

Compte Rendu

Projet Systèmes Embarqués

Guilherme VENTAPANE RODRIGUES
Jarod LECOEUVRE

Sommaire

Introduction.....	3
Méthodologie.....	4
Environnement de Développement et Plateforme Cible.....	4
Flot de Conception.....	5
Partie Logicielle : Du Modèle Keras au Code C.....	6
Modélisation et Apprentissage (Keras).....	6
Développement du Code C (Floating-Point).....	6
Validation Croisée.....	6
Description Fonctionnelle des Couches du Réseau.....	7
Conception Matérielle et Synthèse de Haut Niveau (HLS).....	7
Adaptation du code et Gestion Mémoire.....	8
Gestion des Poids et Autonomie.....	8
Arithmétique à Virgule Fixe.....	8
Optimisation et Exploration de l'Espace de Conception (DSE).....	8
Intégration Système et Métriques d'Évaluation.....	8
Validation.....	10
Validation Fonctionnelle du Système.....	10
Analyse des Ressources (Synthèse HLS).....	10
Performances Comparatives.....	10
Analyse et Conclusion.....	11
Conclusion.....	12
Bilan des réalisations.....	12
Analyse des limitations.....	12
Retour d'expérience.....	12
Annexes.....	13
Synthétisation sur Vivado.....	14

Introduction

Ce projet s'inscrit dans le cadre de la dernière année de spécialisation en Ingénierie des Systèmes Embarqués. Il explore le domaine en plein essor de l'IA embarquée, cherchant à porter des algorithmes d'intelligence artificielle complexes sur des architectures matérielles contraintes.

L'objectif principal est le développement complet et l'accélération matérielle d'un Réseau de Neurones Convolutif (CNN), spécifiquement l'architecture LeNet, dédiée à la reconnaissance de caractères manuscrits. La plateforme cible est une ZedBoard équipée d'un SoC Zynq-7000, combinant un processeur ARM Cortex-A9 et un FPGA Xilinx.

L'implémentation de réseaux de neurones profonds sur des systèmes embarqués soulève plusieurs défis majeurs que ce projet vise à résoudre :

- **Le fossé entre Logiciel et Matériel** : Les frameworks d'IA traditionnels (comme Keras/TensorFlow) fonctionnent avec des nombres à virgule flottante et des allocations dynamiques de mémoire, ce qui est peu adapté aux architectures matérielles embarquées. L'implémentation matérielle nécessite souvent des unités de calcul sur mesure (RTL) en VHDL ou Verilog, dont le développement manuel est long et complexe.
- **La nécessité de la Synthèse de Haut Niveau (HLS)** : Pour combler ce fossé, ce projet utilise la HLS (High Level Synthesis). Cette technologie permet de convertir un code de haut niveau (C/C++) en une description matérielle (RTL). Cependant, tout code C n'est pas directement synthétisable. Le défi est donc de produire un code "HLS compliant" : statique, sans allocation dynamique, et utilisant des boucles déterministes.
- **Contraintes de calcul et conversion en virgule fixe** : Les opérations en virgule flottante sont coûteuses en ressources et en énergie sur FPGA. Une étape critique du projet est donc la conversion des poids et des calculs du modèle en **arithmétique à virgule fixe** (16, 32 ou 64 bits) sans perdre la précision de l'inférence.

Le projet vise à démontrer la faisabilité et l'efficacité de cette approche à travers plusieurs jalons :

1. **Développement logiciel** : Création et entraînement du modèle LeNet (Keras) et écriture des fonctions C (conv, pool, fc) compatibles HLS.
2. **Optimisation matérielle (DSE)** : Exploration de l'espace de conception (Design Space Exploration) pour maximiser l'utilisation des ressources FPGA (parallélisme, pipelining, gestion mémoire) via l'outil Vivado HLS.
3. **Intégration Système** : Implémentation complète sur la ZedBoard via l'outil SDSoc pour créer un système fonctionnel sous Linux.

L'objectif final est d'atteindre une accélération matérielle avec un **gain de vitesse d'au moins 4x** et une **amélioration de l'efficacité énergétique d'un facteur 100** par rapport à une exécution purement logicielle sur le processeur ARM.

Méthodologie

Cette section décrit l'environnement matériel et logiciel utilisé, ainsi que le flot de conception adopté pour porter le réseau de neurones LeNet sur FPGA via la Synthèse de Haut Niveau (HLS).

Environnement de Développement et Plateforme Cible

Le projet s'appuie sur la plateforme de développement **ZedBoard**, construite autour du System-on-Chip (SoC) **Zynq-7000** de Xilinx. Cette architecture hétérogène est au cœur de notre stratégie d'accélération : elle intègre sur une même puce un système de traitement (PS) dual-core **ARM Cortex-A9** et une logique programmable (PL) issue de la technologie FPGA. Cette dualité permet une approche de co-design matériel/logiciel, où le processeur, fonctionnant sous un environnement Linux embarqué, gère l'orchestration générale, tandis que le tissu FPGA est dévolu à l'exécution massivement parallèle des couches du réseau de neurones.

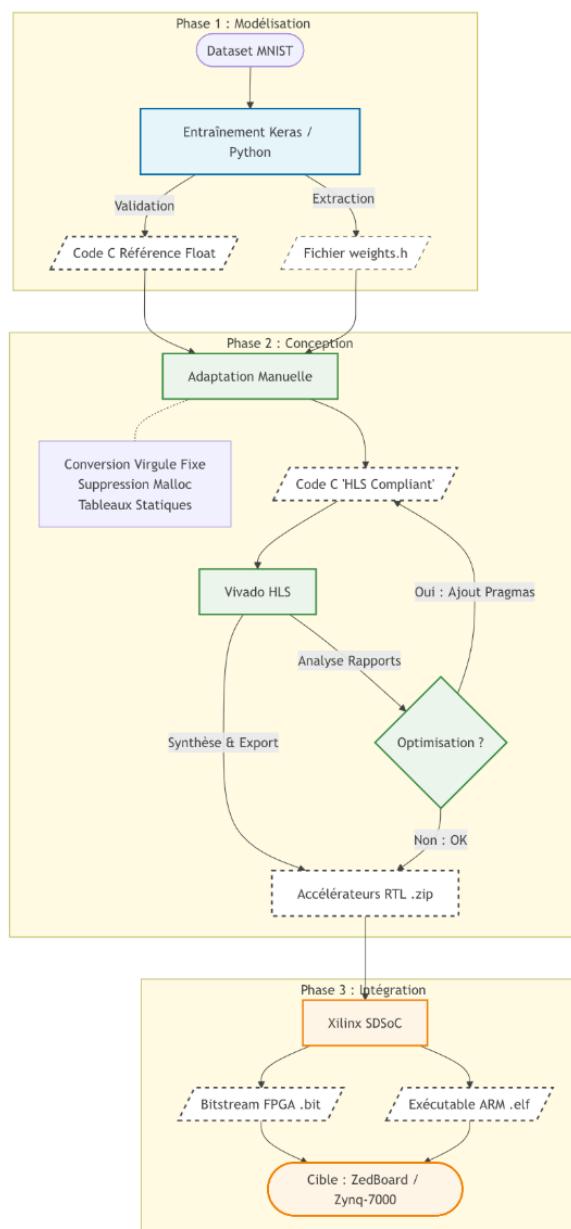
L'environnement de développement, déployé sur une station de travail **Linux**, s'articule autour d'une chaîne d'outils Xilinx et de frameworks d'IA standards :

- **Keras / TensorFlow** : Utilisés pour la phase initiale de modélisation du CNN LeNet, l'entraînement sur le dataset MNIST et l'extraction des poids synaptiques.
- **Vivado HLS** : Outil central de la synthèse de haut niveau, il permet de convertir les spécifications algorithmiques C/C++ en blocs de propriété intellectuelle (IP) matériels (RTL en VHDL/Verilog).
- **SDSoC** : Environnement de développement système utilisé pour l'intégration finale. Il automatise la génération de l'exécutable complet pour la ZedBoard, incluant le bitstream pour le FPGA et les pilotes logiciels nécessaires à la communication entre le processeur et les accélérateurs.

Flot de Conception

Le diagramme illustre les trois phases successives de notre méthodologie :

- Phase Logicielle (Bleu)** : L'entraînement du modèle LeNet sous Keras permet de figer les poids et de valider l'algorithme via un code C de référence en virgule flottante.
- Phase HLS (Vert)** : Le code est converti manuellement en virgule fixe et adapté aux contraintes matérielles (suppression de l'allocation dynamique). L'outil Vivado HLS réalise ensuite la synthèse en RTL. Une boucle d'itération (DSE) permet d'insérer des directives (*pragmas*) pour optimiser le parallélisme jusqu'à satisfaction des contraintes de performance.
- Phase Système (Orange)** : L'outil SDSoc assemble les accélérateurs matériels et le code hôte pour générer le système complet exécutable sur le SoC Zynq-7000 de la ZedBoard.



Partie Logicielle : Du Modèle Keras au Code C

Cette phase constitue le socle algorithmique du projet. Elle vise à traduire le modèle théorique de haut niveau (Python) en une implémentation logicielle bas niveau (C) fonctionnellement équivalente, qui servira de Golden Reference pour la suite du développement.

Modélisation et Apprentissage (Keras)

Le point de départ est le script `lenet_keras.py`, décrivant une architecture LeNet modifiée pour être plus adaptée à l'accélération matérielle. Nous avons utilisé ce script avec le framework TensorFlow/Keras pour entraîner le réseau sur la base de données MNIST. Une fois la précision (*accuracy*) jugée satisfaisante, les poids synaptiques et les biais du réseau ont été sauvegardés dans un fichier au format standard HDF5 (.h5). Contrairement à la future version embarquée, cette première étape de validation logicielle conserve la flexibilité de lire ce fichier de poids externe dynamiquement via la librairie HDF5.

Développement du Code C (Floating-Point)

L'objectif suivant était de reproduire l'inférence du réseau en langage C standard. Nous sommes partis d'un squelette de projet fourni dans le répertoire `FLOAT` , contenant le programme principal mais dont les fonctions de traitement (couches) n'étaient disponibles que sous forme d'objets binaires. Notre travail a consisté à réécrire intégralement le code source de ces fonctions (`conv1`, `pool1`, `conv2`, `pool2`, `fc1`, `fc2`). Cette version utilise l'arithmétique en virgule flottante et charge le fichier `.h5` généré précédemment pour valider la correction de nos algorithmes par rapport au modèle Python

Validation Croisée

Cette version logicielle utilise l'arithmétique en virgule flottante (float), identique à celle utilisée par Keras lors de l'entraînement. Elle nous a permis de valider la correction de nos algorithmes de convolution et de pooling en comparant les prédictions du programme C avec celles du modèle Python. Cette étape est cruciale : elle garantit que toute dégradation de performance observée ultérieurement sera uniquement due à la quantification (passage en virgule fixe) et non à une erreur de logique.

```
Reading weights  
Opening labels file  
Processing mnist/t10k-images-idx3-ubyte[09999].pgm  
  
Softmax output:  
0.00% 0.00% 0.00% 0.00% 0.00% 0.00% 100.00% 0.00% 0.00% 0.00%  
  
Predicted: 6      Actual: 6  
TOTAL PROCESSING TIME (gettimeofday): 23.000000 s  
  
Errors : 216 / 10000  
Success rate = 97.839996%
```

Description Fonctionnelle des Couches du Réseau

Le modèle LeNet utilisé pour ce projet est un réseau de neurones convolutif (CNN) structuré en une série de couches transformant l'image d'entrée en une prédiction de classe (chiffre 0 à 9). Chaque couche correspond à une fonction C spécifique à développer et accélérer :

- **Couche conv1 (Convolution 1) :**
Cette première couche reçoit l'image brute (taille 28x28 pixels) en entrée. Elle applique un ensemble de 6 filtres (kernels) pour extraire les caractéristiques visuelles élémentaires (contours, lignes). Cela génère 6 cartes de caractéristiques (feature maps) en sortie.
- **Couche pool1 (Pooling 1) :**
Cette couche de sous-échantillonnage réduit la dimension spatiale des cartes générées par conv1 pour diminuer la quantité de calculs et éviter le surapprentissage. Elle produit 6 cartes de taille réduite (14x14).
- **Couche conv2 (Convolution 2) :**
Seconde étape d'extraction de caractéristiques, elle prend les résultats de pool1 et applique de nouveaux filtres plus complexes combinant les informations précédentes. Elle génère 16 cartes de caractéristiques d'une taille généralement réduite par l'absence de padding (10x10).
- **Couche pool2 (Pooling 2) :**
Seconde couche de sous-échantillonnage opérant sur les résultats de conv2. Elle réduit encore la dimension spatiale pour produire 16 cartes de taille 5x5, préparant les données pour la partie classification.
- **Couches fc1 et fc2 (Fully Connected / Dense) :**
Ces couches finales réalisent la classification proprement dite. Les données 2D issues de pool2 sont "aplatis" en un vecteur 1D.
 - **fc1** : Connecte les entrées à une couche dense de neurones (généralement 120 neurones).
 - **fc2** : Connecte la couche précédente à la couche de sortie composée de 10 neurones, correspondant aux 10 chiffres possibles (0-9), fournissant le score de probabilité pour chaque classe.

Ces fonctions (`conv1`, `pool1`, `conv2`, `pool2`, `fc1`, `fc2`) constituent les blocs fondamentaux qui doivent être convertis en code compatible HLS, passés en virgule fixe, puis synthétisés en accélérateurs matériels.

Conception Matérielle et Synthèse de Haut Niveau (HLS)

L'étape centrale du projet consiste à transformer les fonctions logicielles C (couches de convolution, pooling, fully connected) en accélérateurs matériels (RTL) via l'outil **Vivado HLS**. Cette transition impose une refonte stricte du code pour le rendre « HLS compliant »

Adaptation du code et Gestion Mémoire

Le code C standard n'est pas directement synthétisable. Nous avons dû supprimer toutes les constructions logicielles abstraites, notamment l'allocation dynamique de mémoire (malloc), au profit de tableaux de taille statique mappés directement sur les ressources mémoire du FPGA. De même, pour permettre au synthétiseur d'optimiser le parallélisme, toutes les boucles ont été rendues déterministes, avec des bornes d'itération fixes et connues à la compilation.

Gestion des Poids et Autonomie

Contrairement à la version logicielle sous Python/Keras, le système embarqué ne peut supporter la librairie HDF5 pour la lecture des poids. Nous avons donc extrait les poids appris lors de l'entraînement pour les intégrer directement dans le code source via un fichier d'en-tête (.h). Ces poids sont stockés dans des tableaux constants, rendant l'accélérateur totalement autonome vis-à-vis du système de fichiers.

Arithmétique à Virgule Fixe

L'implémentation de calculs en virgule flottante (float) sur FPGA est coûteuse en ressources et complexe. Nous avons donc converti l'intégralité de la chaîne d'inférence en arithmétique à virgule fixe. Cette étape a nécessité une analyse précise pour définir le format de données (ex: entiers 16 ou 32 bits) offrant le meilleur compromis entre l'occupation des ressources logiques et la minimisation de la perte de précision par rapport au modèle de référence.

Optimisation et Exploration de l'Espace de Conception (DSE)

Une fois le code fonctionnel, nous avons procédé à l'optimisation des performances via l'ajout de directives (pragmas) HLS. Cette exploration (DSE) a permis de guider l'outil de synthèse pour paralléliser les traitements, notamment grâce au pipelining des boucles internes et au partitionnement des tableaux pour augmenter la bande passante mémoire.

Intégration Système et Métriques d'Évaluation

La phase finale du projet vise à déployer le réseau de neurones complet sur l'architecture hétérogène du Zynq-7000. Cette intégration est réalisée via l'environnement SDSoc, qui automatise la conception conjointe logiciel/matériel (*HW/SW Co-design*). L'outil prend en charge la génération du bitstream pour la partie FPGA ainsi que la création des pilotes logiciels (drivers) et des mécanismes de transfert de données nécessaires à la communication entre le processeur ARM (exécutant l'OS Linux) et les accélérateurs matériels.

L'objectif est de produire une démonstration fonctionnelle de reconnaissance de caractères où le processeur délègue les tâches de calcul intensif (convolutions, pooling) à la logique programmable. La validation de cette architecture repose sur une analyse quantitative

stricte. Nous visons un gain de vitesse de traitement d'un facteur minimal de 4 et une amélioration de l'efficacité énergétique d'un facteur 100 par rapport à une exécution purement logicielle sur les cœurs Cortex-A9. Pour évaluer ces performances de manière globale, nous utiliserons le produit énergie-délai (*Energy-Delay Product*) comme métrique de référence dans notre rapport final.

Validation

Cette section présente la validation fonctionnelle du système complet sur la ZedBoard et l'analyse quantitative des performances. N'ayant pas abouti à une version matérielle parallélisée fonctionnelle, notre analyse compare les performances mesurées de la version logicielle (SW) aux performances de la version matérielle séquentielle (HW_SEQ) estimées par l'outil de synthèse.

Validation Fonctionnelle du Système

Nous avons validé le bon fonctionnement de l'environnement Linux embarqué et de la chaîne de compilation croisée.

1. **Démarrage de l'OS :** La communication série avec la ZedBoard est fonctionnelle. Nous avons pu accéder au système de fichiers et vérifier le montage de la partition contenant nos exécutables.
2. **Exécution de l'Inférence :** Le binaire `mnist_cnn_zedboard.elf` a été exécuté avec succès.
3. **Correction Algorithmique :** Le programme produit bien des vecteurs de probabilités (Softmax) en sortie, confirmant que le modèle LeNet fonctionne correctement sur le processeur ARM.

Analyse des Ressources (Synthèse HLS)

L'analyse de l'architecture matérielle séquentielle (HW_SEQ) s'appuie sur les rapports générés par Vivado HLS :

- **Utilisation des Ressources :** Le design n'utilise que 4 blocs DSP48E (soit ~1% du total disponible). Cette très faible occupation confirme que le circuit effectue les opérations mathématiques de manière séquentielle, sans exploiter le parallélisme spatial du FPGA.
- **Latence de Calcul :** Le rapport estime la latence de traitement pour une seule image à 64 694 947 cycles d'horloge.

Performances Comparatives

Nous comparons ici le temps d'exécution réel du logiciel (ARM) face à la performance de l'accélérateur séquentiel (FPGA).

Données et Méthode de Calcul :

- **Version SW (Mesurée) :** Lors de nos tests sur carte, nous avons relevé un temps total de 600 secondes pour le traitement de l'intégralité du jeu de test (10 000 images). Cela correspond à une moyenne de 60 ms par image.

- **Version HW_SEQ (Estimée) :** La partie logique (PL) du Zynq fonctionnant à une fréquence standard de 100 MHz, nous déduisons le temps de traitement à partir des cycles rapportés par Vivado :

$$T_{HW} = \frac{\text{Cycles}}{\text{Fréquence}} = \frac{64\,694\,947}{100\,000\,000} \approx 0.647 \text{ s} = \mathbf{647 \text{ ms}}$$

version	Temps global	Temps moy/img	speedUp
HW_SEQ	~6 470 s	647 ms	0.06 (Ralentissement x16)
SW (ARM Cortex-A9 Acceleré)	600 s	60 ms	1.0

Analyse et Conclusion

Les résultats mettent en évidence un ralentissement d'un facteur 16 en passant du logiciel au matériel séquentiel.

Ce résultat s'explique par deux facteurs architecturaux :

1. **Le différentiel de fréquence :** Le processeur ARM tourne à 667 MHz, soit 6,6 fois plus vite que le FPGA (100 MHz).
2. **L'absence de parallélisme :** Sans les directives d'optimisation (**UNROLL**, **PIPELINE**), le FPGA exécute les instructions séquentiellement, tout comme le processeur. Il subit donc sa fréquence d'horloge plus faible sans pouvoir la compenser par un traitement de masse.

Cette expérience démontre que le simple portage d'un code C sur FPGA est insuffisant. Pour qu'un accélérateur matériel soit pertinent, il est impératif d'utiliser des directives de synthèse (HLS pragmas) pour paralléliser les boucles de calcul et ainsi dépasser les performances d'un processeur généraliste.

Conclusion

Ce projet de fin d'études en systèmes embarqués avait pour objectif le déploiement et l'accélération d'un réseau de neurones convolutif (LeNet) sur une architecture hétérogène SoC Zynq-7000. Le défi principal résidait dans l'utilisation de la Synthèse de Haut Niveau (HLS) pour combler le fossé entre la modélisation logicielle abstraite et l'implémentation matérielle physique.

Bilan des réalisations

Nous avons mené avec succès la majeure partie du flux de conception, validant plusieurs jalons techniques critiques :

1. **Modélisation et Preuve de concept** : L'entraînement du modèle sous Keras et le développement d'une référence logicielle en C ont permis de valider l'algorithme.
2. **Conversion HLS** : Nous avons surmonté les contraintes de la synthèse matérielle en convertissant le code en arithmétique à virgule fixe et en supprimant toute allocation dynamique de mémoire.
3. **Intégration Système** : La réussite de l'implémentation sous SDSoC, avec un OS Linux fonctionnel exécutant notre binaire sur la ZedBoard, démontre notre maîtrise de la chaîne de compilation croisée et de l'environnement matériel.

Analyse des limitations

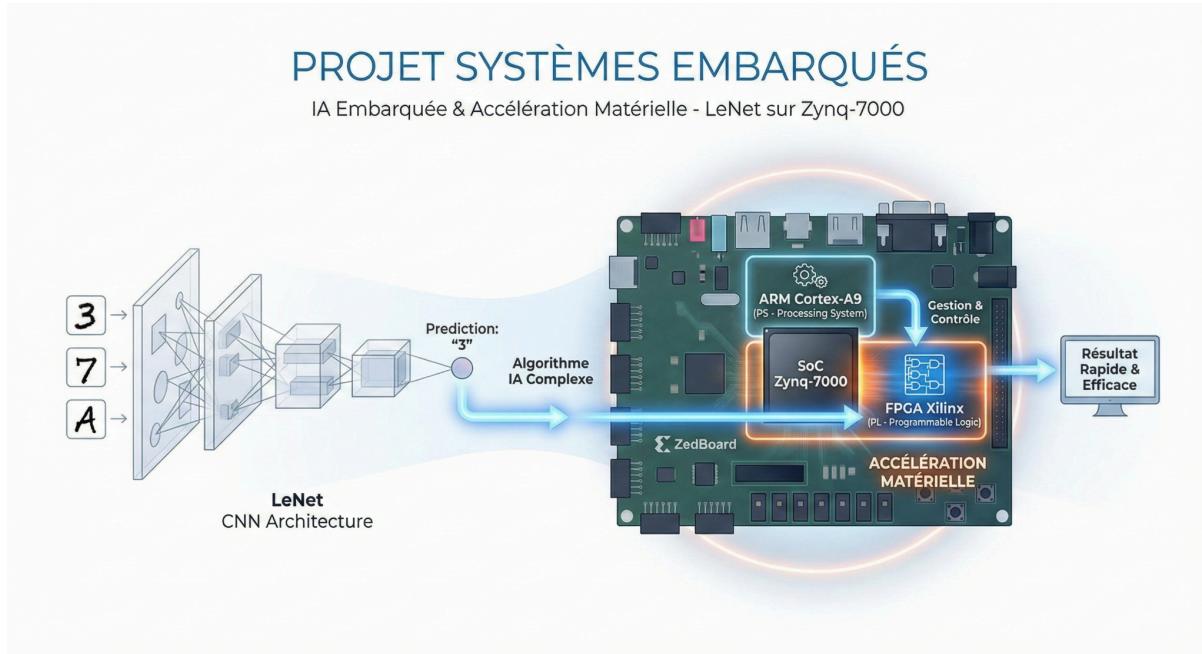
Cependant, nous n'avons pas pu atteindre le dernier objectif de performance fixé (accélération x4). La version finale implémentée est une version matérielle séquentielle, avec accélération, (HW_SEQ) qui s'avère plus lente que l'exécution logicielle sur le processeur ARM. Cet échec partiel sur le plan des performances s'explique par la difficulté rencontrée lors de l'étape d'**Exploration de l'Espace de Conception (DSE)**. Nous n'avons pas réussi à stabiliser une version parallélisée (HW_PAR) intégrant les directives d'optimisation (**PIPELINE**, **UNROLL**). Sans ce parallélisme spatial, le FPGA subit sa fréquence d'horloge inférieure (100 MHz contre 667 MHz pour le CPU) sans pouvoir compenser par un traitement massif des données.

Retour d'expérience

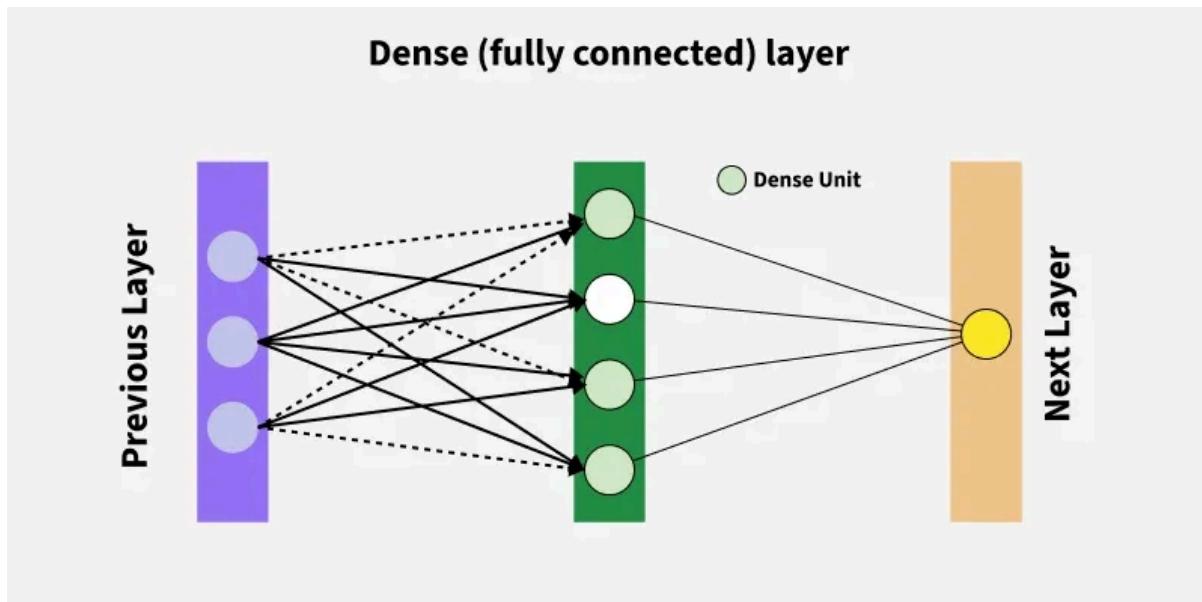
Malgré cette limitation finale, ce projet a été extrêmement formateur. Il nous a permis d'appréhender concrètement les défis du co-design matériel/logiciel et de comprendre que le portage sur FPGA n'est pas "magique" : l'accélération ne provient pas du support matériel en soi, mais de la capacité de l'ingénieur à architecturer son code pour exploiter le parallélisme. Le système fonctionnel que nous avons produit constitue une base saine sur laquelle il ne reste plus qu'à itérer les directives de synthèse pour débloquer la puissance du FPGA.

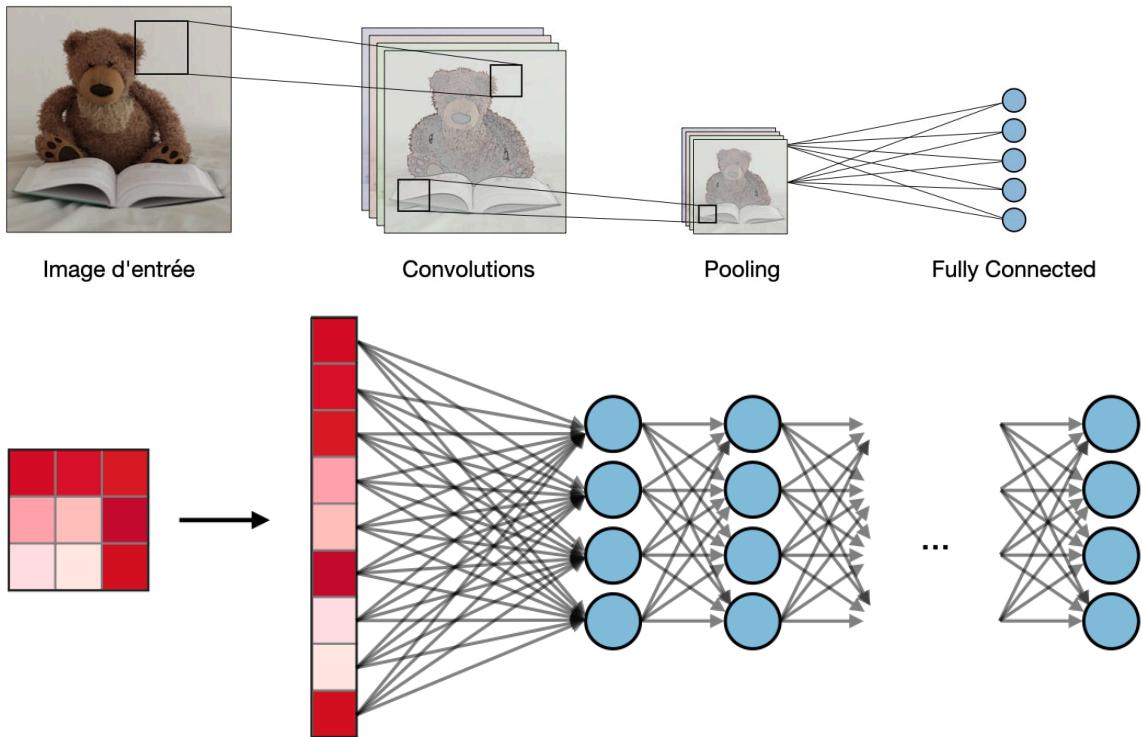
Annexes

Schéma résumé du projet

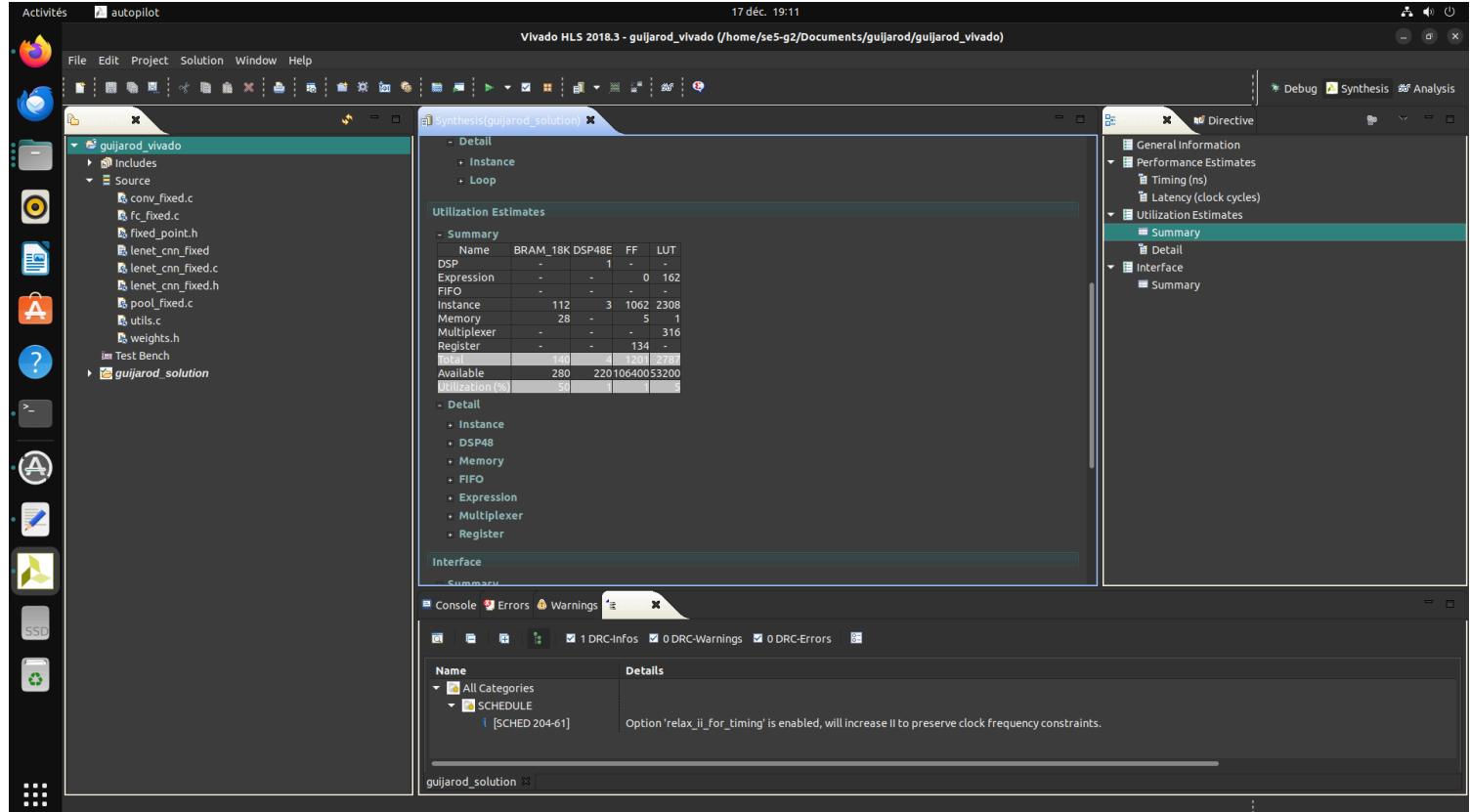
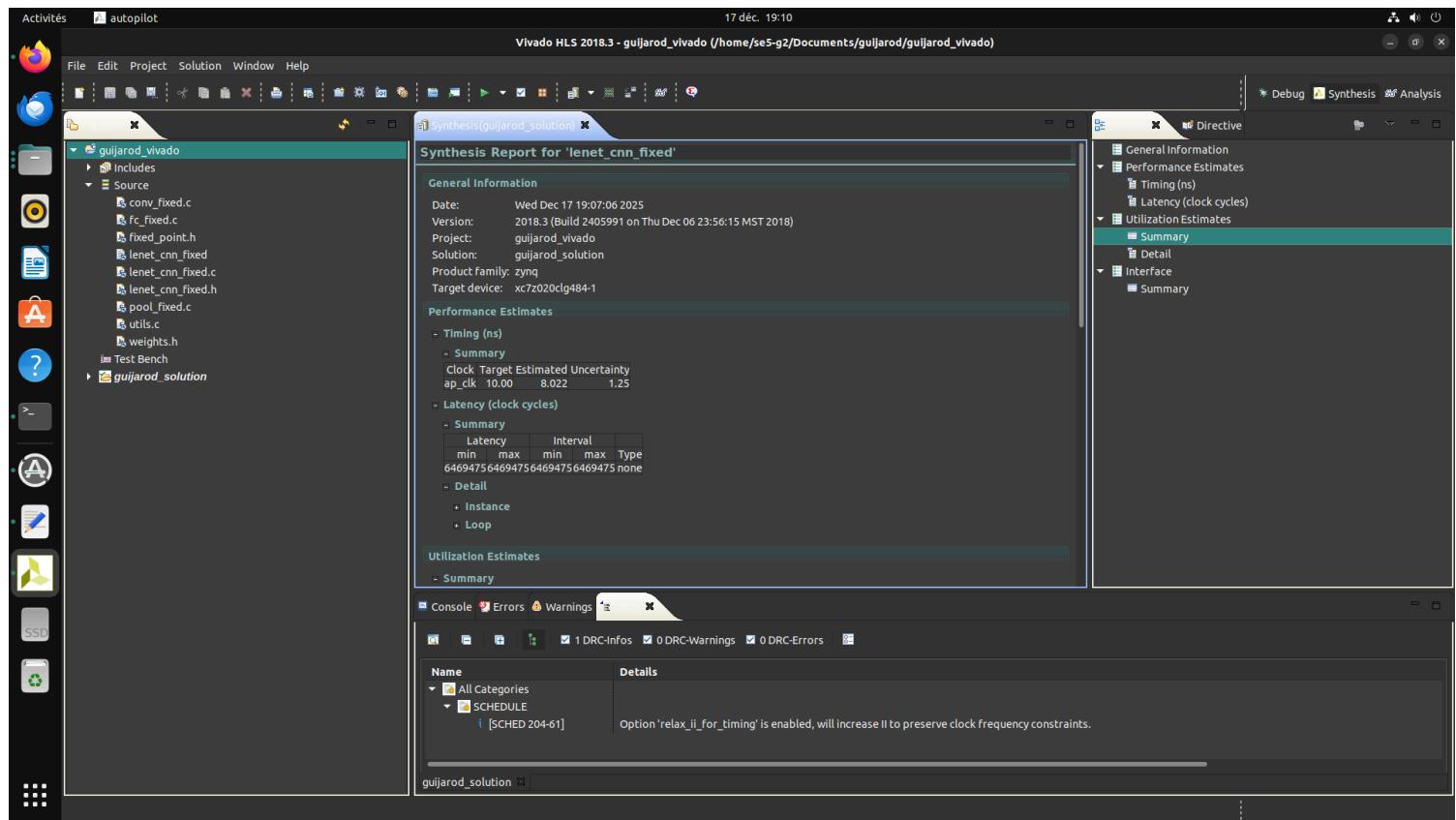


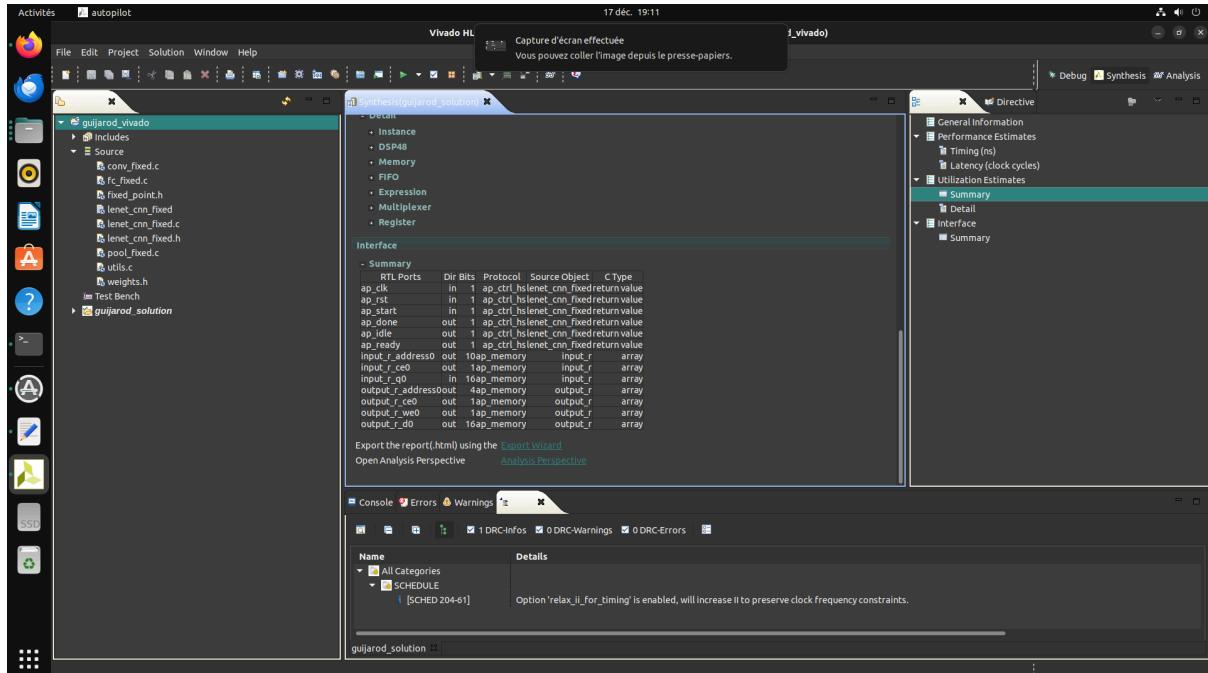
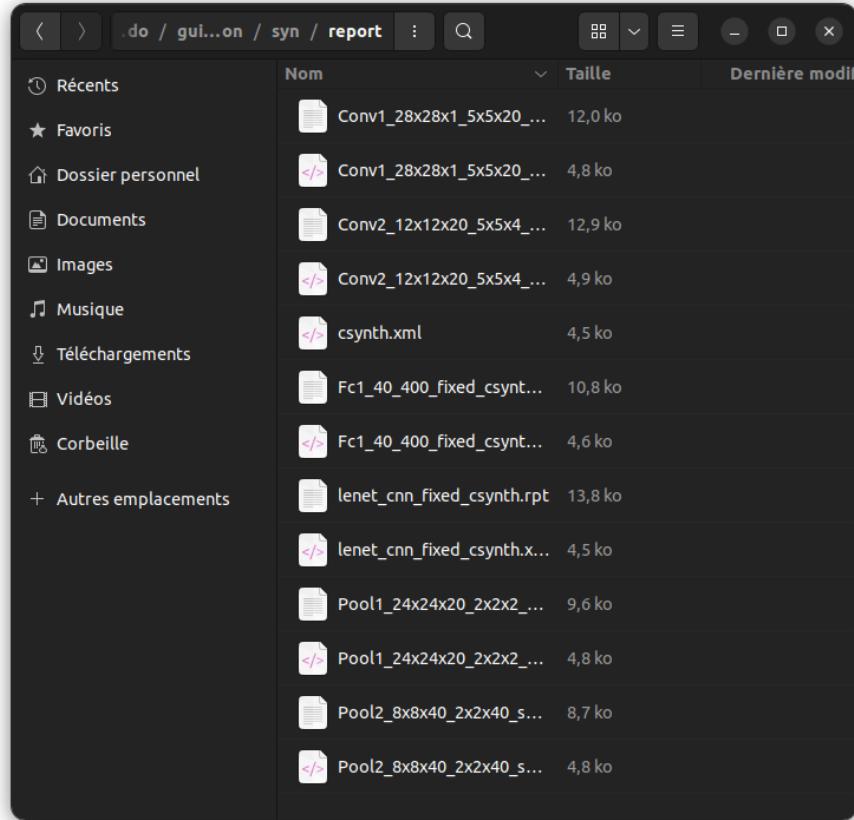
Schémas explicatifs





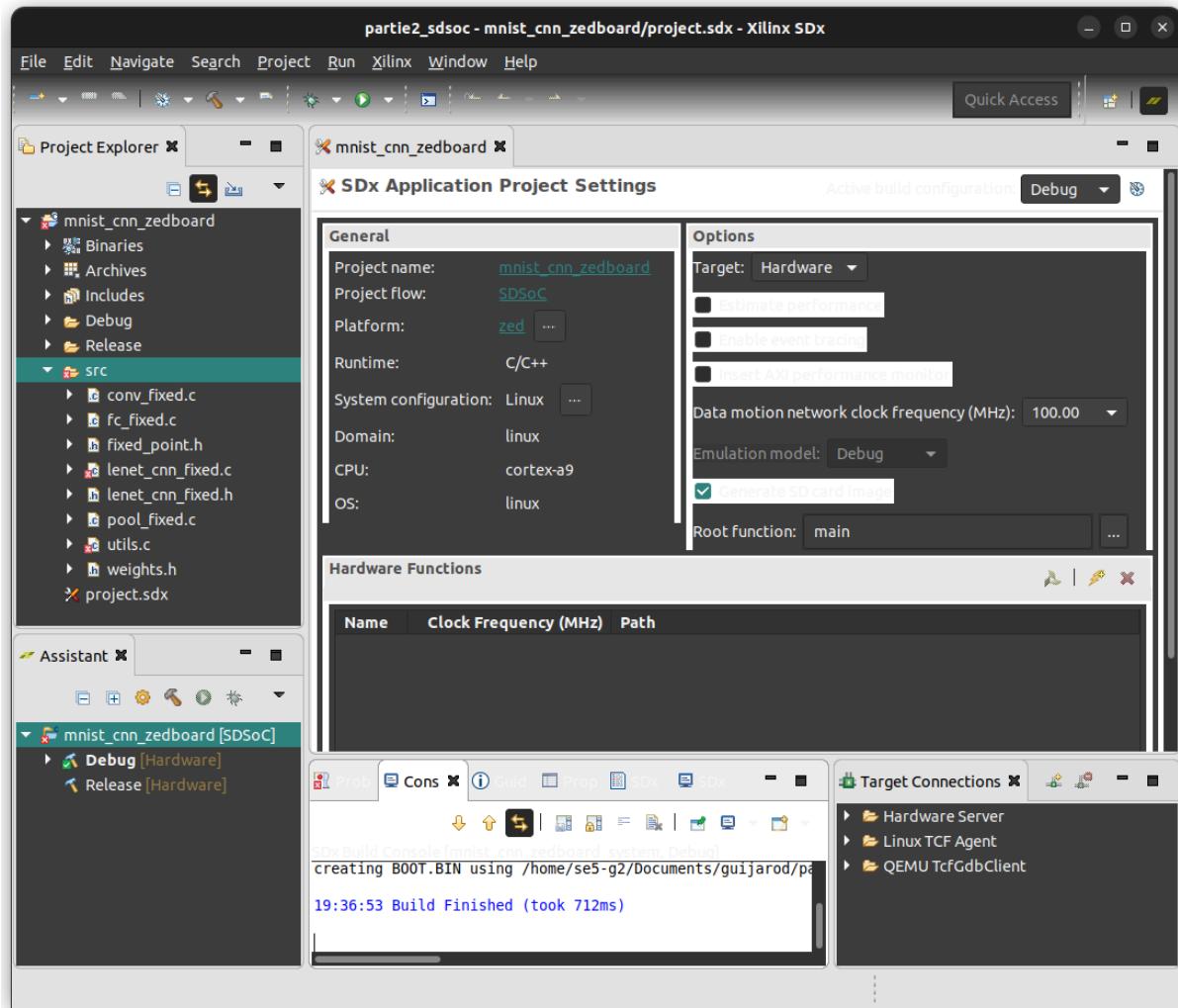
Synthétisation sur Vivado





Partie sur SDSoc

Après le build



Validation des fichiers générés

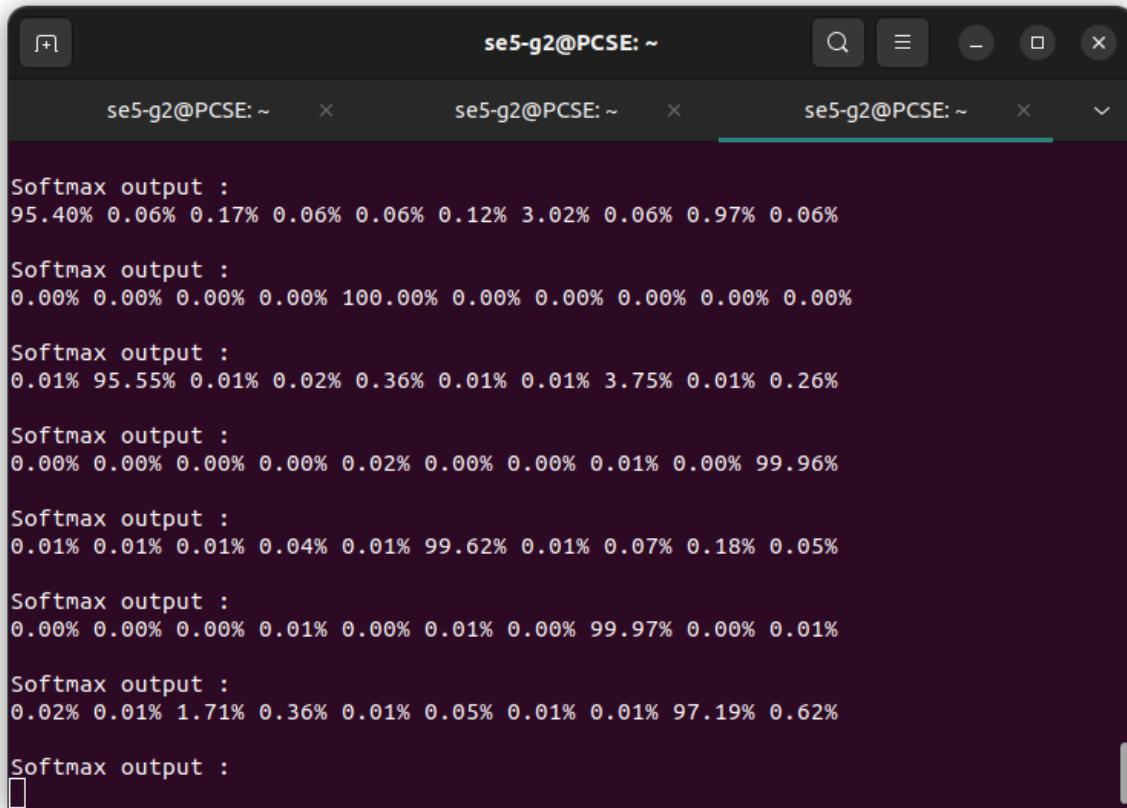
The screenshot shows a file explorer window with the following details:

- Path: / pa...soc / mn... rd / Debug
- File List:

 - guidance.html (1,7 ko)
 - guidance.pb (251 octets)
 - makefile (1,6 ko)
 - mnist_cnn_zedboard.elf (595,1 ko)
 - mnist_cnn_zedboard.elf.... (4,0 Mo)
 - mnist_cnn_zedboard_De... (9,4 ko)
 - mnist_cnn_zedboard_De... (4,7 ko)
 - objects.mk (231 octets)
 - removeFiles.sh (13 octets)
 - sd_card (4 éléments)
 - _sds (6 éléments)
 - sources.mk (484 octets)
 - src (11 éléments)

Partie hardware

exécution sans accélérateur



The screenshot shows a terminal window with three tabs open, all titled "se5-g2@PCSE: ~". The terminal background is dark purple. The first tab is active and displays the following text:

```
Softmax output :  
95.40% 0.06% 0.17% 0.06% 0.06% 0.12% 3.02% 0.06% 0.97% 0.06%  
  
Softmax output :  
0.00% 0.00% 0.00% 0.00% 100.00% 0.00% 0.00% 0.00% 0.00% 0.00%  
  
Softmax output :  
0.01% 95.55% 0.01% 0.02% 0.36% 0.01% 0.01% 3.75% 0.01% 0.26%  
  
Softmax output :  
0.00% 0.00% 0.00% 0.00% 0.02% 0.00% 0.00% 0.01% 0.00% 99.96%  
  
Softmax output :  
0.01% 0.01% 0.01% 0.04% 0.01% 99.62% 0.01% 0.07% 0.18% 0.05%  
  
Softmax output :  
0.00% 0.00% 0.00% 0.01% 0.00% 0.01% 0.00% 99.97% 0.00% 0.01%  
  
Softmax output :  
0.02% 0.01% 1.71% 0.36% 0.01% 0.05% 0.01% 0.01% 97.19% 0.62%  
  
Softmax output :  
□
```