TRAVAUX PRATIQUES - V1.0

UNIVERSITÉ DE BRETAGNE-OCCIDENTALE

M2 - LSE

Implémentation d'un réseau de neurones sur FPGA

-

De Python au FPGA

Date: 18 Janvier 2023

Auteur :
Tanguy LE PENNEC (email : etudiants@tanguylepennec.fr)

Ressources :
L'ensemble des ressources de ce TP sont disponibles à ce dépôt Github : https://github.com/CaptainKey

Vous devez **impérativement** envoyer l'ensemble de vos réponses dans un document sous format pdf à l'adresse : etudiants@tanguylepennec.fr avec vos projets Vitis HLS, Vivado et Vitis IDE dans une archive .zip.
Les documents en dehors des fichiers .pdf ne seront pas acceptés

Contents

1 L'organisation de ce TP

Le TP est réalisé pour que vous soyez au maximum indépendant et donc que vous puissiez avancer à votre rythme, que vous ayez des facilités ou des difficultés, tout le monde devrait y trouver son compte. **Cependant, il est certain que vous allez avoir des questions ! Alors, n'hésitez pas à poser des questions**.

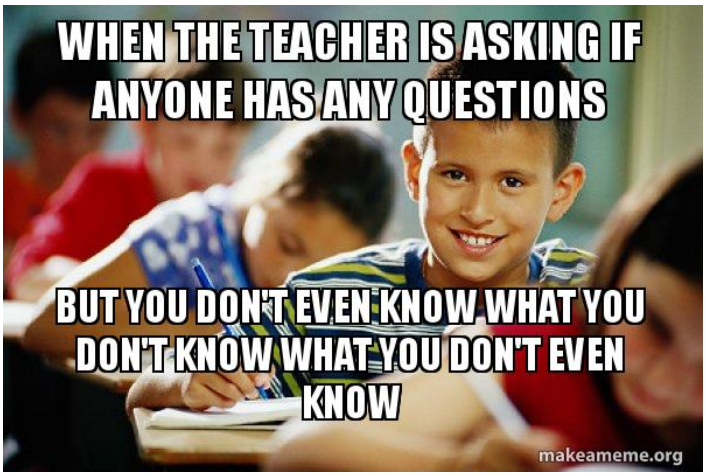

Figure 1

Enfin, le document comprend des informations, prenez le temps de les lire.

Figure 2

2 Les objectifs de ce TP

- Réaliser un flot HLS complet, de la création d'une IP sur Vivado HLS, son intégration dans un design par Vivado et enfin l'exploitation du design complet avec un ARM sur une carte d'évaluation (ZedBoard).

- Comprendre les challenges liés à l'implémentation des réseaux de neurones sur FPGA : complexité des architectures, limitations des ressources logiques, choix dans la conception.

- Comprendre les différences d'exploitation entre un CPU, un GPU, et FPGA.

- Comprendre et réaliser l'exploitation d'une architecture développer en Python et l'exploiter sur FPGA

3 Introduction

3.1 Contexte

Les réseaux de neurones sont des algorithmes formidables qui permettent de résoudre des tâches complexes. Les progrès les plus spectaculaires ont pu être constatés dans le domaine de la classification des images où ils surpassent toutes les techniques existantes depuis 2012.
Cette capacité magnifique à résoudre des problèmes a cependant des inconvénients. En effet, ces algorithmes ont besoin de machines puissantes pour être exécutés. La complexité des tâches qu'ils résolvent augmente leur complexité avec les années et donc la demande en unités de calcul spécialisées capable de les exécuter en un temps record.
L'industrie du calcul haute performance est partagée entre deux idéologies, la première est de produire des puces spécifiques optimisées pour le calcul, c'est le cas de NVIDIA qui adapte ses GPU spécifiquement pour des tâches de machine learning ou encore d'Apple M1, M2 qui ont à l'intérieur de leurs puces des ressources dédiées également aux algorithmes de machine learning. L'autre pari est de rendre disponible un hardware qui peut évoluer avec le développement des nouveaux algorithmes ou architectures. C'est le pari d'AMD via le rachat de Xilinx en 2022 pour 50 milliards de dollars. C'est cette dernière vision qu'illustre ce TP.
Les FPGA sont également utilisés dans de nombreux autres domaines en dehors de l'intelligence artificielle : la cybersécurité (traitement des attaques DDOS), les télécommunications (5G), les cryptomonnaies (minage : POW / validateur : POS), etc.

3.2 Les outils du deep learning

L'industrie du machine learning et plus spécifiquement de l'apprentissage profond (*deep learning*) a donné naissance à de multiples outils pour permettre de développer rapidement de nouvelles architectures et ainsi déployer le deep learning dans des domaines où il pourrait apporter une valeur ajoutée (exemple : médecine, finance, etc.). La majorité des outils ou frameworks sont gratuits : Pytorch[1], Tensorflow[2], Keras[3] etc. Cela pour favoriser l'adoption, car derrière chaque framework se cache une grande entreprise ou un laboratoire qui souhaite voir ses outils les plus utilisés : Facebook pour Pytorch ou encore Google pour Tensorflow.
Ces frameworks encapsulent la théorie, la logique et les mathématiques pour l'inférence et l'entrainement des réseaux de neurones. Le développement de nouvelles architectures consiste donc en la réutilisation et l'organisation de couches paramétrables, mais optimisées dans leurs exécutions. Les frameworks de ML sont en Python, cependant, ce dernier n'est qu'une interface. En effet, bien que l'utilisateur réalise ses instructions en Python, celles-ci font appel à du C ou C++ précompilés afin d'obtenir

[1] https://pytorch.org/
[2] https://www.tensorflow.org/
[3] https://keras.io/

les meilleures performances. Le C et le C++ sont des langages compilés, proches de la machine contrairement à Python qui est un langage interprété. Les Frameworks disposent aussi de la possibilité d'interfacer le code sur GPU à partir d'une simple ligne de commande, cela permet facilement d'obtenir une accélération sans pour autant changer son programme.

3.3 La synthèse de haut niveau

Les GPU ont des bibliothèques éprouvées pour les exploiter (exemple : NVIDIA CUDA) ce qui a permis d'intégrer leur exploitation dans les bibliothèques de machine learning. Cela n'est pas le cas des FPGA. En effet, le développement sur ce type de matériel nécessitait l'utilisation d'un langage de description industriel comme le VHDL ou le Verilog, on pourrait le comparer à l'utilisation de l'assembleur pour réaliser un programme informatique. L'industrie a donc évolué pour attirer de nouveaux utilisateurs et favoriser le développement de communautés, de bibliothèques. L'aboutissement de cette évolution est la mise en place de la méthode High Level Synthesis (HLS). Cette méthode permet de transpiler un code C++ en VHDL ou Verilog selon les choix de l'utilisateur. Cela a permis un changement d'échelle, de paradigme comparable à l'utilisation du C plutôt que de l'assembleur. Ainsi, on peut spécifier la logique que l'on souhaite implémenter et un outil va en extraire l'équivalent en langage de description industriel.
L'inconvénient principal de la méthode HLS est que l'utilisateur perd le contrôle sur le code généré. Ainsi, on perd en performance, mais d'un autre côté, on gagne un temps considérable dans le développement du design. Cependant, il peut imposer des contraintes sur la génération du design grâce à des directives préconfigurées pour spécifier le déroulement de boucles, des interfaces, etc.

3.4 Les outils de AMD-Xilinx

Il existe différents fabricants de FPGA : AMD-Xilinx, Altera, Lattice, etc. Pour exploiter ces derniers, ils proposent leurs outils (souvent propriétaires). Dans notre cas, nous allons utiliser les outils de AMD-Xilinx. AMD-Xilinx est le leader des FPGA. Il existe 3 grands outils qui ont évolué avec les versions de la Vivado Design Suite :

- Vivado HLS ou Vitis HLS : Dans cet outil, on utilise la méthode HLS pour créer des IP qui sont le résultat de la transcription d'un code C++ en VHDL ou Verilog

- Vivado : Dans cet outil, on réalise un schéma destiné à être implémenté sur un FPGA. On peut ainsi intégrer des IP issues de l'outil HLS.

- Vitis ou Vivado SDK : dans cet outil, on réalise l'exploitation de la partie CPU. (Depuis l'intégration de Vitis dans la suite des outils, celui-ci s'est également tourné vers l'exploitation plus spécifique des cartes Alveo, mais de l'IA en général)

3.5 Les architectures des puces AMD-Xilinx

Les puces qui sont créées par l'entreprise AMD-Xilinx sont conçues pour réaliser une coopération CPU / FPGA. Dans ce contexte on fait référence au CPU en tant que partie Programmable Logic (PL) et au FPGA en tant que partie Programmable Logic (PL). Dans ce TP, nous allons réaliser nos implémentations pour une puce Zynq-7000, son architecture est visible à la figure3

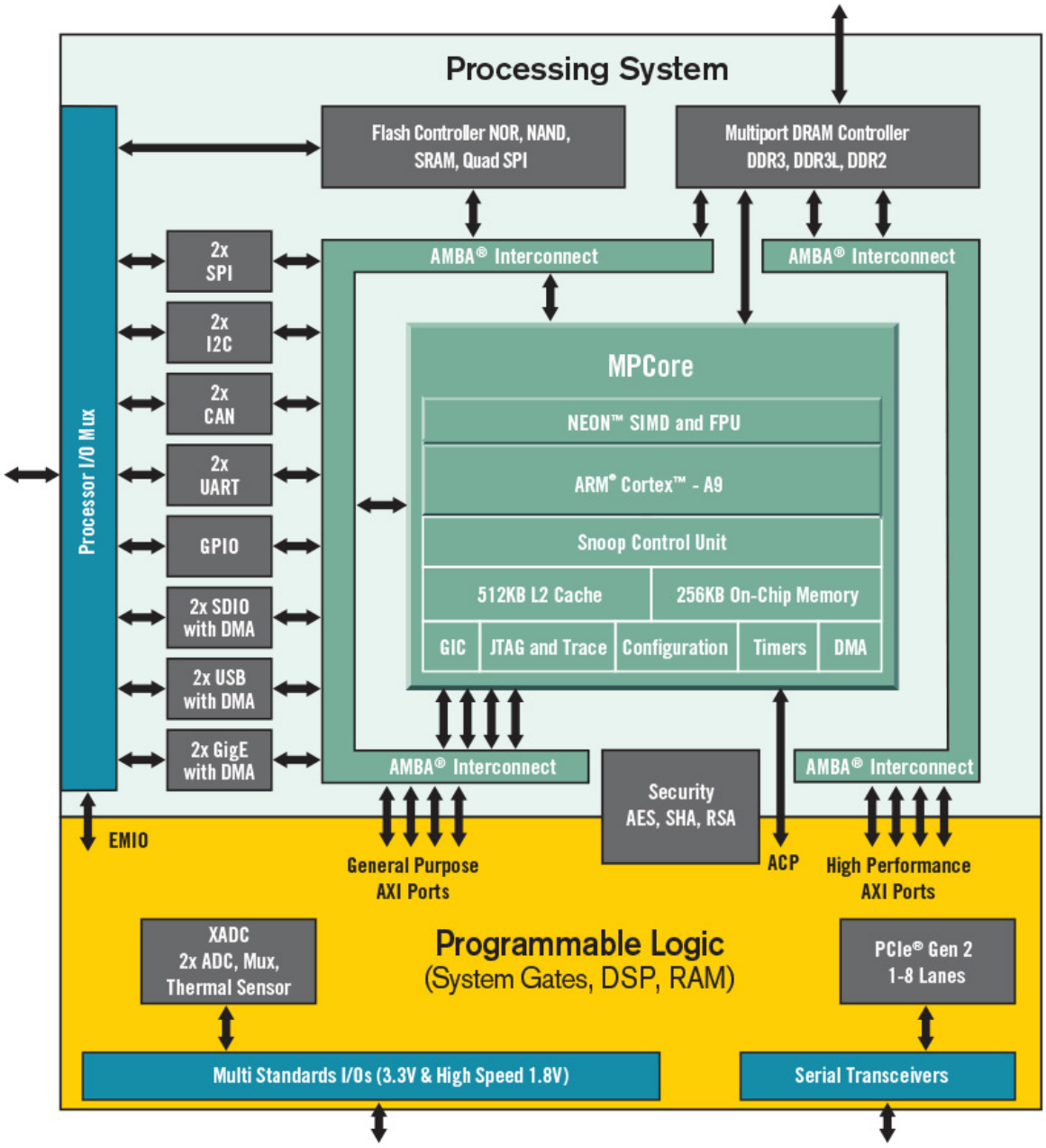

Figure 3: Architecture Zynq-7000

4 Python : Un pour les dominer tous

Il existe de nombreux langages de programmation, tous avec leurs spécificités qui les rendent adaptés à certaines tâches plutôt qu'à d'autres : Go pour les applications DevOps(exemple : Docker[4], Kubernetes[5]), C/C++ ou encore Rust pour les applications embarquées à faible mémoire, JavaScript pour les applications web, etc.

Python s'est forgé une communauté solide dans le traitement et la visualisation de données. Sa syntaxe est très simple à apprendre pour les néophytes, car faiblement typée. Il est alors aisé de développer des programmes en peu de temps. Outre sa facilité de développement, il existe de nombreuses bibliothèques éprouvées pour la visualisation et le traitement de données. Cela donne un avantage unique à Python dans ce domaine. Vous pouvez consulter la figure4 qui vous illustre différentes bibliothèques dans l'écosystème Python.

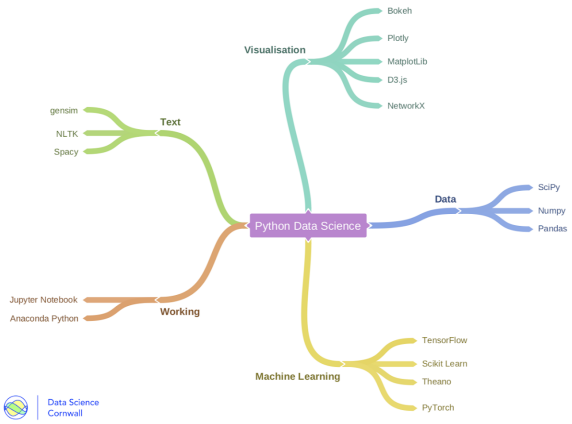

Figure 4: Les libraires en Python

[4]https://www.docker.com/
[5]https://kubernetes.io/fr/

4.1 Visualisation d'une architecture d'un réseau de neurones

Il peut être difficile de se représenter l'architecture des réseaux de neurones. En effet, la plupart ont des dizaines ou des centaines de couches, il est alors difficile de se représenter l'ensemble de l'architecture et d'y associer une complexité.
Il existe différents outils qui permettent de détailler une architecture de manière lisible pour un humain. Voici quelques-uns d'entre eux.

4.1.1 Netron

Figure 5: Netron

Netron est un visualiseur pour modèles de réseaux de neurones. Il prend en entrée un fichier de modèle (comme un fichier .onnx ou .h5) et affiche une représentation graphique de la structure du réseau, y compris les couches, les connexions entre les couches, et les poids des paramètres. Il permet également d'explorer les données du modèle et de les visualiser de différentes manières. Il est développé par Lutz Roeder et est accessible en tant que site web à l'adresse suivante : **https://netron.app/**.
Le code source de *Netron* est également disponible sur le GitHub de Lutz Roeder : **https://github.com/lutzroeder/netron**

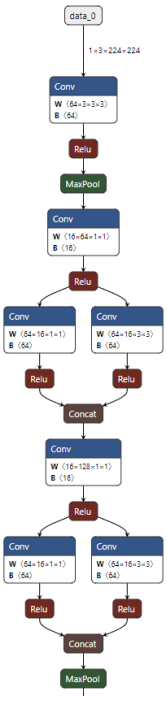

Figure 6: Exemple de visualisation avec Netron

4.1.2 torchsummary

torchsummary est un package python qui permet d'afficher de manière claire et concise les informations sur l'architecture d'un réseau de neurones défini avec PyTorch. Il peut afficher les informations sur les couches du réseau, le nombre de paramètres, le nombre de flops, les entrées et sorties de chaque couche, etc. Il est donc très utile pour comprendre rapidement les caractéristiques d'un réseau de neurones et pour faciliter le débogage et l'optimisation des architectures.
Pour en savoir plus : **https://pypi.org/project/torch-summary/**

4.2 Une architecture parmi d'autres

Il existe de nombreuses architectures de réseaux de neurones[6]. Dans ce TP, nous allons nous concentrer sur les réseaux de neurones de convolution qui sont principalement destinés au traitement de données organisées sous forme de grille (exemple : image, son).
Ici, nous allons utiliser le réseau de neurones LeNet-5 qui est certes plus petit que ses frères et sœurs, mais qui nous permettra de parcourir les différentes problématiques liées à une implémentation FPGA.

Question : A l'aide des fichiers lenet_architecture.py et resnet_architecture.py, des outils cités précédemment et Google Colab, quelles différences constatez-vous entre ces architectures ? Que pouvez-vous en conclure dans le contexte d'une implémentation FPGA ?

Question : Détailler en quelques phrases et avec un schéma l'architecture LeNet (nombre de couches, type de couches, rôle, nombre de paramètres, dimension des entrées et des sorties)

4.3 Entrainement d'une architecture avec Python

Question : Quelles sont les différences entre les fichiers lenet5_cpu.py et lenet5_gpu.py ? Qu'en concluez-vous ?

Question : A l'aide de Google Colab, exécutez les fichiers lenet5_cpu.py et lenet5_gpu.py. Que pouvez-vous constater au niveau du temps d'exécution ? Pourquoi ? Selon vous, quel serait le résultat avec des architectures plus importantes ?

Question :A l'aide de Google Colab, exécutez les fichiers resnet_cpu.py et resnet_gpu.py. Que pouvez-vous dire sur l'architecture du réseau de neurones utilisé ainsi que sur les temps d'exécution ? Quels sont les différences avec les fichiers lenet5_cpu.py et lenet5_gpu.py ? Pourquoi ?

[6] https://modelzoo.co/

4.4 Une architecture pour de multiples tâches

Question : Quelles sont les différences observables entre les fichiers lenet5_mnist.py et lenet5_cpu.py ? Que pouvez-vous en déduire ?

4.5 Récupérer les paramètres

Question : Après avoir réalisé l'entrainement du réseau LeNet à l'aide du fichier lenet_cifar.py, réalisé l'extraction des paramètres du réseau dans un fichier. **Si vous êtes bloqué à cette étape, passer à la partie suivante**

5 Vivado HLS

5.1 Avant propos sur Vivado HLSs

Vivado HLS va nous permettre de créer un ou plusieurs projets dans lesquels nous allons écrire la logique que l'on souhaite intégrer à notre IP[7]. À partir de notre logique C++ Vivado HLS va en extraire une description logique : VHDL ou Verilog suivant le choix de l'utilisateur. Grâce à l'utilisation du C++ il est relativement facile de produire du code complexe.

ATTENTION : Bien que l'on puisse utiliser du C++, certaines fonctionnalités ne sont pas disponibles comme l'allocation dynamique (malloc). En effet, Vivado HLS a besoin de connaitre toutes les adresses mémoires afin de pouvoir y associés les ressources logiques qui sont fixées à la fin de l'exportation de notre IP. Pouvoir réaliser des allocations dynamiques à l'intérieur de notre design FPGA induirait que l'on peut créer des ressources à partir de rien. Il faut donc contraire de manière certaine l'ensemble des ressources dont on a besoin.

Dans Vivado HLS nous avons 4 grandes phases :

- La phase de C-Simulation : Dans cette phase on vérifie le fonctionnement fonctionnel de notre application. Est-ce qu'il fonctionne correctement ? Est-ce que pour une entrée j'obtiens la bonne sortie ?

- La phase de Synthèse : Dans cette phase, Vivado HLS va traduire proposer une estimation des ressources logiques (LUT, mémoires : RAM, ROM, FIFO, etc). Elle donne également une indication sur le temps d'inférence.

- La phase Co-Simulation RTL : Dans cette phase on vérifie au niveau logique si notre programme fonctionne correctement. Il est obligatoire d'avoir réalisé la synthèse avant de réaliser cette phase, car Vivado HLS prend le design de synthèse comme référence et simule son fonctionnement.[8]

- La phase de création de l'IP et son exportation :

[7] Le résultat final de Vivado HLS

[8] Cette tâche est souvent la plus longue

5.2 Réalisation de l'IP associé à LeNet

Dans cette partie vous utiliserez les fichiers du dossier sources

Question : Expliquer le rôle de chaque fichier dans le dossier **source** ?

Question : Réaliser un projet Vitis HLS à l'aide des fichiers disponibles?

Question : Expliquer à quoi correspond les fonctions **layer0**,**layer1**,**layer2**,**layer3**,**layer4**,**layer5**,**layer6**. Faites le lien avec l'architecture LeNet

Question : Quelles sont les précisions utilisés pour les types ? Expliquer le rôle de chacun.

Question : Qu'est-ce qu'un bus AXI ?

Question : Qu'est-ce qu'un bus S_AXI_LITE ?

Question : A l'aide de vos connaissances et en respectant le design, ajouter les paramètres sauvegardé précedemment

Question : Ajouter des directives pour spécifier que les entrées de votre IP sont des bus AXI

Question : Ajouter une directives pour contrôler l'IP à l'aide d'un S_AXI_LITE

Question : Que permet la directive **Dataflow** ? Ajouter la à votre IP

Question : Réaliser la synthèse de l'IP. Que pouvez-vous dire sur la consommation de ressources logiques ?

Question : Créer un testbench pour vérifier le bon fonctionnement de l'IP

Question : Réaliser la simultion C. Comparer le résultat de simulation avec celui obtenu par Python. Que constatez vous au niveau des résultats ? Pourquoi ?

Question : Réaliser la co-simulation avec waveform. Décrire les comportements sur le waveform des bus de l'IP : INPUT et OUTPUT

Question : Réaliser l'exportation de l'IP. Comparer les ressources finales utilisés avec celles estimés lors de la synthèse. Que pouvez-vous en déduire ?

6 Vivado

6.1 Avant-propos sur Vivado

6.1.1 Création d'un design

Pour créer un design, il faut créer un **Block Design** grâce à l'onglet **Create Block Design**. Vous pouvez ensuite réaliser la synthèse (onglet : **Run Synthesis**), l'implémentation (onglet : **run implementation**) et enfin générer le bitstream[9] (onglet : **Generate Bitsream**)

6.1.2 Ajouter une IP réalisée grâce à Vivado HLS / Vitis HLS

Pour permettre à Vivado de visualiser des IP préalablement créées par Vivado HLs il est nécessaire d'ajouter le répertoire qui les contient via les onglets : **Project Manager** → **Settings** → **IP** → **Repository**. Cliquer sur **+** pour ajouter le chemin de l'IP (voir figure 7).

6.2 Exporter le design pour l'exploiter avec Vivado SDK ou Vitis IDE

Une fois que vous avez généré le Bitstream, vous devez exporter votre design afin de pouvoir l'exploiter depuis Vitis IDE. Pour ce faire aller dans le menu **File** → **Export** → . Lors du choix du type de plateforme, laisser **Fixed** puis cliquer sur **next**. Ensuite sélectionner le choix **include bitsteam** pour inclure le bitstream à l'exportation du design puis cliquer sur **next**. Enfin, sélectionner un dossier d'exportation et terminer l'exportation du design.
Vérifier à la suite de l'exportation qu'un fichier .xsa a bien été créé.

[9] Le bitstream est un fichier binaire avec une extension .bit. Il comprend le schéma logique destiné à être implémenté sur la partie PL.

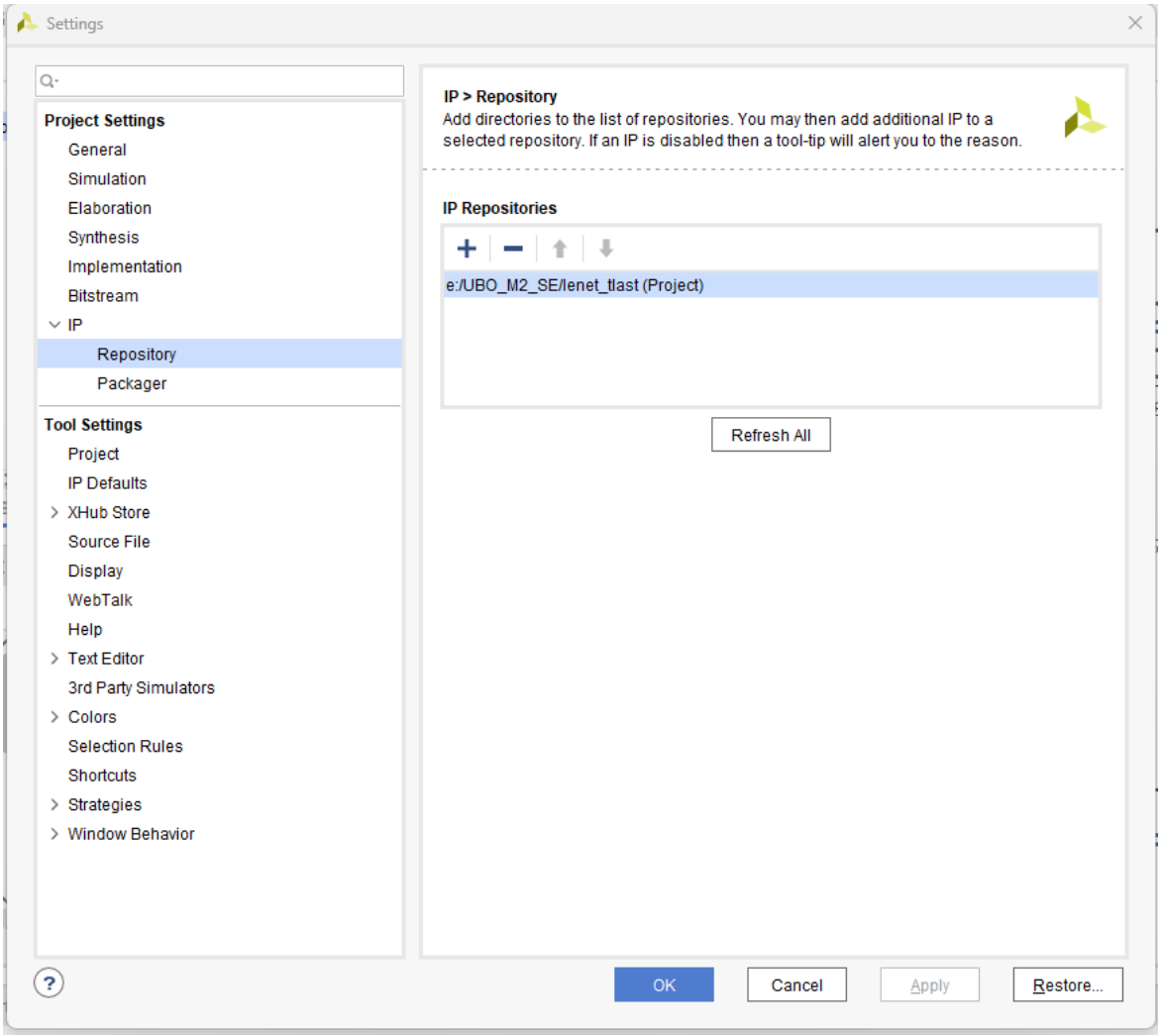

Figure 7: Vivado - Fenêtre des repos

6.3 Réalisation du design global

Question : Qu'est-ce qu'une DMA ?

Question : A l'aide de vos connaissances réaliser le design de la figure8

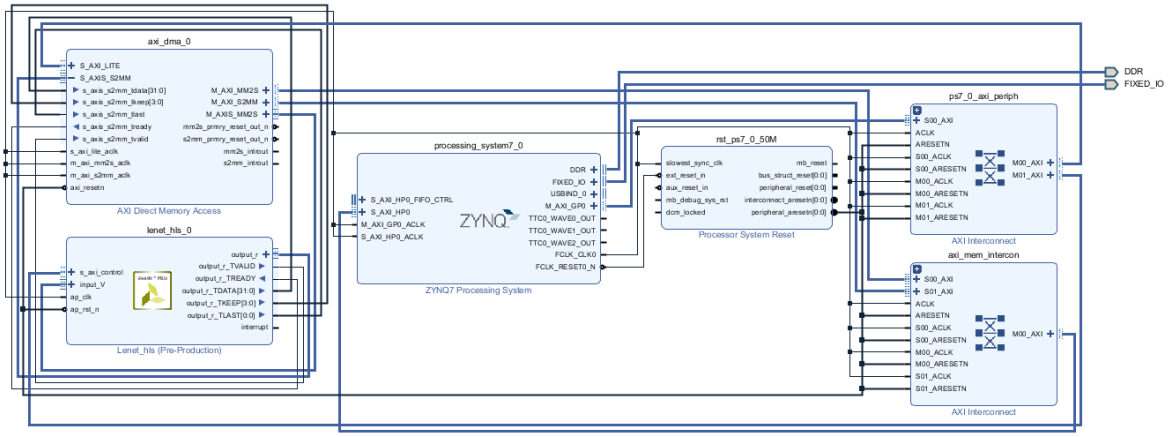

Figure 8: LeNet - Design Vivado

Question : Expliquer l'utilité de l'ensemble des ports pour les IPs suivantes : axi_dma_0, lenet_hls_0 et processing_system7_0

Question : Quelles sont les avantages que procure Vivado HLS dans votre conception avec Vivado ?

Question : Réaliser la synthèse, l'implémentation, la génération du Bitstream

Question : Consulter les métriques concernant la consommation des ressources logiques. Comparer avec ceux obtenus avec Vitis HLS. Qu'en concluez-vous ?

7 Vitis ou Vivado SDK

7.1 Avant-propos sur Vitis et/ou Vivado SDK

Vitis[10] permet d'exploiter la partie CPU des puces Xilinx[11]. Il est cependant à noter que Vitis permet de réaliser un flot de conception différent que celui illustré dans ce TP pour certaines cartes : https://www.xilinx.com/products/boards-and-kits/alveo.html. Les Alveos sont des cartes FPGA orientées pour les centres de données (datacenter)

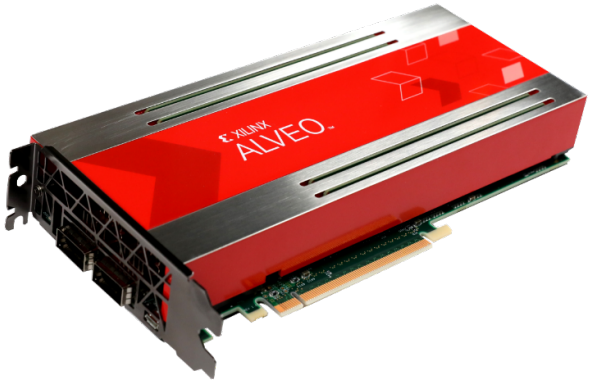

Figure 9: Carte FPGA Xilinx - Alveo

7.1.1 Créer une plateforme

Dans le cadre de ce TP il est nécessaire de spécifier l'hardware que nous souhaitons exploiter. Après avoir réalisé l'exportation du design dans Vivado, un fichier .xsa a été créée. C'est ce fichier qui va être utilisé par Vitis IDE pour prendre connaissance du design.
Pour créer une nouvelle plateforme, aller dans le menu **File** → **New** → **Plateform Project..** puis :

1. Définir un nom pour la plateforme
2. Sélectionner le .xsa précédemment créé avec le bouton **Browse**.
3. Cliquer sur **Finish**

À la suite du clique sur le bouton finish un projet apparait dans l'arborescence de Vitis IDE. Cliquer sur le fichier **plateform.spr**
Après avoir sélectionné **Board Support Package** cliquer sur le bouton **Modify BSP Settings** et cliquer sur l'option **xilffs**

[10] Anciennement VivadoSDK
[11] souvent un ARM

7.1.2 Créer un programme pour l'ARM

Pour créer un programme à destination de la partie PS aller dans le menu **File** → **New** → **Application Project..** puis :

1. Sélectionner la plateforme associée au programme.
2. Définissez un nom pour le projet
3. créer le projet à partir du template **Hello World**

7.2 Exploitation de la partie PS

Question : Expliquer ce qu'est le BSP

Question : A quoi sert la librarie xilffs ?

Question : Dans le projet de la plateforme associé à votre design que contient le répertoire **export** ? Expliquer l'utilité des différents fichiers.

Question : Réaliser un programme pour contrôler l'IP associé à LeNet

Question : Réaliser un programme pour envoyer des données à partir de la DMA à l'IP LeNet

Question : Combien de données et données

8 Implémentation sur Zynq-7000 - Zedboard

Avant de mettre sous tension votre carte Zedboard, brancher les câbles micro-USB sur les connecteurs **PROG** et **JTAG** sur votre carte. Branchez votre carte sur une prise secteur et mettez la sous-tension.
À l'allumage une diode verte devrait être visible.
Réaliser la programmation de la partie PL pour vérifier que la carte est bien détecté à l'aide du menu **Xilinx** → **Program FPGA**.
Si aucune erreur n'a été détectée, lancez votre programme d'application associé.

Question : Que constater vous sur la sortie FPGA?

Question : Comparer les résultats de la carte Zeboard avec les résultats HLS et Python

9 Conclusion & Ouverture

Question : Que pouvez-vous dire du flot de dévéloppement que nous avons utilisés ici ? Quels sont les avantages, les inconvénients ?

Question : Avec vos nouvelles connaissances, proposer des améliorations au design lenet réalisé ici ?

Question : Dans ce TP nous avons passer une unique image à des fins de démonstrations, modifier le code de l'ARM pour classifier l'ensemble des données de test présente dans un fichier binaire sur la carte SD.

Question : Réaliser une implémentation du réseau LeNet-5 ayant pour but de classifier les images de la base MNIST