PBD - HLS - Deuxième partie : ARM et IP matérielle

Groupe 1 :
LEGOUEIX Nicolas
MALLET Julien
VIGUIER Franck

24 novembre 2021

1 Modifications du code et exécution

1.1 Modifications

L'utilisation du réseau de neurones nécessite d'avoir calculé les poids des différentes couches du réseau lors de la phase d'entraînement. Ceux-ci sont sauvegardés dans des fichiers **.wei** que nous avons utilisé dans la première partie du projet pour faire fonctionner la reconnaissance de caractères sur le processeur de notre ordinateur.

La lecture des fichiers **.wei** n'est pas directement possible sur le FPGA car ce dernier ne possède pas un système de fichier permettant de faire une lecture facile et rapide des poids. Pour utiliser les poids des différentes couches du réseau nous avons créer un fichier c (et son en-tête) pour stocker toutes les valeurs des poids dans des tableaux. Il nous suffit ensuite d'inclure le fichier d'en-tête au niveau du fichier **main.c**, pour que notre réseau de neurones puisse utiliser les tableaux de poids lors du calcul de la sortie.
Ces fichiers de code et d'en-tête sont générés automatiquement par le programme que nous avons développé et qui reprend le code de la fonction *InitHostMem()* pour la lecture des fichiers **.wei** et y ajoute la sérialisation vers les deux fichiers **.c** et **.h**

La seconde manipulation que nous avons effectué à été l'ajout de la librairie **math** qui offre la fonction *tanh()*, utilisées comme fonction d'activation de notre réseau de neurones.
En effet, la commande d'inclusion de la librairie **math.h** utilisée jusque là ne fonctionne pas sous Vivado directement. Il est donc nécessaire de l'ajouter manuellement via les options de compilation de la manière suivante depuis le SDK :

- clicker droit sur le nom du projet concerné dans l'explorateur de projets sur la gauche de la fenêtre, puis sélectionner "C/C++ Build Settings"
- Dans la fenêtre qui s'ouvre, naviguer dans "ARM v7 gcc linker" puis dans "Libraries"
- Selectionner "add" et taper "m" dans la fenêtre qui s'ouvre puis valider

1.2 Exécution sur FPGA et capture de sortie

Une fois le Bitstream envoyé sur la carte Zedboard, l'exécution du programme en C est faite de la manière suivante :

- clicker droit sur le nom du projet concerné dans l'explorateur de projets sur la gauche de la fenêtre
- selectionner "Run As" puis "Launch On Hardware (System Debugger)"
- le programme est alors compilé et, en l'absence d'erreurs, envoyé sur la carte qui démarrera alors l'exécution.

Les appels à la fonction *printf()* effectuent leur sortie sur l'interface UART de la carte. Elles sont donc consultables sur un ordinateur hote à l'aide d'un cable USB et d'un logiciel capable d'effectuer une liaison série. Par exemple, Vivado fournis un terminal dans son SDK capable de cette tâche. Il ne fonctionnait cependant pas correctement sous Linux sur la machine que nous avons utilisé, nous avons donc utilisé Minicom.
Par défaut, la sortie UART est réglée a 115200 bauds et Linux lui affecte la TTY nommée ACM0. Pour s'y connecter, il suffit donc d'entrer la commande suivante dans un terminal :

```
minicom -D /dev/ttyACM0 -b 115200
```

2 Profilage matériel

2.1 Contexte

Le profilage logiciel, notamment à l'aide de la fonction *dtime()* comme lors de la première partie s'avère beaucoup plus compliqué à mettre en oeuvre une fois que le programme doit tourner sur le FPGA. Nous avons donc opté pour un profilage matériel à l'aide de la libraire **xtime_l.h**. Cette dernière fournit la fonction *XTime_GetTime()* qui permet d'obtenir directement le temps de manière très précise en interrogeant les registre de temps du FPGA. Ce temps, exprimé en nombre de cycles doit être converti en secondes à l'aide d'une variable globale **COUNTS_PER_SECONDS** qui représente la fréquence de fonctionnement du timer du processeur Cortex A9. Cette dernière est égale à la moitié de la fréquence de fonctionnement du processeur.
Il suffit alors de déclarer deux variables de type **Xtime** et de récupérer le temps actuel avant et après l'exécution de chaque calcul de couches de neurones afin de savoir précisément le temps qu'il a fallu au FPGA afin de faire ces calculs.

Il convient également de noter qu'en raison du fait que ce profilage s'effectue sur le Cortex A9 et non plus sur un Ryzen 7 4700U comme lors de la première partie, le temps pris pour une exécution du programme ne peut pas être comparé. En revanche, nous pouvons comparer le pourcentage de temps passé dans chaque fonction. Nous devrions alors retrouver des résultats similaires, et devrions pouvoir baser notre parallélisation en fonction de ces valeurs.
Enfin, la fonction *InitHostMem()* ne figure pas sur ces résultats contrairement aux résultats du premier rapport car cette fonction n'est plus utilisée. Elle représentait cependant un temps relativement négligeable.

2.2 Résultats

Voici les résultats obtenus pour une seule exécution :

Name	Time (timer ck)	Time (us)	Time (%)
TOTAL	18095410	27170.29	100.0
Layer3	9829427	14215.94	54.32
Layer4	6572460	9868.56	36.32
Layer2	1495386	2245.32	8.26
Layer5	34430	51.70	0.19
Layer1	3712	5.57	0.02

2.3 IP Leds

Note à propos de l'affichage du résultat sur les LEDs de la carte Zedboard : Pour une raison indéterminée, la LED ne s'allume qu'une fois sur deux. En effet, la première exécution affiche correctement le résultat, mais lors d'une deuxième exécution, la LED s'éteint comme attendu mais ne se rallume pas sur le bon résultat. Enfin, une troisième exécution affiche correctement le résultat et ainsi de suite...

3 Partitionnement

Les résultats obtenus après avoir effectué le profilage du réseau de neurones nous ont permis d'identifier 2 fonctions qu'il serait judicieux de faire exécuter sur la partie **pl** (programmable logic) du FPGA :

— calculateLayer3
— calculateLayer4

Ce partitionnement donnerait le synoptique suivant :

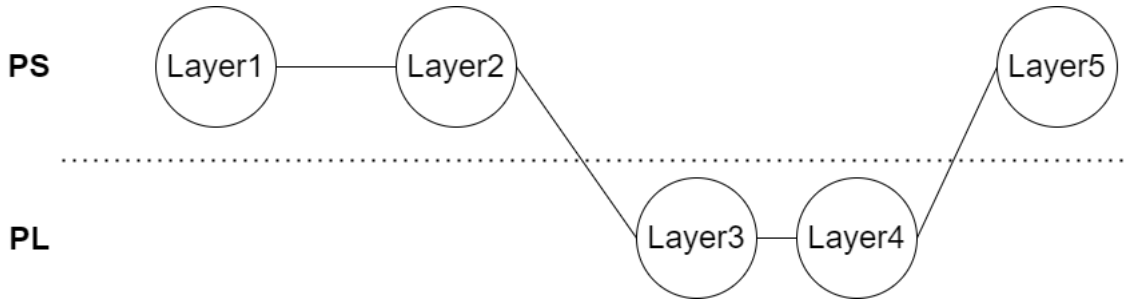

FIGURE 1 – première idée de partitionnement

Cependant, il est nécessaire de garder à l'esprit le coût élevé des communications entre les parties PS et PL. Il pourrait donc être intéressant d'éviter la transition entre les Layer4 et 5, et basculant la fonction Layer5 en PL, on obtiendrait alors le synoptique suivant :

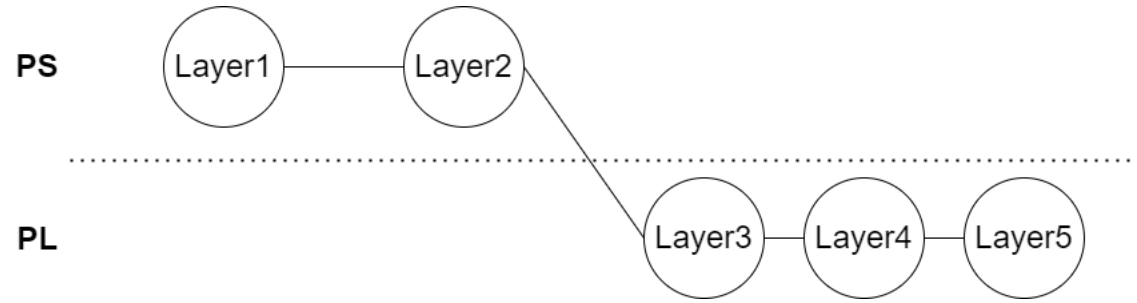

FIGURE 2 – partitionnement amélioré

3.1 Remarque

Il convient de noter qu'ici, aucune parallélisation n'est réalisée. Notre idée est ici d'accélérer les taches prenant le plus de temps en les passant en PL tout en limitant les coupures PS/PL. Nous ne pensons pas que la parallélisation ne soit possible car chaque fonction layer dépend de la sortie de la précédente, et nécessite donc que cette dernière termine afin de pouvoir démarrer.