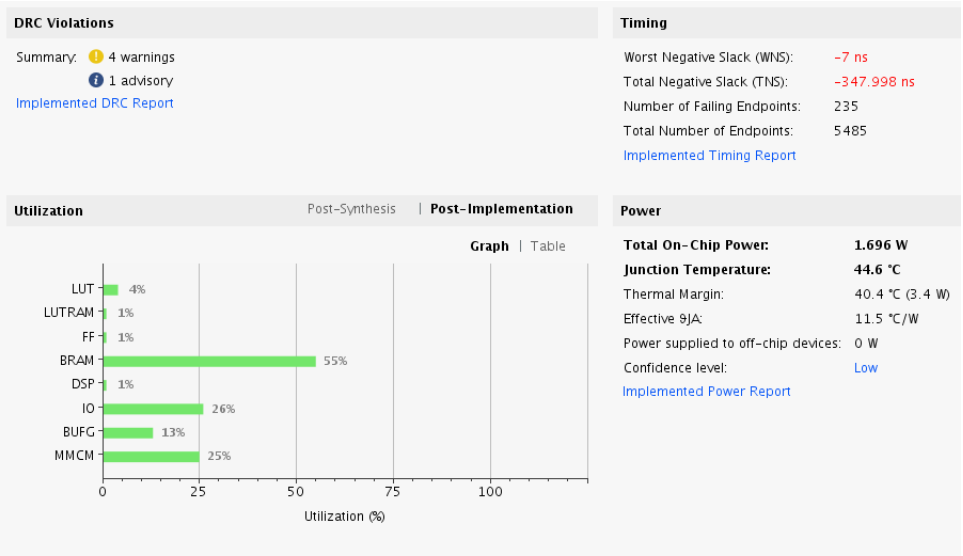


FIGURE 11 – Extrait du résumé après génération du bitstream sur Vivado pour le projet HDMI_PROC_TEST

4.5 Résultats

Le premier résultat, celui qui a été présenté, est un échec partiel. La synthèse du CNN à pu être effectuée entièrement et on peut donc obtenir un affichage sur écran d'une image et du résultat de son traitement par le filtre CNN en overlay. Néanmoins, l'algorithme du CNN sur lequel se base cette implémentation présentant des défauts que nous n'avons pas pu résoudre à temps, le traitement synthétisé et donc le résultat affiché sont donc également faux.
Après la présentation, l'algorithme a pu être corrigé, donnant les résultats attendus en test. Une synthèse complète et fonctionnelle de cette version a pu être faite, mais malheureusement pas programmée sur la carte. Nous nous attendons cependant au résultat attendu, c'est à dire un affichage identique à la version précédente (image traitée et overlay), mais avec cette fois-ci un traitement fonctionnel de l'image.

5 Résultats et améliorations futures

Lors de la démonstration, réalisée en salle de projet, nous avons pu montrer le fonctionnement de la convolution seule qui réaliser un filtre détecteur de contour. Nous avons pu montrer le fonctionnement du CNN sur FPGA Zybo et l'affichage de l'image avec un masque mais à ce moment, notre algorithme ne donnait pas le bon résultat en sortie du CNN. Après cela, nous avons finalement pu résoudre le problème et obtenir les taux de résultats attendus, il s'agissait d'une erreur d'index dans la convolution qui n'influe pas sur le design du module mais impacte suffisamment l'algorithme pour fausser le CNN.
Nous avons pu refaire la génération de bitstream avec cette nouvelle version sans toutefois pouvoir la tester sur la FPGA.

La première chose à faire dans l'optique de continuer le projet serait de tester l'affichage du CNN donnant les résultats attendus. Au-delà de ce test, le détecteur de crête comme le CNN peuvent être améliorés.
Comme dit plus haut, le contraste du détecteur de crête pourrait être affiné. On pourrait également tenter de réduire le temps de traitement de l'image par le module, ce qui est également le cas du CNN complet. En effet, si le CNN a été implémenté, l'architecture du module n'a as été optimisée, et c'est la recherche de cette optimisation qui pourrait être au coeur d'une poursuite de notre travail.

Rajoutons que dans l'état de notre CNN actuel, une seule image est affichée en sortie. Une modification de la fonction de display pourrait sans doute permettre d'afficher les quelques images suivant celle-ci dans le traitement dans le batch. En parallèle, une version de la convolution permettant un affichage multiple d'une image traitée par différents filtres était en développement lors du rendu du projet. Bien que synthétisable par CatapultC et Vivado, ce module ne fonctionnait pas, et est encore à l'état de test (le code dédié à cette partie est donné en archive). Comme ce module aurait permit en divisant l'écran l'affichage d'un nombre d'image choisi par l'utilisateur en amont de la synthèse, son utilisation aurait pu permettre l'affichage de plusieurs résultat issus du CNN en l'incorporant au design du CNN complet
L'algorithme d'affichage multiple tel que décrit ici étant trop complexe pour l'usage souhaité au sein du CNN, une simple modification de la fonction display serait à envisager en premier lieu, mais la complétion de ce module et l'ajout à l'architecture globale pourrait être un objectif intéressant dans le cadre d'une poursuite du projet.

Conclusion

Ce projet a pour nous été difficile, puisque n'ayant pas pu obtenir un algorithme fonctionnel à temps, nous avons pris du retard pour les synthèses, retard qui ne nous a pas permit de mener à bout tous les travaux que nous avions entrepris au cours du projet. Au final, nous obtenons une implémentation fonctionnelle du CNN, mais l'algorithme étant faux et n'ayant pas pu être corrigé dans les phases finales du projet, nous n'arrivons pas vraiment au résultat attendu.
Cette expérience a néanmoins été formatrice, puisque nous confrontant à ce type d'obstacle que nous rencontrerons forcément au cours de notre carrière.

Ce projet nous a cependantt permis de nous familiariser avec l'utilisation de CatapultC pour la synthèse HLS, le développement de modules matériels depuis un programme écrit en C. Nous avons également pu prendre en main le logiciel Vivado pour la synthèse FPGA et la génération de bitstream. De plus, ce projet nous a permis de mettre en pratique les notions vues en cours d'adéquation algorithme et architecture concernant le traitement d'image, les mémoires matérielles et les calculs en virgule fixe.

Nous avons également une meilleure compréhension du fonctionnement des Réseaux de neurones convolutifs, de leur intérêt mais également de leur limites.

Troisième partie

Annexes

Cette partie comprend quelques informations utiles quant à l'utilisation de l'archive du projet fournie avec le présent document.

1 Structure du répertoire

```
TOP
├── Bitstreams // contient les bitstreams générés
│   └── old //contient les bitstreams des anciennes versions
├── code
│   ├── CPP
│   │   ├── PreHLS
│   │   │   ├── CNN // sources pour la synthèse du CNN
│   │   │   ├── Convolution // sources pour la synthèse de la convolution uniquement
│   │   │   └── EdgeDetector // sources pour la synthèse de l'Edge detector
│   │   └── TESTS
│   │       ├── CNN_FULL // tous les fichiers nécessaire au test du CNN complet en C
│   │       │   sur linux
│   │       ├── EdgeDetector // tous les fichiers nécessaire au test de l'Edge
│   │       │   Detector en C sur linux
│   │       └── MultipleDisplay // les fichiers pour un affichage de plusieurs
│   │           convolution (non terminé)
│   └── Python // les fichiers de référence pour l'algorithme du CNN en Python
├── doc //contient ce rapport
├── EDF_files //les fichiers edf générés par Catapult C
├── masks // les 10 fichiers servant de masques au format pgm
└── scripts //des scripts utiles à la génération de différents fichiers
```

FIGURE 12 – Structure global du répertoire projet_CNN

2 Référence algorithmique (Python)

```
Python
├── generated_pgm //dossier pour les fichiers images générés
├── batches.meta.txt//contient les noms des classes d'image
├── clean.sh//script pour nettoyer le répertoire
├── CNN_coeff_3x3.txt//contient les coefficients du CNN entrainé
├── CNN_upgrade.py//contient la classe CNN et les classes layers
├── CNN.py//ancienne version du CNN
├── dicoCoeff.py//construit un dictionnaire de coefficient à partir d'un fichier
├── data_batch_1.bin
├── generateImageHeader.py//crée header avec image et sa normalisation
├── generatePNGTraces.sh//lance testCIFAR10.py et convertit certains pgm en png
├── ioPGM.py//fonctions d'écriture/lecture dans des fichiers
├── README.txt
├── testCIFAR10.py//script de test de CNN_upgrade.py sur un batch
└── testLayers.py//script de test des layers + affichage pgm
```

FIGURE 13 – Contenu du dossier Python

2.1 Utilisations des scripts

testCIFAR10.py Pour lancer la procédure de test du CNN complet sur un ensemble d'image (contenues dans le fichier data_batch_1.bin. On utilise la commande suivante dans un terminal.

```
python testCifar10.py
```

Pour changer le nombre d'image traitée, il suffit de modifier la valeur dans la boucle while du fichier "testCIFAR10.py".

```
while imglder.shift<1000:
```

testLayers.py Pour lancer la procédure de test qui va générer les images intermédiaires du traitement d'une image au format pgm ascii, on utilise la commande :

```
python testLayers.py
```

Les images de la première partie de ce rapport au format png sont générées grâce au script "generatePNGTraces.sh".

```
./generatePNGTraces.sh
```

dicoCoeff.py Si ce fichier est lancé comme un script python, il génère un fichier .h avec les différents tableaux de coefficients pour le CNN en C.

3 Pré-HLS (C/C++)

Pour fonctionner, les tests suivant nécessitent l'installation des librairies ac_types et ac_math [1].

```
make install_ac_libs
source sourceme.sh
```

La première commande télécharge les librairies depuis github.com dans le dossier libs (cf fig.12). La seconde exporte les variables d'environement qui pointent sur ces librairies.

3.1 Tests du CNN complet

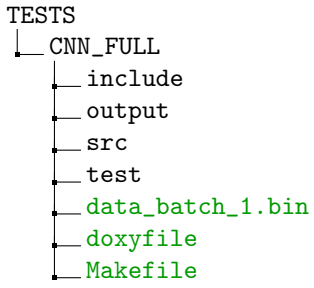


FIGURE 14 – Structure du répertoire CNN_FULL

CNN_test.exe Pour tester le fonctionnement du CNN tel qu'il est implémenté sur FPGA, on utilise les commandes :

```
make CNN_test.exe
./CNN_test.exe
```

Cela crée un fichier output.pgm dans le dossier output qui est l'affichage de sortie produit à destination de l'écran.

CNN_Batch.exe Pour valider le fonctionnement du CNN sur un batch d'image entier, on utilise les commandes :

```
make CNN_Batch.exe
./CNN_Batch.exe -vo 100
```

L'option -v permet d'afficher les résultats du CNN pour chaque image traitée dans la sortie standard. L'option -o permet d'activer l'écriture des fichiers intermédiaires dans le dossier output/ au format pgm. Enfin le nombre correspond au nombre d'image à traiter par le programme.

3.2 Tests de l'Edge detector

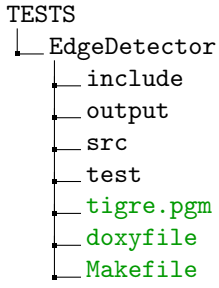


FIGURE 15 – Structure du répertoire CNN_FULL

EdgeDetector.exe Pour tester l'Edge Detector, on utilise les commandes :

```
make EdgeDetector.exe
./EdgeDetector.exe
```

Le programme prend en entrée l'image tigre.pgm et écrit la sortie dans un fichier output.pgm dans output.