



Joint design of deep neural networks and FPGA dataflow architectures for semantic segmentation in autonomous vehicles

Hugo Le Blevec

► To cite this version:

Hugo Le Blevec. Joint design of deep neural networks and FPGA dataflow architectures for semantic segmentation in autonomous vehicles. Machine Learning [cs.LG]. Ecole nationale supérieure Mines-Télécom Atlantique, 2024. English. ⟨NNT : 2024IMTA0445⟩. ⟨tel-05206281⟩

HAL Id: tel-05206281

<https://theses.hal.science/tel-05206281v1>

Submitted on 11 Aug 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



HAL Authorization

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
MINES-TÉLÉCOM ATLANTIQUE BRETAGNE
PAYS DE LA LOIRE – IMT ATLANTIQUE

ÉCOLE DOCTORALE N° 648
Sciences pour l'Ingénieur et le Numérique
Spécialité : *Électronique*

Par

Hugo Le Blevec

Joint design of deep neural networks and FPGA dataflow architectures for semantic segmentation in autonomous vehicles

Thèse présentée et soutenue à IMT Atlantique, Brest, le 6 Décembre 2024
Unité de recherche : Lab-STICC, CNRS UMR 6285
Thèse N° : 2024IMTA0445

Rapporteurs avant soutenance :

Olivier SENTIEYS Professeur des Universités, Université de Rennes, Rennes
Andrea PINNA Professeur des Universités, Sorbonne Université, Paris

Composition du Jury :

Président :	Frédéric PÉTRO	Professeur des Universités, Grenoble INP, Grenoble
Examineurs :	Olivier SENTIEYS Andrea PINNA Anastasia VOLKOVA Camille LEROUX Thomas PREÜBER	Professeur des Universités, Université de Rennes, Rennes Professeur des Universités, Sorbonne Université, Paris Chargée de recherche, Inria, Lyon Maître de conférences, Bordeaux INP, Bordeaux Principal Research Engineer, AMD, Dresden
Dir. de thèse :	Matthieu ARZEL	Professeur, IMT Atlantique, Brest
Co-encadrant :	Mathieu LÉONARDON	Maître de conférences, IMT Atlantique, Brest

Invité(s) :

Jakoba PETRI-KOENIG Research Engineer, AMD, Dublin

Contents

Acknowledgments	9
Résumé long	11
Résumé	23
Abstract	25
Introduction	27
Preamble	27
Context	27
Outline and contributions	29
1 Deep learning for semantic segmentation and super-resolution	33
1.1 Model building and training	34
1.1.1 Compositional approach	34
1.1.2 Loss function	35
1.1.3 Optimizer	35
1.1.4 Scheduler	36
1.2 Operators	36
1.2.1 Convolution layer	36
1.2.2 Fully-connected layer	37
1.2.3 Pooling layers	38
1.2.4 Upsampling layers	38
1.2.5 Activation layers	40
1.2.6 Batch normalization layer	41
1.3 Model compression	41
1.3.1 Quantization	42
1.3.2 Efficient convolutions	46
1.3.3 Knowledge distillation	49
1.3.4 Pruning	49
1.3.5 Neural Architecture Search	50
1.4 Semantic segmentation	50
1.4.1 Task definition, metrics and applications	50
1.4.2 Datasets	52
1.4.3 Models history overview	53
1.4.4 Real-time semantic segmentation	54
1.5 Super-resolution	57
1.5.1 Task definition and metrics	57
1.5.2 BSD dataset	57
1.5.3 Models overview	58

1.6	Summary	58
2	Deploying Deep Learning Models on FPGA	59
2.1	Basics of FPGA architectures	60
2.2	Interests and challenges	61
2.3	Architectures and frameworks	64
2.3.1	Deep learning Processing Units	64
2.3.2	Pipelined dataflow architectures	65
2.3.3	Choice of framework	66
2.4	FPGA architectures for real-time semantic segmentation	67
2.4.1	CamVid	67
2.4.2	Cityscapes	68
2.4.3	Observations	69
2.5	Summary	70
3	FINN: a framework for fast, scalable quantized neural network inference on FPGAs - Principles and contributions	71
3.1	Detailed introduction	72
3.1.1	Hardware Library	72
3.1.2	Compiler	73
3.2	Dealing with residual networks	78
3.3	Contributions to the framework	80
3.3.1	New <i>MakeMaxPoolNHWC</i> transformation	81
3.3.2	The <i>SplitLargeFIFOs</i> transformation	82
3.3.3	Improved support for nearest neighbor upsampling	82
3.3.4	Support of transposed convolutions	83
3.4	Summary	84
4	Co-designing semantic segmentation models for pipelined dataflow hardware architectures	85
4.1	First study-case: ResNet18-UNet	86
4.1.1	Motivations	86
4.1.2	Structure adaptations for compatibility	88
4.1.3	Training and quantization	89
4.1.4	Implementation results	91
4.1.5	Discussions	93
4.2	Second study-case: ENet	93
4.2.1	Motivations	93
4.2.2	Model definition	93
4.2.3	Training	96
4.2.4	Model adptations	98
4.2.5	Implementation results	100
4.3	Hardware-oriented design space exploration of ENet architecture	102
4.3.1	Comparison against previous FPGA implementations	104
4.4	Performance evaluation against GPU	106
4.5	Summary and discussions	109
5	Upsampling method co-design	111
5.1	Context and motivations	112
5.2	Training setup, quantization, and results	113
5.3	Comparative implementation results	114
5.3.1	Inference transformations	114

5.3.2	Two-times upscaling	116
5.3.3	Higher upscaling ratios	117
5.4	Improving transposed convolution performance	118
5.5	Summary	120
Conclusion		121
	Summary	121
	Perspectives	123
Bibliography		127

Acronyms

ASIC Application-Specific Integrated Circuit.

BRAM Block Random Access Memory.

CLB Configurable Logic Block.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

DMA Direct Memory Access.

DPU Deep-learning Processing Unit.

DSP Digital Signal Processing.

FF Flip-Flop.

FIFO First In First Out.

FPGA Field-Programmable Gate Array.

FPS Frames Per Second.

GPU Graphics Processing Unit.

HDL Hardware Description Language.

HLS High-Level Synthesis.

IoU Intersection over Union.

LUT Look-Up Table.

MAC Multiply-Accumulate.

mIoU mean Intersection over Union.

MSE Mean Square Error.

NAS Neural Architecture Search.

ONNX Open Neural Network eXchange.

PA Pixel Accuracy.

PE Processing Element.

PReLU Parametric Rectified Linear Unit.

PSNR Peak Signal-to-Noise Ratio.

PTQ Post-Training Quantization.

QAT Quantization Aware Training.

ReLU Rectified Linear Unit.

RTL Register Transfer Level.

SGD Stochastic Gradient Descent.

SIMD Single Instruction Multiple Data.

SoC System-on-Chip.

SSIM Structural Similarity Index Measure.

STE Straight-Through Estimator.

UAV Unmanned Aerial Vehicle.

URAM Ultra Random Access Memory.

ViT Vision Transformer.

Acknowledgments

First and foremost, I would like to thank Andrea Pinna and Olivier Sentieys for their review of this manuscript and for taking part in the defense as jury members. I would also like to thank all the other jury members, Frédéric Pétrot, Anastasia Volkova, Camille Leroux, Thomas Preußner, and Jakoba Petri-Koenig for being part of the defense and for the interesting questions and discussions.

I also want to express my deepest gratitude to my supervisors, Matthieu Arzel and Mathieu Léonardon, for offering me this thesis and for their guidance, trust, and understanding throughout the three and a half years of this Ph.D. Many thanks to the researchers of the Mathematical & Electrical Engineering department of IMT Atlantique with whom I have had the opportunity to collaborate and interact. In particular, I want to send my warmest thanks to all the BRAIn team members, whether they are permanent researchers, PhD students, research engineers, or interns, for all the lively and interesting moments and discussions I have had the immense privilege of sharing with them.

I want to sincerely thank Michaela Blott and Yaman Umuroglu from AMD Research in Dublin for offering me an internship, as well as all the other employees of AMD that I had the opportunity to meet and work with. In particular, I want to express my sincere gratitude to Thomas Preußner, Ian Colbert, and Jakoba Petri-Koenig for all the help they provided and all the interesting collaborations, both during and after the internship, that have had an immense influence on the results of this thesis.

Finally, I want to thank my friends and my family for their love and continuous support throughout this thesis and every day of my life.

Résumé long

Introduction

Au cours des dix dernières années, l'apprentissage automatique s'est imposé comme un paradigme révolutionnaire en informatique, capable de résoudre des problèmes complexes grâce à l'entraînement d'algorithmes sur des exemples, contrairement aux approches classiques basées sur des solutions explicites. L'apprentissage profond, sous-discipline de l'apprentissage automatique, a gagné en notoriété depuis le succès d'Alex-Net en 2012 lors du concours ImageNet, devenant un outil incontournable dans des domaines variés comme la vision par ordinateur, la reconnaissance audio et le traitement du langage naturel. Ce succès est attribué à l'accès facilité à de grandes quantités de données et à des avancées matérielles, notamment les *Graphics Processing Units* (GPUs), qui accélèrent les calculs intensifs des réseaux neuronaux.

Malgré leurs performances, les réseaux de neurones profonds posent des défis lorsqu'ils doivent être déployés sur des systèmes embarqués, tels que les véhicules autonomes, les drones ou les robots. Ces systèmes nécessitent des inférences rapides (faible latence) et une consommation énergétique limitée, ce qui rend les GPUs traditionnels inadaptés en raison de leur forte consommation. Les GPUs embarqués, développés par Nvidia, et les FPGA (circuits logiques reconfigurables) offrent des alternatives intéressantes. Les FPGA, en particulier, sont plus efficaces pour les tâches de vision embarquée, notamment en termes de consommation énergétique et d'intégration au système, mais leur mise en œuvre est complexe, surtout pour des modèles intensifs comme les réseaux de neurones.

Les recherches actuelles se concentrent principalement sur la réduction de l'empreinte computationnelle et mémoire des DNN via des techniques de compression, tout en maintenant un équilibre entre performance matérielle et précision. Une question clé réside dans l'adéquation de la conception du réseau aux exigences matérielles spécifiques, au-delà de simples compressions. Cette thèse explore ces relations en étudiant comment la conception algorithmique des réseaux influence leurs performances en termes de précision, latence et débit dans le contexte d'implémentations sur FPGA.

Cette thèse se concentre sur la tâche de segmentation sémantique, connue pour sa complexité et son exigence en calcul haute résolution, ce qui en fait un défi pour les FPGA. La segmentation sémantique est critique pour des applications comme les drones de recherche et sauvetage, qui ont besoin d'analyser une scène inconnue et complexe. Cette thèse a d'ailleurs contribué au projet LEASARD, financé par le Labex Comin-labs (ANR-10-LABX-07-01). Grâce à l'outil open-source FINN, des architectures adaptées aux FPGA sont générées pour deux modèles de segmentation sémantique, permettant une exploration des performances matérielles et algorithmiques. Les réseaux sont entraînés et évalués sur des données de scènes urbaines, souvent utilisées pour les voitures autonomes, mais les conclusions sont transposables à d'autres systèmes

embarqués, tels que les drones de recherche et sauvetage.

En complément, la thèse étudie la co-conception des méthodes de suréchantillonnage et des architectures matérielles, dans le cadre de la super-résolution, une composante essentielle des réseaux utilisés. Cette approche vise à optimiser l'équilibre entre précision, latence et consommation énergétique, tout en répondant aux contraintes des systèmes embarqués.

L'apprentissage profond pour la segmentation sémantique et la super-résolution

Ce premier chapitre sert d'introduction aux différents concepts employés lors de cette thèse et sert de guide à la compréhension du reste du manuscrit.

Il commence par une présentation des bases de l'apprentissage profond. Les réseaux de neurones profonds y sont décrits comme des modèles composés de couches mathématiques capables d'apprendre à partir de données. Le processus d'entraînement repose sur la propagation avant (pour générer une prédiction) et la propagation arrière (pour mettre à jour les paramètres en minimisant une fonction de perte). Les différentes composantes de ce processus telles que les optimiseurs couramment utilisés, comme SGD (*Stochastic Gradient Descent*) et Adam, les planificateurs de taux d'apprentissage et les fonctions de perte utilisées, sont détaillés. Différents types de couches (convolutions, pooling, activations, normalisation par *batch*) sont également expliqués, soulignant leurs rôles clés dans le traitement des données.

Par ailleurs, les réseaux de neurones modernes étant souvent trop gourmands en calculs pour être utilisés directement sur des systèmes embarqués, tels que les drones ou les véhicules autonomes, ce chapitre détaille plusieurs techniques dites de compression permettant de rendre ces modèles plus légers :

- Quantification : réduction de la précision des poids et activations (par exemple, passer de FP32 à INT8) pour diminuer la consommation d'énergie et l'empreinte mémoire.
- Convolutions efficaces : optimisations des convolutions (par exemple, convolutions séparables ou dilatées) pour réduire les coûts de calcul tout en maintenant les performances.
- Distillation des connaissances : transfert des connaissances d'un modèle complexe (« enseignant ») vers un modèle plus simple (« étudiant ») pour améliorer la performance de ce dernier.
- Elagage : suppression de poids ou de structures inutiles dans un réseau pour réduire la taille du modèle sans compromettre excessivement la précision.
- Recherche d'architecture neuronale (NAS) : automatisation de la conception de réseaux de neurones pour trouver des architectures optimisées.

Dans un second temps, ce chapitre introduit une définition et un état de l'art des tâches d'apprentissage profond étudiées dans cette thèse, à savoir la segmentation sémantique et la super-résolution.

La segmentation sémantique est le principal sujet d'étude du manuscrit. Elle consiste à attribuer une classe à chaque pixel d'une image, permettant ainsi de comprendre les scènes complexes. Les applications sont nombreuses, allant des véhicules autonomes

(analyse de scènes routières) à l'imagerie médicale (segmentation d'organes ou de tissus), en passant par l'agriculture (classification des cultures et détection des maladies). Les défis incluent l'obtention de données étiquetées en grande quantité, le déséquilibre des classes dans les images, et les besoins en ressources pour traiter des images haute résolution. La métrique principale permettant d'évaluer les modèles de réseaux de neurones sur cette tâche est la mIoU (*mean intersection over union*) qui se définit comme la moyenne des rapports entre intersection et union entre la prédiction et l'exemple annoté. Pour cette tâche, nous avons utilisé deux ensembles de données populaires et adaptés à l'entraînement de réseaux de neurones pour la segmentation de scènes urbaines, à savoir Cityscapes et CamVid.

Une évolution des modèles de segmentation sémantique est présentée, depuis les premiers réseaux comme FCN et U-Net jusqu'aux modèles récents basés sur les Transformers, tels que SETR et InternImage. Un accent particulier est mis sur les modèles en temps réel, conçus pour répondre à des contraintes strictes de latence et de débit. Ces modèles, bien qu'ils sacrifient une partie de leur précision, utilisent des structures de réseaux efficaces pour fonctionner dans des systèmes comme les véhicules autonomes, où le traitement rapide des données est essentiel.

Bien que secondaire dans ce manuscrit, la tâche de super-résolution est également abordée. Elle consiste à augmenter la résolution des images tout en préservant leur qualité. Les modèles utilisés vont des premières approches, comme SRCNN et VDSR, aux réseaux génératifs adversaires (GANs) et aux Transformers. Les métriques couramment utilisées, telles que le PSNR (rapport signal sur bruit pic) et le SSIM (indice de similarité structurelle), permettent d'évaluer la qualité des reconstructions. Le dataset BSD (Berkeley Segmentation Dataset) est utilisé pour les expériences, fournissant un ensemble diversifié d'images pour l'entraînement et les tests.

Deployment de modèles d'apprentissage profond sur FPGA

Ce second chapitre explore les principes fondamentaux des FPGA (Field-Programmable Gate Arrays) et leur potentiel pour exécuter des réseaux neuronaux profonds dans des systèmes embarqués. Il met en lumière les avantages et les défis des FPGA, tout en présentant les outils et cadres existants pour faciliter leur utilisation. Enfin, il passe en revue les recherches précédentes sur la mise en œuvre de modèles de segmentation sémantique en temps réel sur FPGA.

Les FPGA sont des dispositifs matériels reconfigurables qui diffèrent des ASICs (circuits spécifiques à une application) ou des CPU et GPU, dont l'architecture est fixe. Les FPGA sont composés de blocs logiques configurables (CLBs), de blocs DSP optimisés pour les opérations de multiplication-accumulation (MAC), et de mémoires RAM embarquées (BRAM). Leur reconfigurabilité permet de concevoir presque toutes les fonctions de traitement numérique.

La programmation des FPGA s'effectue traditionnellement à un niveau bas via des langages de description matérielle (HDL) qui modélisent les signaux et les opérateurs logiques. Toutefois, des outils de synthèse de haut niveau (HLS) ont émergé pour simplifier cette tâche en utilisant des langages comme le C++. Cela réduit la complexité du processus, bien que l'architecture générée soit souvent moins optimisée.

Les FPGA présentent plusieurs avantages, notamment :

- Une faible consommation énergétique : idéal pour les systèmes embarqués ayant des contraintes en puissance. Cette capacité a notamment été illustrée lors du

concours DAC-SDC de 2018, où les compétiteurs devaient implémenter des réseaux de détection d'objet sur FPGA ou sur GPU embarqué. Les résultats montrent que, pour des performances similaires, les FPGA consomment deux à trois fois moins d'énergie.

- **Parallélisme élevé** : grâce à leur structure en grille, les FPGA sont particulièrement efficaces pour exécuter des fonctions linéaires et logiques en parallèle.
- **Reconfigurabilité** : les FPGA peuvent être reprogrammés après déploiement, ce qui les rend flexibles pour s'adapter à de nouveaux cas d'usage.

Cependant, leur utilisation pour embarquer des réseaux de neurones pose des défis importants :

- **Ressources limitées** : la mémoire embarquée est souvent insuffisante pour des modèles complexes, obligeant à des compromis tels que la compression ou la simplification des modèles. À titre d'exemple, les puces FPGA parmi les plus grandes du marché ont 80 Mb de mémoire interne, ce qui est ridicule comparé aux Gb de mémoires disponibles sur les GPUs embarqués.
- **Complexité de l'implémentation** : la programmation des FPGA requiert une expertise en électronique numérique. Bien que les outils HLS réduisent cette difficulté, ils n'éliminent pas la nécessité d'optimisations et d'ajustements manuels.

Afin de faciliter l'implémentation de réseaux de neurones sur FPGA, de nombreux outils ont été développés, et peuvent être distingués par la catégorie d'architectures qui est utilisée pour déployer le réseau :

- **Unités de traitement de deep learning (DPU)** : ces moteurs de traitement sont génériques et agnostiques à la structure du réseau. Ils réutilisent la même unité pour toutes les couches, ce qui facilite l'implémentation et préserve la précision. Cependant, les transferts fréquents de données entre les unités augmentent la latence. Exemples : Vitis AI (AMD), OpenVINO (Intel), Gemmini (UC Berkeley).
- **Architectures pipeline à flux de données** : chaque couche du réseau est déployée sur une unité dédiée, opérant en parallèle selon un pipeline. Ces architectures minimisent la latence et les transferts de données, mais elles consomment davantage de ressources et nécessitent une compression significative des modèles. Exemples : FINN (AMD), HLS4ML (CERN), fpgaConvNet (Imperial College).

Pour cette thèse, nous avons choisi de travailler avec FINN, un projet open-source d'AMD, pour sa flexibilité, sa compatibilité avec PyTorch et sa capacité à produire des architectures à faible latence. Il permet de générer des architectures FPGA complètes de type pipeline à flux de données, avec un support étendu pour diverses couches réseau. Bien qu'il nécessite une compréhension approfondie de ses concepts pour des modèles complexes comme ceux de la segmentation sémantique, il reste un outil clé pour explorer la co-conception entre réseaux neuronaux et architectures matérielles.

Enfin, le chapitre termine par un état de l'art des précédentes implémentations de réseaux de neurones sur FPGA pour la segmentation sémantique des jeux de données Cityscapes et CamVid, en mettant en valeur les types d'architectures utilisés. Celui-ci met en évidence une tendance générale à utiliser des outils HLS (High-Level Synthesis) pour générer les architectures matérielles. Il apparaît également que les architectures à flux de données en pipeline tendent à offrir une latence plus faible que les unités de traitement spécialisées (DPUs).

L'approche typique pour cette tâche consiste à sélectionner un réseau de neurones, appliquer une quantification et/ou un élagage, puis procéder à l'implémentation, parfois accompagnée de discussions sur les optimisations matérielles et les choix de conception. Cependant, les arguments concernant le choix du réseau de neurones sont souvent succincts, voire absents, et rares sont les études qui justifient en détail pourquoi un réseau spécifique est préféré à un autre. Cela souligne la nécessité de recherches supplémentaires sur la relation entre l'architecture des réseaux de neurones et l'exploration des compromis entre précision et performances matérielles. Par ailleurs, il est difficile d'extraire des méthodologies claires pouvant être appliquées à différents réseaux ou contextes. Cette difficulté est exacerbée par le manque de codes open-source ou de descriptions matérielles accessibles dans ce domaine de recherche.

FINN : un outil pour l'inférence rapide et modulable de réseaux de neurones quantifiés sur FPGA - Principes et contributions

Ce troisième chapitre a pour objectif d'introduire les concepts principaux de FINN ainsi que les différentes contributions que nous avons apportées à l'outil pour nous permettre de l'utiliser pour implémenter les réseaux de segmentation sémantique étudiés.

FINN est un outil open-source conçu pour déployer des réseaux de neurones quantifiés sur des FPGA, exploitant des blocs HLS et RTL. Ces réseaux utilisent une quantification entière uniforme des poids et activations, permettant une meilleure efficacité énergétique et un débit plus élevé comparé aux formats en virgule flottante. Cet outil repose sur deux composantes principales : une bibliothèque matérielle composée de modules HLS ou RTL configurables et un "compilateur" qui permet, à partir d'une représentation du réseau sous forme de graphe, d'associer les opérateurs du réseau aux éléments de la bibliothèque matérielle, ainsi que d'automatiser les étapes de synthèse et d'implémentation matérielle. Cette combinaison permet de produire des architectures à débit élevé et faible latence, adaptées à des tâches spécifiques.

La bibliothèque matérielle contient des modules pour des opérations courantes dans les réseaux de neurones. Un exemple clé est l'Unité Matrice-Vecteur-Activation (MVAU), qui remplace les convolutions par des multiplications matricielles simplifiées en utilisant la méthode Im2Col. Par ailleurs, les opérations de normalisation de batch et d'activations sont fusionnées avec la multiplication matricielle dans ce seul module matériel grâce à des comparaisons avec différentes valeurs de seuil. Chaque composant matériel de FINN, comme le MVAU, est hautement configurable :

- Paramètres de parallélisme : les utilisateurs peuvent ajuster le nombre de lignes SIMD (*Single Instruction Multiple Data*) et d'unités de traitement (*Processing Elements*, PEs).
- Types de ressources : les additionneurs et multiplieurs peuvent utiliser des LUT (*Look-Up Tables*) ou des blocs DSP (*Digital Signal Processing*) tandis que les mémoires peuvent utiliser des LUT, des BRAM (Block RAMs) ou encore des URAMs (Ultra RAMs) selon les choix de l'utilisateur.

Le compilateur de FINN est une bibliothèque Python qui transforme des graphes QONNX (une version augmentée du format ONNX pour gérer la quantification) en architectures matérielles. Cette conversion passe par plusieurs étapes :

1. Streamlining (Simplification du graphe) :

- Suppression des opérations en virgule flottante (normalisation batch, scalaires de quantification) en les absorbant dans des transformations linéaires.
 - Conversion des activations quantifiées en opérateurs de seuil (MultiThreshold).
 - Transformation des convolutions classiques en opérations Im2Col suivies de multiplications matricielles.
 - Changement de format de données de NCHW (canal en premier) à NHWC (canal en dernier), pour une meilleure compatibilité avec le matériel.
2. Conversion en Couches Matérielles : les nœuds simplifiés sont remplacés par leurs équivalents matériels HLS/RTL.
3. Synthèse et Intégration :
- Les opérateurs sont paramétrés (nombre de PEs et de SIMDs) et une estimation de l'utilisation des ressources et du débit est réalisée.
 - Le code HLS ou RTL de chaque opérateur est généré et synthétisé.
 - Des mémoires tampons sous forme de FIFOs (*First In First Out*) sont insérées entre les blocs pour compenser les différences de latence et garantir le débit.
 - Les modules sont connectés entre eux afin de réaliser le chemin de données du réseau et l'architecture complète est implémentée.

Bien que FINN soit performant pour des réseaux à la structure simple, son intégration avec des réseaux résiduels tel qu'utilisés en segmentation sémantique soulève plusieurs difficultés. En effet, FINN est initialement conçu pour des topologies linéaires où chaque couche a une seule entrée et une seule sortie. Les connexions résiduelles et branches multiples nécessitent des transformations spécifiques pour éviter des incohérences dans les graphes. De plus, les branches résiduelles introduisent des différences de latence, nécessitant une synchronisation précise au niveau des nœuds de fusion. Une mauvaise configuration des FIFOs peut entraîner des blocages (*deadlocks*). Pour ces raisons, nous avons dû apporter des modifications à l'outil sous la forme de nouveaux éléments permettant de faciliter l'intégration de réseaux résiduels. Ces nouveaux éléments faisant désormais partie intégrante du code officiel du projet, ils contribuent à l'enrichir. Nos contributions apportées à FINN sont les suivantes :

- Transformation *MakeMaxPoolNHWC* : cette transformation résout les problèmes rencontrés lorsque des nœuds MaxPool ont plusieurs sorties. Une version améliorée insère des nœuds de transposition là où nécessaire pour maintenir la cohérence des formats de données.
- Transformation *SplitLargeFIFOs* : en réponse à la limite de profondeur des FIFOs imposée par Vivado (32768 valeurs), cette transformation divise les grands tampons en plus petits, optimisant par la même occasion l'utilisation de la mémoire.
- Support pour le suréchantillonnage par plus proche voisin : l'intégration des couches de suréchantillonnage par la méthode du plus proche voisin utilise la transformation *MakeScaleResizeNHWC*, qui ajuste les échelles de redimensionnement et les formats des données pour permettre à l'étape de Streamlining de se dérouler correctement.
- Support des Convolutions Transposées : à travers une collaboration avec AMD, nous avons aidé au développement des différents éléments permettant une inté-

gration dans FINN des convolutions transposées, une méthode de suréchantillonnage des images populaires dans les réseaux de segmentation sémantique et de super-résolution.

Ces contributions à l'outils ont permis l'implémentation des réseaux de segmentation sémantique présentés dans le chapitre suivant.

Co-conception de modèles de segmentation sémantique pour les architectures matérielles en flux de données

Ce quatrième chapitre présente les principales contributions de cette thèse. La première est l'implémentation d'un réseau ResNet18-Unet sur AMD Alveo U250, ce qui constitue une première étape dans le défi de l'implémentation d'un réseau de segmentation sémantique avec un flux de données pipeliné. Cette contribution a été publiée lors d'une conférence internationale. Ensuite, l'auteur présente une amélioration de la configuration d'entraînement d'ENet, un réseau de segmentation sémantique léger, et son implémentation sur le FPGA AMD ZU19EG. Cette implémentation atteint un débit et une latence compatibles avec l'inférence en temps réel et a été améliorée grâce à une exploration de l'espace de conception des architectures de réseaux de neurones. Le modèle final est moins complexe que le modèle initial et conserve sa précision, avec une très légère baisse lors de la quantification INT4. La mise en œuvre de ce modèle montre des résultats supérieurs en termes de mIoU, de débit et de latence par rapport aux implémentations précédentes sur FPGA. De plus, une comparaison avec les GPU intégrés met en évidence une efficacité énergétique 1,15 fois supérieure à celle d'un Nvidia Jetson AGX Xavier.

Pour une première tentative de déploiement d'un réseau de segmentation sémantique, une architecture de type UNet a été choisie. L'architecture UNet repose sur un encodeur et un décodeur symétriques, reliés par des connexions directes entre des couches associées. L'encodeur est basé sur ResNet18, préféré à ResNet50 pour son meilleur rapport précision/nombre de paramètres. Le décodeur UNet reconstruit les dimensions d'entrée grâce à des blocs appelés UnetBlocks, qui utilisent les sorties des couches de l'encodeur (ResBlocks) via des connexions latérales. Le suréchantillonnage est effectué par interpolation aux plus proches voisins suivie d'une convolution. Le modèle final compte 14,3M de paramètres.

Avec les paramètres d'entraînement choisis, le réseau atteint 68% de mIoU sur Cityscapes. Afin de faciliter l'implémentation sur FPGA, nous avons appliqué une quantification des poids et des activations du réseau, en utilisant une méthode de QAT (*Quantization-Aware Training*). En expérimentant avec différents niveaux de quantification, nous avons déterminé que le meilleur compromis se trouvait dans l'utilisation de 4 bits pour les poids et les activations, sauf pour la dernière et la première couche qui sont mises sur 8 bits. Ce schéma de quantification permet d'atteindre près de 63% de mIoU.

Lors de l'implémentation avec FINN, nous avons dû insérer des FIFOs de très grandes tailles dans le réseau à cause des longues connexions entre l'encodeur et le décodeur afin d'éviter tout blocage des données, ce qui a augmenté considérablement la quantité de mémoire nécessaire pour l'implémentation du réseau. En effet, l'architecture finale utilise près de 80% des BRAM de l'Alveo U250. Cette implémentation, en utilisant des images de résolution 256x256, atteint 23 IPS (images par seconde) et une latence de 44 ms par image avec une horloge à 134 MHz.

Cette première expérience a produit des résultats encourageants en faveur des architectures de flux de données en pipeline, montrant qu'elles peuvent être utilisées pour implémenter des réseaux de segmentation sémantique. Elle a permis d'acquérir une meilleure compréhension de FINN, qui, malgré ses limites, offre des outils pour dépasser ces contraintes et intégrer de nouveaux éléments pour améliorer le support des réseaux complexes. L'expérience a également fourni des enseignements précieux sur les architectures de flux de données en pipeline, notamment le rôle des buffers et les défis liés à l'ajout d'un raccourci entre l'encodeur et le décodeur, une modification mineure en logiciel mais problématique en matériel. Bien que des améliorations aient été identifiées pour potentiellement répondre aux exigences des applications en temps réel, il est apparu que ce réseau serait difficilement performant en raison des longues connexions et du nombre de paramètres élevés selon les standards FPGA.

Cela nous a amené vers notre seconde contribution qui concerne l'implémentation d'ENet, un réseau de type encoder-decoder composé de multiples blocs inspirés par les blocs "bottlenecks" des ResNets, et publié par A.Paszke en 2016. Malgré son nombre important de couches, ENet reste compact avec seulement 0,37 million de paramètres, atteignant une mIoU de 58,3% sur Cityscapes et 51,3% sur CamVid.

ENet est l'un des plus anciens modèles de segmentation sémantique en temps réel, et depuis sa publication en 2016, plusieurs modèles ont émergé qui le surpassent en termes de précision de segmentation. Cependant, dans la littérature sur l'apprentissage profond, il existe souvent un biais où les architectures de réseaux de neurones sont comparées tout en modifiant les routines d'entraînement, et les gains de performance sont attribués exclusivement à l'architecture. Dans le domaine de la classification d'images, cela a été clairement démontré que, avec des techniques d'entraînement appropriées, les réseaux convolutionnels peuvent rivaliser avec les transformers, souvent considérés comme supérieurs en raison de leur architecture. Nous avons constaté une tendance similaire dans nos expériences sur les réseaux de segmentation sémantique. Ainsi, un ENet bien entraîné peut offrir d'excellentes performances tout en restant extrêmement léger.

En effet, l'optimisation de l'entraînement d'ENet grâce à des hyperparamètres améliorés et des techniques efficaces d'augmentation des données augmente significativement sa précision, permettant d'atteindre 70,5% de mIoU sur Cityscapes et 77,6 % sur CamVid, tout en améliorant davantage son ratio performance/paramètres. Cela rend ENet encore plus attractif pour les systèmes embarqués aux ressources limitées. Ces résultats soulignent l'importance d'une procédure d'entraînement adaptée et montrent que le choix d'un réseau neuronal ne devrait pas se baser uniquement sur les précisions rapportées par les auteurs. Ces performances initiales peuvent davantage refléter la configuration d'entraînement que l'architecture elle-même. Finalement, opter pour un réseau de neurones léger n'est pas aussi restrictif pour les performances en segmentation qu'il pourrait sembler à première vue.

Afin d'implémenter ENet avec FINN, nous avons dû apporter des modifications à la structure du réseau, pour prendre en compte les spécificités de l'outil, ainsi que les leçons apprises de l'implémentation de ResNet-UNet concernant le lien entre structure du réseau et performances matérielles. Ces modifications permettent l'implémentation, mais augmentent la complexité du réseau en terme de nombre de paramètres et d'opérations. Nous avons également appliqué la même méthode de quantification que pour le ResNet18-Unet. Le réseau atteint 69,75% de mIoU sur Cityscapes en quantifié. L'architecture implémentée sur le FPGA AMD ZU19EG utilise environ 60% des LUT et des BRAM et permet d'atteindre un débit de 47,5 IPS et une latence à 21 ms avec une

horloge à 250 MHz, en utilisant des images d'une résolution de 512x256 pixels. Ces performances sont satisfaisantes et sont compatibles avec une segmentation en temps réel.

Après avoir obtenu une implémentation d'un modèle capable de fonctionner sur notre cible avec des performances acceptables, nous voulions examiner si la modification d'éléments de l'architecture pourrait conduire à un meilleur équilibre entre les performances et les ressources, ce que nous avons fait en explorant l'espace de conception. Ceci a permis de mettre en lumière que remplacer les convolutions transposées par une interpolation aux plus proche voisin suivie éventuellement d'une convolution était très bénéfique pour l'équilibre entre performances et complexité puisque, sans introduire de baisse de la mIoU, cela permet de considérablement réduire le nombre de paramètres et d'opérations du réseau, le rendant légèrement moins complexes que l'original, et permettant d'utiliser moins de ressources du FPGA pour son implémentation.

Cette version du réseau, que nous avons nommé P-ENet, a été entraînée à nouveau afin d'améliorer ses performances jusqu'à 70,3% de mIoU sur Cityscapes. Cette architecture de réseau a ensuite été déclinée en deux implémentations matérielles. La première, P-ENet_S, atteint 126 IPS et une latence moyenne de 8 ms, tandis qu'une version utilisant un plus grand parallélisme et donc plus de ressources, appelée P-ENet_M, atteint 226 IPS et 4,4 ms de latence. Une version adaptée à la résolution d'images de CamVid, appelée P-ENet_C a également été réalisée, atteignant 175 IPS et une mIoU à 76,26%. Ces performances surpassent celles de l'état de l'art des implémentations de réseaux de segmentation sémantique sur FPGA, que cela soit sur Cityscapes ou CamVid. Cependant, ces performances sont accompagnées d'une utilisation de ressources plus importantes que la plupart des travaux précédents.

Ce chapitre fini sur une comparaison des performances avec deux GPUs embarqués, la carte Jetson AGX Xavier et la carte Jetson Orin Nano. Les mesures effectuées indiquent que notre implémentation sur FPGA atteint des performances comparables avec celles des GPUs, tout en étant légèrement plus efficace énergétiquement que la Jetson AGX Xavier. Toutefois, le FPGA n'atteint pas l'efficacité énergétique de la Jetson Orin Nano, celle-ci étant fabriquée à partir d'une technologie de transistors bien plus récente.

Co-conception de la méthode d'échantillonnage

Ce cinquième et dernier chapitre présente une étude de la co-conception des méthodes de suréchantillonnage dans les réseaux de neurones convolutifs, réalisée en partenariat avec AMD. L'objectif est d'identifier la méthode de suréchantillonnage optimale pour les implémentations sur FPGA en utilisant une architecture de flux de données optimisée avec FINN. Les recherches antérieures menées sur GPU ont montré que les méthodes comme la convolution sous-pixel offrent une meilleure convergence pendant l'entraînement, tandis que la convolution transposée est plus efficace en termes d'énergie et de mémoire pendant l'inférence. Cependant, ces études ont été menées en précision flottante. Cette étude examine si ces observations sont valables dans le domaine quantifié et sur des architectures FPGA.

Pour ce faire, le modèle ESPCN, conçu pour la super-résolution, a été choisi comme base. Ce modèle, léger et simple, facilite les tests sur FPGA et l'expérimentation avec différents paramètres de quantification et facteurs de mise à l'échelle. Bien que moins performant pour des applications avancées de super-résolution, ESPCN est pertinent pour des systèmes embarqués à faible consommation.

L'entraînement du modèle ESPCN a été réalisé sur le jeu de données BSD300 en utilisant soit une convolution sous-pixel (originale au modèle), une convolution transposée ou une interpolation au plus proche voisin suivie d'une convolution (NNRC) pour suréchantillonner les images. Chaque méthode a été testée avec des facteurs d'agrandissement (α) de 2, 3 et 4, sur des quantifications en 3, 4, et 8 bits ainsi qu'en virgule flottante (32 bits). Une quantification de type QAT a été utilisée pour ajuster les poids et les activations, avec des stratégies de quantification uniformes par canal ou par tenseur selon les couches.

Les résultats indiquent qu'aucun algorithme ne surpasse systématiquement les autres. Toutefois, la méthode NNRC montre un léger avantage pour $\alpha = 2$, tandis que la convolution transposée est meilleure pour $\alpha \geq 3$. La convolution sous-pixel reste compétitive mais ne dépasse pas les autres méthodes.

Nous avons également exploré l'impact des conversions des méthodes de suréchantillonnage entre l'entraînement et l'inférence sur FPGA :

- NNRC convertie en convolution transposée (D-NN).
- Convolution sous-pixel convertie en convolution transposée (D-SP).
- Convolution transposée native (TC).

Des expériences ont été effectuées avec des facteurs d'agrandissement de 2, 4 et 8. Pour $\alpha = 2$, les tests ont été réalisés sur une carte FPGA Kria KV260. Pour des facteurs $\alpha = 4$ et $\alpha = 8$, les tests ont été déplacés sur une carte FPGA ZCU104 pour surmonter les limites de ressources, et le suréchantillonnage a été effectué en utilisant une ou plusieurs couches. Les résultats montrent que la convolution NNRC consomme moins de ressources que les méthodes converties. D-SP et TC consomment des ressources similaires, mais restent moins efficaces que NNRC. Pour les facteurs $\alpha = 4$ et $\alpha = 8$, les observations indiquent que le suréchantillonnage divisé sur plusieurs couches est plus efficace en termes de consommation de ressources et de performances.

Les implémentations actuelles de convolution transposée sur FPGA utilisent une approche appelée fractionally-strided convolution. Cependant, cette méthode produit des pixels nuls dans les cartes de caractéristiques, entraînant des calculs inutiles. Pour améliorer cette inefficacité, des alternatives comme REVD (Reverse Looping Deconvolution) et REVD2 ont été proposées. Ces algorithmes évitent les pixels nuls en travaillant directement dans l'espace des poids ou en optimisant les indices pour réduire les opérations redondantes. REVD2, en particulier, s'est révélé plus rapide et économe en énergie dans des modèles analytiques GPU et sera intégré dans FINN pour de futures expérimentations.

Conclusion

Les réseaux de neurones profonds dominent de nombreuses tâches complexes de vision par ordinateur, notamment la segmentation sémantique, qui est cruciale pour des applications comme les véhicules autonomes et les drones. Ces tâches nécessitent une exécution rapide, efficace et à faible consommation énergétique. Cependant, les GPU traditionnels, bien qu'efficaces pour l'entraînement, ne conviennent pas toujours à ces contraintes dans des environnements embarqués. Les FPGA, par leur flexibilité et leur efficacité énergétique, représentent une alternative prometteuse. Toutefois, leur implémentation est parfois difficile à cause des flots de conception complexes et des ressources matérielles limitées. Ces défis nécessitent des modèles de réseaux adaptés, équi-

librant performance et efficacité matérielle. Cette thèse démontre que la structure du réseau joue un rôle essentiel dans l'optimisation des architectures matérielles pour répondre à ces exigences.

Pour répondre à ces défis, cette thèse s'est appuyée sur l'outil open-source FINN, conçu pour développer des architectures matérielles à faible latence adaptées aux réseaux de neurones. Des contributions spécifiques ont été apportées à FINN, notamment des transformations pour faciliter le déploiement de réseaux à structure résiduelle et l'intégration des convolutions transposées. Ces ajouts ont permis une meilleure prise en charge des réseaux de segmentation sémantique et ont jeté les bases des expérimentations présentées.

Une première application de ces travaux a porté sur le réseau ResNet18-UNet, une architecture encodeur-décodeur typique pour la segmentation sémantique. Les résultats obtenus ont mis en lumière le rôle des éléments structurels tels que les connexions directes entre l'encodeur et le décodeur, qui peuvent introduire des problèmes matériels non anticipés. Ces observations soulignent l'importance d'une conception soignée des réseaux des neurones, en tenant compte des contraintes matérielles.

Dans une seconde application, la thèse a exploré ENet, un modèle de segmentation conçu pour l'inférence en temps réel. Ce réseau, initialement moins performant que l'état de l'art, a bénéficié d'améliorations substantielles grâce à de nouveaux paramètres d'entraînement et à des techniques d'augmentation de données. Ces ajustements ont permis d'augmenter la précision d'ENet de plus de 12 % sur le jeu de données Cityscapes et de 26 % sur CamVid, démontrant que des réseaux légers bien entraînés peuvent rivaliser avec des modèles plus complexes.

Enfin, une exploration orientée matériel a permis de concevoir P-ENet, une version optimisée d'ENet adaptée aux contraintes des FPGA. Cette architecture atteint une vitesse d'exécution de 226 FPS avec une latence de seulement 4,2 ms, tout en conservant une précision proche de l'état de l'art (70,3 % de mIoU en quantification INT4). Cette conception met en évidence l'importance de repenser les architectures de réseaux en fonction des exigences matérielles, plutôt que d'appliquer simplement des modèles complexes avec des taux de compression élevés.

L'évaluation des performances énergétiques a révélé que les FPGA sont généralement plus efficaces que les GPU embarqués comme le Jetson AGX Xavier, bien qu'ils restent en retrait par rapport aux GPU de dernière génération tels que le Jetson Orin Nano. Cependant, les FPGA offrent une plus grande flexibilité dans l'exploration des compromis entre puissance et précision, notamment grâce à des options de quantification variées et des conceptions adaptées à différents cas d'utilisation. De plus, leur capacité à s'intégrer directement aux capteurs, sans interfaces logicielles ou matérielles supplémentaires, réduit la latence et simplifie la conception des systèmes embarqués.

Ainsi, cette thèse souligne l'importance d'une co-conception du matériel et des réseaux de neurones pour optimiser les performances des systèmes embarqués. Les résultats obtenus démontrent que des architectures adaptées peuvent rivaliser avec des solutions plus complexes tout en offrant une efficacité supérieure dans des contextes contraints.

Résumé

Au cours de la dernière décennie, l'apprentissage automatique a révolutionné l'informatique, avec un rôle central de l'apprentissage profond dans l'avancement de domaines tels que la classification d'images, la détection d'objets et le traitement du langage naturel. Bien que les réseaux de neurones profonds aient démontré des performances exceptionnelles, leur déploiement dans des systèmes à ressources limitées tels que les véhicules autonomes et les drones présente des défis importants. Ces systèmes nécessitent des solutions à faible latence et à faible consommation d'énergie, et bien que les Graphics Processing Units (GPUs) soient efficaces pour les tâches à grande échelle, ils ne sont pas optimisés pour ces environnements. Cette thèse explore l'implémentation de modèles d'apprentissage profond, en particulier la segmentation sémantique, sur des Field-Programmable Gate Arrays (FPGAs), qui sont plus adaptés aux applications embarquées en raison de leurs capacités de traitement parallèle et de leur faible consommation d'énergie. Cependant, les FPGAs présentent des limitations en termes de ressources de calcul et de mémoire, faisant du déploiement de modèles complexes un véritable défi. Bien que les techniques de compression réduisent l'empreinte mémoire et computationnelle, elles peuvent dégrader la précision des modèles, nécessitant un équilibre soigneux entre les contraintes matérielles et les performances algorithmiques. Par ailleurs, cela soulève la question de si les architectures de réseaux neuronaux existantes, souvent choisies pour leurs performances spécifiques à une tâche, sont intrinsèquement adaptées au déploiement sur FPGA. En utilisant l'outil open-source FINN, nous générons des architectures flux de données adaptées aux FPGA, en nous concentrant sur l'obtention d'un débit élevé et d'une faible latence pour les tâches de segmentation sémantique.

Cette thèse contribue au déploiement des réseaux de neurones profonds sur FPGAs en améliorant l'outil FINN avec de nouveaux éléments, permettant un meilleur support des tâches de segmentation sémantique. L'implémentation de deux modèles encodeur-décodeur constitue le cœur de notre travail. Premièrement, nous avons implémenté un modèle ResNet18-UNet, révélant des informations sur les architectures de flux de données en pipeline, notamment le rôle des mémoires tampons et les défis matériels liés à des éléments structurels comme les raccourcis entre l'encodeur et le décodeur. Cela a motivé la recherche d'optimisations conjointes des architectures de réseaux neuronaux et du matériel, ainsi que des considérations minutieuses sur la structure des modèles, ce qui nous a conduit à notre deuxième contribution, l'implémentation d'ENet, un réseau à faible complexité conçu pour l'inférence en temps réel. En optimisant sa configuration d'entraînement, y compris l'augmentation de données, nous avons amélioré ses performances de segmentation de plus de 12% sur le jeu de données Cityscapes et de 26% sur CamVid. Cela a démontré que des réseaux plus simples, avec un entraînement approprié, peuvent obtenir des résultats compétitifs par rapport à des architectures plus complexes. De plus, des modifications structurelles nous ont permis de répondre aux exigences d'inférence en temps réel sur FPGA, atteignant 47 FPS avec une latence de 21

ms, même avec une quantification en INT4, tout en conservant la précision du modèle. Nous avons développé une version optimisée, P-ENet, grâce à une exploration de l'espace de conception en prenant en compte les performances matérielles. P-ENet atteint 70,3% de mean Intersection over Union (mIoU) avec une quantification en INT4, seulement 0,2% en dessous de la référence, mais réduit encore davantage les paramètres et la complexité computationnelle du modèle, permettant son implémentation sur FPGAs pour atteindre 226 FPS avec une latence de seulement 4,2 ms. Cela démontre que se concentrer sur la co-conception de l'architecture de réseau de neurones et de l'architecture matérielle fournit un meilleur compromis entre débit et précision, par rapport à l'application simple de la compression sur des réseaux performants. Les comparaisons de puissance et d'énergie des FPGAs par rapport aux GPUs embarqués ont montré que les FPGAs offrent une efficacité énergétique 1,15 fois supérieure aux GPUs embarqués tels que la Jetson AGX Xavier. De plus, la flexibilité des FPGAs, en particulier en matière de quantification et d'optimisation de l'utilisation d'énergie, les rend attrayants pour les applications à faible consommation et haute efficacité. Enfin, notre dernière contribution, l'étude des méthodes de suréchantillonnage d'images dans les tâches de super-résolution, met en évidence la supériorité de la convolution redimensionnée par le plus proche voisin pour l'inférence sur FPGA et démontre également que les performances des réseaux de neurones sur GPUs ne se transfèrent pas toujours directement sur FPGAs, renforçant ainsi l'importance d'une conception spécifique au matériel.

Abstract

In the last decade, machine learning has revolutionized computer science, with deep learning playing a pivotal role in advancing fields like image classification, object detection, and natural language processing. While deep neural networks have demonstrated outstanding performance, deploying them in resource-constrained systems such as autonomous vehicles and drones presents significant challenges. These systems require low-latency, power-efficient solutions, and common GPUs, though effective for large-scale tasks, are not optimized for such environments. This thesis investigates the implementation of deep learning models, specifically semantic segmentation, on FPGAs, which are more suitable for embedded applications due to their parallel processing capabilities and lower power consumption. However, FPGAs come with limitations in computational and memory resources, making the deployment of complex models a challenging task. While compression techniques reduce memory and computational footprints, they may degrade model accuracy, requiring a careful balance between hardware constraints and algorithmic performance. On the other hand, this work questions whether existing neural network architectures, typically selected for their task-specific performance, are inherently adapted for FPGA deployment. Using the open-source FINN framework, we generate dataflow architectures tailored for FPGA, focusing on achieving high throughput and low latency for semantic segmentation tasks.

This thesis contributes to the deployment of deep neural networks on FPGAs by enhancing the open-source FINN framework with new elements, enabling better support for semantic segmentation tasks. The implementation of two encoder-decoder models is central to our work. First, we implemented a ResNet18-UNet model, revealing insights into pipelined dataflow architectures, particularly the roles of buffers and hardware challenges caused by structural elements like shortcuts between the encoder and decoder. As it motivates the search for joint optimizations of neural network and hardware architectures and careful considerations of model structures, it led to our second contribution, the implementation of ENet, a low-complexity network designed for real-time inference. By optimizing its training setup, including data augmentation, we improved its segmentation performance by over 12% on the Cityscapes dataset and 26% on CamVid. This demonstrated that simpler networks, with appropriate training, can achieve competitive results against more complex architectures. Further, structural modifications allowed us to meet real-time inference requirements on FPGA, achieving 47 FPS with 21 ms latency, even with INT4 quantization, while retaining model accuracy. We developed an optimized version, P-ENet, through hardware-aware design space exploration. P-ENet achieves 70.3% mIoU with INT4 quantization, only 0.2% below baseline, but further reduces the model's parameters and computational complexity, and allows its implementation on FPGA to reach 226 FPS with just 4.2 ms latency. This demonstrates that focusing on co-designing the neural network and hardware architecture provides a better trade-off between throughput and accuracy compared to

simply applying compression to high-performing networks. Power and energy comparisons of FPGAs versus embedded GPUs showed that FPGAs offer a 1.15 times superior power efficiency over embedded GPUs like the Jetson AGX Xavier. Moreover, the flexibility of FPGAs, particularly in quantization and power-usage optimization, makes them attractive for low-power, high-efficiency applications. Additionally, our last contribution, the study of image upsampling methods in super-resolution tasks, highlights the superiority of nearest-neighbor resized convolution for FPGA inference and further demonstrates that neural network performance on GPUs does not always translate directly to FPGAs, reinforcing the importance of hardware-specific design.

Introduction

Preamble

The research work in this thesis was carried out in the Mathematical and Electrical Engineering department of IMT Atlantique in Brest, France. It was partly funded by French government support granted to the Labex CominLabs excellence laboratory and managed by the National Research Agency in the “Investing for the Future” program under reference ANR-10-LABX-07-01. In this program, the project associated to this work is LEASARD (Low-Energy Deep Neural Networks for Autonomous Search-And-Rescue Drones).

It also led to a collaboration with the Integrated Communications and AI Lab of AMD Research & Advanced Development, including a 6-month research internship at AMD Dublin, Ireland.

Context

The past decade has seen the rise in popularity of machine learning as a new paradigm that has revolutionized computer science. While the classical approach of defining explicit algorithmic solutions fails to provide satisfactory answers to several very complex or ill-posed problems, machine learning introduces the method of training an algorithm through examples to find an implicit solution. Deep learning, a subset of machine learning, has obtained particular attention since the breakthrough made by AlexNet [1] in the 2012 ImageNet Large Scale Visual Recognition Challenge [2], by becoming the first deep learning algorithm to win the image classification contest. Since then, deep learning has become predominant in several fields such as machine vision, audio recognition, image generation, and natural language processing. Although the origins of deep learning lie in the 1950s, its recent success is mainly explained by the easier access to large quantities of data, as well as advances in computing hardware through the popularisation of GPUs, of which the great parallel processing capabilities are found to be extremely beneficial to accelerate operations performed in deep learning algorithms.

A significant proportion of research done on deep learning seeks to leverage an ever-increasing amount of computational power in the attempt to train more computationally intensive algorithms to achieve the highest score possible on benchmarks. Meanwhile, part of it focuses on answering some of the main challenges faced by deep learning, such as training with only a few examples [3], [4], robustness to noisy inputs [5], decentralization [6], interpretability [7] or deployment in heavily constrained systems. The latter is of particular interest for embedded systems like autonomous vehicles, drones, or robotics in general, as they benefit from the potency of deep learning to infer decisions based on sensor information through solving tasks such as object detection, semantic segmentation, and Simultaneous Localization And Mapping (SLAM),

but require the processing to be done within a certain latency and with limited power consumption. In the context of autonomous vehicles for example, these constraints can be associated with requiring the decision to be made within milliseconds¹ to ensure a fast reaction of the vehicle to the environments and needing the system's power consumption to remain below certain thresholds to ensure a sufficient battery autonomy. However, deep learning algorithms, referred to as deep neural networks, were not initially designed to be particularly efficient on these metrics, and deploying them on embedded systems is a challenge not only from an algorithmic optimization point-of-view but also from a hardware one, as the high computational power of conventional GPUs comes at the cost of high energy consumption, making them inadequate devices for such power-constrained systems.

Therefore, GPU manufacturers, and particularly Nvidia, have designed a new family of GPU boards, referred to as embedded GPUs, which are specifically meant for running trained deep learning models in systems with low latency and power consumption requirements. While they have the advantage of providing the same ease of use and a straightforward design flow for programmers as regular GPUs, they are challenged by another type of hardware device, the FPGAs. Indeed, FPGA implementations have proven to be more efficient for embedded machine vision tasks, especially regarding power consumption [8]. These reconfigurable devices are, by design, very adequate for parallel processing, and as such, are widely used to accelerate digital signal processing algorithms, making them appropriate for deep learning. Moreover, they are more suitable from a system integration perspective, enabling a direct interface to the sensors without requiring several additional software stacks or hardware buses. On the other hand, they have relatively low computational and memory resources and traditionally require the use of complex design flows, which makes the implementation of very intensive algorithms like deep learning models particularly challenging. Fortunately, years of research in hardware architecture design have led to the creation of tools that automatize the design and implementation of hardware architectures for deep learning on FPGA, significantly reducing the development time.

This allows designers to refocus their efforts on algorithmic optimizations to find solutions to the challenge of implementing deep learning models on FPGA of achieving great performance in terms of throughput and latency while maintaining the correctness of the algorithm. Reducing the computation and memory footprints of deep neural networks can be done through the use of compression techniques, leading to lower hardware resource requirements, power consumption, and, to some extent, latency. For high compression rates, this algorithmic complexity reduction can lead to a degradation of the accuracy of the neural network on the task it was trained for. The goal is then to find the best balance between hardware and algorithmic performances, depending on the requirements of the system. Minimizing latency, hardware resources usage, and power consumption can be more crucial than maximizing accuracy. On the other hand, if the requirements of the system are met, improving the accuracy of the model within these requirements is also of great interest.

However, a less studied aspect of the problem is that, when considering the implementation of a deep learning model in a resource-constrained system and attempting to achieve certain throughput and latency requirements, leveraging compression techniques is not the only way to explore the trade-off between accuracy and hardware performance. In fact, the question that may arise is whether the chosen neural network's design is the most adequate in the first place, and if selecting a model only

¹For instance, a car driving at 50 km/h only has 70 ms to detect and dodge an obstacle standing a meter away from it.

based on its performance on the task is appropriate. The relationships between the algorithmic design of the neural network and the achievable performance in terms of accuracy, throughput, and latency in the context of FPGA implementation are not trivial. Studying these relationships is precisely the subject of this thesis.

We choose to conduct this study on neural networks trained for the machine vision task of semantic segmentation, as it is known for employing complex neural network structures and requiring the computation of high-resolution inputs. Thus, their implementation on FPGA is particularly challenging and not as thoroughly studied as other more popular machine vision tasks, which makes them a particularly interesting research topic to investigate. Moreover, it is a critical task for Search-And-Rescue drones, as outlined in the LEASARD project funding this thesis. We decided to make use of the open-source framework FINN [9], [10] to generate dataflow architectures for FPGA which are tailored for the neural network to be implemented and are designed to yield high throughput and low latency. We take advantage of this tool to study the implementation of two semantic segmentation models and conduct a hardware-aware exploration of the neural network design space. As autonomous cars are a popular application of embedded semantic segmentation deep neural networks with heavy latency and power constraints, we conduct this study using neural networks trained and evaluated on datasets of urban driving scenes. However, the results and findings of this thesis can be transposed to other types of embedded systems that take advantage of semantic segmentation with a demand for low latency, such as Unmanned Aerial Vehicles (UAVs). Since image upsampling is an essential part of the neural networks we use in this thesis, we also study the co-design of upsampling methods and hardware architectures using the context of the super-resolution task to carry out our experiments.

Outline and contributions

This manuscript is organized into five chapters. Chapter 1 outlines the fundamental principles of deep learning, explaining the construction and training of algorithms, and presenting the different operators utilized in this thesis. It also covers compression techniques designed to lower the computational and memory requirements of deep learning, thereby facilitating their deployment on devices with limited resources and power constraints. Additionally, it introduces semantic segmentation, the task we selected as challenging for hardware implementation, and examines the current leading models dedicated to this task. Lastly, it provides a concise overview of super-resolution, which is the secondary task involving upsampling used in our experiments.

Chapter 2 delves into the subject of implementation of neural networks on FPGAs. It offers an introductory overview of FPGA circuits, including the various programming techniques employed in this domain. Following this, the chapter elaborates on the benefits of utilizing FPGAs for deep neural network inference, discussing their advantage compared to GPUs. Additionally, the chapter addresses the complexities and hurdles associated with deploying neural networks on FPGAs, showcasing several methods and tools designed to ease this deployment process, and details the reasons leading to the selection of FINN as the tool used for our experiments in this thesis. Finally, it provides a review of prior research focused on implementing semantic segmentation networks for real-time applications that benchmark their performance on one of the two urban road scene datasets employed in this thesis.

In Chapter 3, an overview of the fundamental concepts behind FINN is presented to offer an understanding of its practical applications. This comprehensive introduction

sets the stage for discussing the obstacles and uncovered constraints, which were hindering the effective implementation of semantic segmentation networks studied in this thesis. Proposed solutions to address these challenges are described, which involve integrating new components into the tool, thereby enhancing its support for semantic segmentation networks. These open-source contributions are officially incorporated into the FINN project to help other developers overcome similar challenges they may encounter.

Chapter 4 presents the main contributions of this thesis, starting with the implementation of a ResNet18-UNet neural network on FPGA, giving a first insight into how the structure of neural networks influences the performance of dataflow hardware architectures. The second contribution is the methodology followed to improve and implement ENet, a low-complexity semantic segmentation network. Although appearing to perform worse than most state-of-the-art models, we proposed major improvements so that its initial accuracy is increased up to a more competitive standard by employing a better training setup, highlighting that the comparison of models based on their reported accuracy is skewed by the differences in training setups. The hardware architecture obtained from this model is able to meet the throughput and latency requirements of real-time applications. Through an exploration of the neural network design space, we investigate the influence of structural changes on the algorithmic and hardware performances, selecting the most efficient design. This results in an implementation outperforming the state of the art of semantic segmentation networks for autonomous driving on FPGA. This chapter is concluded by a detailed comparison of the performance of the FPGA design against two embedded GPUs, showing a better power efficiency than the GPU of the Nvidia Jetson AGX Xavier, fostering discussions on the role of FPGAs for embedded deep learning, highlighting their interest in exploring the design space to achieve power-efficient solutions.

Finally, Chapter 5 presents our last scientific contribution, which focuses specifically on image upsampling methods in deep learning, and studies the relationship between the algorithm used and the hardware performance. ESPCN, a lightweight super-resolution network is used as a baseline to carry out this study. The results obtained identify one method as outperforming the others. Interestingly, this study leads to different outcomes than a prior one made on GPU and highlights the importance of considering the core differences between these hardware devices when designing neural networks.

The research conducted in this thesis led to publications and scientific communications:

- Peer-reviewed publications:
 - H. Younes, H. Le Blevec, M. Léonardon, V. Gripon, "Inter-Operability of Compression Techniques for Efficient Deployment of CNNs on Microcontrollers" in *International Conference on System-Integrated Intelligence (SYSINT)*, 2022 [11].
 - H. Le Blevec, M. Léonardon, H. Tessier and M. Arzel, "Pipelined Architecture for a Semantic Segmentation Neural Network on FPGA," in *30th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2023 [12].
 - H. Le Blevec, M. Léonardon, H. Tessier and M. Arzel, "Co-conception d'un réseau de neurones convolutif de segmentation sémantique et de son architecture matérielle", in *29^{ème} Colloque sur le traitement du signal et des images, GRETSI*, 2023 [13].

- Communications:
 - H. Le Blevec, "Segmentation sémantique à base de réseaux de neurones profonds embarqués sur FPGA", in *IMT Webinar Data&IA*, 2022.
 - H. Le Blevec, "FINN Tutorial at FPL'22: Streamlining semantic segmentation networks with FINN", in *32nd International Conference on Field Programmable Logic and Applications (FPL)*, 2022.
 - H. Le Blevec, "FINN Tutorial at FPL'23: Semantic segmentation with FINN: ResNet18-UNet on Alveo U250", in *33rd International Conference on Field Programmable Logic and Applications (FPL)*, 2023.
 - H. Le Blevec, M. Léonardon, and M. Arzel, "Revisiting ENet for more efficient real-time embedded semantic segmentation on FPGA", *GdR SoC2*, 2024.
 - H. Le Blevec, M. Léonardon, and M. Arzel, "FINN Tutorial at FPL'24: Revisiting ENet for more efficient real-time embedded semantic segmentation on FPGA", in *34th International Conference on Field Programmable Logic and Applications (FPL)*, 2024 [14].

Chapter 1

Deep learning for semantic segmentation and super-resolution

Contents

1.1	Model building and training	34
1.1.1	Compositional approach	34
1.1.2	Loss function	35
1.1.3	Optimizer	35
1.1.4	Scheduler	36
1.2	Operators	36
1.2.1	Convolution layer	36
1.2.2	Fully-connected layer	37
1.2.3	Pooling layers	38
1.2.4	Upsampling layers	38
1.2.5	Activation layers	40
1.2.6	Batch normalization layer	41
1.3	Model compression	41
1.3.1	Quantization	42
1.3.2	Efficient convolutions	46
1.3.3	Knowledge distillation	49
1.3.4	Pruning	49
1.3.5	Neural Architecture Search	50
1.4	Semantic segmentation	50
1.4.1	Task definition, metrics and applications	50
1.4.2	Datasets	52
1.4.3	Models history overview	53
1.4.4	Real-time semantic segmentation	54
1.5	Super-resolution	57
1.5.1	Task definition and metrics	57
1.5.2	BSD dataset	57
1.5.3	Models overview	58
1.6	Summary	58

This chapter aims to provide the notions necessary to understand the task studied in this thesis. It is not meant to be exhaustive on the subjects presented, but to supply useful information related to the topic of this manuscript. We start with a general introduction to the basic concepts of deep learning and describe the elements that are used to train deep learning models. Then we define the list of operators, or layers, that are used to build deep neural networks. In the third section, we explore the topic of model compression techniques that can be used to reduce the computational and memory costs of deep learning, making them easier to deploy on devices with high resources and power constraints. Then, we present semantic segmentation, the main task of interest in this thesis, and delve into the state of the art of models designed for this task. Finally, we briefly explore super-resolution, a second task used in our experiments with upsampling, presented in the last chapter.

1.1 Model building and training

1.1.1 Compositional approach

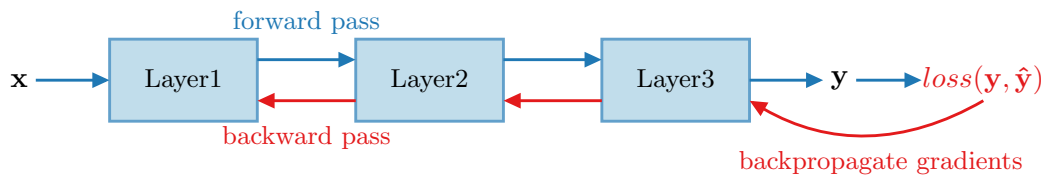


Figure 1: Illustration of the training process of deep neural networks. x is the input vector, y the prediction, and $\hat{y}$ the expected output.

Deep learning is a type of machine learning method that uses structures called artificial neural networks. These neural networks, or models, are a composition of mathematical functions called layers, some of which contain parameters that are meant to process abstract representations of data such as images, audio, or text in the form of multidimensional matrices called tensors. They can be trained to recognize patterns within the data to solve complex problems for which no explicit solution exists, like image classification or speech recognition. In the case of supervised learning, the training is an iterative process that involves using several examples from a dataset selected as meaningful for the learned task to provide predictions that can be compared against a predefined ground truth using a loss function to evaluate the accuracy of the prediction. The goal is then to gradually modify the parameters of the network to find a minimum for the loss function using an optimization algorithm. In this context, each step of processing an input and computing a loss is called the forward pass, while the parameter update is the backward pass. This process is repeated until the whole training dataset has been iterated over, forming an epoch. Note that other training methods, such as self-supervised learning, do not require labeled examples to train the model, but we do not make use of them in this manuscript. Once the training phase is completed, the trainable parameters are fixed, and the neural network can be used as a function to perform predictions on new data, which is called the inference phase.

We detail the loss functions and how the optimizer tunes trainable parameters in the following sections.

1.1.2 Loss function

This is the function used to calculate the error between the prediction and the labeled ground truth. Many loss functions exist, each with its own particularities and use cases, but we make use of two of them in this manuscript:

- The Mean Square Error: $MSE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$
- The Cross-Entropy: $CE(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{N} \sum_{i=1}^N \sum_{c=1}^C y_{i,c} \log(\hat{y}_{i,c})$

In both cases, $\mathbf{y}$ is the prediction, $\hat{\mathbf{y}}$ is the expected output, C is the number of object classes in the dataset, and N is the number of samples (or batches) of data processed in parallel.

1.1.3 Optimizer

1.1.3.1 Stochastic Gradient Descent

This is the optimization algorithm used during training to minimize the loss function. Several algorithms exist in the literature, one of the most popular being the Stochastic Gradient Descent (SGD) [15]. In short, at every iteration step t this algorithm computes stochastic approximations of the gradients $g_{i,t} = \frac{\delta \mathcal{L}}{\delta w_{i,t}}$ of the loss function $\mathcal{L}$ with respect to each parameter $w_{i,t}$ of the model and produces an update:

$$w_{i,t+1} = (1 - \lambda \eta_t) w_{i,t} + \alpha \Delta w_{i,t} - \eta_t g_{i,t} \quad (1)$$

where λ is the weight decay factor, η_t is the learning rate at iteration t , α is the momentum and $\Delta w_{i,t}$ is the parameter update performed during the previous iteration. Obtaining the gradient of the loss function with respect to each weight is done using a backpropagation algorithm. It starts from the gradient of the loss relative to the output $\frac{\delta \mathcal{L}}{\delta y}$, and using the chain rule to propagate the gradient through the network, computing the gradients with respect to the weights and inputs of each layer each time.

1.1.3.2 Adam

The Adam optimizer (Adaptive Moment Estimation) is also a widely used optimization algorithm. This method merges the benefits of two widely used optimization techniques, AdaGrad and RMSProp, by tailoring the learning rate for individual parameters. Adam monitors both the first moment (mean) and the second moment (uncentered variance) of the gradients, which enables dynamic adjustment of the learning rate. It employs two parameters, β_1 and β_2 , to control the decay rates of these moment estimates. Additionally, the learning rate is modified with a bias correction to offset the initial bias towards zero. For each iteration step t , the first moment estimate $m_{i,t}$ and second moment estimate $v_{i,t}$ are updated based on the gradient of the loss function with respect to the parameters $g_{i,t}$:

$$m_{i,t} = \beta_1 m_{i,t-1} + (1 - \beta_1) g_{i,t} \quad (2)$$

$$v_{i,t} = \beta_2 v_{i,t-1} + (1 - \beta_2) g_{i,t}^2 \quad (3)$$

Then, to mitigate the initial bias in the estimates of the moments, Adam applies bias correction and computes the corrected first moment estimate $\hat{m}_{i,t} = \frac{m_{i,t}}{1 - \beta_1^{i,t}}$ and corrected the second moment estimate $\hat{v}_{i,t} = \frac{v_{i,t}}{1 - \beta_2^{i,t}}$, where $\beta_1^{i,t}$ and $\beta_2^{i,t}$ are the bias correction

factors that gradually reduce the bias as the optimization progresses, and converge towards 0 as t increases. Finally, the weight update can be computed as :

$$w_{i,t+1} = w_{i,t} - \frac{\alpha \hat{m}_{i,t}}{\sqrt{\hat{v}_{i,t}} + \epsilon} \quad (4)$$

where ϵ is a small constant (typically $1e-8$) used for numerical stability.

1.1.4 Scheduler

This is a function that dictates the variations of the learning rate over the iterations, decreasing it over time. The collection of scheduling functions is vast, but we focus on the following two:

- The MultiStepLR scheduler: $\eta_{t+1} = \begin{cases} \gamma \eta_t & \text{if } t \in [T_0, \dots, T_N] \\ \eta_t & \text{otherwise} \end{cases}$ with $0 < \gamma \leq 1$ and $[T_0, \dots, T_N]$ a list of predefined epochs (also known as milestones) in which the learning rate is modified.
- The PolynomialLR scheduler: $\eta_{t+1} = \eta_t (1 - \frac{t}{T_{tot}})^p$ with $0 < p \leq 1$ and T_{tot} is the total number of iterations.

Additionally, an initial warm-up phase can be used in which case, for a certain number of iterations, the learning rate is increased up to its maximum value, before the decreasing of the scheduler starts.

1.2 Operators

This section provides mathematical definitions of the operators (or layers) that make up the deep neural networks studied in this manuscript. In the following, we call "activations" the inputs of the layers, whereas the trainable parameters are called "weights" if they are involved in multiplication with the inputs, or "biases" if they are added.

1.2.1 Convolution layer

Convolution layers are very common components of neural networks for machine vision, due to their ability to extract information from images. Neural networks that employ convolutions as their principal element are referred to as Convolutional Neural Networks (CNNs). It is inspired by convolutions in traditional signal processing but extends its principles. In a 2D convolution layer, the input is a three-dimensional tensor $\mathbf{x} = (x_{c,h,r}) \in \mathbb{R}^{C \times H \times R}$ where C is the number of input channels and H and R are, respectively, the height and width of the two-dimensional images (or feature maps) contained within this structure. The layer performs a discrete convolution between $\mathbf{x}$ and a four-dimensional weight tensor $\mathbf{w} = (w_{c',c,i,j}) \in \mathbb{R}^{C' \times C \times k_h \times k_r}$ where C' is the number of output channels and k_h and k_r are the dimensions of the convolution kernels, followed by addition with a bias vector $\mathbf{b} = (b_{c'}) \in \mathbb{R}^{C'}$. In the simplest case, the output tensor $\mathbf{y} = (y_{c',h',r'}) \in \mathbb{R}^{C' \times H' \times R'}$ is obtained by:

$$\forall c' \in [1, C'], \forall h' \in [1, H'], \forall r' \in [1, R']$$

$$y_{c',h',r'} = \sum_{c=1}^C \sum_{i=1}^{k_h} \sum_{j=1}^{k_r} x_{c,s_h \cdot h' + i - p_h - 1, s_r \cdot r' + j - p_r - 1} \cdot w_{c',c,i,j} + b_{c'} \quad (5)$$

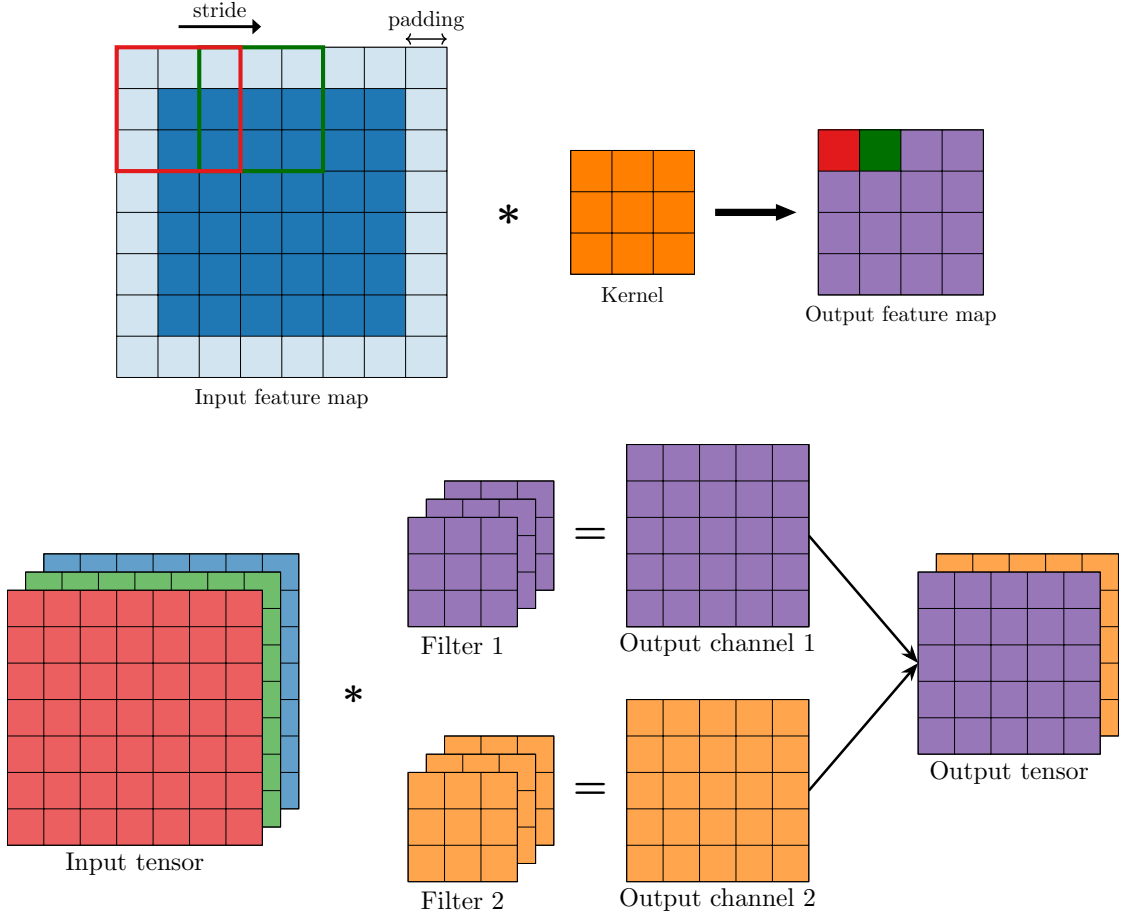


Figure 2: Illustration of convolution layers. Top: 2D perspective of the convolution of a 7×7 input with a 3×3 kernel with a padding of 1 and a stride of 2. Bottom: 3D perspective of the convolution of a $3 \times 7 \times 7$ tensor with a $2 \times 3 \times 3 \times 3$ weight tensor, assuming no padding and a stride of 1. $*$ is the convolution operator.

where p_h and p_r are the vertical and horizontal padding values, and s_h and s_r are the vertical and horizontal stride values. Padding and stride are additional parameters that can be used to manipulate the shape of the output. As illustrated in Figure 2, padding controls the amount of zeros that are put around the input feature maps, while the stride is defined by how many pixels the window of the input that is convoluted with the weight kernel is shifted on each iteration of the computation. The dimensions of the output feature maps can be computed as:

$$H' = \left\lceil \frac{H + 2p_h - (k_h - 1) - 1}{s_h} + 1 \right\rceil \quad (6)$$

$$R' = \left\lceil \frac{R + 2p_r - (k_r - 1) - 1}{s_r} + 1 \right\rceil \quad (7)$$

1.2.2 Fully-connected layer

In a fully-connected layer (also known as linear layer), each output activation is connected to all the input ones. It is among other things, the central piece of a classifier, a part of classification networks, responsible for providing the prediction from the features extracted by the convolutions. The input is a vector $\mathbf{x} \in \mathbb{R}^C$, where C to which a weight matrix $\mathbf{w} \in \mathbb{R}^{C \times C'}$ is multiplied, before being added to a bias vector $\mathbf{b} \in \mathbb{R}^{C'}$.

The output is therefore a vector $\mathbf{y} \in \mathbb{R}^{C'}$ such that:

$$\forall c' \in \llbracket 1, C' \rrbracket, y_{c'} = \sum_{c=1}^C x_c w_{c,c'} + b_{c'} \quad (8)$$

1.2.3 Pooling layers

Although convolutions can be used to reduce the dimensions of feature maps, it can sometimes be beneficial to do it without involving costly computations and learnable parameters. This is the purpose of pooling layers, and include two variants.

Average pooling

This layer averages the values of the input tensor $\mathbf{x} \in \mathbb{R}^{C \times H \times R}$ within a defined $k_h \times k_r$ window with a vertical stride s_h and horizontal stride s_r , producing an output tensor $\mathbf{y} \in \mathbb{R}^{C \times H' \times R'}$ such that:

$$\begin{aligned} &\forall c \in \llbracket 1, C \rrbracket, \forall h' \in \llbracket 1, H' \rrbracket, \forall r' \in \llbracket 1, R' \rrbracket \\ y_{c,h',r'} &= \frac{1}{k_h k_r} \sum_{i=0}^{k_h-1} \sum_{j=0}^{k_r-1} x_{c, s_h \cdot h' + i, s_r \cdot r' + j} \end{aligned} \quad (9)$$

Max pooling

This variant computes the maximum of values within the window rather than the average, such that:

$$\begin{aligned} &\forall c \in \llbracket 1, C \rrbracket, \forall h' \in \llbracket 1, H' \rrbracket, \forall r' \in \llbracket 1, R' \rrbracket \\ y_{c,h',r'} &= \max_{i \in \llbracket 0, k_h-1 \rrbracket} \max_{j \in \llbracket 0, k_r-1 \rrbracket} x_{c, s_h \cdot h' + i, s_r \cdot r' + j} \end{aligned} \quad (10)$$

1.2.4 Upsampling layers

Several machine vision tasks require an upsampling of feature maps, whether it is to increase the resolution of the initial input or to recover its original dimensions in the event that the image has been downsampled by previous layers in the network. The methods commonly used in neural networks to perform upsampling can be organized into two categories.

1.2.4.1 Interpolation-based upsampling

Interpolation-based upsampling techniques use two-dimensional interpolation functions to infer new values from nearby ones, usually using nearest-neighbor interpolation (direct reproduction of the closest pixel), bilinear, or bicubic interpolation, the two latter being, respectively, linear or cubic interpolation on each of the axes of the feature map. Another upsampling method that can be associated with this category is max unpooling, which is a partial inverse of max pooling. It uses the output of a max pooling, more precisely the indices of the maximal values that were selected in the initial feature maps, and reconstructs them back to their shape prior to the max pooling by replacing non-maximal values with zero.

1.2.4.2 Convolution-based upsampling

As the name suggests, these methods are based in part on convolutions and, therefore, use learned parameters, as opposed to interpolation-based methods. Within this category, three main techniques are commonly used in deep neural networks.

Transposed convolution

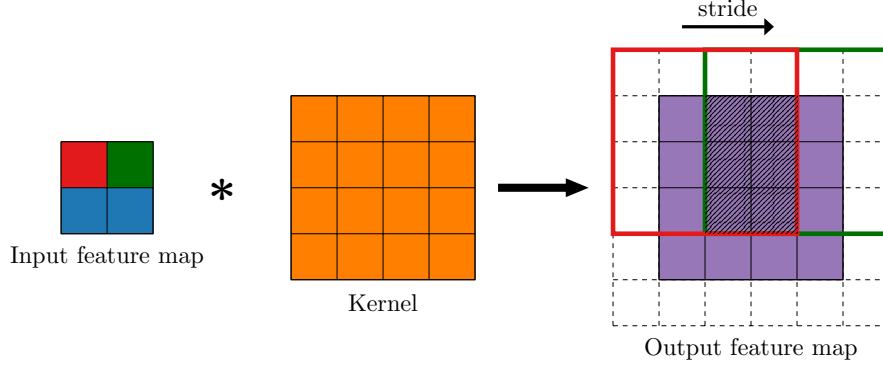


Figure 3: Illustration of transposed convolution of a 2×2 input with a 4×4 kernel, using a stride of 2 and padding of 1, leading to a 4×4 output. Hashed boxes represent pixels where there are overlapping sums during the computation.

Popularized by Zeiler *et al.* [16], a transposed convolution can be seen as the opposite of a standard convolution and is often called deconvolution, in the sense that it performs a convolution-like operation to produce an output with a larger spatial size, while a regular convolution produces an output with a similar or lower spatial size. In the standard deconvolution algorithm, for an input $\mathbf{x} \in \mathbb{R}^{C \times H \times R}$ and an output $\mathbf{y} \in \mathbb{R}^{C' \times H' \times R'}$ and a weight tensor $\mathbf{w} \in \mathbb{R}^{C \times C' \times k_h \times k_r}$ the computation of the output tensor $\mathbf{y}$ can be expressed as:

$$\forall c' \in \llbracket 1, C' \rrbracket, \forall h' \in \llbracket 1, H' \rrbracket, \forall r' \in \llbracket 1, R' \rrbracket$$

$$y_{c', h', r'} = \sum_{c=1}^C \sum_{i=1}^{k_h} \sum_{j=1}^{k_r} x_{c, \frac{h'-i+p_h}{s_h}-1, \frac{r'-j+p_r}{s_r}-1} \cdot w_{c, c', i, j} + b_{c'} \quad (11)$$

Each input pixel is multiplied by a kernel, which results in a window the size of the kernel that is slid over the output space to determine the output pixels. As illustrated in Figure 3, when the stride is smaller than the kernel dimensions, it introduces several overlapping sums which have been shown to introduce checkerboard artifacts during training [17] and require storing partial sums during inference which lead to resource and complexity overheads [18]–[21]. For this reason, it is usually never implemented this way in practice, in favor of algorithms that solve this overlapping sums issue. One of the most common methods is fractionally-strided convolution [22]. It performs a pixel padding, introducing $s - 1$ zeros between each pixel of the input, padding the input with $k - p - 1$ zeros around the feature maps, before applying a regular convolution with transposed kernels (as in usual matrix transpositions) and a stride of 1, as illustrated in Figure 4. This implementation is used in two most common deep learning software frameworks such PyTorch [23] and TensorFlow [24].

Sub-pixel convolution

Sub-pixel convolution was introduced by Shi *et al.* [25] and uses a regular convolution to increase the depth of the tensor, multiplying it by α^2 , where α is the upsampling factor. Then it is followed by a pixel shuffling algorithm (see Algorithm 1 below) to reshape the feature maps to the desired dimensions, which also removes the extra channels created

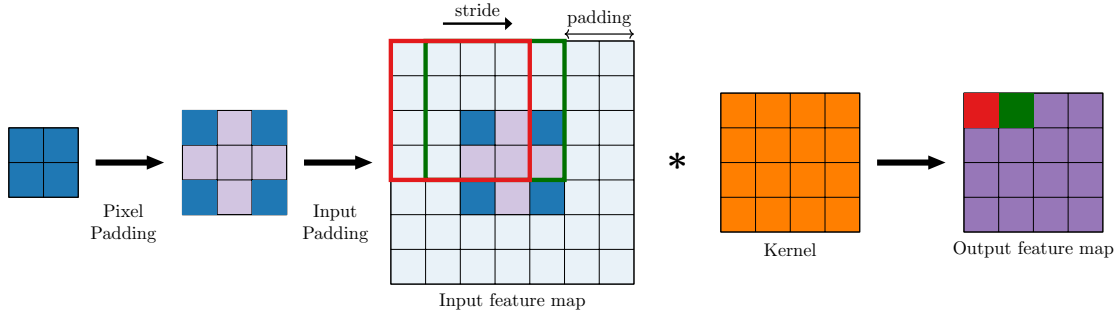


Figure 4: Illustration of the fractionally-strided convolution algorithm, that implements a transposed convolution of a 2×2 input with a 4×4 kernel, using a stride of 2 and padding of 1, leading to a 4×4 output.

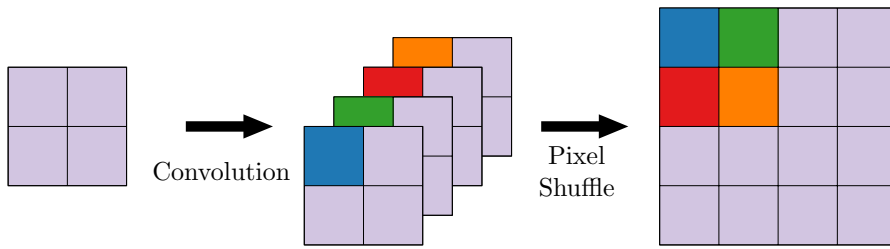


Figure 5: Illustration of the subpixel convolution of a 2×2 input with an upsampling factor $\alpha = 2$.

previously, as illustrated in Figure 5. It is a widely used method for upsampling in super-resolution neural networks [26].

Algorithm 1 Pixel Shuffle

```

procedure PIXEL SHUFFLE
  for  $c' = 0, c'++$  while  $c' < C'$  do
    for  $o_r = 0, o_r++$  while  $o_r < R'$  do
      for  $o_h = 0, o_h++$  while  $o_h < H'$  do
         $i_h \leftarrow \lfloor \frac{o_h}{\alpha} \rfloor$ 
         $i_r \leftarrow \lfloor \frac{o_r}{\alpha} \rfloor$ 
         $c \leftarrow \alpha^2 c' + \alpha \text{mod}(o_h, \alpha) + \text{mod}(o_r, \alpha)$ 
         $y[c', o_h, o_r] \leftarrow x[c, i_h, i_r]$ 

```

Resize convolution

Introduced by Dong *et al.* [27], resize convolution performs an interpolation upsampling, usually nearest-neighbors, followed by a regular "same-padded" convolution with a stride of 1, which means that the padding is chosen so the spatial dimensions of the feature maps are unchanged by the convolution, as illustrated in Figure 6.

1.2.5 Activation layers

Activation layers are nonlinear functions usually inserted after layers that perform linear operations like fully-connected layers or convolution layers and are essential for deep learning models to function, as they prevent the model from being reducible to a single linear operation. A wide range of activation layers can be found in the literature, although we only use the following in this thesis:

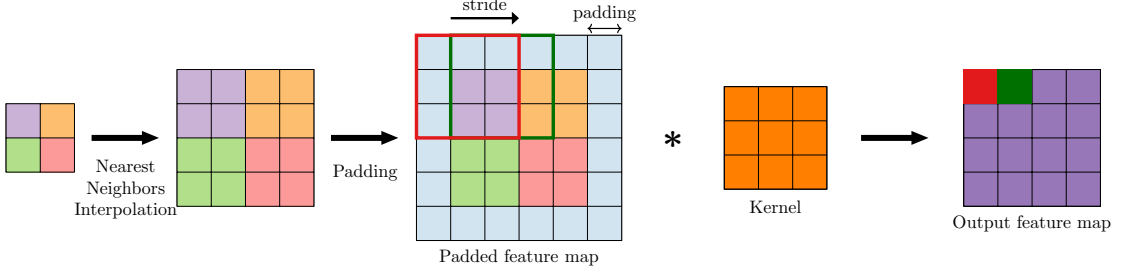


Figure 6: Illustration of the nearest-neighbor resize convolution of a $\times 2$ upsampling of 2×2 input. The convolution following the interpolation uses a kernel size of 3, which means a zero-padding of 1 needs to be performed to get a 4×4 output.

- The Rectified Linear Unit (ReLU), defined by $\forall x \in \mathbb{R}, \text{ReLU}(x) = \max(0, x)$ and one of the most commonly found activation layers.
- The Parametric Rectified Linear Unit (PReLU), a variant of ReLU defined by $\forall x \in \mathbb{R}, \text{PReLU}(x) = \max(0, x) + \alpha \min(0, x)$, where α is a learnable parameter.
- The Hyperbolic Tangent (Tanh) defined by $\forall x \in \mathbb{R}, \text{Tanh}(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$.

1.2.6 Batch normalization layer

During training, the inputs from the dataset are often processed by batches of input to better use hardware parallelism capabilities and speed up training. In this context, the use of batch normalization layers has been shown to increase the speed of convergence of deep neural networks [28]. They are inserted at various points within the network, usually between convolutions (or, in some cases, fully-connected layers) and activation layers, and use runtime statistics along with trainable parameters to normalize the tensors. For an input tensor $\mathbf{x} \in \mathbb{R}^N \times \mathbb{R}^{C \times H \times R}$ where N denotes the number of inputs in a batch, the output $\mathbf{y} \in \mathbb{R}^N \times \mathbb{R}^{C \times H \times R}$ is obtained by:

$$\forall n \in \llbracket 1, N \rrbracket, \forall c \in \llbracket 1, C \rrbracket, \forall h \in \llbracket 1, H \rrbracket, \forall r \in \llbracket 1, R \rrbracket$$

$$y_{n,c,h,r} = \frac{x_{n,c,h,r} - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma_c + \beta_c \quad (12)$$

where μ and σ are respectively the mean and standard deviation over the batch dimension, γ_c and β_c are vectors in $\mathbb{R}^C$ of parameters to be learned, respectively called the weights and biases, and ϵ is a scalar used for numerical stability.

1.3 Model compression

Deep learning has proven to be a very powerful solution for many machine vision tasks. However, the large number of computations required to perform deep neural network inferences makes them difficult to use on devices used in embedded systems that are severely limited in computation resources, memory, and power budget. This has become more and more of a challenge, as state-of-the-art models tend to require an increasing amount of computations over the years, making the deployment of performing deep neural networks in embedded systems a major hurdle. To address this challenge, several methods can be used to reduce the computational complexity of models without substantially sacrificing their accuracy. These methods that are collected under the name of "model compression" aim to find an optimum between the performance of

the network (*i.e.* how well it performs on its task, measured by appropriate metrics) and the cost of running such network, measured by indicators such as the number of operations, power consumption or latency. This optimization problem represents a vast design space, especially because the cost of a neural network is determined by numerous elements. Therefore, each method used for model compression explores a different axis of optimization, such as designing efficient layers, reducing the number of parameters, reducing the hardware cost of operations, or finding the best hyperparameters. In this section, we review the most popular compression methods, restricting ourselves to methods applicable to CNNs, as model compression can be different in the context of transformers and foundation models.

1.3.1 Quantization

1.3.1.1 Principle

In computer architectures, data are commonly represented in 32-bit floating point format (FP32). This format offers a large and flexible dynamic range for representing floating point numbers in computations. However, it is also very costly in terms of energy compared to representations using fewer bits such as the 16-bit floating point (FP16), or fixed-point quantization. For example, an addition using 8-bit integer quantization (INT8) is 30x more energy efficient than an addition with FP32 [29]. Moreover, using fewer bits to represent the data also reduces the memory footprint and the amount of data movement, which are also very costly. The idea behind quantization in deep learning is to find efficient data representations for the weights and activations that are less costly than the initial FP32 format while minimizing accuracy degradation. Several studies have been conducted on quantization to reduce the cost of training models, however, we will restrict ourselves to presenting the quantization methods for inference, as this is the main interest for deploying deep neural networks in embedded systems. More precisely, we will discuss fixed point quantization (and by extension, integer quantization), since this is the primary goal in the context of implementing deep neural networks on digital circuits, especially on FPGA. Indeed, FPGA architectures are, by design, much more efficient with fixed-point (or integer) operations than they are with floating-point [30], which enhances the interest in quantizing models in fixed-points for this task. Note that fixed-point quantization can also benefit other types of hardware, provided that they have processing units optimized to perform operations with this precision, which is usually the case of micro-controllers, but also recent generations of GPU architectures.

In the remainder of this manuscript, we will sometimes mention integer formats when discussing quantization. From a hardware perspective, they are equivalent; knowing whether the data are actual integers or fixed-point decimals is only relevant for interpreting the results of computations. Therefore, we will treat them as if they were identical.

Quantization for deep learning is an extremely vast research topic, and this section only reviews the main principles to help understand the choices we make in our experiments with quantization, without delving too much into the very advanced techniques.

1.3.1.2 Uniform quantization

To perform quantization, we need to define an operation that maps the large range of values of FP32 to a smaller, finite set of integer values. In other words, we need to introduce a mapping function between the subset of real values that can be represented

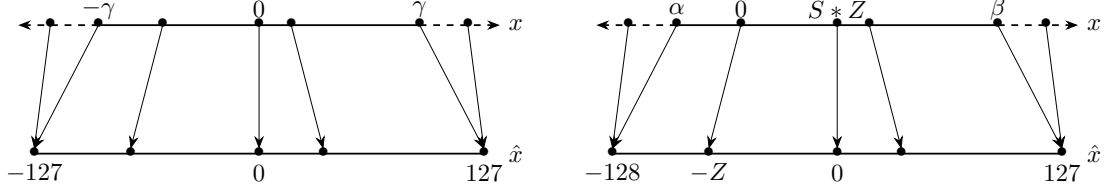


Figure 7: Illustration of symmetric (or scale) quantization and asymmetric (or affine) quantization in the example of 8-bit integer format. Figure adapted from [31].

in FP32 format, and the subset of integer values defined by the integer format. A real value x can be converted to a quantized value $\hat{x}$ by the quantize function Q , such as:

$$\hat{x} = Q(x) = \text{clip}(\text{round}(\frac{x}{S}), \alpha, \beta) - Z \quad (13)$$

where S is a real-valued number called the scaling factor, Z is an integer called zero point. Here, round is the rounding operation that approximates a real number to the closest integer, and clip is a clipping function defined by:

$$\text{clip}(x, \alpha, \beta) = \begin{cases} \alpha, & x < \alpha \\ x, & \alpha \leq x \leq \beta \\ \beta, & x \geq \beta \end{cases} \quad (14)$$

where $[\alpha, \beta]$ is called the clipping range.

Since the quantized values are uniformly spaced, this method is referred to as uniform quantization. The real values can be recovered through an operation called dequantization, but only partially due to the rounding and clipping operations:

$$\tilde{x} = S(\hat{x} + Z) \quad (15)$$

1.3.1.3 Calibration and symmetry

The important aspect when implementing uniform quantization is the determination of the scaling factor, the zero, and the clipping range. To do so, there are usually two ways forward, both illustrated in Figure 7. The zero and the scaling factor are determined on the basis of the choice of clipping range, which is called calibration. This calibration can be either dynamic, *i.e.* computed dynamically during inference for each activation tensor, or static, *i.e.* predetermined before the inference. A dynamic quantization is often beneficial to the accuracy, as it leads to a representation that is closer to the distribution of data, but requires a substantial overhead of computation, which is why static quantization is generally favored.

Affine quantization

The most straightforward way to determine the clipping range is to base them on the minimum x_{min} and maximum x_{max} of the value distribution of the input to be quantized. Doing so will, most of the time, lead to a clipping range that is not symmetric with respect to the origin. In this scheme, called affine quantization, or asymmetric quantization, the scaling factor and zero are determined such as:

$$S = \frac{\beta - \alpha}{2^b - 1} \quad (16)$$

$$Z = \text{round}(\frac{\alpha}{S}) + 2^{b-1} \quad (17)$$

with b the number of bits allocated to represent the data also called the bit width [31], [32].

Scale quantization

Another possible choice is a symmetric clipping range, also based on the minimum and maximum values of x , but where the lower and upper bounds are chosen so that $|\alpha| = |\beta| = \gamma$. This choice is called scale quantization or symmetric quantization. It is symmetric around zero, which means that $Z = 0$, and the scaling factor is determined as:

$$S = \frac{\gamma}{2^{b-1} - 1} \quad (18)$$

This has the principal advantage of simplifying the quantization process and leads to a reduction of the computational cost of the quantization, but offers a clipping range less relevant to the value distribution of data, which might be detrimental when the repartition of positive and negative values is very imbalanced. Moreover, since the dynamic range of signed integer quantization format is $\llbracket -2^{b-1}, 2^{b-1} - 1 \rrbracket$, it is by nature asymmetric, which means that using a symmetric quantization drops one value out of the range, which might be problematic for low bit widths [31], [32].

1.3.1.4 Non-uniform quantization

Rather than using evenly spaced quantization intervals, some works have studied non-uniform quantization [33]–[36]. There, a set of discrete non-uniformly spaced quantization levels $\{Q_i \in \mathbb{N} | i \in \llbracket 0, 2^b - 1 \rrbracket\}$, as well as non-uniformly spaced thresholds $\{T_i \in \mathbb{R} | i \in \llbracket 0, 2^b - 1 \rrbracket\}$ are defined. The quantization function is then defined by:

$$Q(x) = Q_i, \text{ if } T_i \leq x < T_{i+1} \quad (19)$$

This type of quantization is more flexible than uniform quantization, and can better represent the dynamic range of values of the signal, especially in scenarios where the distribution is a bell curve with long tails, which can happen with weights in neural networks, generally leading to a better accuracy [33], [37], [38]. However, non-uniform quantization is difficult to use in practice, as it requires more quantize/dequantize steps for computations, limiting the benefits of quantization, and is more difficult to implement in hardware, which is why it is often disregarded in favor of uniform quantization.

1.3.1.5 Granularity

The next element that must be chosen when implementing quantization is how the quantization parameters (*i.e.* scaling factor, zero, clipping range) should be shared between elements of the same tensor, whether it is an activation tensor or a weight tensor. This element is referred to as granularity, and several levels can be identified, from finest to coarsest:

- element-wise quantization, where every individual value of a tensor has its own parameters,
- channel-wise quantization, in which quantization parameters are unique to each channel for three-dimensional tensors (*e.g.* convolutions), or row / column for two-dimensional tensors (such as in fully-connected layers),
- group-wise quantization, where the parameters are attributed to groups of rows or columns or groups of channels,
- layer-wise quantization, where the whole layer or tensor shares the same parameters.

Element-wise quantization is rather unpractical, as it makes matrix computations much more difficult to implement, and even if group-wise quantization shows good results in some contexts, it also brings extra costs. In fact, if we consider, for example, a quantized matrix multiplication $\mathbf{y} = \mathbf{x} \cdot \mathbf{w}$ (which is how, most of the time, operations of fully-connected or convolution layers are implemented in hardware) with scale quantization, the values of the output matrix $\mathbf{y} = (y_{i,j}) \in \mathbb{R}^{M \times N}$ can be approximated from the quantized inputs $\hat{\mathbf{x}} = (\hat{x}_{i,k}) \in \mathbb{R}^{M \times P}$ and $\hat{\mathbf{w}} = (\hat{w}_{k,j}) \in \mathbb{R}^{P \times N}$ by dequantizing them before performing the multiplication:

$$y_{i,j} = \sum_{k=1}^P x_{i,k} w_{k,j} \approx \sum_{k=1}^P \tilde{x}_{i,k} \tilde{w}_{k,j} = \sum_{k=1}^P s_{x,i,k} \hat{x}_{i,k} \cdot s_{w,k,j} \hat{w}_{k,j} \quad (20)$$

Therefore, using integer matrix multiplication requires the scales to be independent of k so they can be factored out of the sum:

$$y_{i,j} = s_{x,i} s_{w,j} \sum_{k=1}^P \hat{x}_{i,k} \hat{w}_{k,j} \quad (21)$$

Thus, the quantization granularity cannot be per-element and must be per-tensor or per-channel (per-row or per-column for fully-connected layers). For performance reasons, channel-wise quantization is usually favored for weights as it shows a significant increase in accuracy compared to layer-wise quantization, which, conversely, is best for activations [32].

1.3.1.6 Fine-tuning

Most of the time, introducing quantization will lead to a degradation in accuracy, a phenomenon often referred to as quantization error or quantization noise. Minimizing the gap between the quantized and floating-point accuracies is an important step of neural network quantization, called fine-tuning, which can be done with, or without retraining the model.

Post-Training Quantization

Post-Training Quantization (PTQ) starts with a pre-trained networks with 32-bit floating point weights and applies the quantization without additional training, but uses a calibration phase to determine the quantization parameters based on the statistical distribution of values of weights and activations, performing a few inferences to measure the average distribution of activations throughout the network. This method has the advantage of introducing almost no overhead, as no retraining is needed, but can introduce significant degradation of the accuracy. To mitigate this degradation, several techniques can be employed, such as bias correction, cross-layer equalization, or better rounding functions for approximation.

Quantization-Aware Training

In some cases, PTQ methods are insufficient to mitigate the accuracy degradation. This is often the case when using very low bit widths (*i.e.* 4 bits and lower). As a solution to these scenarios, a second approach is to perform quantization during training, to have the network learn to adapt to this restricted range of values. This is referred to as Quantization Aware Training (QAT). This method can also start from floating point weights from a previous training, and perform training for a few epochs using training data, or start completely from scratch. In both cases, it often leads to greater accuracy than

PTQ, as it allows the model to find better adaptations and more accurate quantization parameters, but at the cost of slower training.

Usually, the operations of the forward and backward pass are done using quantized weights and activations but represented in FP32 format, and a quantization step is performed after the weight update. Doing so allows the network to retain enough precision for the weight updates, which are usually very small values compared to the weights, and performing the update with fewer bits would often lead to no weight updates [39]. In Section 1.1.3, we presented how SGD computes the weight updates, based on the gradient of the loss function, itself based on the gradients of each layer. However, the quantization function Q of equation 13 is by definition non-derivative due to the rounding operation. To circumvent this issue, QAT uses the Straight-Through Estimator (STE) [40], which approximates the rounding function to an identity function, setting its derivative to 1. Despite its simplicity, this estimator works very well in practice and remains the most widely used technique for training quantized models. It is also possible to learn the quantization parameters on top of learning to adjust the weights. Previous studies have demonstrated the advantage of learning the clipping range [41], the quantization levels and intervals [42] or the scaling factors [43], [44].

1.3.2 Efficient convolutions

Convolutions are by far the most computationally intensive layer of a CNN, it is, therefore, natural to wonder to what extent it would be possible to reduce the complexity of convolutions, without degrading the performance of the network. Several techniques have been designed over the years to address this question. The computational complexity $\mathcal{C}$ of a convolution layer can be expressed as:

$$\mathcal{C} = \mathcal{O}(H' \cdot R' \cdot k_h \cdot k_r \cdot C \cdot C') \quad (22)$$

using the same notations as Equation 5. Reducing the complexity of convolutions is then a question of how to reduce one of these parameters without compromising their behavior, and maintaining the same receptive field, to make sure they are still able to correctly extract features from images.

1.3.2.1 Spatially separable convolutions

The idea behind this method comes from the observation that a $n \times m$ matrix can sometimes be expressed as the product of a $n \times 1$ and a $1 \times m$ vector. Therefore, some convolution kernels can be split along their spatial dimensions in a similar way, transforming a convolution with a $k_h \times k_r$ kernel into a convolution with a $k_h \times 1$ kernel, followed by a convolution with a $1 \times k_r$ kernel. This method, also called asymmetric convolutions requires less computations. Indeed, the complexities of the new convolution layers are:

$$\mathcal{C}_1 = \mathcal{O}(H' \cdot R' \cdot k_h \cdot 1 \cdot C \cdot C'_1) \quad (23)$$

$$\mathcal{C}_2 = \mathcal{O}(H' \cdot R' \cdot 1 \cdot k_r \cdot C_2 \cdot C') \quad (24)$$

We have $C'_1 = C_2 = \min(C, C')$, and if we assume $\min(C, C') = C$ (which is almost always the case), the total complexity is then:

$$\mathcal{C} = \mathcal{C}_1 + \mathcal{C}_2 = \mathcal{O}(H' \cdot R' \cdot C \cdot (C \cdot k_h + C' \cdot k_r)) \quad (25)$$

Since $(C \cdot k_h + C' \cdot k_r) < k_h \cdot k_r \cdot C \cdot C'$, this technique can save a lot of computations, especially for high kernel dimensions.

1.3.2.2 Depth-wise separable convolutions

As we have seen before a convolution projects a tensor onto a space of different depth and spatial resolution. In a regular convolution, this happens in one step. Depth-wise separable convolutions, introduced in MobileNets [45], separate this into two steps: a depth-wise convolution, which applies only one kernel to each input channel and is equivalent, in complexity, to a regular convolution where $C' = 1$, followed by a point-wise convolution, which is a simple convolution with a 1×1 kernel, which only changes the depth of the tensor. The interest is that, in this configuration, the complexities are:

$$\mathcal{C}_1 = \mathcal{O}(H' \cdot R' \cdot k_h \cdot k_r \cdot C \cdot 1) \quad (26)$$

$$\mathcal{C}_2 = \mathcal{O}(H' \cdot R' \cdot 1 \cdot 1 \cdot C \cdot C') \quad (27)$$

Which gives a total complexity such as:

$$\mathcal{C} = \mathcal{C}_1 + \mathcal{C}_2 = \mathcal{O}(H' \cdot R' \cdot C \cdot (k_h \cdot k_r + C')) \quad (28)$$

This requires much less computations than a regular convolution, however, in some cases, it can significantly decrease the accuracy of the model, especially in networks with already few parameters.

1.3.2.3 Grouped convolutions

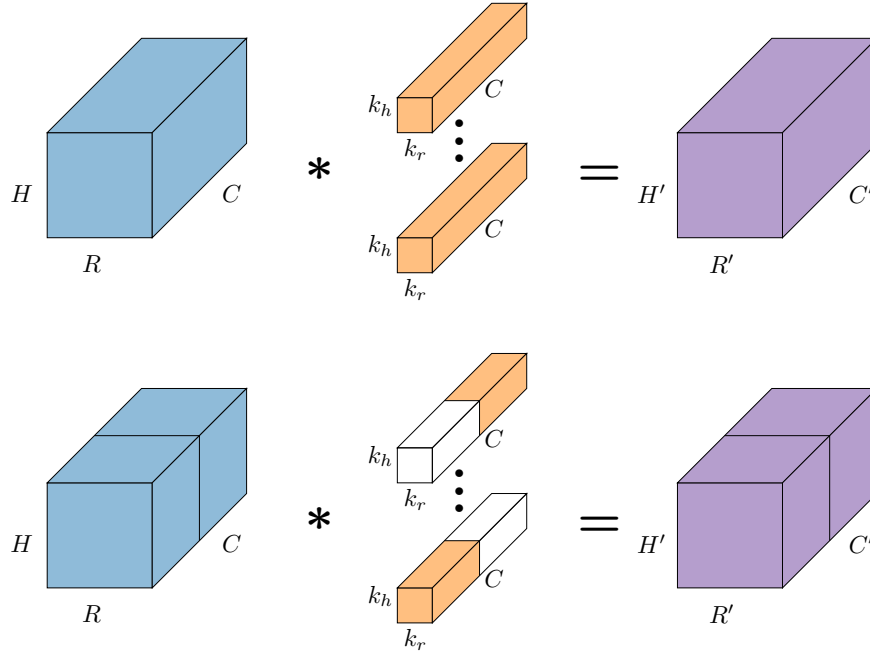


Figure 8: Illustration of the difference between a regular convolution (top) and grouped convolution with two groups (bottom), where half of the filters are applied to a corresponding half of the input channels.

As illustrated in Figure 8, grouped convolutions separate the convolution filter into g groups, with each group being paired with a group of channels of the input. Each group is convolved independently, which acts as performing g convolutions with $\frac{C}{g}$

input channels, and $\frac{C'}{g}$ output channels, resulting in g times less computations:

$$\begin{aligned}\mathcal{C} &= \mathcal{O}(g \cdot (H' \cdot R' \cdot k_h \cdot k_r \cdot \frac{C}{g} \cdot \frac{C'}{g})) \\ &= \mathcal{O}(\frac{H' \cdot R' \cdot k_h \cdot k_r \cdot C \cdot C'}{g})\end{aligned}\tag{29}$$

In the special case where $g = C'$, this is equivalent to the depth-wise convolution presented earlier. This method was originally introduced in AlexNet [1] as a way to split the computation between multiple GPUs but has later been shown to be a very efficient method to perform convolutions, even increasing the accuracy in some cases [46].

1.3.2.4 Bottleneck convolutions

ResNets [47], a very popular family of CNNs for image classification, introduced the use of bottleneck modules to reduce the complexity of convolutions. These modules use a point-wise convolution to project to a space with a reduced number of channels, then perform the convolution with larger kernels, before expanding back to the original depth with another point-wise convolution. Reducing the depth makes the internal convolution much less costly, with the reduction typically being larger than the cost of the two additional point-wise convolutions. Indeed, the complexities of each layer are then:

$$\mathcal{C}_1 = \mathcal{O}(H' \cdot R' \cdot 1 \cdot 1 \cdot C \cdot C^{proj})\tag{30}$$

$$\mathcal{C}_2 = \mathcal{O}(H' \cdot R' \cdot k_h \cdot k_r \cdot C^{proj} \cdot C^{proj})\tag{31}$$

$$\mathcal{C}_3 = \mathcal{O}(H' \cdot R' \cdot 1 \cdot 1 \cdot C^{proj} \cdot C')\tag{32}$$

This gives a total complexity such as:

$$\mathcal{C} = \mathcal{C}_1 + \mathcal{C}_2 + \mathcal{C}_3 = \mathcal{O}(H' \cdot R' \cdot C^{proj} \cdot (C + k_h \cdot k_r \cdot C^{proj} + C'))\tag{33}$$

This is generally lower than a regular convolution, especially for deep networks.

1.3.2.5 Dilated convolutions

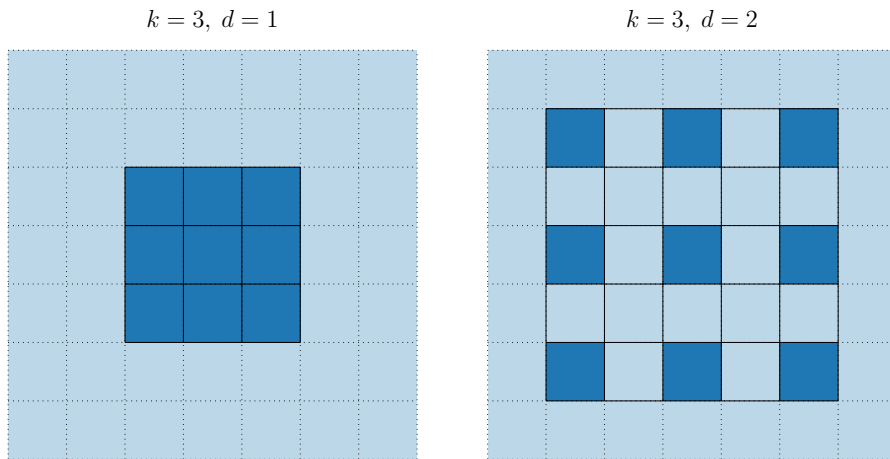


Figure 9: Illustration of kernel dilation. The 3×3 kernel with dilation rate $d = 2$ (right) covers a larger area of the feature map than the regular kernel (left), with the same number of parameters.

In dilated convolutions, or "atrous" convolutions, a number $d - 1$ of zeros are inserted between the weights in convolutions kernels, with d being called the dilation rate. Doing so widens the kernels, allowing the convolution to have a larger receptive field (also called the field of view), without requiring more parameters. For example, a 3×3 kernel with a dilation rate $d = 2$ will cover the same area as a 5×5 kernel, as illustrated in Figure 9, without using more parameters or more computations (provided that the processing hardware is able to correctly handle kernel sparsity). Even if it seems to be skipping values, this is usually compensated by the stride. It makes the convolution better at capturing global information, but worse at understanding fine-grained elements, which is why in practice, several dilated convolutions with different dilation rates are used across the network to mitigate this. A large receptive field is very interesting for semantic segmentation since the images are rather large, and being able to cover it entirely is important to reach good accuracy. Therefore, doing so without employing large kernels is of great benefit.

1.3.3 Knowledge distillation

Knowledge distillation [48] was introduced as a way to compress large and well-performing models into smaller ones that are best suited for real-case scenarios by using the knowledge acquired by the large "teacher" model to help the smaller "student" model reach better accuracy than if it was trained alone. During training, the predictions of the teacher are used alongside the predictions of the student to compute the loss, such as:

$$\mathcal{L} = \alpha \cdot \mathcal{L}_{CE}(\mathbf{y}, \hat{\mathbf{y}}) + (1 - \alpha) \cdot \mathcal{L}_{KD}(\mathbf{y}_{soft}, \hat{\mathbf{y}}_{soft}) \quad (34)$$

where $\mathcal{L}_{CE}$ is the cross-entropy loss computed between the hard labels $\mathbf{y}$ and the hard prediction $\hat{\mathbf{y}}$ of the student, $\mathcal{L}_{KD}$ is the knowledge distillation loss computed between the soft labels $\mathbf{y}_{soft}$ and the soft output $\hat{\mathbf{y}}_{soft}$ of the teacher, and α is a parameter used to balance the two losses. The soft outputs and labels are vectors of probabilities measuring the probability of each class to be the one represented by the image (or the pixel in the case of semantic segmentation), and are used to compute the loss associated with the teacher, usually based on the Kullback-Leiber (KL) divergence, which, given to probability distributions P and Q is defined by:

$$KL(P||Q) = \sum_{i \in \mathcal{I}} P(i) \log\left(\frac{P(i)}{Q(i)}\right) \quad (35)$$

where $\mathcal{I}$ is the sample space, *i.e.* the indices of classes in the case of deep learning. Knowledge distillation can also be used to train a student on a small dataset based on a teacher trained on the same task, but with a larger dataset, a method called transfer learning.

1.3.4 Pruning

Neural network pruning is based on the idea that there is a lot of redundancy in the weights of a deep neural network, making them inefficient. Therefore pruning attempts to remove selected parts of the network to reduce its cost, following a strategy that does not reduce too much the accuracy of the model. It can be removing weights from layers in a non-structured manner, referred to as "non-structured pruning", leaving behind sparse kernels. This approach leads to a good performance-to-parameters trade-off but is rarely implemented in practice, as sparsity is not handled well by common processing hardware [49]. Nonetheless, some works in the literature have studied the design of hardware circuits for sparse matrix computations [50]. A second approach, more

avored but less efficient, is "structured pruning", which, as the name suggests, prunes whole structures in the network, like complete convolution filters. It also has the interest of reducing the size of intermediate representations (*i.e.* activation tensors) which helps reduce the memory footprint of the network. Whether it is in a structured or unstructured way, deciding which parameters to prune is based on criteria, such as their magnitude (*i.e.* their absolute values) or the magnitude of gradients, discarding the parameters with the lowest metrics, as they contribute less to the network's output.

Numerous studies have shown that pruning yields good results for model compression, especially for models with a high number of parameters. However, some works have shown that there are several inconsistencies in the pruning literature and that training smaller models with good training practices can lead to better results than large pruned models, questioning its interest in practice [51]–[53].

1.3.5 Neural Architecture Search

As discussed before, the design of neural networks is a difficult task, due to the immense design space. To address this issue, a technique called Neural Architecture Search (NAS) has emerged. Its goal is to automate the search for the best neural network possible for a given task. It starts by defining a search space, like the type of layers to use (*e.g.* convolutions, fully-connected, choice of activation layers), their configurations, and the overall topology of the network. Then it defines a strategy to explore the search space, such as randomly sampling architectures from the search space, using evolutionary algorithms or Bayesian optimization. Finally, it defines an evaluation strategy to quickly evaluate the performance of an architecture without requiring to fully train it, to avoid spending resources to train a poorly performing architecture. NAS has shown great potential in finding novel architectures that can be more efficient (*i.e.* using less parameters) than manually-crafted ones. However, it suffers from a great computational cost induced by the search and multiple trainings (even if partial), the difficulty of NAS-generated models to generalize well to unseen data, and biases in the search created by the strategies [54]. Although NAS is not a compression technique per se, it helps finding more dense and efficient architectures and thus can be seen as serving the same purpose as compression.

1.4 Semantic segmentation

In this section, we present semantic segmentation, the main machine vision task studied in this thesis, the associated evaluation metrics, and the datasets we will be using. We also discuss the specific context of real-time systems and how they shape the design of deep neural networks.

1.4.1 Task definition, metrics and applications

1.4.1.1 Definition

While image classification aims to attribute a label to an image to identify the main object represented in it, semantic segmentation operates on a finer level and attributes a class label to each pixel of the image. This helps understand the nature and positions of the different elements of an image and their relationships, even in complex scenarios with overlapping objects or objects appearing at different scales. The final goal is to give meaning to the scene represented and use the understanding of that scene to make decisions. Semantic segmentation stands as one of the most challenging tasks in ma-

chine vision for several reasons. The first challenge is the difficulty of obtaining large labeled datasets that are necessary for deep learning models to perform well as producing fine-grained labeling requires significant amounts of human time. Moreover, it suffers from the imbalance between classes that naturally happens in images obtained from real-life scenarios. Finally, the high computational resources needed to process high-resolution images and the often complex models make it particularly challenging for resource-constrained systems.

1.4.1.2 Metrics

Regarding metrics to evaluate neural networks on this task, the most intuitive one is certainly Pixel Accuracy (PA), which represents the proportion of correctly labeled pixels in the image:

$$PA = \frac{TP + TN}{TP + TN + FP + FN} \quad (36)$$

where TP , TN , FP , and FN are respectively the number of true positives, true negatives, false positives, and false negatives in the prediction. Although straightforward, this metric has a major drawback, as an imbalance between class representations in the dataset often produces misleading results. Indeed, this metric does not evaluate the ability of the model to correctly segment each individual class, and while a model may have a high pixel accuracy, it may still be unable to produce correct labeling of some classes if they are underrepresented. As an example, if we consider a dataset with two classes, one constituting 95% of the pixels in the image and the other the remaining 5%, if a model can correctly identify pixels of the first class, but performs very poorly on the second, it will still yield a very high pixel accuracy, but its performances might be unsuited for the application it is trained for if the second class is of great importance.

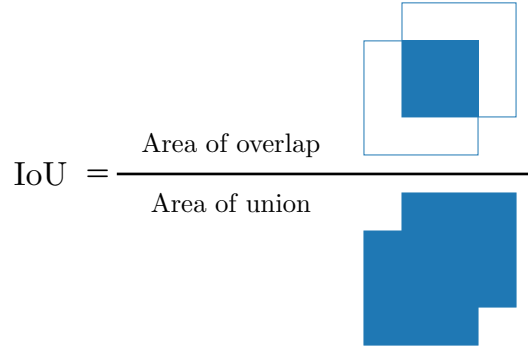


Figure 10: Illustration of the computation of the intersection over union metric.

For this reason, a second metric, named Intersection over Union (IoU) also exists and is computed independently for each class c by:

$$IoU(c) = \frac{prediction(c) \cap target(c)}{prediction(c) \cup target(c)} = \frac{TP(c)}{TP(c) + FP(c) + FN(c)} \quad (37)$$

It is defined as the ratio between the area of overlap and the area of union between the prediction and the ground truth, as illustrated in Figure 10. For easier evaluation and comparison of models, the actual metric used is the mIoU, simply obtained by:

$$mIoU = \sum_{c=1}^C IoU(c) \quad (38)$$

where C is the number of classes in the dataset. Although pixel accuracy might still be present in some papers, mIoU is much more standard in recent works, as it is better suited to properly evaluate the performance of the models. Therefore, in this manuscript, we will only consider this metric when evaluating deep neural networks for semantic segmentation.

1.4.1.3 Applications

Even if it represents a challenging task, semantic segmentation has garnered a lot of interest, as it finds applications in a lot of research fields, such as:

- Autonomous vehicles: analyzing road scenes to help systems understand the driving environment to make informed decisions, such as detecting and avoiding obstacles [55]–[59].
- Medical imaging: segmenting medical scans such as magnetic resonance imaging (MRI), computed tomography (CT), and X-rays to identify abnormalities in tissues or organs [60]–[64].
- Agriculture: identification of crops and weeds to guide precision farming robots, labeling forest tree species or detecting diseases and pest infections in plantations [65]–[69].
- Satellite imaging: extracting road, detecting building, identifying land cover types, ships and oil spilling detection [70]–[74].
- Robotics: indoor navigation and obstacle detection [75]–[78].

1.4.2 Datasets

There is a vast collection of datasets labeled for semantic segmentation. The most popular ones for benchmarking neural networks are ADE20K [79], the Pascal Visual Object Challenge dataset [80], SUN-RGBD [81], and COCO-Stuff [82]. All of these datasets have a wide range of objects from various scenes. However, since this thesis is centered on semantic segmentation applied to autonomous driving, we focus on datasets specially meant to train models to understand urban scenes.

1.4.2.1 CamVid



Figure 11: Example taken from the CamVid dataset with the associated pixel-wise labeling. Source: mi.eng.cam.ac.uk/research/projects/VideoRec/CamVid/

The Cambridge-driving Labeled Video Database [83] (CamVid) is a dataset containing four video sequences captured by a camera mounted on a car at a rate of 30Hz, lasting a total of 22 minutes and 14 seconds in urban and residential environments. The frames of the videos have been sampled at a lower rate of either 1Hz or 15Hz, producing an image dataset of 701 examples in a resolution of 960x720 pixels. The original dataset contains 32 object classes, although benchmarking is done on a subset of 11 classes: building, tree, sky, car, sign symbol, road, pedestrian, fence, column pole, sidewalk, and bicyclist.

1.4.2.2 Cityscapes

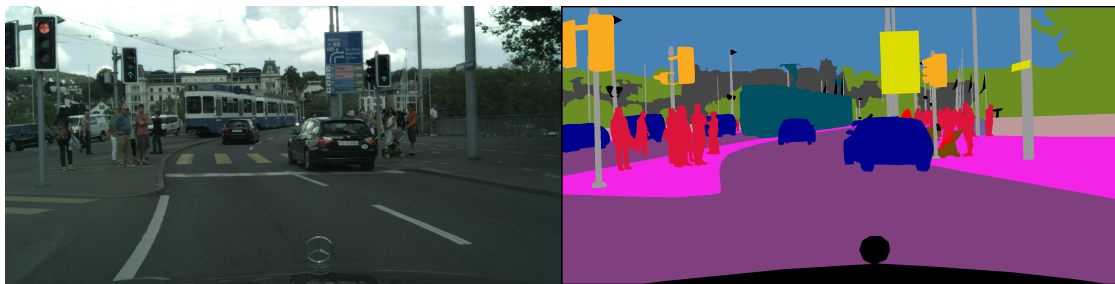


Figure 12: Example from the Cityscapes dataset with the associated dense annotation. Source: cityscapes-dataset.com

The Cityscapes dataset [84] contains 5000 finely-annotated and 20000 coarsely-annotated images of street scenes taken from 50 different cities over several months and under good weather conditions using a camera with a very high resolution of 2048x1024 pixels. The dataset labels contain 30 object classes organized into 8 categories. Benchmarking models on this dataset involves training only on the images with dense annotations, and on 19 classes: road, sidewalk, building, wall, fence, pole, traffic light, traffic sign, vegetation, terrain, sky, person, rider, car, truck, bus, train, motorcycle, and bicycle. The particularly wide images and large amount of examples in the dataset make it both a more interesting and challenging dataset than CamVid, which is why it is more common in recent papers and is often included in works not necessarily targeting autonomous driving applications.

1.4.3 Models history overview

Although less vast than for more popular machine vision tasks like classification, the collection of deep neural networks for semantic segmentation remains rather large. A lot of progress on the performance of the models has been made over the ten-year-long history of semantic segmentation, leading to more and more complex models.

It begins with FCN (2014) [55] which was the first paper to use CNNs to perform semantic segmentation. It introduced the idea of replacing the classifier of a classification neural network with a second convolutional network. While the first convolutional part reduces the spatial dimension of the feature maps, this second network uses transposed convolutions to progressively upsample them back to their original dimensions while learning segmentation masks, thus forming the first convolutional encoder-decoder, a type of architecture that has been the cornerstone of deep neural networks for semantic segmentation.

This principle was then refined by papers like U-Net [85] and SegNet [86] both presented in 2015, introducing the idea of propagating information between the encoder and the decoder to improve segmentation. SegNet propagates only the indices of max pooling layers of the encoder to be used in max unpooling layers in the decoder, while U-Net propagates intermediate outputs from the downsampling layers of the encoder to be used as side-inputs concatenated to the intermediate outputs of the decoder, resulting in a U-shaped network where the decoder part is symmetric of the encoder. U-Net is especially interesting as the simple yet efficient method it introduced is still very used to this day despite being outperformed by more advanced neural networks, especially in the more application-centered field of deep learning, as over two thousand papers¹ that took inspiration from it have been published since its release.

An important change in the way of designing semantic segmentation neural network architectures was made through the DeepLab series (DeepLabv1, v2, v3, v3+, 2016 to 2018) [87]–[89], and more specifically DeepLabv2, which introduced a module named atrous spatial pyramid pooling (ASPP). It uses dilated convolutions with different dilations and a pyramid pooling scheme to have the network operate in different fields of view, which was shown to better capture the different scales of objects present in the scene, leading to better semantic segmentation results.

Inspired by this method, HRNet (2020) [90] pushed the idea of multiscale architectures further and maintains high-resolution representations through the network, while operating convolutions on multiple lower scales before fusing them, which proved to greatly improve segmentation performance.

The introduction of Vision Transformers (ViTs) [91] has had major consequences in the deep learning community and paved the way toward a new generation of deep neural networks, showing to outperform CNNs in various vision tasks. Regarding semantic segmentation, one of the first papers to introduce ViT to the task is SETR (2021) [92], followed by numerous works.

In recent years, access to Internet-scale datasets has led to the creation of very large models called foundation models that use billions of parameters with high generalization capacities that can be easily specialized to perform any machine learning task. The prime example of foundation models applied to semantic segmentation is InternImage (2023) [93], being one of the best-performing models on the task. The same year, the Segment Anything (SA) project [94] made a breakthrough by creating a very versatile model that can segment an image regardless of the object class or context and use it to create a dataset of 11 million images with over one billion segmentation masks, thus creating the largest image segmentation dataset to date. Note, however, that this is not *semantic* segmentation per se, as the masks are unlabeled, but this work still represents a major change in the way machine learning is done in general and how datasets are constructed and used.

1.4.4 Real-time semantic segmentation

In this thesis, we study real-time systems, that is, systems that can operate under real-time constraints. What a "real-time" system means in practice varies greatly depending on the context, but a general definition can be given as a system that "controls an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time" [95]. In other words, it is a system that can complete its task in a limited time frame. In the case of video processing systems,

¹Source: <https://paperswithcode.com/method/u-net>

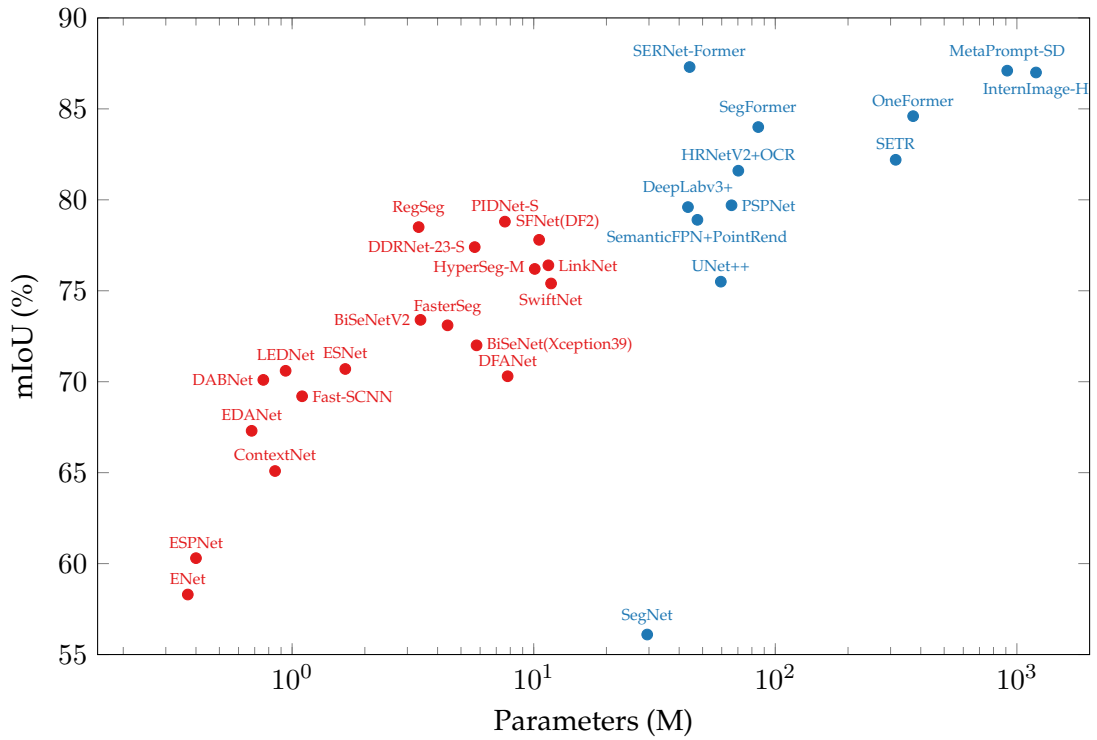


Figure 13: Semantic segmentation performance on Cityscapes measured by the mIoU (in %) relative to the number of parameters (in millions) of popular state-of-the-art models in blue, and real-time models in red.

such as, for example, a semantic segmentation model for autonomous driving, this limit is set so that the system is at least as fast as the video stream in input to ensure that it is processed without loss of data. In practice, this translates into a minimum throughput of 30 Frames Per Second (FPS), as this is the usual sampling rate of video cameras. Regarding latency, it highly depends on the application. For instance, in the context of autonomous driving, the required latency is in the order of tens of milliseconds.

In the meantime, the history of deep learning models tends to follow a general trend of pushing for the highest accuracy on their respective tasks, only using this metric as a figure of merit. This is usually achieved at the cost of larger and larger models with an increasing number of parameters. Having large models is not ideal for real-time systems as, to some extent, more parameters mean more computations, which itself means more computation time. To be accurate, the number of parameters itself is only an imprecise indicator of latency, as the inference time of a model that requires a given amount of computations is highly dependent on the hardware that it is being executed on. Nevertheless, it is safe to assume that a model with 10 million parameters will generally be faster than one with 100 million, or a billion parameters.

In the past decade, as part of the research in semantic segmentation was following its everlasting pursuit of accuracy, a second branch of models emerged, with a different goal: producing networks able to meet real-time constraints. Although they do not reach an accuracy as high, they tend to have fewer parameters than regular models, sometimes by several orders of magnitude, as shown in Figure 13. These real-time semantic segmentation models take advantage of efficient network structures and convolution techniques to greatly reduce the number of computations and try to find the best optimization point between throughput and accuracy.

Category	Model name	Publication Year	Parameters (Millions)	Cityscapes mIoU (%)	CamVid mIoU (%)	FPS
Encoder-decoder	ENet [96]	2016	0.37	58.3	51.3	135.4
	ERFNet [97]	2017	2.1*	69.7	-	125.0
	LinkNet [98]	2017	11.5	76.4	68.3	65.8
	ESPNet [99]	2018	0.4	60.3	-	112
	DABNet [100]	2019	0.76	70.1	66.4	170.2
	EDANet [101]	2019	0.68	67.3	66.4	108.7
	Fast-SCNN [102]	2019	1.1	69.2	-	485.4
	ShelfNet [103]	2019	-	74.8	-	59.2
	SwiftNet [104]	2019	11.8	75.4	72.6	39.3
	ESNet [105]	2019	1.66	70.7	-	63
	RegSeg [106]	2023	3.34	78.5	80.9	38
Multi-branch	BiSeNet (Xception39) [107]	2018	5.8	72.0	65.6	285.2
	ContextNet [102]	2018	0.85	65.09	-	139.2
	ICNet [108]	2018	-	70.6	67.1	30.3
	BiSeNetV2 [109]	2020	3.4*	73.4	76.7	156
	SFNet(DF2) [110]	2021	10.53	77.8	70.4	61
	DDRNet-23-S [111]	2021	5.7	77.4	78.6	101.6
	PIDNet-S[112]	2023	7.6	78.8	80.1	93.2
Attention	DFANet [113]	2019	7.8	70.3	64.7	160
	LEDNet [114]	2019	0.94	70.6	-	71
	FANet-18 [115]	2020	-	75.0	69.0	72
	PP-LiteSeg-B2 [116]	2022	-	78.2	75.0	102.3
Meta	FasterSeg [117]	2019	4.4	73.1	71.1	163.9
	HyperSeg-M [118]	2021	10.1	76.2	-	36.9

Table 1: Comparative table of real-time semantic segmentation models, organized by architecture type. The throughputs in frames-per-second are obtained from the results presented in the original papers, using best-case scenarios. They were evaluated on different hardware devices with different image resolutions, and are not presented here to serve as a direct comparison, but to display the ability of the models to reach real-time capacities.

*: not obtained from the original paper.

Each of these models, presented in Table 1, brings its own innovative idea of network architecture. However, they can be classified into four categories based on the general ideas and methods used in their respective papers, inspired by the methodologies of state-of-the-art semantic segmentation models:

- Encoder-decoder architectures, which we discussed in the previous section, are two-stage convolutional networks. The first stage, the encoder, is either specific to the network or inspired by a classification network, and is used to reduce the spatial dimensions of images while increasing their features (*i.e.* channels). The second stage, the decoder, operates in the opposite direction, gradually recovering the initial resolution of the images through several upsampling layers while reducing the depth.
- Multi-branch networks, that combine global and local information by maintaining different resolutions in parallel in the network, thus extracting features at different scales, before fusing them.
- Attention-based models, which utilize self-attention mechanisms originally designed for language tasks to enable the network to dynamically assign varying levels of importance to different components within an input sequence, thereby focusing on the most relevant parts when generating predictions.
- Models that use "meta-learning" techniques that leverage knowledge gained from

previous tasks to influence the architecture of the network. Methods in this category include NAS, knowledge distillation, and hypernetworks, where a network is trained to set the weights of another network.

This branch of semantic segmentation models is where one should focus interest when attempting to design a real-time system for autonomous driving. As we shall discuss in greater detail in the next chapter, having a lower number of parameters is also a considerable advantage for embedded systems, especially in the case of FPGAs, where memory resources are heavily restricted.

1.5 Super-resolution

Although semantic segmentation is the main focus of this thesis, we also study the super-resolution task in Chapter 5. In that chapter, we use one specific super-resolution model as an example to study the choice and design of upsampling methods. Therefore, our study of real-time super-resolution in this thesis is not as thorough as for semantic segmentation. Nevertheless, in this section, we introduce the basics of super-resolution, the dataset used, and a cursory overview of models.

1.5.1 Task definition and metrics

Super-resolution is the task of enhancing the resolution of an image or a video while maintaining the quality of the image and its content as much as possible. Although several classical image processing techniques address this task, deep neural networks have demonstrated promising results in fields such as medical imaging [119], satellite imaging [120], surveillance [120], image restoration [121], or video decompression [122].

Regarding metrics to measure image reconstruction accuracy, the two most widely used are the Peak Signal-to-Noise Ratio (PSNR) and the Structural Similarity Index Measure (SSIM). The PSNR evaluates pixel-level errors in the reconstruction through the Mean Square Error (MSE) (cf. Section 1.1.2) between the ground-truth image and the reconstructed image:

$$PSNR = 10 \log_{10} \left(\frac{MAX^2}{MSE} \right), \quad (39)$$

where MAX is the maximum value of pixels in the ground truth.

Meanwhile, SSIM aims to measure the similarity between ground truths and reconstructions on a more perceptual basis, taking into account elements such as structures, luminance, and contrast. For a reconstructed image $\mathbf{y}$ and a ground truth $\hat{\mathbf{y}}$, it is obtained by:

$$SSIM(\mathbf{y}, \hat{\mathbf{y}}) = \frac{(2\mu_{\mathbf{y}}\mu_{\hat{\mathbf{y}}} + c_1)(2\sigma_{\mathbf{y}\hat{\mathbf{y}}} + c_2)}{(\mu_{\mathbf{y}}^2 + \mu_{\hat{\mathbf{y}}}^2 + c_1)(\sigma_{\mathbf{y}}^2 + \sigma_{\hat{\mathbf{y}}}^2 + c_2)}, \quad (40)$$

where $\mu_{\mathbf{y}}$ and $\mu_{\hat{\mathbf{y}}}$ are the mean pixel values of $\mathbf{y}$ and $\hat{\mathbf{y}}$, $\sigma_{\mathbf{y}}^2$ and $\sigma_{\hat{\mathbf{y}}}^2$ their respective variances, $\sigma_{\mathbf{y}\hat{\mathbf{y}}}$ is the covariance of $\mathbf{y}$ and $\hat{\mathbf{y}}$, and c_1 and c_2 are two variables meant for stabilization, such as $c_1 = (k_1 L)^2$ and $c_2 = (k_2 L)^2$ with L defined as the dynamic range of pixels, $k_1 = 0.01$, and $k_2 = 0.03$.

1.5.2 BSD dataset

The Berkeley Segmentation Dataset [123] is a popular dataset for super-resolution and grayscale image denoising, and the one we use in our experiments. It contains 300 images of different objects like plants, animals, people, or food, as well as landscapes. Out

of these 300 images, 200 are used for training, and 100 for testing. The low-resolution images are usually obtained by resizing the dataset images that are used as ground truth.

1.5.3 Models overview

Image super-resolution is not as studied as image classification or even semantic segmentation, and there are not as many contributions made to the task. However, several pioneering papers can be identified, starting with SRCNN [27] (2014) which was the first paper to use a CNN for super-resolution, proposing a very straightforward 3-layer network with bicubic interpolation as the main upsampling technique. VDSR [124] (2016) improved the performances of CNNs for super-resolution by proposing a very deep network with 20 layers capable of learning a more complex mapping and introducing the use of residual learning to the task, which uses skip connections to fuse the low-resolution input with the high-resolution generated output, enhancing the reconstruction quality. The Generative Adversarial Networks [125] (GANs), first invented for generating images, demonstrated their interest in super-resolution through works like SRGAN [126] (2017) and ESRGAN [127] (2018), especially for reconstructing images with greater perceptual quality. As attention mechanisms made a breakthrough in image classification and semantic segmentation, it also showed competitive results in super-resolution, with papers like RCAN [128] (2018), using attention gates to help train a deep neural network efficiently and assisting the network in concentrate on significant image features. Much like for other machine vision tasks, current state-of-the-art networks use vision transformers to produce high-resolution images with greater detail and texture that can be easily adapted to different image sizes [129]–[131].

1.6 Summary

In this chapter, we presented the concepts of deep learning for semantic segmentation and super-resolution, and the considerations regarding their deployment in real-time embedded systems. Each of the discussed topics is a vast research subject on its own, and this chapter is only intended to introduce the concepts involved in the remainder of this manuscript, to give a general understanding of the studied applications and their context.

In the following chapter, we will present the context and challenges of deploying neural networks for real-time semantic segmentation on FPGA, and present the previous works that tackled this task.

Chapter 2

Deploying Deep Learning Models on FPGA

Contents

2.1	Basics of FPGA architectures	60
2.2	Interests and challenges	61
2.3	Architectures and frameworks	64
2.3.1	Deep learning Processing Units	64
2.3.2	Pipelined dataflow architectures	65
2.3.3	Choice of framework	66
2.4	FPGA architectures for real-time semantic segmentation	67
2.4.1	CamVid	67
2.4.2	Cityscapes	68
2.4.3	Observations	69
2.5	Summary	70

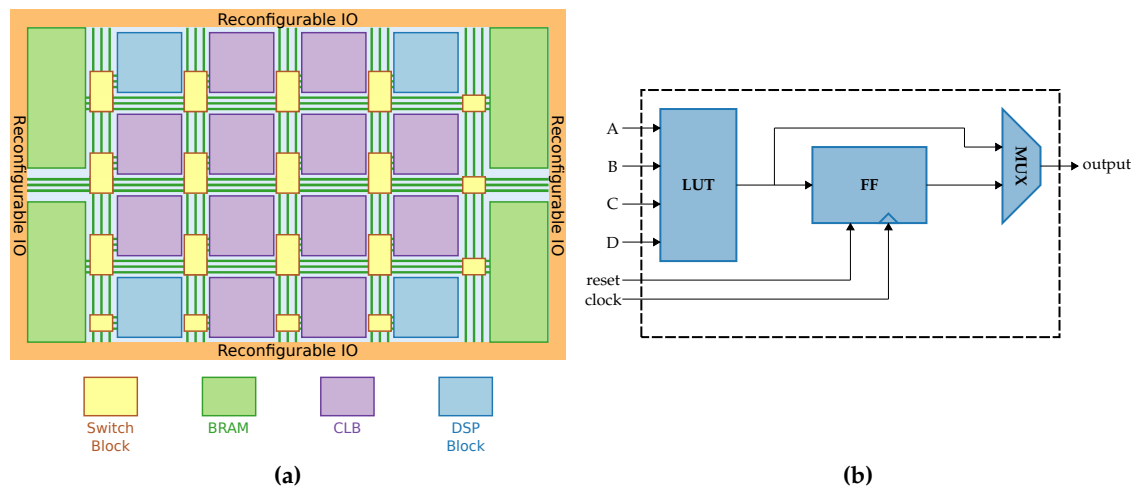


Figure 14: Illustrations of a simplified FPGA circuit (a) and a simplified CLB block (b).

This chapter is centered on the topic of implementing neural networks on FPGAs, starting with a brief introduction to FPGA circuits and the methods employed to program them. We follow with a discussion on the interests of FPGAs as a hardware device to perform the inference of deep neural networks, and to what extent they can be a better solution than GPUs, which are the devices usually used for this purpose. We also show the challenges of deploying neural networks on FPGA and the methods and tools that can be employed to facilitate this process. We end the chapter with a presentation of previous works that studied the implementation of semantic segmentation networks for real-time processing and that benchmarked their work on one of the two studied datasets.

2.1 Basics of FPGA architectures

FPGAs are a type of hardware device that has the unique ability to be reconfigurable, as opposed to Application-Specific Integrated Circuits (ASICs), which are designed and optimized for one task (as data encryption or audio encoding/decoding) or set of tasks, like digital signal processing units, or even Central Processing Units (CPUs) or GPUs, whose instruction sets and architectures are fixed. FPGA architectures have evolved a lot since their invention in 1985, but in essence, they are large-scale arrays of interconnected reconfigurable hardware blocks, surrounded by reconfigurable inputs/outputs (IOs), and reconfigurable interconnections (switch blocks) as shown in Figure 14a. The natures and names of the configurable blocks depend on the FPGA manufacturer, but are generally one of the following types:

- Configurable Logic Blocks (CLBs), which are the core elements of FPGA's reconfigurable logic. They are composed of Look-Up Tables (LUTs) which can be programmed as logic gates, synchronous memory elements in the form of Flip-Flops (FFs), and in most modern versions, carry-adder trees specialized for additions. By using multiples of these blocks, it is therefore possible to program any function, provided that it can be described as a set of combinatorial functions. An illustration of a basic CLB is shown in Figure 14b.
- Digital Signal Processing (DSP) blocks, which are circuits specifically designed to optimize Multiply-Accumulate (MAC) operations, which are essential to digital signal processing functions.

- Block Random Access Memorys (BRAMs), which are synchronous memory elements in the form of dual-port RAMs, are used to store large amounts of data and in scenarios where multiple parallel access to the data is not required.

These hardware blocks can be configured to describe almost any digital processing function, usually by using a functional behavior description of the architecture at a low level of abstraction, called Register Transfer Level (RTL), which models the circuit as a flow of data signals between registers and logic operators. This description is done using a Hardware Description Language (HDL) that can be synthesized into a "netlist" using an electronic design automation tool. This netlist is a description of the connections between the primitive elements of the FPGA. In other words, it is a description of how the described architecture is translated in terms of FPGA resources and connections between them. This netlist is then fitted to the FPGA circuit through a "place and route" algorithm that maps primitives to the actual resources of the FPGA. This workflow is generally complex and requires a lot of expertise in architecture design and the tool used. However, in the past decade, High-Level Synthesis (HLS) was proposed as a higher level of abstraction to help hardware design and reduce engineering costs. HLS uses higher-level descriptions of the architecture, often using high-level programming languages like C++, to generate the architecture description in HDL. This helps the process of FPGA programming by giving a higher level of abstraction that removes the need to do RTL descriptions, at the expense of a usually less optimized architecture on a fine-grained level. Once programmed, the FPGA can be used as a standalone device, directly connected to other components providing inputs and collecting outputs, such as, for example, analog-to-digital converters (ADCs) and digital-to-analog converters (DAC). However, in many cases, it is used alongside a CPU that controls the whole system, with the FPGA as a peripheral device, performing preprocessing steps if necessary, providing inputs, and using outputs for an application executed on the CPU, forming a System-on-Chip (SoC).

2.2 Interests and challenges

A common use for FPGAs is to prototype ASICs. In fact, ASIC manufacturing is expensive, therefore, a reconfigurable hardware device that can be used as a platform to test different architecture choices is quite useful. Moreover, they have a shorter time to market than ASICs, and while the latter has a lower cost per unit for a large number of units, the considerable non-recurring cost of producing new ASIC circuits make FPGA much more suited for applications requiring a low number of units. For these reasons, FPGAs have found their place as accelerators in embedded systems, even in industrial applications. As we have seen in the previous section, their DPS blocks and reconfigurable logic make them very efficient at processing linear functions, composed of additions and multiplications, or logic functions, like comparisons. Moreover, one particular feature of FPGAs that distinguishes them from common processing devices like CPU, is their parallelism. The array structure combined with the grid-like clocking network makes them excellent for processing data in parallel. Most digital signal processing algorithms can take advantage of these two features, proficiency in linear computing and parallelism. For this reason, FPGAs have been widely used in embedded systems to process digital signal processing functions faster than a CPU would. Even though an ASIC dedicated to this function would reach better efficiency than an FPGA, there are cases where an FPGA is more interesting. Indeed, the reconfiguration offers the opportunity to reprogram the FPGA and change the implemented functions, their parameters, or their performance to better adapt to new use cases and issues after

deployment in a system.

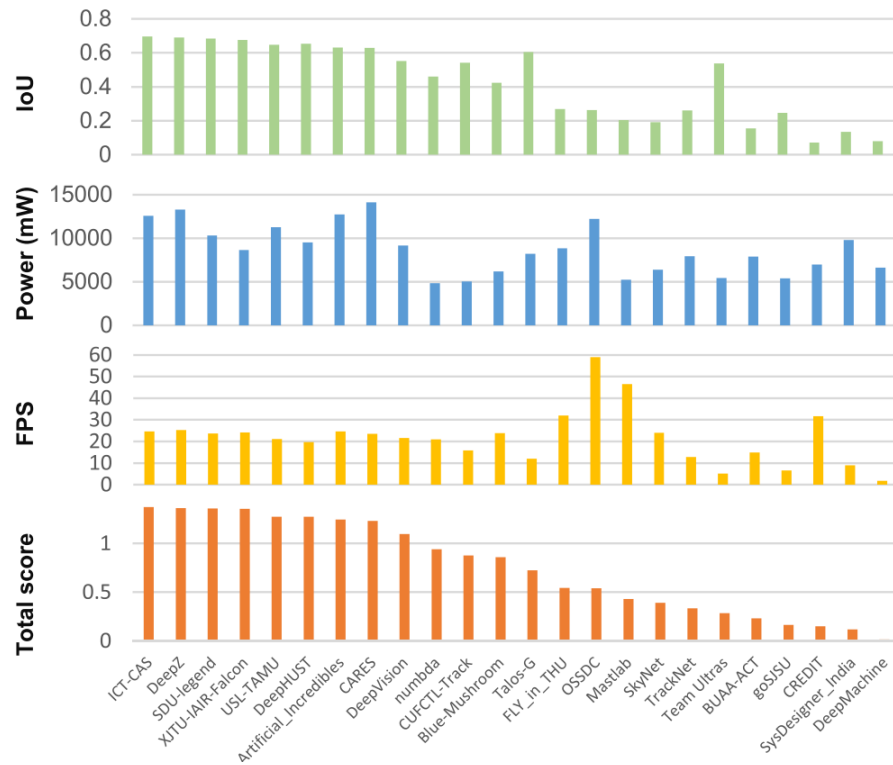
Where it starts to be interesting in our use case is that deep neural networks are essentially digital signal processing algorithms composed mostly of linear functions. Even if there are nonlinear functions such as activation functions, where their implementation on FPGA would be less straightforward, most of the computational cost of deep neural networks lies in convolutions and fully-connected layers, which are only MAC operations, and happen to be extremely compatible with parallelization. The last point is also the main reason why GPUs are so powerful in processing deep neural networks. Their array structure dedicated to parallel computations associated with efficient languages, libraries and compilation chains designed for neural networks makes them undoubtedly the most used hardware devices for the training and inference of deep learning models in cloud computing environments. However, common GPUs are far too power hungry for embedded systems, and in this case, embedded GPUs like the Nvidia Jetson family are much more suited, as they are specifically built for better power efficiency and lower latency than desktop GPUs. One question that arises is: if these GPUs seem like a straightforward solution for deploying neural networks in embedded systems, is there a reason to use FPGAs?

This question is what motivated the System Design Contest (SDC) of the 55th Design Automation Conference (DAC) in 2018 [8]. This contest gave its 110 participants the challenge of implementing a low-power object detection embedded system using any deep neural network they desired but having to choose between an embedded GPU (the Nvidia Jetson TX2) or an FPGA platform (AMD's PYNQ Z1) as the hardware device. The proposed solutions were scored based on accuracy (measured by the IoU), throughput in frames per second (FPS), and power consumption. The scoring results of the GPU and FPGA entries as reported by the conference's publication are displayed in Figure 15. Those results show that, although the GPU implementations are faster most of the time, the FPGA solutions reach similar accuracy while consuming much less power than their GPU counterparts, down to a third of the consumption in some cases, which allowed some FPGA entries to reach similar scores as the top GPU entries, which highlights the interest of FPGAs for deploying neural networks in embedded systems with high power constraints.

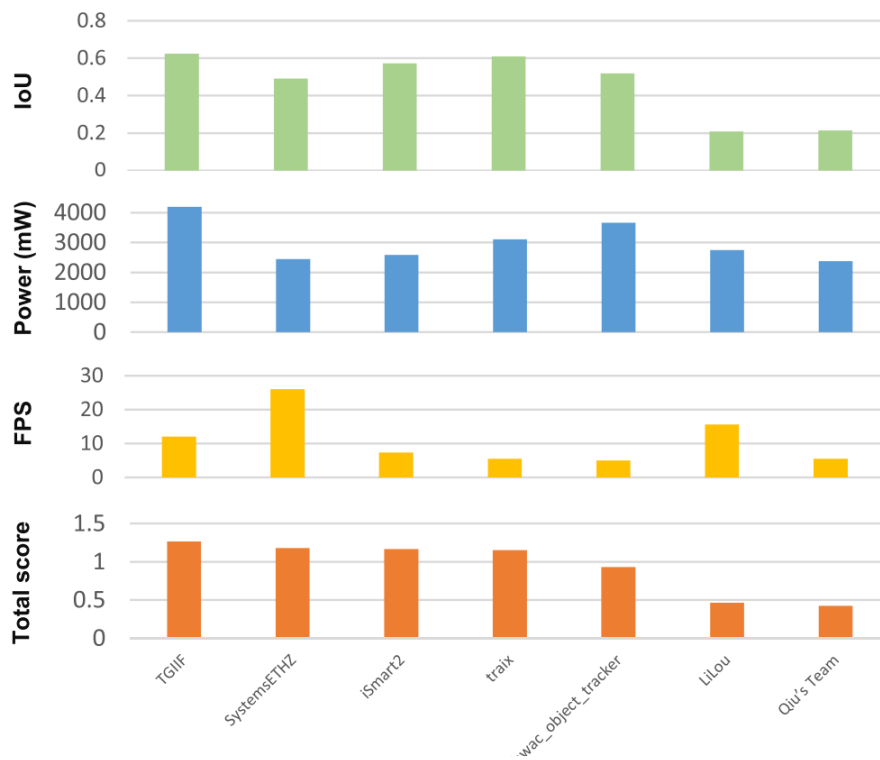
Furthermore, the authors of QuTiBench [132] proposed a benchmark of deep neural networks on heterogeneous hardware and showed that FPGAs can outperform GPUs on throughput and accuracy. They evaluated the performance of two typical classification networks, ResNet50 [47] and GoogleNetV1 [133] on two different devices: a Nvidia Jetson TX2 and an AMD ZCU104 FPGA board. Their results showed that the FPGA provides the highest throughput for both networks and has the best accuracy for GoogleNet, but not for ResNet50.

However, one more thing can be noticed from the results of the DAC-SDC challenge: the great difference in the number of entries between GPU and FPGA. In fact, the authors of the challenge reported that 24 out of 53 participants who chose to use GPU successfully implemented their design, while only 7 out of 61 participants succeeded with FPGA. This can be explained by the much greater difficulty of implementing neural networks on FPGA than it is on GPU, for two principal reasons.

First and foremost, and this is something we have mentioned a few times already, FPGAs are much more constrained in resources than embedded GPUs. As an example, the PYNQ-Z1 used for the challenge only has 5Mb of on-chip RAM. This board is fairly small, but even the ZU19EG, one of the largest AMD FPGAs, only totals 80Mb of RAM. This is ridiculously small compared to the 8GB of memory available to the Jetson TX2



(a) GPU entries



(b) FPGA entries

Figure 15: GPU and FPGA entries of the DAC-SDC Low Power Object Detection Challenge for UAV Applications. Source: [8]©2019 IEEE.

GPU. There is the possibility of using external memory, such as the RAM available on most boards, which is sometimes required to alleviate for high memory footprints of some networks. However, external memory transfers are slow, and using them limits the achievable throughput, making the performances of the design bounded by the memory. Therefore, for performance reasons, a better approach is to try to only use on-chip memory [9], [10], [134]–[136]. This difference means that implementing a model on FPGA requires making more compromises, such as having to compress the model to a point where it starts degrading the accuracy significantly or resolving to select a smaller model that may not perform as well on the task, making it harder to achieve good accuracy and throughput at the same time.

Second, the workflow for deploying networks on FPGA is longer and more difficult. As discussed in the previous section, programming an FPGA requires the design of a dedicated hardware architecture, which is itself a task that requires knowledge of digital electronics and FPGA architectures. Even with the emergence of HLS, which eases and lowers the time cost of designing architectures, there are still more elements to put into consideration, and more trade-offs and optimizations to find that with GPUs, where only software programming of the network and the use of a dedicated compiler (TensorRT [137]) is necessary.

Having the considerations of hardware resources and architecture design hidden behind a level of abstraction plays a lot in favor of GPUs. More time can be spent on finding good training parameters and exploring various compression methods that improve network speed, without worrying about implementation details or lack of hardware resources. In other words, it gives a much smaller design space as finding a design point that maximizes both accuracy and throughput is a matter of deep learning, and not a matter of hardware. This is one of the reasons why GPU entries in the contest generally have better throughput and accuracy than FPGA entries.

In summary, there are clear indications that FPGAs have the potential to surpass embedded GPUs for real-time embedded applications regarding latency, throughput and power consumption. However, implementing deep neural networks on FPGA and reaching good performance is a challenging and time-consuming task that requires joint knowledge of deep learning and FPGA architecture design. Fortunately, various high-level frameworks that help with the hardware architecture design have recently emerged, and are the topic of the next section.

2.3 Architectures and frameworks

As discussed in the previous section, designing hardware architecture for convolutional neural networks on FPGA is a tedious task that requires a lot of time, making it more difficult to explore the design space and find efficient solutions that maximize throughput and accuracy. To address this challenge, several tools or frameworks are being developed to help with the design process by generating FPGA architectures meant for neural network processing. Although they all have their own particularities, they can be organized into two categories based on the type of architecture that they are leveraging.

2.3.1 Deep learning Processing Units

The first category uses a specialized processing engine designed to accelerate standard operations found in neural networks. Although it is not a standard term, we refer to

them as Deep-learning Processing Units (DPUs). They are usually based on systolic array architectures [138], [139] that leverage FPGA parallelism to achieve high throughput. They are meant to be (mostly) agnostic to the neural network architecture and can be used to process the various layers composing neural networks in a sequential manner, which means that basically (almost) any network can be processed by the engine regardless of its structure, provided that the operations are supported. The role of the associated framework is to compile the network into a series of instructions to be given to the processing unit. This type of architecture requires having a microprocessor in conjunction with the FPGA to send the instructions and data to perform the operations. They are generally quite efficient in terms of resources as the same unit is reused throughout the inference, which means two things: they can scale very well to larger networks and having a heavily compressed model is not as important, which means that it is easier to preserve the model accuracy. The main drawback of DPUs is that even if the processing engine itself is able to perform computations at high speed, the multiple data transfers associated with each computation (loading the weights, transferring inputs, and retrieving outputs) lead to larger latency, lower throughput, and increased power consumption. Notable frameworks in this category are:

- VitisAI, a commercial solution from AMD with several proprietary DPU architectures compatible with a limited range of targets. The compiler also offers quantization to 8-bit integers and pruning, and a large collection of pre-compiled models already exists, which makes deployment of common neural networks very fast.
- OpenVINO, the commercial solution from Intel, which offers a solution similar to Vitis AI, but targeted towards Altera FPGAs.
- Tensil, an open-source solution developed by TensilAI, a company independent from FPGA manufacturers. The accelerator is generated using HLS, and has a few tunable parameters, like memory depths or the size of the systolic array. The architecture supports 32-bits and 16-bits fixed-point formats.
- Gemmini [140], a project from the UC Berkeley Architecture Research Group, also using HLS to generate the design, which uses non-standard RISC-V custom instructions, and supports signed 8-bit integer, unsigned 32-bit integer, and 32-bit float data formats. It is part of a larger framework, Chipyard, which is meant for the design of complete systems-on-chip.

2.3.2 Pipelined dataflow architectures

The second category follows a completely different paradigm. Rather than having one large unit to process all layers, each layer is attributed to a processing unit specifically designed for the operations of the layer. Each unit is connected according to the network structure, following the same datapath. The architecture operates as a pipeline, and each unit operates independently and in parallel, thus forming what is called a pipelined dataflow architecture. Since the whole network is deployed on the FPGA, there is usually a limited amount of external data transfers compared to DPUs, as the goal is to only use on-chip memory, so these architectures often achieve much lower latency. However, this full deployment means they require more FPGA resources, which implies that compression plays a much more important role. Moreover, the hardware architecture is completely dependent on the structure of the neural network, which means that the support of a specific neural network is less straightforward and depends much more on the capacities of the framework used to generate the architecture. Therefore, the implementation process can be more difficult than with DPUs, but it offers more customization parameters such as parallelization factors, giving it more flex-

ibility to explore the throughput-resources trade-off, and has the potential to achieve lower latency. Notable frameworks to generate pipelined dataflow architectures are:

- HLS4ML [141], which was originally developed by the CERN for embedded detection of physics particles, and is fully open-source. It uses HLS to generate the processing units based on the description of the neural network and user-defined configuration, such as quantization or to what extent the processing elements should be reused, to balance between throughput and resource usage.
- FINN [9], a fully open-source project from a team of AMD Research. It was originally designed for binary neural networks (*i.e.* with single bit precision) but has been refined over time to support any integer quantization [10]. It is based on a library of HLS templates to generate customizable processing units, as well as a compiler environment to perform the mapping between the neural network layers and HLS templates. Each individual generated hardware module can be parameterized to choose elements such as the parallelism level and which FPGA resources to use for implementation.
- fpgaConvNet [142], a project of Imperial College London. It follows a similar method as FINN, with a tool to create an intermediate representation of the neural network with parallelization parameters that can be optimized to reach desired throughput and latency constraints before being mapped to hardware using HLS templates. The project was originally closed-source, but the authors recently made it available to the community.
- N2D2, a framework to train, quantize, simulate, and deploy neural networks on different targets like CPU, GPU, and FPGA. For the latter, it is through a configurable dataflow architecture called DNeuro. The project is developed by the CEA-List research institute, and the code to deploy on FPGA is currently closed-source.

2.3.3 Choice of framework

All these frameworks are useful for easing the process of deploying neural networks on FPGA, and choosing which framework to work with depends on several considerations such as the purpose of the architecture, the type of application, the level of flexibility and freedom on desired design choices, or the amount of time that can be spent to obtain a solution. From our perspective, which is for real-time embedded solutions, it appears that DPU frameworks, although fairly easy to use, would not offer a satisfying solution regarding latency. The architecture is mostly fixed and does not have enough flexibility to explore hardware design options, like different bit widths for quantization, or balancing out performance and resources. For these reasons, it appears that frameworks that produce pipelined dataflow architectures are more suited for our goal of studying the relationships between neural network architectures and FPGA hardware architectures with the intent of identifying design choices that lead to efficient real-time embedded systems for semantic segmentation.

Regarding the choice of framework within this category, at the time when this thesis began, FINN appeared to be the most interesting choice. It is well supported by the developers, and fully open-source, which means it is both open to contributions and modifiable if needed, and it had the largest amount of layer support. Moreover, it is fully end-to-end, with direct compatibility with the neural network training framework we were already using (PyTorch), and the option to also generate drivers to facilitate the deployment on FPGA SoCs, which allows one to rapidly test the architecture. Although

this framework has all the necessary elements to provide a context in which we could explore hardware and neural network architectures co-design, it has a few limitations and requires a deep understanding of its concepts to be able to use it for complex neural network architectures like the ones often used in semantic segmentation. This point and how to overcome these limitations is the topic of the next chapter, where we present the core concepts of FINN and the contributions that we made to the framework that were necessary to achieve our goals.

2.4 FPGA architectures for real-time semantic segmentation

Before starting to dive into FINN and presenting our contributions, we want to give an overview of the state-of-the-art of FPGA implementations of real-time semantic segmentation models and identify how they relate to the two types of architectures presented above. The papers that we gather in this section are those that explicitly used models trained on CamVid or Cityscapes. We have found some works in the literature that study semantic segmentation, but the datasets used are not the usual benchmarks, which makes it impossible to compare, so we do not include them here.

2.4.1 CamVid

Model	Arch. HW	mIoU	Target FPGA	Res.	Latency	FPS
AlexNet-SFCN [143]	Pipelined	44.90%	ZU9EG	300x225	-	165
SegNet [144]	DPU	41%	Arria 10 GX 1150	480x360	2891 ms	-
UNet [145]	Pipelined	67.8%	Alveo U200	480x352	-	123
BEDN [146]	Pipelined	59.9%	ZU7EV	480x360	-	25.89

Table 2: Comparative table of previous FPGA implementations of semantic segmentation networks benchmarked on CamVid.

Regarding the implementations of neural networks benchmarked on CamVid, Shimoda *et al.* (2019) [143] proposed an implementation of an encoder-decoder model based on FCN [55] using the encoder part of AlexNet [1] and replacing the transposed convolutions of the FCN decoder by bilinear upsampling. They used unstructured pruning to reduce the number of weights of the convolution filters by 94%, and used knowledge distillation to retrain the model after pruning, using the dense model (*i.e.* without pruning) as a teacher. The final model achieves 45% mIoU on CamVid, which is quite low considering that state-of-the-art models at the time were in the 65-70% range [100], [101], [147]. However, the model has only 0.5M parameters and a very simple structure without complicated operations, which makes it easier to implement on FPGA. Concerning hardware, they implemented a hand-made dataflow architecture, with a processing unit using several parallel MAC units for convolutions and a separate processing engine for ReLU and batch normalization layers, while the interpolation is offloaded to the CPU. They also have extra memory elements to store the coordinates of the nonzero pixels in convolution filters to correctly perform convolutions with sparse kernels. The design was deployed on AMD ZU9EG FPGA and reached 165.4 FPS, using inputs with 300x255 pixels of resolution due to memory constraints. The authors did not report any latency; however, for pipelined architectures, the latency can be roughly estimated as the inverse of the throughput multiplied by the number of inputs. To be accurate, the latency would be exactly the inverse when considering only the FPGA, but we must add the latency of data transfers with the SoC that provides the inputs and collects outputs, making this only a rough approximation. This would give a latency of

approximately 6 ms. The authors reported a 24.49x higher power efficiency than a Jetson TX2 embedded GPU. However, their experiments on GPU were done simply with Python inference rather than compiling with TensorRT, which gives performance lower than the actual capacity of the GPU.

The same year, Yu *et al.* proposed an implementation of SegNet [144] using an HLS-based DPU architecture that has processing units for convolution, ReLU, MaxPooling and MaxUnpooling. The pooling indices are quantized to 2 bits and stored in off-chip memory, while the weights and activations are on 8 bits. In addition, the authors merged batch normalization layers with convolutions to reduce the number of required computations. The compressed model achieves 41% accuracy on CamVid, and the architecture was implemented on an Intel Arria 10 GX 1150 FPGA. The author did not report any clear throughput in FPS; however, they reported latency of 2891 ms, which is very high, and highlights that DPUs tend to achieve much larger latencies than pipelined dataflow architectures.

Miyama *et al.* (2021) [145] implemented a modified UNet [85] where the transposed convolutions for upsampling have been replaced by bilinear interpolation. They performed quantization on 3 bits of the network weights and reported a 67.8% mIoU on CamVid. The method used to obtain the architecture is rather unclear, but it appears to be a dataflow architecture with processing units dedicated to each type of layer that are executed in a sequence that is executed multiple times in parallel. They implemented the architecture on AMD Alveo U200 which is more of a FPGA for datacenter applications than for embedded purposes and obtained 123 FPS with 480x352 inputs. They also compared their implementation with one obtained using Vitis AI's DPU, which reached 32 FPS.

Finally, Kim *et al.* (2021) [146] proposed a straightforward binarized encoder-decoder network with a symmetric structure between the encoder and the decoder, using transposed convolutions for upsampling, which they called BEDN. Working with single precision makes hardware implementation much simpler, as a MAC operation can be replaced by an XNOR logic gate (*i.e.* the logical complement of the exclusive OR) and a *popcount* operations (*i.e.* counting the number of ones in the vector of bits), and batch normalization can be expressed as a simple comparison with a threshold. The authors obtained an mIoU of 59.9% on CamVid, which is acceptable for a single precision quantization. The hardware architecture was generated using FINN and is a pipeline composed of processing units dedicated to each layer, containing arrays of processing elements with parameterized sizes. The authors reported a throughput of 25.89 FPS with 480x360 inputs on AMD ZU7EV FPGA, which is a rather small device, and reported a very low power consumption of only 2W.

2.4.2 Cityscapes

Model	Arch. HW	mIoU	Target FPGA	Res.	Latency	FPS
EnetHQ [148]	Pipelined	36.8 %	ZU9EG	240x152	3 ms	-
ENet [149]	DPU	63.1 %	Zynq 7035	1024x512	720 ms	32.9
DeepLabV3+ [150]	DPU	65.29 %	Arria 10 GX 1150	960x960	670 ms	-
FPN [151]	DPU	57.6 %	ZU9EG	512x256	-	128
MobileNet-PSPNet [152]	Pipelined	64.2%	ZU9EG	1024x512	9.1 ms	139

Table 3: Comparative table of previous FPGA implementations of semantic segmentation networks benchmarked on Cityscapes.

Ghielmetti *et al.* (2022) proposed an implementation of ENet [96], a lightweight real-

time semantic segmentation model, using the HLS4ML framework to produce a pipelined dataflow architecture, and improved the implementation of the convolution of the framework by designing a more efficient algorithm. They also performed some optimization of the buffer insertion (*i.e.* inserting memory elements between processing units to balance the differences in the input/output dataflow of each unit) that is necessary for the pipeline to achieve good performance and reduce the memory footprint of the architecture. They modified the original model and replaced some complex convolutions (asymmetric, dilated, and strided) with regular ones, increasing the number of parameters to 1.1M, but also the mIoU to 63.2%. They used a heterogeneous quantization scheme on the model, where layers are assigned different bit widths, and pruned the model to reduce the number of filters in convolutions, halving the number of parameters. This drastically reduced the performance of the model, bringing the mIoU to 36.8%. Interestingly, the pixel accuracy was 81.1%, which was the highest among the different quantization and pruning experiments carried out by the authors and could explain why they kept this version. This is a good example that illustrates the argument we presented in Chapter 1 that pixel accuracy is not a good metric to judge the performance of a semantic segmentation model. The authors also used a low resolution of inputs of 240x152, which allowed them to implement the architecture on AMD ZU9EG FPGA and to obtain a very low latency average of 3 ms, however, it significantly reduces the mIoU of the model.

Jia *et al.* (2021) [149] also implemented ENet, but used the Vitis AI DPU to deploy the network on AMD Zynq 7035 with INT8 quantization, with which they report 63.1% mIoU. Thanks to the low resources requirements of using a DPU, they have been able to use a large input size of 1024x512 pixels and obtain 32.9 FPS of throughput. Regarding latency, considering only the DPU processing time gives an average latency of 30.38 ms, however, when taking into account the latency of data transfers, it adds up to 720 ms.

Mori *et al.* (2022) [150] implemented DeepLabV3+ [89] on Intel Arria 10 GX 1150 FPGA, using a handmade DPU design. They performed channel pruning by simulating the latency of the architecture and using it as a parameter to a genetic algorithm to select the more optimized pruning configuration, reducing the total latency by 2.44x compared to the baseline, and obtained an mIoU of 65.29% on Cityscapes, only 2% behind the baseline. The authors resized the images to a 960x960 resolution and did not report a throughput in FPS, but noted a latency of 670 ms.

Fang *et al.* (2018) [151] designed a DPU using HLS to accelerate a semantic segmentation network based on FPN [153], and obtained an mIoU of 57.6%. They obtained a throughput of 128 FPS with 512x256 inputs when deploying this architecture to AMD ZU9EG. Their architecture can also be used for accelerating an object detection network.

Finally, Sada *et al.* (2019) [152] developed a pipelined dataflow architecture for a sparse encoder-decoder based on MobileNet [45] and PSPNet [154], using a similar unstructured weight pruning technique as [143], achieving 64.2% mIoU. The authors remain very vague about how they actually implemented the architecture. However, they took an interesting route by proposing a tiling algorithm, which allowed them to keep the image resolution as high as 1024x512 for inference by segmenting 224x224 tiles and reconstructing the image back as a post-processing step. Although the tiling reduces the overall throughput, they still reported 139 FPS for the whole inference.

2.4.3 Observations

This overview of the literature on semantic segmentation of road scenes on FPGA exhibits that there is a general trend of using HLS, either through frameworks or custom-

made, to generate the hardware architectures and that pipelined dataflow architectures tend to achieve lower latency than DPUs. Moreover, the general approach to the task is through the selection of a network, application of quantization and/or pruning, and implementation, sometimes including discussion of hardware optimizations and hardware design choices. However, the discussions regarding the choice of neural network are often very brief, if ever present, and rarely highlight the reasons why a specific neural network was chosen over others. Furthermore, when the original network is modified to reduce its complexity and facilitate the integration, there is a lack of comprehensive studies of these changes and how they influence the end results. In general, we observe that the model compression and hardware optimizations side of the task is much more studied than the neural network design side. This appeals for more research of the relationship between neural network architecture and the exploration of the trade-off between accuracy and hardware performances. Moreover, it is rather difficult to extract clear methodologies to apply them to different networks, or different contexts. This is even more difficult because this research field lacks open-source code or hardware descriptions.

2.5 Summary

In this chapter, we presented what are FPGAs and how they can contribute to real-time embedded neural networks. We displayed clear evidence that they can be more suitable than embedded GPUs in some situations. We discussed the difficulties in implementing neural networks on FPGA and the tools that can be used to facilitate this process. We selected FINN, an open-source framework, as the most adapted solution to obtain hardware architectures of neural networks that achieve real-time performance and as an open environment that is suited for our goal of hardware-aware exploration of the neural network design space. In the next chapter, we dive into the core concepts of FINN, identify its strengths and weaknesses, and present the enhancements that we brought to the tool to overcome the few elements that were preventing us from reaching our goal.

Chapter 3

FINN: a framework for fast, scalable quantized neural network inference on FPGAs - Principles and contributions

Contents

3.1	Detailed introduction	72
3.1.1	Hardware Library	72
3.1.2	Compiler	73
3.2	Dealing with residual networks	78
3.3	Contributions to the framework	80
3.3.1	New <i>MakeMaxPoolNHWC</i> transformation	81
3.3.2	The <i>SplitLargeFIFOs</i> transformation	82
3.3.3	Improved support for nearest neighbor upsampling	82
3.3.4	Support of transposed convolutions	83
3.4	Summary	84

This chapter is centered around FINN, an open-source framework to deploy neural networks on FPGA for inference. We give a detailed introduction to FINN, to provide an overview of its concepts and principles. This allows us to introduce discussions on the issues and limitations we identified that was preventing the implementation of semantic segmentation networks and the solutions we proposed to overcome these issues. These solutions come in the form of new elements added to the tool that contribute to improving the support for semantic segmentation networks and are now merged with the official FINN GitHub repository.

3.1 Detailed introduction

FINN is an open-source tool that leverages templated HLS and RTL building blocks to produce a scalable pipelined dataflow hardware architecture from a graph-based representation of a quantized neural network. It supports arbitrary precision of weights and activations, as long as it is uniform scale integer quantization. We already discussed how most hardware implementations, especially FPGAs, are much more efficient regarding throughput and power when using integer quantization (or fixed point) than with floating point. Moreover, scale quantization is the most hardware-friendly method of quantizing neural networks, as it greatly simplifies the computations between elements with different quantization parameters. The extra logic necessary for more flexible quantization methods to perform computations on hardware makes it usually not worth the extra points in accuracy it can bring [31], [32].

FINN is built on two main components: a hardware library and a compiler environment. In this section, we provide explanations of these components to give a comprehensive understanding of the tool and help introduce later discussions on its limitations and how they constrain neural network design and justify the need to introduce new elements.

3.1.1 Hardware Library

The FINN hardware library is a collection of customizable C++ HLS (Vitis HLS, to be specific) functions and RTL modules that describe layers and operators commonly found in neural networks and operate in a streaming dataflow manner, consuming and producing data whenever available. Each block can be tailored to meet the user's requirements through a set of attributes ranging from layer-specific to parallelism-related parameters, allowing the manipulation of each block's latency and resource usage independently. Each block is synthesized separately using the AMD tools (Vitis HLS and/or Vivado) before joining them by implementing AXIStream (a common streaming handshake protocol in hardware design) interfaces between them, producing a block design called the *Stitched IP* whose structure reproduces the neural network it implements.

To give an example of a hardware block, the MatrixVectorActivationUnit (MVAU) is a key element to the implementation of the most fundamental and computationally heavy element of Convolutional Neural Networks (CNNs): the Convolution-BatchNorm-Activation set, also known as "convolution block". The convolution is transformed into a matrix-matrix operation using an ImageToColumn (Im2Col) algorithm [155] and is decomposed into multiple matrix-vector computations [9]. The BatchNorm layer and the activation layer that follows are converted into a simple comparison with thresholds and merged with matrix multiplication to form the MVAU, whereas the conversion from tensor to matrix (*i.e.* the hardware implementation of Im2Col) is performed by another element called the SlidingWindowGenerator (or ConvolutionInputGenerator).

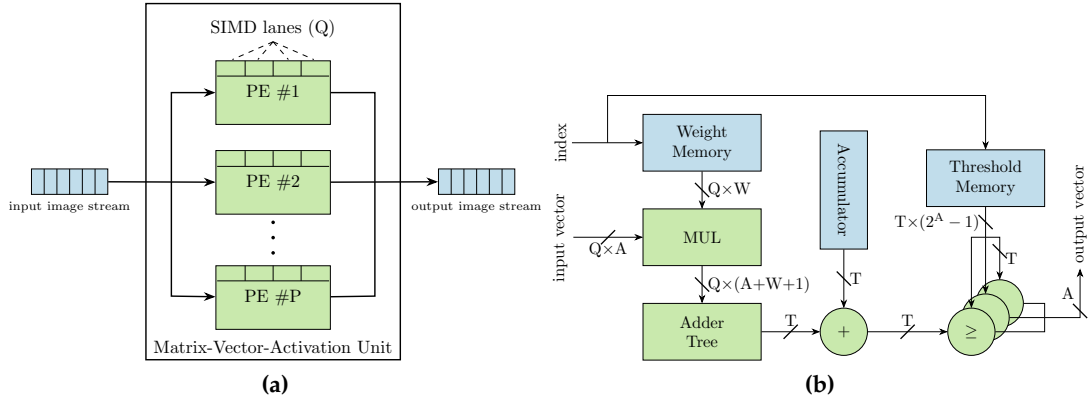


Figure 16: The MVAU SIMD systolic array (a) and its PEs (b). The PEs are operating a multiplication-accumulation with A -bit inputs and W -bit weights followed by a comparison with T -bit threshold values. The amount Q of SIMD lanes and P of PEs can be customized to obtain a chosen parallelism. Figures reproduced from [10].

The MVAU is organized as a systolic array (see Figure 16a) with a Single Instruction Multiple Data (SIMD) architecture. Each Processing Element (PE) of the array operates a multiplication-accumulation with weight values stored in its local memory and applies a comparison with threshold values stored in local memory, as shown in Figure 16b. The amounts Q of SIMD lanes and P of PEs can be manually set and are unique to each MVAU. The matrix-vector product is folded so that, in each cycle, a Q -wide and P -high tile of the input matrix is processed by the systolic array. The total computation of one matrix-matrix multiplication requires a number of such matrix-vector products equal to the number of pixels in the output image. Each hardware module of FINN has similar parallelism parameters, although, the parallelism of most of them is only one-dimensional. This allows balancing the latency and resource usage of each pipeline element, which is essential to optimize the architecture to achieve the desired throughput while remaining within the capabilities of the target FPGA. On top of that, some modules have options to select the type of resources that must be used for MACs (LUTs or DSPs) and memory (LUTs or BRAMs).

3.1.2 Compiler

The FINN compiler is a Python library whose purpose is to create and customize the RTL/HLS code of appropriate hardware blocks from a description of the neural network, as well as automate every step of the compilation of HLS code, synthesis, and implementation necessary to produce an architecture that can be deployed on target.

It starts from a representation of the neural network in the Open Neural Network eXchange (ONNX) format, which displays the layers of the network as nodes and the input/outputs of the network as edges, thus forming a graph. This graph originally contains nodes symbolizing standard operators commonly found in neural networks. Since the ONNX format does not naturally handle quantization, FINN uses as input a graph in the QONNX [156], [157] format, which is a superset of ONNX that adds several custom nodes to support quantization, as well as operators used to represent elements used in intermediate steps of FINN, such as the *Im2Col* operator or the *MultiThreshold* operator that implements the comparison with thresholds to perform activation layers and batch normalization layers. Additionally, FINN defines several custom nodes to represent its specific operators, which are meant to represent the different elements of the hardware library, which are referred to as *hardware layers*.

This graph can be modified by adding, deleting, or modifying nodes or edges. This is achieved through operations called *transformations*, which are used to produce a new graph that contains only nodes representing operators that are direct representations of a hardware block of the hardware library, with only quantized inputs and outputs remaining. To do so, the graph goes through two series of transformations that constitute the main parts of integrating a neural network on FPGA with FINN, which are called *streamlining* and *conversion to hardware layers*. The nature and order of the transformations that must be performed to achieve these steps are extremely network-dependent.

3.1.2.1 Streamlining

The principle of streamlining [158] is to modify the network graph to map it to the hardware layers afterward. It simplifies the graph and removes remaining floating-point operations, such as batch normalization or multiplication, with scaling factors for quantization/dequantization. It takes advantage of the fact that, as mentioned previously, quantized activation functions (*i.e.* returning a quantized output, acting as both an activation function and a quantizer) can be seen as comparisons with thresholds (the so-called *MultiThreshold* operator) and leverage the ability of subsequent linear operations to be merged and absorbed into thresholds by updating their values. More precisely, any uniform b -bit quantized activation function f can be expressed as a successive thresholding operation T followed by a linear transformation. The successive thresholding operator T is defined as, for a list of unevenly spaced thresholds $\mathbf{t} = \{t_i \in \mathbb{R} | i \in \llbracket 0, 2^b - 1 \rrbracket\}$ and an input $x \in \mathbb{R}$, it returns the number of threshold values to which x is greater than or equal to:

$$f(x) = aT(x) + b \quad (41)$$

$$T(x, \mathbf{t}) = \begin{cases} 0 & \text{if } x < t_0 \\ i & \text{if } t_{i-1} \leq x < t_i, i \neq 0 \end{cases} \quad (42)$$

where a and b are real values, usually the scaling factor and zero of the uniform quantizer. The thresholds can be different for each input channel, depending on the quantization granularity. This is a very efficient technique to represent non-linear quantized activation functions. However, the number of thresholds grows exponentially with the bitwidth b , making it more suitable for lower ones.

Based on that, any linear operation that occurs before the activation function can be absorbed into the successive thresholding operator, by updating the threshold values, as per the following inequation:

$$\begin{aligned} t_i &\leq a \cdot x + b < t_{i+1} \\ \frac{t_i - b}{a} &\leq x < \frac{t_{i+1} - b}{a} \end{aligned} \quad (43)$$

provided that $a > 0$. This is very useful for dealing with scaling factors and batch normalization, as the latter is essentially a channel-wise linear function. In fact, based on equation 12 of section 1.2:

$$\begin{aligned} y_{l,n,c,h,r} &= \frac{x_{l,n,c,h,r} - \mu_l}{\sqrt{\sigma_l^2 + \epsilon}} \gamma_{l,c} + \beta_{c,l} \\ &= \frac{\gamma_{l,c}}{\sqrt{\sigma_l^2 + \epsilon}} x_{l,n,c,h,r} + \beta_{c,l} - \frac{\gamma_{l,c} \mu_l}{\sqrt{\sigma_l^2 + \epsilon}} \\ &= a \cdot x_{l,n,c,h,r} + b \end{aligned} \quad (44)$$

with $a = \frac{\gamma_{l,c}}{\sqrt{\sigma_l^2 + \epsilon}}$ and $b = \beta_{c,l} - \frac{\gamma_{l,c}\mu_l}{\sqrt{\sigma_l^2 + \epsilon}}$, which, for each channel, are fixed scalar values at inference time. Therefore, the batch normalization parameters can be organized as two vectors of as many floating-point elements as there are input channels.

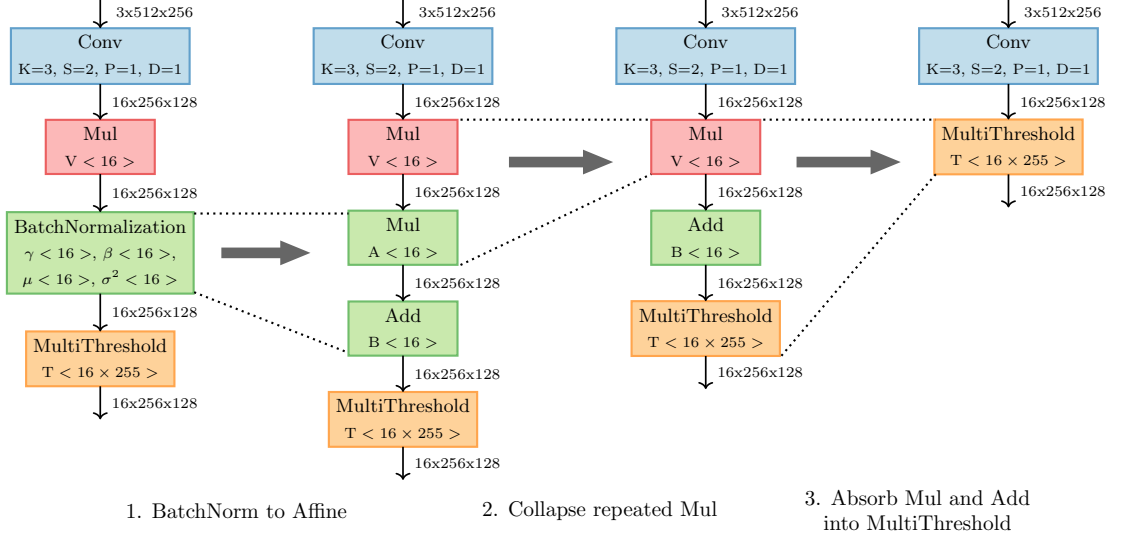


Figure 17: Illustration of the floating-point elements removal as part of the streamlining of a convolution block. The notation $\langle \rangle$ indicates the dimensions of a matrix or vector.

To illustrate how this is put together in practice, we provide an example of floating-point elements removal using streamlining in Figure 17. We take the common example of an ONNX graph of a convolution block operating on a $3 \times 512 \times 256$ input (*i.e.* three channels of 512×256 feature maps). It starts with a *Conv* operator with a kernel of size $K = 3$, stride $S = 1$, padding $P = 1$, and dilation $D = 1$ and 16 output channels. Then it is followed by a *Mul* which is a multiplication with a vector of length 16 there to apply the quantization scaling, as we took the example of channel-wise scale quantization, which means each channel has a different scaling factor. We then have the *BatchNormalization* node with its associated parameters that are also 16-element long vectors, followed by the *MultiThreshold*, which uses a threshold matrix of dimensions 16×255 in our example where we took $b = 8$. Note that, if we wanted to be accurate, the *MultiThreshold* node should be followed by a *Mul* node that multiplies with the scaling factor associated with the quantized activation function. We did not include it in the illustration for simplicity, as it would be dealt with by absorbing it into the following nodes in practice. To get rid of floating-point elements we can perform the following transformations:

1. Convert batch normalization to an affine function following equation 44, transforming the *BatchNormalization* node into a *Mul* node with a 16-element long vector A followed by an *Add* node with a 16-element long vector B.
2. Collapse the two channel-wise multiplications in a single *Mul* node by changing the multiplication vector V.
3. Absorb the channel-wise multiplications and additions in the *MultiThreshold* node, following inequation 43.
4. Round the floating-point thresholds to the nearest greater integer (*i.e.* applying the *ceil* operator), which does not change the result of the comparisons, as the input is also an integer.

These are some of the most frequent transformations that happen in the streamlining of a CNN to get rid of floating-point elements, but several other transformations are present in the compiler and should be used depending on the network's structure.

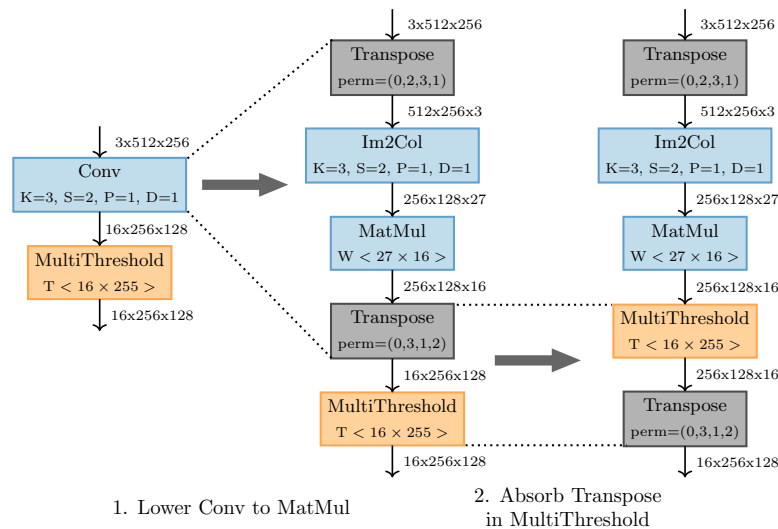


Figure 18: Illustration of the lowering of convolution and data layout conversion as part of the streamlining of a classical convolution block. The notation $\langle \rangle$ indicates the dimensions of a matrix or vector.

The next core element is the lowering of convolutions, which is their conversion to an *Im2Col* followed by a matrix multiplication. We illustrate this transformation by an example displayed in Figure 18, where we take the graph obtained after the transformations presented above, and transform the *Conv* node into an *Im2Col* node and a *MatMul* node which performs the multiplication with the weight matrix. As can be noticed in the figure, doing so also requires changing the order of the input and output tensors' dimensions, moving the channel dimension to the last position. Indeed, the *Im2Col* node, as well as all the FINN custom nodes, use a channel-last data layout, referred to as the *NHWC* data layout (for Number of batches, Height, Width, Channels, with *N* always being 1 in the graph in practice, which is why it is not represented in the figure), as opposed to the usual *NCHW* data layout used in common deep learning frameworks. The reason is related to the design of hardware modules, as having a channel-last layout requires less buffering, since in a convolution, output pixels depend on all the input channels, but not all the input pixels. As a result, the hardware templates use a *NHWC* layout, with a channel-wise SIMD parallelism. However, switching from one layout to the other is not straightforward, as the graph must remain consistent regarding the layout of the inputs and outputs of nodes during streamlining, so the model can be properly executed for testing, and most standard ONNX nodes are meant exclusively for *NCHW* layouts. Therefore, *Transpose* nodes are inserted when necessary, to preserve the data layout consistency of the graph. They implement a simple permutation of dimensions based on a *perm* parameter which defines how the dimensions should be reordered. These nodes are here only for intermediate steps during the streamlining and have no hardware equivalent whatsoever, which means they must be removed before the hardware conversion. This can be done by moving them through other nodes when allowed and updating their data layouts, as shown in the second step of Figure 18, with the example of a *Transpose* node being pushed through a *MultiThreshold* node. The process is then to do so until the *Transpose* node is adjacent to another *Transpose* with a symmetric permutation, in which case they will cancel each other, and can be safely removed from the graph.

These three steps, removing floating-point operations, lowering convolutions, and converting to *NHWC* data layout are the fundamentals of streamlining, which is an essential step to correctly map neural network layers to hardware layers in the following step. It constitutes the major part of integration work when working with FINN, as there is no universal selection and ordering of transformations that would work for every network.

3.1.2.2 Conversion to hardware layers

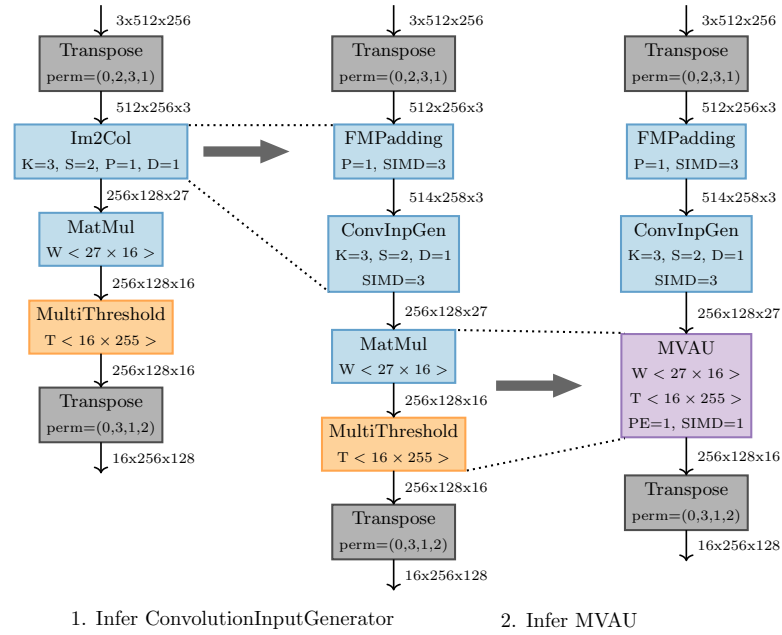


Figure 19: Conversion of a classical convolution block to hardware layers. The notation $\langle \rangle$ indicates the dimensions of a matrix or vector.

Once the graph has been correctly streamlined and every node contains only quantized elements in the channel-last format, the nodes can be replaced by their hardware equivalents. For each of them, a new node that represents a hardware module will be inferred in its place, with attributes that will later serve as parameters for the hardware templates. Following the same example we illustrate the conversion to hardware layers of a convolution block in Figure 19. The *Im2Col* node is converted into a *FMPadding* node which applies the input padding, followed by the *ConvInpGen* which produces the sliding windows for the matrix multiplication. Then, the *MatMul* and *MultiThreshold* nodes are merged to form the *MVAU* that we mentioned earlier. Each hardware node has parallelism parameters in the form of SIMD lanes and, if applicable, PEs.

3.1.2.3 Synthesis and integration

Once such a graph is obtained, the parameters of the operators can be tuned and profiled to estimate the throughput and resource usage of the network, which gives the user the possibility to explore the design space without requiring synthesis. The tool then uses the HLS/RTL templates associated with each node to produce HLS code that can be synthesized into hardware modules, which themselves can be interconnected to reproduce the dataflow of the network's graph.

Since the produced hardware architecture works as a pipeline, buffers in the form of First In First Out (FIFO) memories need to be inserted between each operator to compensate for the differences in dataflow between the blocks and ensure that these differences do not introduce bottlenecks, all to guarantee the achievement of the theoretical throughput. To perform this step, the tool uses RTL simulation to measure the required depth (*i.e.* the number of values it can contain) of buffers. Then, the final architecture can be synthesized and implemented using AMD's tools workflow (Vivado or Vitis, depending on the target).

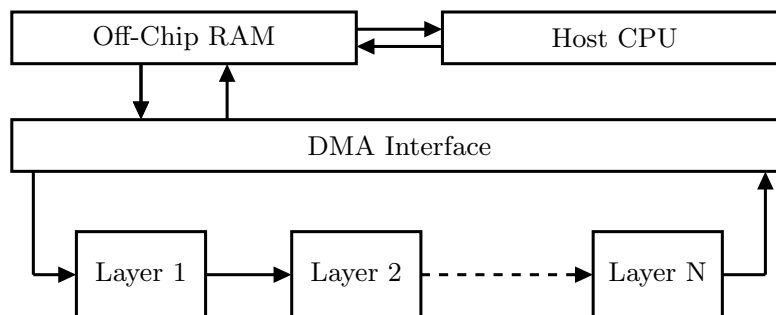


Figure 20: Illustration of the transfer of data between the CPU and the neural network hardware architecture deployed on FPGA through a DMA interface.

The design can then be used independently, or inserted in a larger design. When the FPGA is part of a SoC with a CPU, which is common in evaluation boards meant to prototype and test FPGA designs, a usual way of establishing communication between the CPU and the FPGA is to use Direct Memory Access (DMA) interfaces, which, as the name suggests, implement interfaces between the SoC memory and the FPGA, allowing the transfer of data between both. The CPU can then be used to perform pre-processing on the data if necessary and use the interface to send inputs to the design and retrieve outputs after they have been processed. This is a very convenient way of rapidly deploying and testing applications.

3.2 Dealing with residual networks

As we presented above, FINN is a collection of hardware operators and transformations to map a neural network structure onto them. Therefore, the feasibility of a network to be integrated with FINN depends on whether the layers it contains are supported (*i.e.* they have a hardware equivalent) and whether the structure can be streamlined correctly. This can limit the range of networks that can be selected when designing a system, depending on how complex the task is, and how complex the networks that are trained on this task are. FINN was created with the intent to help deploy rather simple neural networks, like multi-layer perceptions (networks with only fully-connected and activation layers), or simple CNNs, and has been progressively enhanced as needed by the developers, or external contributors, to carry out their projects. When we started learning about how to use FINN and started to experiment with it on semantic segmentation networks, we faced a few challenges with their implementations.

FINN was mostly meant to deploy rather simple neural networks, like multi-layer perceptions (networks with only fully-connected and activation layers), or simple CNNs, and required some improvements to support the studied semantic segmentation networks. In fact, they tend to employ complex operators such as different types of efficient convolutions, some of which were not supported by FINN, which constrained the choice of the network to work with and implied making compromises, adapting

the layers used in the network, or developing new elements in the framework when needed. This point will be approached in greater detail in the next chapter where we present our experiments and contributions.

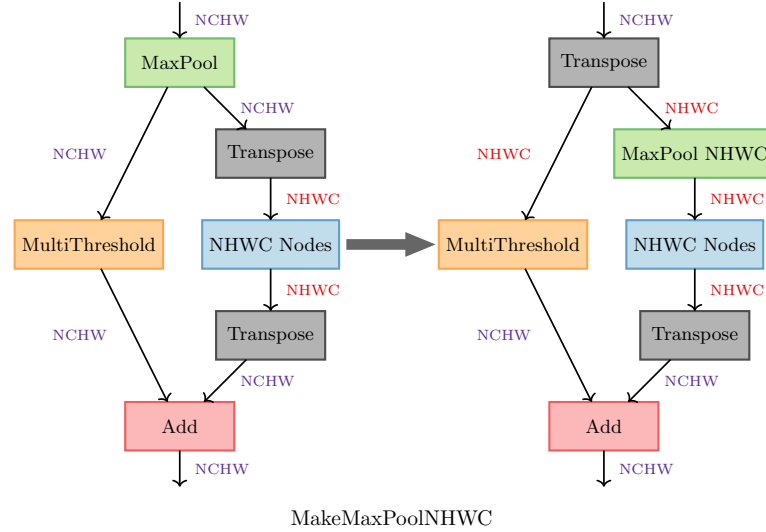


Figure 21: Illustration of applying the *MakeMaxPoolNHWC* transformation when the *MaxPool* node is a fork node.

Moreover, they tend to have complex structures, either with different branches or skip connections between layers in the case of encoder-decoder structures. FINN’s streamlining transformations were designed with linear topologies in mind, *i.e.* networks with a straightforward structure where every layer has only one input and one output, without any form of branching. Therefore, many elements during streamlining do not initially work correctly with residual topologies, *i.e.* network structures with skip connections. A successful attempt at integrating a residual network (ResNet50) with FINN had been made before the start of this thesis [159], which set a foot in making FINN compatible with such networks. However, our first experiment with semantic segmentation models exhibited that some elements were still difficult to streamline correctly, due to an incompatibility between the structure of the network and the way the transformations are implemented in the framework.

The first issue encountered was related to the *MakeMaxPoolNHWC* transformation, which as the name suggests, transforms a standard *MaxPool* node into its variant with *NHWC* data layout, so it can be mapped to its hardware equivalent correctly. This transformation introduces *Transpose* nodes before and after the max pooling to perform the conversion between different layouts. However, in the event where the *MaxPool* node is a fork node (*i.e.* a node with more than one output), applying this transformation will lead to a situation as shown in Figure 21. The transformation does not account for the particular structure of the network and creates a situation that is not only incorrect from a functional point of view, as the left branch is no longer connected to the output of the *MaxPool* node but also creates an inconsistency in data layouts between the input and the output of the *MultiThreshold* node.

Another important element that can be an issue when working with residual topologies is the placement of FIFOs. There is a functional difference between FIFOs insertion in the case of linear networks and for residual networks, as displayed in Figure 22. When there is only one branch, incorrectly sized buffers might lead to a bottleneck that will lower the achievable throughput, as we discussed earlier. However, in the case of

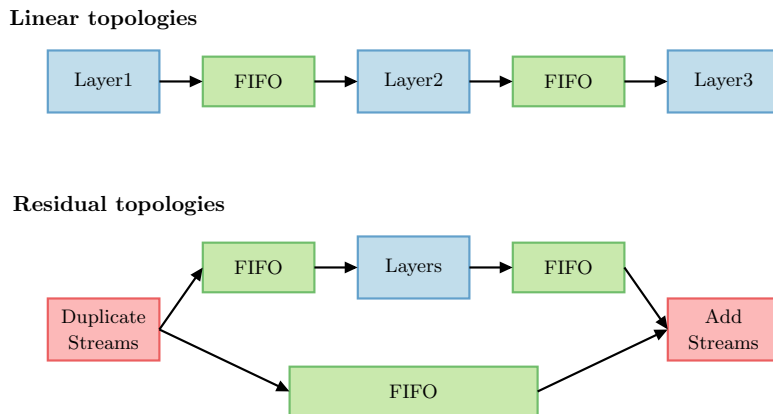


Figure 22: Illustration of the difference between linear and residual topologies regarding the insertion of FIFOs.

networks with skip connections, there is an additional problem. In most cases, one of the two branches will hold much more computations than the other, which means that in terms of dataflow, one path is faster than the other in providing data. However, there is synchronization to be made at the point where the two paths are eventually merged, as the *AddStreams* node is waiting for both branches to provide input so they can be added together. If the buffer on the light branch is not deep enough to compensate for this difference in latency, and depending on how the layers in the heavy branch are consuming and producing data, it might lead to the pipeline stalling completely, resulting in a situation called a deadlock, in which no output can be produced. We found that, in many cases, the automatic simulation-based insertion of FIFOs led to incorrect results, with some buffers being too small, creating such deadlock. Moreover, in some cases, it was impossible to create buffers that would be large enough to solve the issue, due to limitations of the tool used for synthesis (Vivado), which limits the size of each individual FIFO element to a depth of 2^{15} .

These issues incarnate some of the limitations of FINN regarding the integration of semantic segmentation networks. However, thanks to its open-source code, we have been able to design new elements to overcome these difficulties, contributing to the improvement of the framework at the same time.

3.3 Contributions to the framework

In order to solve the issues encountered when attempting to integrate the networks studied in our experiments, as well as a way to contribute to the tool by improving the support for semantic segmentation networks, we introduced new elements to the framework. These elements are in the form of new transformations to address the points presented in the previous section. We also improved the support of upsample layers. Part of these new elements, as well as multiple small contributions in the form of bug fixes, are now part of the official FINN GitHub repository. Although they are presented without context here, they will be related to our experiments in the next chapter, where we will present our main contributions to real-time semantic segmentation on FPGA.

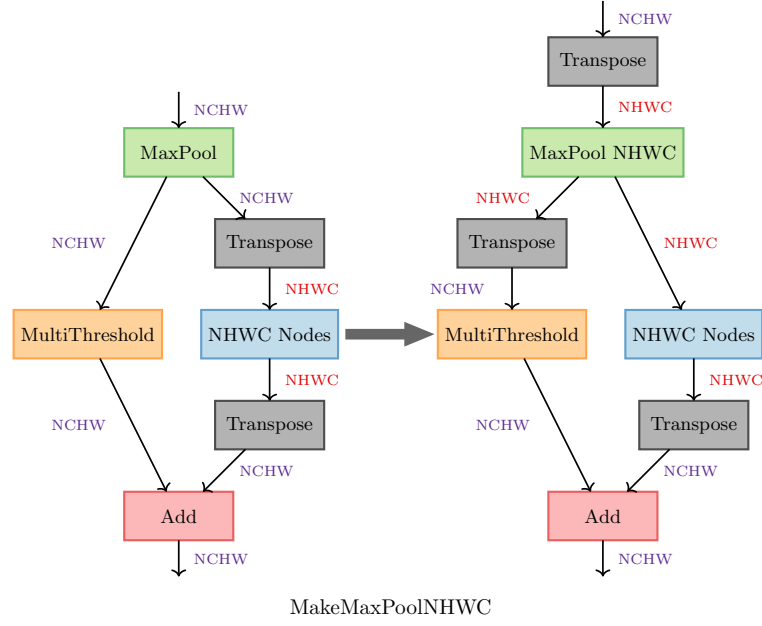


Figure 23: Illustration of the new *MakeMaxPoolNHWC* transformation that we introduced as a solution to the problem illustrated in Figure 21

3.3.1 New *MakeMaxPoolNHWC* transformation

The issue of the *MakeMaxPoolNHWC* transformation discussed in the previous section comes from the way the transformation was originally implemented, which is as follows:

1. Check if the consumer node (*i.e.* the node whose input is the output of the current node) is a *Transpose* with a (0, 2, 3, 1) permutation (NCHW to NHWC), which means the node following the *Transpose* is in channel-last layout.
2. If so, change the data layout of the *MaxPool* node and swap its position with the consumer *Transpose* node.
3. If not, and if the producer node (*i.e.* the node whose output is the input of the current node) is a *Transpose* with a (0, 3, 1, 2) permutation (NHWC to NCHW), change the data layout of the *MaxPool* node and swap its position with the producer *Transpose* node.

In other words, it searches for a $\{MaxPool \rightarrow Transpose\}$ or a $\{Transpose \rightarrow MaxPool\}$ pattern, and swaps the position of the nodes. However, as we mentioned earlier, this algorithm does not take into account that the *MaxPool* node can be a fork node, in which case it has more than one consumer. Therefore, we modified the transformation to take this into account. If the node is a fork, and at least one consumer is a *Transpose*, rather than swapping them, we remove it, and insert new *Transpose* nodes before and after the max pooling, on every other branch, to keep the layouts consistent, as illustrated in Figure 23, which solves the issue. We identified and proposed solutions to these issues in the context of the semantic segmentation networks we studied, and this work could be extended to improve FINN’s support for multibranch networks as a whole.

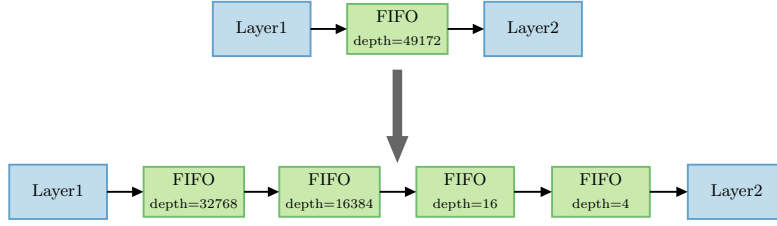


Figure 24: Illustration of the *SplitLargeFIFOs* transformation.

3.3.2 The *SplitLargeFIFOs* transformation

As mentioned in the previous section, Vivado limits the FIFO depth to 2^{15} , which can be too small in some cases, and lead to deadlocks. To overcome this issue, we initially proposed a transformation that splits FIFOs deeper than that into as many smaller FIFO as needed. The FINN development team identified that there was another underlying problem with FIFOs in Vivado, as it constrains them to have a depth being exactly a power of two. This means that, if for example a depth of 49172 is needed, Vivado would implement a FIFO whose depth is the next power of two, which is 65536, thus wasting a lot of memory resources. To circumvent this, FINN has been improved for our original transformation to give the current *SplitLargeFIFOs* transformation illustrated in Figure 24, where a large FIFO is decomposed into smaller ones, each with a depth being a power of two so that no memory resources are wasted.

3.3.3 Improved support for nearest neighbor upsampling

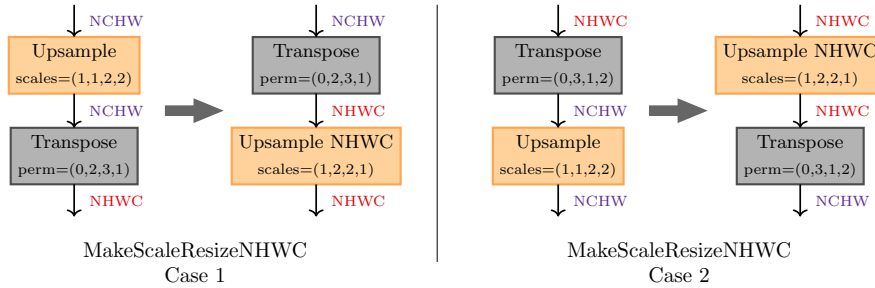


Figure 25: Illustration of the two scenarios where *MakeScaleResizeNHWC* transformation can be applied, with the example of an *Upsample* node with $2 \times$ upsampling. The tensors are represented with three dimensions but are actually 4-dimensional, with the batch size dimension always set to one.

As presented in Chapter 1, upsampling is a major element of semantic segmentation networks and, therefore, naturally became a matter of interest in our experiments. It turned out that, initially, only nearest-neighbor upsampling was supported. However, although there was an HLS implementation, some elements in the compiler were missing to fully support it. Precisely, although the mapping to the associated hardware layer requires channel-last layouts for its input and output, similar to any other node, there was no existing transformation to convert the layouts. Therefore, we designed a new transformation to convert the layout of upsampling nodes to *NHWC*. This *MakeScaleResizeNHWC* transformation uses a method similar to the *MakeMaxPoolNHWC* transformation and can be applied to *Upsample* nodes or *Resize* nodes, provided that they only use scale ratios to perform the resizing (each dimension of the input is multiplied by a certain value, rather than specifying the output dimensions directly). Similar to the max pooling, we identify two cases, depending on if the pattern is $\{Upsample \rightarrow Transpose\}$

or $\{Transpose \rightarrow Upsample\}$, as illustrated in Figure 25. In any case, the *scales* attribute of the node, which is the list of scale ratios to be applied to the input dimensions, must be permuted so the operation remains consistent regarding the size of the output.

3.3.4 Support of transposed convolutions

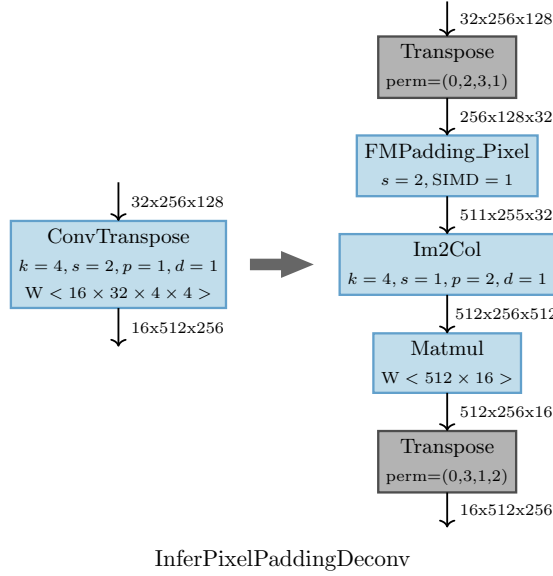


Figure 26: Illustration of the *InferPixelPaddingDeconv* transformation with the example of a $2 \times$ upsampling. The tensors are represented with three dimensions but are actually 4-dimensional, with the batch size dimension always set to one.

As part of our experiments with super-resolution that we will be presented in Chapter 5, we wanted to explore different upsampling methods. To not be limited to only one upsampling method, we added support for transposed convolutions, as they are a very common way of performing upsampling, not only in super-resolution networks but also in semantic segmentation ones. The integration in FINN of transposed convolution was done as a collaboration with members of AMD Research and uses an implementation as fractionnally-strided convolutions [22]. It is based on the already existing implementation of convolutions, with the addition of a module that performs the pixel padding, *i.e.* the insertion of a number of zeros between the pixels of the inputs based on the transposed convolution's stride (see Chapter 1, Section 1.2.4). The HLS implementation of this module was carried out by the researchers at AMD. We took care of the compiler part, by creating a custom ONNX node to represent the pixel padding, named *FMPadding_Pixel*, and designing a new transformation. This transformation, called *InferPixelPaddingDeconv* and represented in Figure 26, splits a transposed convolution of stride s , kernel size k , dilation d and padding p into a pixel padding of $s - 1$ zeros followed by a regular convolution with the same kernel size, a stride of 1, and regular padding with $d(k - 1) - p$ zeros. The weight tensor of the transposed convolution is initially of dimensions $C' \times C \times k \times k$, with C and C' respectively the number of input and output channels. The kernel matrices are transposed, and the tensor's dimensions are permuted to be $C \times C' \times k \times k$ as per the fractionnally-strided convolution algorithm, and the convolution is directly lowered to an *Im2Col* and a *MatMul* during the transformation, so it is ready to be converted to hardware layers.

3.4 Summary

In this chapter, we presented the core concepts of FINN, which is built as a combination of a library of HLS and RTL templates and a compiler environment to map a neural network graph onto a graph of hardware modules that follow the network's dataflow. We identified several limitations regarding the implementation of semantic segmentation networks with FINN, especially due to the frequent problems with residual structures. We introduced new elements in the tool to increase the support for the deployment of semantic segmentation models, in the form of new transformations, and integration of transposed convolutions in the framework. These elements, which directly contribute to improving the tool, were developed concurrently with our experiments presented in the following two chapters. The next chapter, in particular, will present our experiments related to the deployment of two semantic segmentation networks on FPGA, which represent the main contributions of this thesis.

Chapter 4

Co-designing semantic segmentation models for pipelined dataflow hardware architectures

Contents

4.1	First study-case: ResNet18-UNet	86
4.1.1	Motivations	86
4.1.2	Structure adaptations for compatibility	88
4.1.3	Training and quantization	89
4.1.4	Implementation results	91
4.1.5	Discussions	93
4.2	Second study-case: ENet	93
4.2.1	Motivations	93
4.2.2	Model definition	93
4.2.3	Training	96
4.2.4	Model adptations	98
4.2.5	Implementation results	100
4.3	Hardware-oriented design space exploration of ENet architecture . .	102
4.3.1	Comparison against previous FPGA implementations	104
4.4	Performance evaluation against GPU	106
4.5	Summary and discussions	109

In this chapter, we present the main contributions of this thesis. The first is the implementation of a ResNet18-Unet network on AMD Alveo U250, which is a first step in the challenge of implementing a semantic segmentation network with pipelined dataflow architectures, fostering discussions on the joint design of neural networks and hardware architectures. This contribution has been published at the 30th IEEE International Conference on Electronics, Circuits, and Systems (ICECS) [12]. The code used is open-source and is released on Github¹.

Then, we present the improvement through a better training setup of ENet, a lightweight semantic segmentation network and its implementation on AMD ZU19EG FPGA. This implementation, achieving throughput and latency compatible with real-time inference, is further improved through a hardware-oriented design space exploration of neural network architectures which identified design choices leading to a more efficient model. This final model has even lower complexity than the initial model, and retain its accuracy, with a very small drop with INT4 quantization. The implementation of this model shows state-of-the-art results of mIoU, throughput, and latency compared to previous implementations of semantic segmentation networks on FPGA trained on either Cityscapes or Camvid. We show a detailed comparison against embedded GPUs, exhibiting a 1.15x better power efficiency than a Nvidia Jetson AGX Xavier. This contribution is part of a paper currently under review, and the code is available on Github².

4.1 First study-case: ResNet18-UNet

4.1.1 Motivations

For the first attempt to deploy a semantic segmentation network, we chose to use a UNet type of architecture. Although UNets are hardly the best-performing semantic segmentation neural networks, we chose this for several reasons. First, we had to make a choice that was realistic regarding the capacities of FPGAs, especially the memory resources. As we mentioned in Chapter 2, the amount of on-chip memories is very restricted, which makes it impossible to deploy large networks with a streamlined architecture, except if off-chip memory is used, which would usually significantly slow down the design and increase its power consumption, and is not desirable. Even when considering quantization, the range of implementable networks is still limited. As an example, a network with 10M parameters would take 80Mb of memory when using INT8 quantization, which is the total amount of memory available on large FPGAs. Going lower than 8 bits is an option, but it would usually imply a drop in accuracy, which means that there is a trade-off to be made, and selecting a large network for its high performance only to significantly lower them by applying very low bit quantization to make it fit on the FPGA might not be the best choice. A similar argument could be made for pruning, as light architectures tend to perform as well as large, heavily pruned architectures.

Second, the network structure and operators used must be compatible with FINN. As of the version that was available at the time (v0.8.1), efficient implementations of convolutions like grouped convolutions or dilated convolutions, as well as complicated structures like pyramid pooling were not supported by the framework. However, these are heavily employed in real-time semantic segmentation models.

With these considerations in mind, the choice quickly narrows down to encoder-decoder CNNs that only uses very basic operators. Therefore, we chose to go with a UNet ar-

¹<https://github.com/hleblevec/finn-semseg>

²<https://github.com/hleblevec/P-ENet>

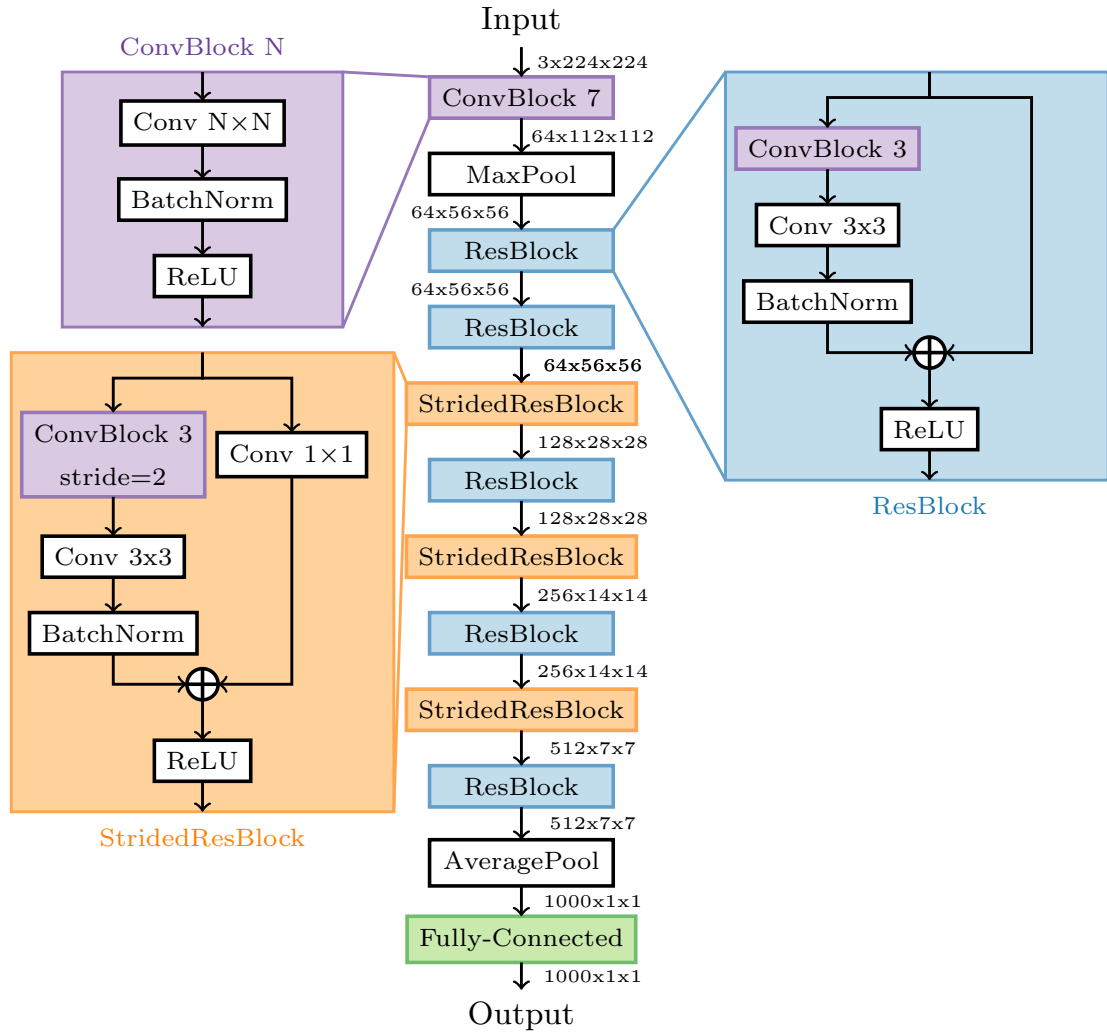


Figure 27: Illustration of the ResNet18 image classification network with input image size of 224x224, which is the resolution of images from the ImageNet dataset [160].

chitecture, as it is a historical network that is still very popular in many application fields, and constitutes a good baseline for a first experiment. Building a UNet is quite simple, and starts with the choice of an encoder, usually inspired by a classification network, and follows by building a decoder that is symmetrical to the encoder in terms of intermediate tensor sizes, and connecting the associated parts of the encoder and the decoder with long shortcuts.

For the choice of the encoder, we wanted to take it off a ResNet, as they are very popular image classification networks, and are very standard in the operators they employ. Moreover, a previous implementation of a ResNet with FINN had already been made, so we knew it was possible to use them, and we had an example to start from. Rather than using ResNet50 like this previous implementation, we opted for ResNet18 instead, as it has a better accuracy-to-number-of-parameters ratio [47]. As illustrated in Figure 27, it employs mostly residual blocks, and alternates between regular ones and downsampling blocks that use a stride of 2 for the first convolution, halving the spatial dimensions of the feature maps, while doubling their depth. It ends with an average pooling layer and a fully-connected layer for the classification.

To build a UNet based on this network, we remove the last two layers, and plug in

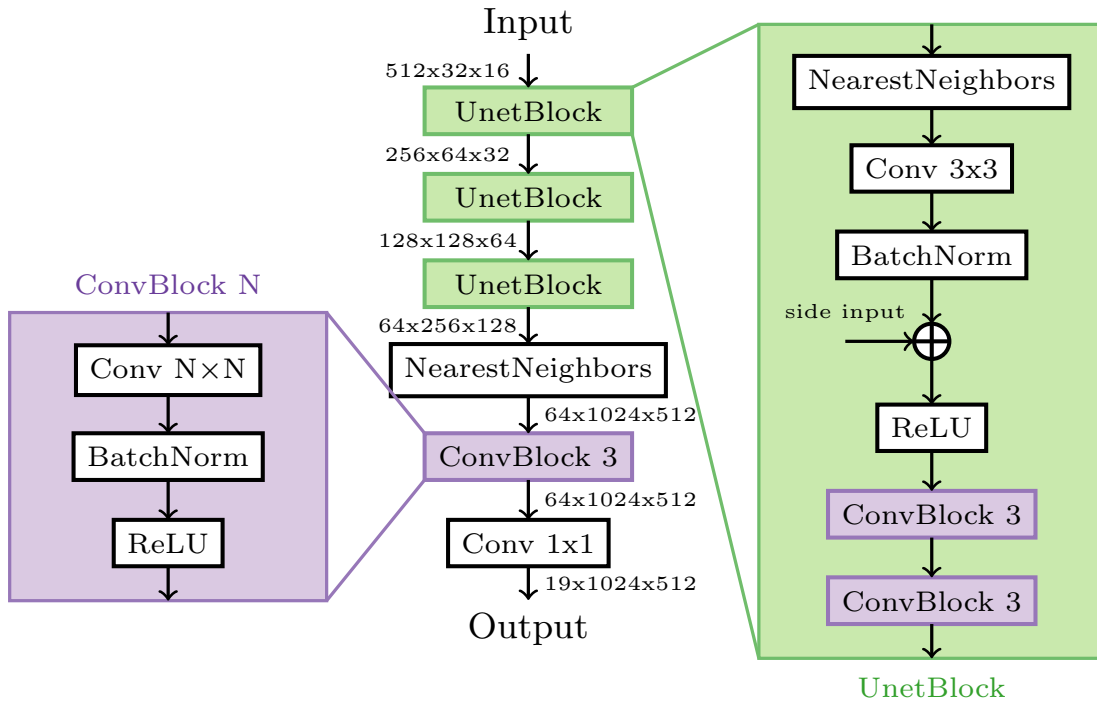


Figure 28: Illustration of the UNet decoder that can be connected to the ResNet18 encoder. Image sizes are set to produce a 1024x512 output, as can be used for training on Cityscapes, with the output having 19 channels, corresponding to the number of classes in the dataset.

the UNet decoder, whose structure is represented in Figure 28. The UnetBlocks are designed to upsample the feature maps, doubling their dimensions, while halving the depth, which is the opposite of what each set of two ResBlocks do, besides the first one, which is why we end up with three UNetBlocks.

Each of these blocks uses the output of the associated ResBlock as a side input to be added to the tensor obtained after the upsampling. For upsampling, we used nearest-neighbors resize convolution, a nearest-neighbors interpolation followed by a convolution, as nearest-neighbors upsampling was the only upsampling method supported by this version of FINN. The decoder ends with an additional upsampling by 4 to finalize retrieving the input image size and a pointwise convolution to obtain an output with as many channels as classes in the dataset, leading to a model with 14.3M parameters. The segmentation output can be obtained from the hard output by computing the maximum value of each pixel across the channel dimension.

4.1.2 Structure adaptations for compatibility

All the operators used in the network are compatible with FINN, however, there is one thing that must be modified to make it fully compatible with FINN. As discussed in Chapter 3, streamlining residual networks in FINN is rarely straightforward. As we mentioned before, a significant part of streamlining revolves around absorbing elements into thresholds, to remove floating point elements. This is done by placing those floating-point elements before a *MultiThreshold* node in the graph. It is usually easy to do with linear topologies, as those elements can be pushed through other nodes invariant by it. However, when a node such as a floating-point scalar multiplication, is pushed past a fork node, it must be placed on all the branches emerging from said node

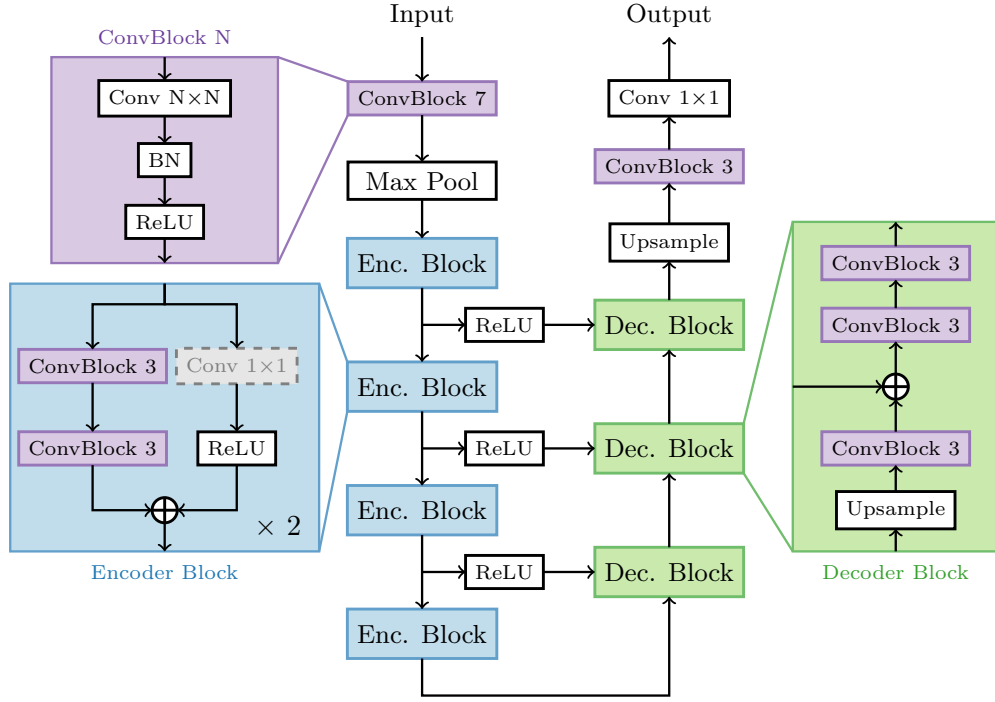


Figure 29: Illustration of the modified ResNet18-UNet architecture [12] ©2023 IEEE.

for the transformation to be valid, as it would otherwise change the network's function. Suppose one of the branches is empty, which happens in ResBlocks, or on the shortcuts between the encoder and the decoder. In that case, this floating-point element has nothing to be absorbed in, and most of the time cannot be pushed past the element-wise additions used to rejoin the branches. To overcome this problem, we use a method that was employed to integrate the ResNet50 [159] and move the ReLU layers present after each element-wise addition, to before the addition. Having the same activation layers on both branches before the addition allows the absorption of floating-point elements, and allows the network to be streamlined. Since activation layers are not linear, this operation changes the behavior of the network, so it must happen as a direct modification of the network architecture and requires retraining the network to adapt to this change. This can be seen as a simple yet essential approach to the co-design between the neural network and the hardware architecture. Applying this modification gives the general network structure shown in Figure 29. The creation of the new *MakeMaxPoolNHWC* and *MakeScaleResizeNHWC* transformations presented in Chapter 3 was necessary for the streamlining of this network.

4.1.3 Training and quantization

In this experiment, and all the following, we used the Python framework Pytorch to train the network. For this network, we only performed training on the Cityscapes dataset. We trained for 200 epochs using SGD as an optimizer with a weight decay of $5e^{-4}$ and a momentum of 0.9, a Cross-Entropy loss function, a MultiStep learning rate scheduler that starts with a 0.01 learning rate and divides it by 10 at epochs 100 and 133, and a batch size of 8. With these training parameters, the baseline network reaches 68% mIoU with FP32 precision, while the version with moved activation layers achieves 67.8%.

As we have mentioned several times already, quantization is almost mandatory when

attempting to deploy a neural network on FPGAs, as floating-point operations with large bit widths consume too many resources to be viable, unless the network is extremely tiny. Therefore, some integer quantization is necessary. However, the choice of precision for weights and activations (*i.e.* the number of bits to represent them) is not straightforward and revolves around finding an appropriate balance between model accuracy and hardware resource consumption. Therefore, this choice depends on the amount of available resources on the FPGA. In this experiment, our target FPGA is the AMD Alveo U250 accelerator card. It is a rather large FPGA embedded on a board that is more dedicated towards acceleration for data center applications than for embedded systems. However, we wanted to make sure that we would have enough resources available for our first experiment, so we chose this FPGA as a starting point. Note that, although the training of the network is done with an input resolution of 1024x512, we need to resize the inputs to 256x256 for inference, to account for the memory limitations and the inability of FINN to process rectangular inputs (in the version we used at the time).

Knowing how many resources are available, we can explore the trade-offs between quantization and resource utilization for our network to select the appropriate quantization. We used the Python library *Brevitas* [161] to perform QAT on our model, using symmetric uniform quantization with per-channel scaling factors that are learned along with the weights during training [43]. This library, built upon Pytorch, enables easy exploration of neural network quantization and supports the export of the model to QONNX which facilitates the interface with FINN. We trained using the same parameters as for the floating-point version, with precisions ranging from 2 to 8 bits, identical for weights and activations. To evaluate and compare the resource utilization associated with each quantization, we fixed the parallelization parameters of the network's layers in FINN to obtain an estimated throughput of approximately 40 FPS with a 200 MHz clock rate for the FPGA. We selected this to have something that would be over 30 FPS in the estimation, giving it the potential to meet the requirements of real-time applications. We then used FINN to get an estimate of the resource usage of the architecture. Please note that these are pre-synthesis estimates and do not take into account the potential optimizations of the synthesis tool or routing and placement that are specific to the target FPGA. It also does not take into account elements that are added in the later steps of the tool. These elements count things like DMA controllers that are necessary for interfacing between the FPGA and the off-chip memory containing the input and output batches of images, or FIFOs necessary to handle the differences of latency and data requirements of each operation and ensure the architecture can meet the target throughput. However, it gives accurate enough results to explore the design space without having to go through the long process of completely implementing the architecture.

Figure 30 displays the mIoU per estimated LUT utilization and per estimated BRAM utilization for each studied quantization. It shows that after 6 bits there is almost no gain in accuracy, which means that even if we had more resources available, there would be no reason for us to go beyond this quantization. Looking at the mIoU per LUT usage indicates that 5 bits seem to be the best compromise between LUT utilization and model accuracy, as it is a point of inflection in the curve. However, the BRAM utilization is a bit too close to the available amount, and after adding the other required elements in the architecture specific to the deployment on Alveo boards, as well as the necessary buffers, we could not implement the design because the BRAM utilization was exceeding the available amount. We therefore favour a 4-bit integer quantization which is a better trade-off between model accuracy and resource utilization. Moreover,

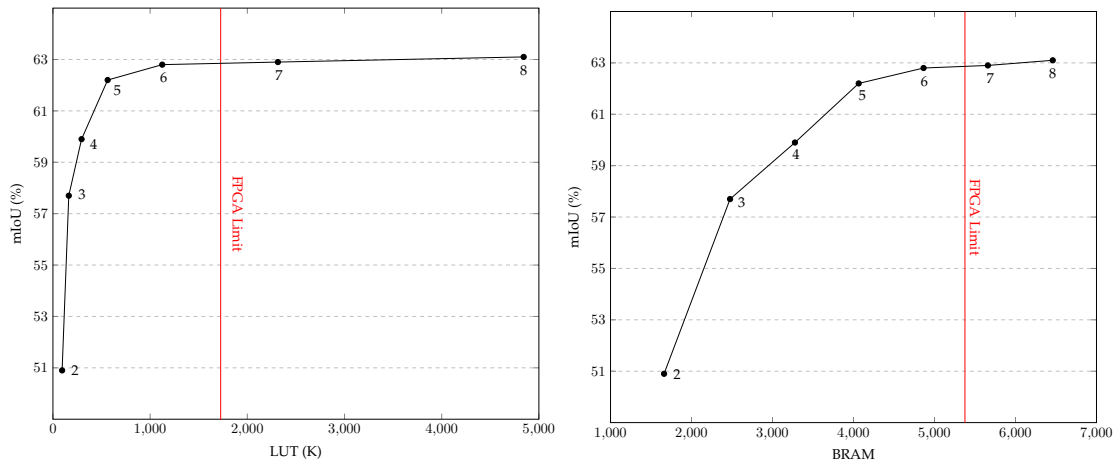


Figure 30: mIoU per estimated LUT utilization (left) and BRAM utilization (right) for different integer quantization bit widths.

although a 5-bit quantization seems to offer a better accuracy-to-resources trade-off than 4 bits, there are other limitations to the design space to be taken into account. The throughput of the network depends on the latency of the slowest layer, which is often an MVAU. Its latency is defined by the size of the input and the number of SIMD lanes and Processing Elements (PE) that set the parallelism level of said layer. If we consider the size of the input image to be a fixed parameter, lowering the latency can be done by increasing the SIMD and PE parameters. There is, however, a limit to how much these can be increased. Due to limitations in Vitis HLS, the size of the AXI interface of each HLS block cannot be greater than 8191. The size of the interface of an MVAU is $SIMD \times PE \times b$ with b being the bit width. This means that for a fixed bit width, there is a limit to the achievable latency of each layer and therefore, to the network's throughput and global latency.

We further improved the model accuracy by putting only the first and last layers to 8 bits. They are much more sensitive to quantization, and maintaining a more moderate quantization is very beneficial to accuracy while bringing a relatively small increase in resources [162]. Doing so brings the LUT and BRAM estimations to 367,322 and 3,327 respectively while allowing us to achieve 62.9% mIoU.

4.1.4 Implementation results

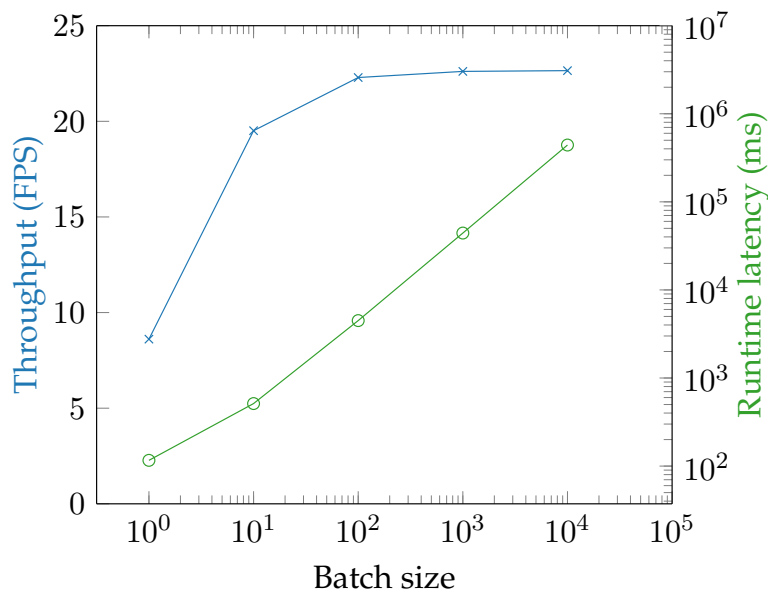
The presence of long bypasses between decoders and encoders causes starvation issues. Indeed, these shortcuts between different points in the network need to be sufficiently buffered. However, the required FIFO sizes exceed the limits of Vivado that were previously mentioned. Therefore, the additions made to FINN (*SplitLargeFIFOs*), as described in the last chapter, allowed us to identify and solve this issue, which was discovered during our experiments. The resources used by the final design are displayed in Table 4. As expected, the LUT and BRAM utilization is higher than the initial estimations, as these did not consider every element. In addition to the usual resources that we introduced in Chapter 2, we also make use Ultra Random Access Memory (URAM), a special type of memory present in the AMD Ultrascale FPGA family, and are deeper memory banks. Therefore all the large FIFOs in the network were placed on URAM to take advantage of this additional memory resource and reduce the BRAM usage. The maximum frequency of the design is 134 MHz, which is low for the considered device.

We used the driver based on the *Pynq* library provided by FINN to evaluate the through-

Resource	Utilization	Available	Utilization Percentage
LUT	490,265	1,726,240	28.4%
FF	569,529	3,456,000	16.4%
BRAM	4,260	5,376	79%
URAM	388	1,280	30%
DSP	1,043	12,288	8%

Table 4: Utilization of resources of our network on AMD Alveo U250 FPGA

put and latency of the design. The version of the library targeting Alveo is built upon the Xilinx Runtime Library (XRT) that handles the communication with the accelerator through the PCI Express interface that connects the Alveo card to the host machine. The network needs to be fed batches of inputs rather than a single image to reach its best achievable performance. This is due to FINN’s pipeline needing to be filled with data to be optimized [163] and the design of PCIe interfaces. We measured the runtime latency as the time between the moment we send the batch of data to the card, to the time we receive it. As such, it takes into account not just the execution time, but also the data transfer time composed of the PCIe transfer time and the input and output DMAs processing time. We carried out measurements with different batch sizes, from 1 to 10^4 by power-of-ten steps. The results are displayed in Figure 31.

**Figure 31:** Semantic segmentation throughput in FPS (blue, cross) and runtime latency in ms (green, circles) of the network on Alveo U250 [12] ©2023 IEEE.

As expected, the throughput increases with the batch size and reaches its maximum value of 22.7 FPS with a batch of at least 100 images. This is significantly lower than the initial target of 40 FPS.

The latency evolves almost linearly with the batch size with an overhead corresponding to the interface’s latency that is visible for small batch sizes (1 or 10) but becomes negligible for larger batch sizes. Doing a linear regression gives an intercept of 71 ms and a slope of 44 ms per input. Such latency is probably a little too high to be compliant with real-time autonomous driving but remains within the expected order of magnitude.

4.1.5 Discussions

This first experiment led to encouraging results in favor of pipelined dataflow architectures, demonstrating that they could be used to implement semantic segmentation networks, and has been valuable from a learning point of view. It helped us acquire deeper knowledge of FINN and how, although it has its limits, it offers an environment that gives all the tools to go beyond these limits and implement new elements to enhance the support for complex networks. It also taught us valuable lessons about pipelined dataflow architectures in general, especially regarding the roles of buffers and how adding a shortcut between the encoder and the decoder, which seems insignificant in software, might lead to many issues from the hardware side. Therefore when targeting this type of hardware architecture, there are some considerations to be aware of that go beyond the limitations of the framework used and are related to the general behavior of pipelined dataflow architectures. Even if we identified ways to improve the design to maybe meet the requirements of real-time applications, we realized this network would probably never be very efficient. Despite that it appeared as a fine choice *a priori*, the long shortcuts and the number of parameters, which is low from a deep learning perspective but is quite high by FPGA implementation standards, mean that it would be difficult, if even possible, to obtain an efficient design that would be fast enough without sacrificing too much of the accuracy. We thought we would be better off investing time in developing new features and new layer support for FINN so it would allow us to implement a more efficient network. Coincidentally, added support for dilated convolutions and convolutions with rectangular inputs thanks to a new *ConvolutionInputGenerator* developed by another external contributor, as well as support for transposed convolutions to which we contributed, opened a new range of networks that were possible to implement with FINN, and led to our second experiment.

4.2 Second study-case: ENet

4.2.1 Motivations

With the conclusions drawn from our first implementation of a semantic segmentation network on FPGA with FINN, we wanted to aim for a more efficient, lighter neural network architecture that we could deploy on an FPGA that would be more realistic for an embedded system. As will be presented below, knowing the constraints inherent to FINN and pipelined dataflow architectures in general, ENet [96] appeared to be the perfect candidate amongst the variety of real-time semantic segmentation networks. For the FPGA target, we chose to go with the AMD Zynq Ultrascale+ ZU19EG that is placed on the TySOM-3A-ZU19EG board from Aldec. This FPGA is much more in the range of embedded systems in terms of power consumption but is still quite high-end and has enough resources for us to explore different parallelism configurations so we can hopefully find a configuration that meets real-time systems requirements.

4.2.2 Model definition

ENet [96] is a convolutional encoder-decoder neural network composed of multiple bottleneck blocks, taking inspiration from ResNets [47]. Each block begins with a point-wise convolution to reduce depth, followed by a primary convolution whose type and parameters are determined by the block type, and concludes with another 1x1 convolution to increase depth. There are three varieties of bottleneck blocks: *RDDNeck* (Figure 32b), *ASNeck* (Figure 32c), and *UBNeck* (Figure 32d). In the *RDDNeck*, the main convo-

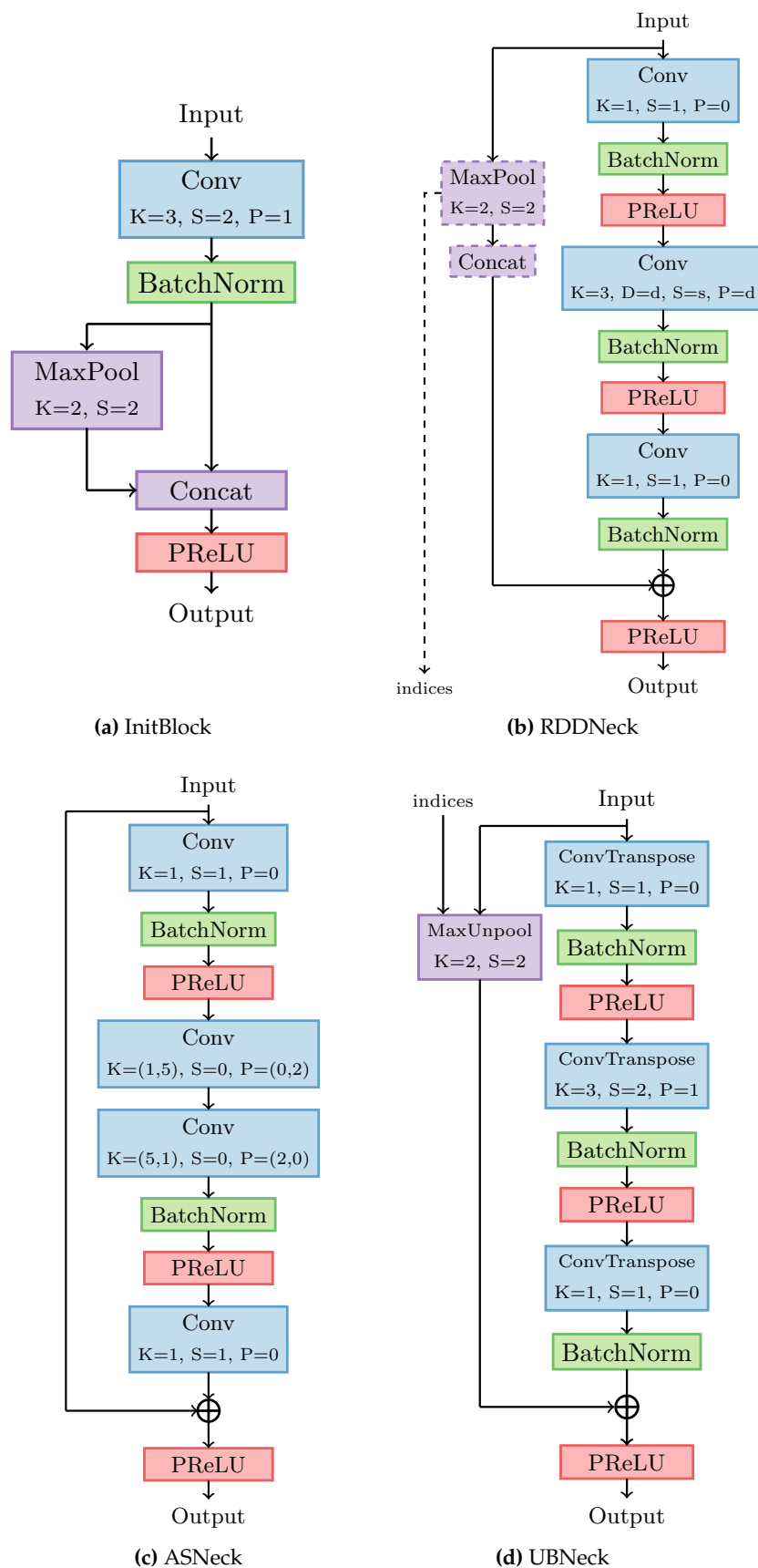


Figure 32: The InitialBlock (a) as well as RDDneck (b), ASNeck (c) and UBNeck (d) the bottleneck blocks of ENet [96]. RDDNeck can be used in downsampling mode (dashed lines), in which case the second convolution has a stride $s = 2$ and a MaxPooling layer followed by a concatenation with zeros is added to the shortcut. The pooling indices of these MaxPooling layers are then used as inputs to the MaxUnpool layers in UBNeck.

Name	Type	Output size
InitBlock	-	$16 \times 256 \times 128$
RDDNeck	downsampling	$64 \times 128 \times 64$
4x RDDneck	-	$64 \times 128 \times 64$
RDDneck	downsampling	$128 \times 64 \times 32$
RDDNeck	-	$128 \times 64 \times 32$
RDDNeck	dilation 2	$128 \times 64 \times 32$
ASNeck	-	$128 \times 64 \times 32$
RDDNeck	dilation 4	$128 \times 64 \times 32$
RDDNeck	-	$128 \times 64 \times 32$
RDDNeck	dilation 8	$128 \times 64 \times 32$
ASNeck	-	$128 \times 64 \times 32$
RDDNeck	dilation 16	$128 \times 64 \times 32$
<i>Repeat previous section, without first RDDNeck</i>		
UBNeck	-	$64 \times 128 \times 64$
RDDNeck	-	$64 \times 128 \times 64$
RDDneck	-	$64 \times 128 \times 64$
UBNeck	-	$16 \times 256 \times 128$
RDDNeck	-	$16 \times 256 \times 128$
Transposed Conv.	-	$19 \times 512 \times 256$

Table 5: ENet architecture. The output sizes are for a 512x256 input. Table adapted from [96].

lution utilizes a 3x3 kernel and may be regular, dilated, or strided depending on the position in the network. ASNeck features a 5x5 convolution divided into two asymmetric convolutions: a 5x1 convolution followed by a 1x5 convolution. Finally, the UBNeck, a key component of the decoder, uses a transposed convolution with a 3x3 kernel as its primary convolution layer to upsample the feature maps. Each convolution is followed by batch normalization layers and PReLU activation layers. Each bottleneck module includes an extra "skip" branch, which is combined with the primary branch via element-wise addition. In RDDNeck, this skip branch includes a MaxPooling layer if the module performs downsampling. If the block's output has more channels than its input, a zero-filled tensor is concatenated after the MaxPooling layer to equalize the difference. In UBNeck, the skip branch incorporates a MaxUnpooling layer, leveraging the MaxPooling indices from previous downsampling modules to create an encoder-decoder connection, similar to SegNet [86]. Additionally, the network starts with an *InitialBlock* (Figure 32a), which performs initial input downsampling. At the end of the network, a single transposed convolution with a 3x3 kernel is used to obtain the segmentation output. The bottleneck modules are sequenced and parameterized as shown in Table 5. They are divided into sections, each having identical feature map dimensions.

ENet introduces a different perspective on encoder-decoder architectures compared to SegNet or UNet. While these two networks have a highly symmetric design with a mirrored encoder and decoder, ENet uses a large encoder and a small decoder. The encoder functions similarly to traditional classification architectures, processing and filtering information on lower-resolution data while the decoder's role is limited to upsampling the encoder's output and refining finer details. ENet's encoder first two blocks significantly reduce the input size and use a small set of feature maps, based on the idea that visual information is spatially redundant and can be compressed into a more efficient

representation. The initial layers are intended to serve as feature extractors rather than directly contributing to classification. To ensure a wide receptive field for context-aware segmentation without employing excessive downsampling which would be detrimental to the accuracy, the encoder leverages dilated convolutions. These replaced the main convolutional layers in bottleneck modules at the smallest resolutions, leading to a significant accuracy increase, with no additional computational cost. The best results were achieved by interleaving the dilated convolutions with regular and asymmetric bottleneck modules, rather than arranging them sequentially. This is also what allows to use of a smaller decoder, as more downsampling would require more upsampling blocks in the decoder. Despite the network's substantial number of layers, it is very compact, containing just 0.37M parameters, achieving an mIoU of 58.3% on Cityscapes and 51.3% on CamVid.

ENet is one of the oldest real-time semantic segmentation models, and since its publication in 2016, several models have emerged that outperform it in terms of segmentation accuracy, as can be seen in Table 1 of Chapter 1. Nevertheless, in light of what we learned from our experiments with integrating semantic segmentation networks with FINN, we found that ENet had several advantages over these models, which justifies our choice. First, it is extremely lightweight and requires very few computations compared to other models, giving it a good performance-to-parameter trade-off, which makes it a relevant candidate for implementation on FPGAs. Moreover, even though the accuracy reported by the authors is quite low, much research has been conducted to propose better methods to train models for semantic segmentation, like data augmentation and careful training parameters selection. Data augmentation in particular, a method of creating more examples for training from a dataset through a variety of image processing functions, has shown to be very beneficial to the network performance. As we will see in the following section, training the model with such a setup has the potential to drastically increase the initial performance of the model and bring it closer to the state of the art, exhibiting that differences in performances between neural networks cannot be only attributed to differences in architectures. Finally, it uses mostly operators supported by FINN, or ones that can be easily replaced, and has a structure that can be made compatible. Indeed, we learned that local shortcuts of ResNet blocks were mostly fine, and the only issue here would be the long shortcuts between the RDDNeck blocks of the encoder and the UBNeck blocks of the decoder. Still, they are related to the association of max pooling and max unpooling for the downsampling and upsampling, which can be replaced, removing the need for long shortcuts, as we will discuss later in Section 4.2.4. Since the network make extensive use of dilated and transposed convolutions, the implementation would have probably been impossible without the recent enhancements of FINN. Moreover, since the new *ConvolutionInputGenerator* supports rectangular inputs, we can finally work with a resolution that is more fitting to the images of semantic segmentation datasets. We center our experimentations mainly around Cityscapes and use a resolution of 512x256 pixels to balance out resources usage. We also experiment a little with CamVid, as it is also a popular dataset that has been used in many previous FPGA implementations of semantic segmentation networks for autonomous driving applications. The resolution commonly used for inference on CamVid with FPGAs is 480x360 pixels, so this is the one we use as well.

4.2.3 Training

To enhance the model accuracy, we draw inspiration from advancements in training hyperparameters and data augmentation techniques for semantic segmentation mod-

Model	Year	Param. (M)	mIoU	mIoU*
ENet [96]	2016	0.37	58.3%	70.5%
ESPNet [99]	2018	0.4	60.3%	69.6%
DABNet [100]	2019	0.76	70.1%	72.9%
EDANet [101]	2019	0.68	67.3%	69.8%
Fast-SCNN [147]	2019	1.1	69.2%	70.3%
ESNet [105]	2019	1.66	70.7%	73.8%

Table 6: Original mIoU on Cityscapes as reported by the authors of several real-time encoder-decoder semantic segmentation networks and the obtained mIoU after retraining, noted mIoU*.

els since the release of ENet. We train the network based on the configuration used by the authors of RegSeg [106]. Their strategy involved using SGD with a momentum of 0.9 and employing a Cross-Entropy loss function. The training span comprised 1000 epochs with a batch size of 8. The learning rate begins at 0.005 and gradually increases to 0.05 over 3000 iterations following a linear warm-up, after which it is modified using a polynomial scheduler. A weight decay of 10^{-5} is applied on learned parameters, excluding the BatchNorm ones. Additionally, extensive data augmentation techniques are utilized. Images from the Cityscapes dataset undergo horizontal flipping and *RandomResizeCrop* within a resize range of [400, 1600] and a crop dimension of 768×768 . For each image, two random operations from the RandAugment [164] set are chosen and applied with a magnitude of 0.2. Furthermore, class uniform sampling [165] is employed during training to ensure an even distribution of initial samples and centroids. These settings with ENet resulted in a mIoU of 70.5%, a significant improvement from the original 58.3%.

For the CamVid dataset, implementing the same approach as RegSeg authors and fine-tuning the model pre-trained on CityScapes did not yield satisfactory results with ENet. Thus, we opted to train from scratch using the same parameters as CityScapes, except for a resize window of [288, 1152], a cropping size of 720×960 , and without applying RandAugment operations. This configuration enabled us to achieve an mIoU of 77.6%, significantly improving from the initial 51.3%.

In the deep learning literature, there has often been a bias where neural network architectures are compared while altering the training routines, and the performance gains are then attributed solely to the architecture. In the field of image classification, this was clearly demonstrated, for example, by Wightman *et al.* [166], which showed that with proper training techniques, convolutional networks could perform just as well as transformers, which were then considered inherently superior due to their architecture. We observed exactly the same trend in our experiments on semantic segmentation networks. The consequence is that a well-trained ENet achieves very good performance while being extremely lightweight.

Optimizing ENet with improved hyperparameters and effective data augmentation significantly enhances its accuracy, while further improving its performance-to-parameter ratio, making ENet even more appealing for resource-limited embedded systems. This shows the importance of a good training procedure, and that when choosing a neural network to implement, comparing them solely on the accuracy reported by the authors may not be sufficient to forge an opinion, as initial results might be more related to the chosen training setup than the architecture itself. In the end, going for a lightweight neural network is not as much of a limit to the achievable segmentation performance as it might seem at first glance. To draw a fairer comparison with other models, we

also trained other encoder-decoder real-time semantic segmentation networks with the same setting (when we could find a working Pytorch implementation). The results, presented in Table 6, show that there is also an improvement of the mIoU on Cityscapes for these models, but not as significant as with ENet. With all the models put on an equal footing, ENet clearly appears to be a justified choice.

4.2.4 Model adaptations

As discussed earlier, there are aspects of the structure and used operators of ENet that are not compatible with FINN. Therefore, they must be replaced to produce a network that can be correctly implemented. Please note that these incompatibilities are time-bound (as of the writing of this manuscript, the current FINN version is 0.10.0.), and could be lifted in the future, as the tool’s development advances. The new versions of the blocks are shown in Figure 33 and contain the following modifications:

Activation replacement: In all blocks, we replace the *PReLU* activation layers by regular *ReLU*s.

Transposed Convolution reshaping: The *ConvTranspose* nodes present in UBNeck and at the end of the network are here to perform x2 upsampling and there are several ways to choose the parameters of a transposed convolution to achieve a certain output size. In the original paper, the authors chose a kernel of size $K = 3$, a stride of size $S = 2$, input padding $P = 1$, and output padding $P_{out} = 1$. This is probably the most efficient way to do x2 upsampling computation-wise. However, due to how transposed convolutions are implemented in HLS in FINN, we cannot use output padding. Therefore, we increase the kernel size to $K = 4$ to compensate. This gives the same output size but without output padding.

Early layers replacement: In the InitialBlock, we remove the *MaxPool* + *Concat* association and adapt the *Conv* parameters to remain coherent with the output size. This is necessary as the *Concat* operation is only partially supported by the tool.

Downsampling replacement: For similar reasons, we replace the *MaxPool* + *Cat* appearing on the side branch of RDDNeck when downsampling is required by a *Conv* with stride $S = 2$ and a 1x1 kernel.

Upsampling replacement: In UBNeck, we replace the unsupported *MaxUnpool* layer originally present in the side branch with a *ConvTranspose* layer with identical parameters as the main branch one. With this and the removal of *MaxPool* layers in RDDNeck, there is no longer a need to transfer information from the encoder to the decoder through long shortcuts.

Activation move: We move the *ReLU* layers that are applied after an *Add* to before it, on both branches, in every ASNeck or RDDNeck block (except when RDDNeck is used for downsampling, which happens twice in the whole network).

Activation insertion: We introduce a *ReLU* between the two asymmetrical *Conv* layers of ASNeck to circumvent the difficulties of the compiler in dealing with convolutions that are not followed by an activation layer.

Early layers replacement: In the InitialBlock, we remove the *MaxPool* + *Cat* association and adapt the *Conv* parameters to remain coherent with the output size.

To evaluate the impact of these alterations on the model’s performance in the semantic segmentation task, we trained various versions of the model on Cityscapes, incrementally incorporating the changes mentioned previously. We followed the settings out-

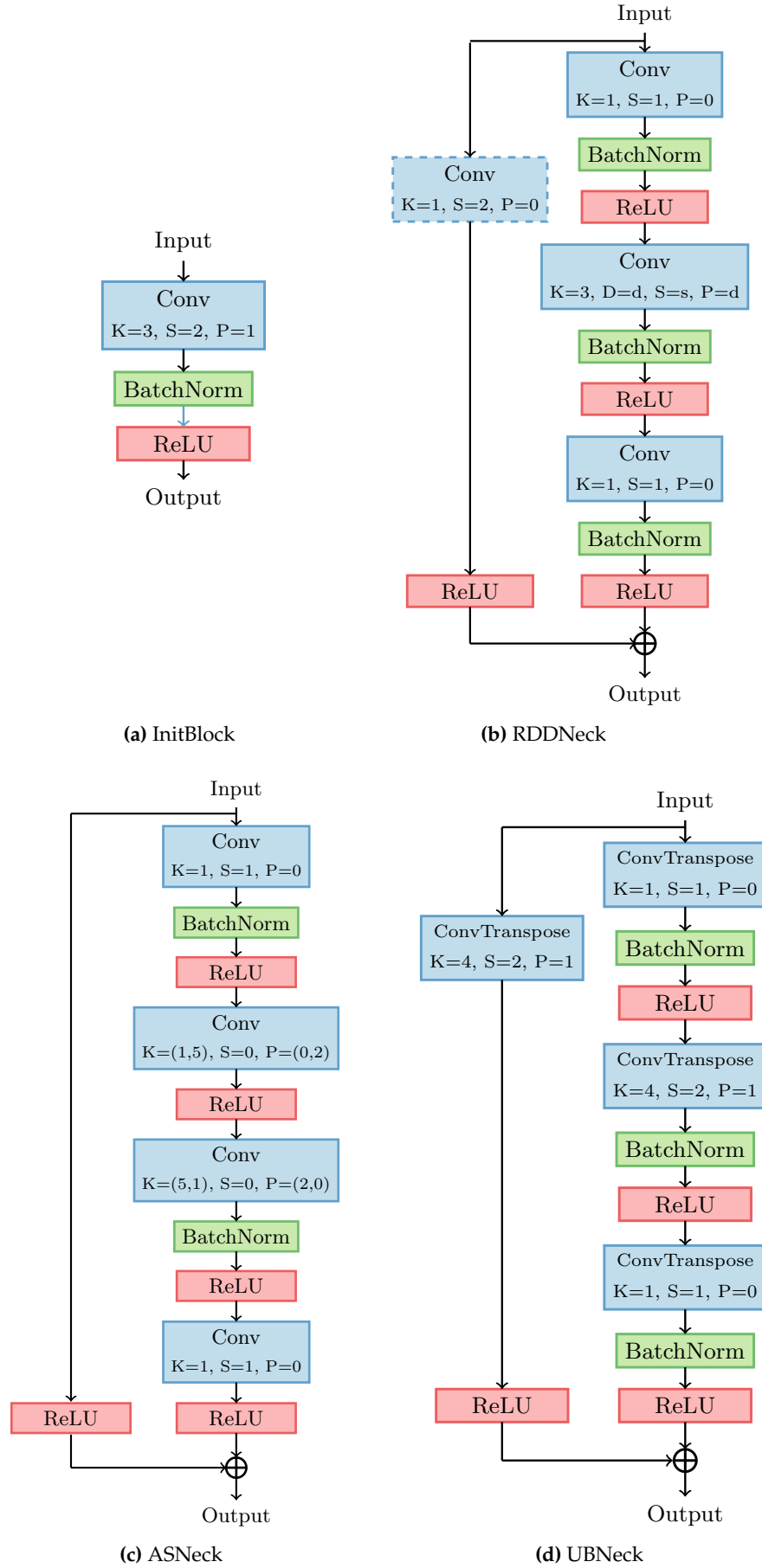


Figure 33: The modified InitialBlock (a) as well as RDDneck (b), ASNeck (c), and UBNeck (d), the bottleneck blocks of our new ENet architecture

Modification	mIoU	Parameters (K)	MACs (G)
Baseline (FP)	70.5	354	1.47
+ Act. repl.	70.84	354	1.47
+ TC reshape	70.90	365	1.87
+ Up. repl.	71.5	504	3.46
+ Down. repl.	71.38	513	3.48
+ Early lay. rep.	71	513	3.48
+ Act. move	70.08	513	3.48
+ Act. ins.	69.75	513	3.48

Table 7: Evaluation results of additive variations on Cityscapes using quantized models. The presented mIoU is the best out of three trainings. The baseline ENet is in mixed floating-point precision (FP32 and FP16). MACs numbers are displayed for a 512x256 input size.

lined in Section 4.2.3 but preferred to assess with quantized models to observe their effect in the quantized domain. We found this more relevant than floating-point results, as our model eventually needs to be quantized for implementation on an FPGA. Although it would also be valuable to analyze the modifications’ impact on floating-point performance, the extensive training time and our limited access to GPUs for training led us to focus solely on quantized models.

We selected the same quantization parameters used during the training of ResNet18-UNet, as they demonstrated a favorable balance between accuracy and resource efficiency. While alternative quantization strategies might yield improved results for this specific network, a comprehensive study of the model’s quantization remains an avenue for future exploration. For the present work, however, we opted to retain INT4 for most layers, with INT8 reserved for the first and last layers, and conducted all our training experiments using this quantization approach.

Table 7 presents the results, highlighting several modifications that lead to a marginal improvement in mIoU. While the placement of *ReLU*s before the branch addition is the most adverse modification, version 2.0 of our model, now FINN-compatible and FPGA-ready, shows only a modest 1% reduction in mIoU compared to the original, a decrease well within acceptable bounds. This slight reduction is accompanied by a notable increase in parameters and MACs compared to the baseline model. Nevertheless, the new version remains significantly smaller than other real-time semantic segmentation networks, making it well-suited for FPGA deployment.

4.2.5 Implementation results

To find the appropriate parallelization parameters for our hardware blocks, we use a mix of automatization and manual tuning to obtain a design that could achieve at least 30 FPS with a 200 MHz clock based on the estimations after conversion to hardware layers. More precisely, with the parameters chosen, it would reach 38 FPS. The last thing remaining is the placement of FIFOs. This step is now much easier without long side branches due to our modifications. Nevertheless, the initial results of the automatic FIFO sizing that we performed led to very large FIFOs throughout the architecture, and the overall memory requirement largely exceeded the capacities of the AMD Zynq Ultrascale+ ZU19EG FPGA we wanted to target. However, several RTL simulations exhibited that the depth of the generated FIFOs could be greatly reduced and still preserve the throughput. Therefore, we limit the FIFOs to a maximum depth of 2^{15} , greatly

reducing the amount of memory required to implement the design, and allowing it to fit the target. Table 8 displays the usage of the resources of our design on the target, which can actually run on a 250 MHz clock. It uses 58% of LUTs, 66% of BRAMs, and 20% of FFs. None of the URAM are used by the design, which means that there are still many memory resources available. Moreover, although we could have chosen to use DSPs to perform some of the computations, they would be largely underutilized with INT4 computations, so we only use LUTs.

Resource	Utilization	Available	Utilization %
LUT	304,371	522,720	58.23
FF	210,693	1,045,440	20.15
BRAM	649.5	984	66.01

Table 8: Resource utilization of our design on AMD ZCU19EG FPGA. Negligible amounts of URAM and DSP are used for the DMA interfaces.

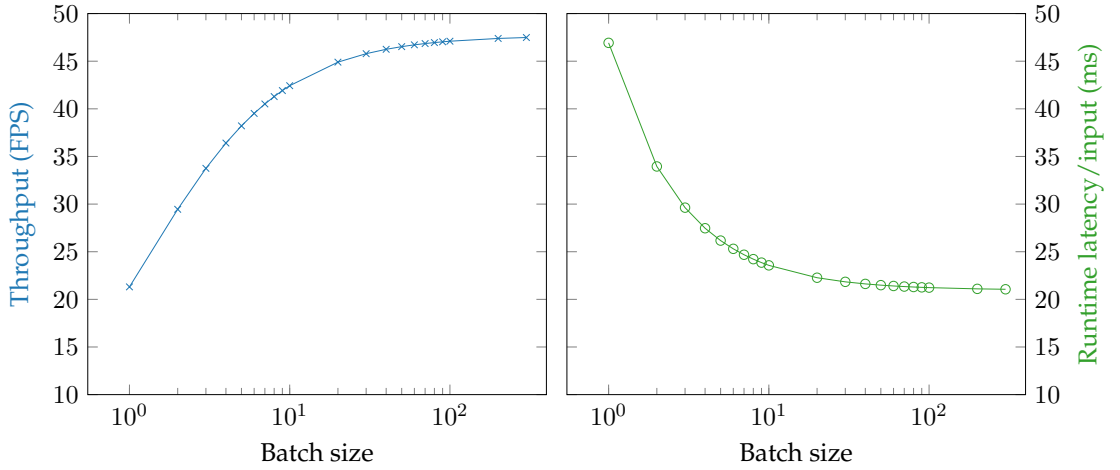


Figure 34: Semantic segmentation throughput in frames-per-second (left) and runtime latency per input in ms (right) of the network on Xilinx ZU19EG using a 250MHz clock and 512x256 input size.

We measured the throughput and latency of our design on target using the driver made with the *Pynq* library, experimenting on different batch sizes. The results, displayed in Figure 34, highlight once more the influence of batch size on throughput. The design reaches up to 47.5 FPS with a batch of 300 images but already runs at 47.1 FPS with 100 inputs. The latency behaves as an affine function of the batch size, with an initial 26 ms attributed to data transfer and driver internal behavior, and a 21 ms/input slope. These results prove the ability of our design to respond to real-time constraints.

The prior knowledge obtained from our first experiment led to this successful attempt at building a real-time semantic segmentation FPGA-based system. It shows how important the co-design of neural networks and hardware architecture is important to achieve good performances. However, having an implementation of a neural network able to meet real-time constraints of throughput was only the first step. The next one is to see how we can improve these performances by using our implementation as a starting point and explore neural network design options in the hope to find a more efficient network from an hardware point of view.

4.3 Hardware-oriented design space exploration of ENet architecture

Name	Proj. Ratio	Up.	SC1	SC2	mIoU (%)	Param. (K)	MACs (G)	LUTs (K)	BRAMs 18K
ENetV2	4	TC	No	No	69.75	513	3.48	88.7	335
ENetV2 _{SC1}	4	TC	Yes	No	70.13	513	4.02	98	360
ENetV2 _{SC2}	4	TC	No	Yes	67.84	513	4.56	101.9	374
ENetV2 _{SC}	4	TC	Yes	Yes	69.47	513	5.09	112.0	400
ENetV2 _{PR8}	8	TC	No	No	63	317	2.99	88.6	256
ENetV2 _{NN}	4	NN	No	No	69.76	350	1.11	71	245

Table 9: Evaluation results on Cityscapes and complexity estimations of the different versions of the quantized model. "TC": transposed convolutions, "NN":nearest-neighbors, "SC": shortcut.

Now that we have a model capable of running on our target with acceptable performance, we explore if modifying elements of the architecture could lead to a better balance between performance and resources and conduct an exploration of the design space. It would be impossible to be completely exhaustive because the design space is infinite and training a model is very costly in both time and resources, especially with QAT. In fact, between training and converting the model to hardware layers with FINN, evaluating one configuration takes several hundred hours. Therefore, we chose a limited set of modifications that we identified as interesting based on computational and hardware complexity, such as changing the projection ratio in the bottleneck blocks, the upsampling method used, or reinstating long shortcuts to evaluate if the resources overhead of using them could be worthwhile. Moreover, we want to keep the general idea of ENet's architecture because it has proven to be efficient, and modifying too much of the global amount of convolutions or the kernel parameters in the network would likely be very detrimental to the accuracy as their configuration influences the field of view of the network, which must remain large enough to cover the wide images of Cityscapes.

First, we suggest increasing the projection ratio, which is a parameter that determines the dimension of the intermediate space used for projection in all bottleneck modules. Specifically, for a given ratio pr and a number C of input channels, the initial pointwise convolution in each bottleneck module will generate a tensor with $C' = \frac{C}{pr}$ channels. The main expensive convolution of the module will be carried out with this tensor size, so increasing the projection ratio will notably decrease the computational requirements of each block and, consequently, the entire network.

Second, we aim to try a different upsampling method. This is an essential component of semantic segmentation networks, and several methods to execute it exist in deep learning. It can result in varying performance and computational demands. Besides transposed convolution, the only other upsampling method fully supported by FINN is nearest-neighbors upsampling. Therefore, we propose replacing the transposed convolutions used for upsampling in the network with nearest-neighbors, possibly followed by a 1×1 convolution when projection to a lower-dimensional space is needed.

Lastly, since we eliminated the connections between the encoder and decoder by substituting *MaxUnpool* layers with transposed convolutions, we want to investigate if reinstating them would improve the mIoU and, if so, determine the additional cost in

volved. As we cannot depend on MaxPooling indices as in the original network, we utilize the output of the downsampling RDDNeck module of the decoder as a side input to the corresponding UBNeck module of the decoder in a manner similar to UNet. However, unlike UNet, the feature map sizes do not match, so an extra upsampling layer is required in the connections. To minimize the impact of this additional layer, we reuse the existing layer already present in the side branch of the UBNeck, so it does not increase the parameter count.

Table 9 presents training outcomes on Cityscapes for various model versions. Each model is compatible with FINN and could be implemented on FPGA. We also evaluated LUT and BRAM usage for each using estimations, maintaining roughly the same estimated throughput between each model via parallelism parameters. Interestingly, adding the second (innermost) shortcut appears to reduce mIoU, while the first shortcut provides minimal enhancement. In any case (first, second, and both shortcuts), it requires more resources without bringing any gain in accuracy, making the reintroduction of shortcuts rather uninteresting. Meanwhile, increasing the projection ratio to 8 offers significant resource savings but results in a steep decline in mIoU, rendering this version impractical. The model with nearest-neighbors upsampling is the most promising, showing the least resource usage without compromising mIoU, which is surprising as we would expect transposed convolutions to perform better. Additionally, the nearest-neighbors version has a comparable parameter count to the baseline model and requires fewer computations (1.11G vs. 1.47G MACs for a 512x256 input).

Consequently, this model has been selected as the final version. To enhance its performance and minimize the difference with the baseline model, we performed fine-tuning for 200 epochs using a cosine scheduler and an initial learning rate of $5 \cdot 10^{-4}$. This process elevated the mIoU to 70.33%, which is only 0.2% behind the floating point baseline after retraining.

Model	Dataset	Inp. Size	LUT	BRAM 18Kb	FF
P-ENet _S	Cityscapes	512x256	232,930 (44.56%)	929 (47.21%)	186,646 (17.85%)
P-ENet _M	Cityscapes	512x256	256,992 (49.16%)	914 (46.44%)	205,756 (19.68%)
P-ENet _C	CamVid	480x360	273,563 (52.33%)	985 (50.05%)	216,730 (20.73%)

Table 10: Resources utilization of the different versions of the design on AMD ZU19EG.

We implemented this model on the ZU19EG FPGA using FINN, introducing two model variants: P-ENet_S and P-ENet_M. The latter increases parallelism to boost throughput and leverages more FPGA resources. Resource consumption for both versions is presented in Table 10. None of the designs employ DSPs, and memory is shared between BRAMs and LUTs. Both of these versions end up being able to run on a 300MHz clock. Throughput and latency metrics were measured at this clock rate, as shown in Figure 35, demonstrating a dependency on batch size, a trait of pipelined design that requires sufficient inputs to achieve maximum performance, as the initial latency (*i.e.* the time before the first output) becomes negligible in front of the total latency with a large batch size. This peak is reached at approximately 50 inputs, where the P-ENet_S version achieves 126 FPS throughput, while the P-ENet_M model achieves 226 FPS. Latency measurements reflect the same dependency, reaching minimums of 8 ms and 4.4 ms per input for the two versions, respectively, as depicted in Figure 35. Surprisingly, the faster, more parallel version does not use a lot more resources than the small ver-

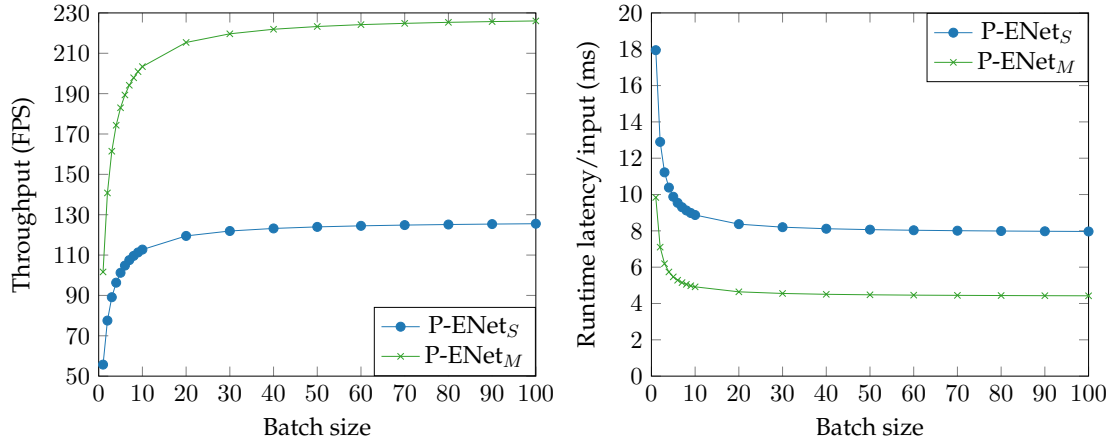


Figure 35: Throughput in frames-per-second (top) and runtime latency per input in ms (bottom) of the network on ZU19EG using a 300MHz clock and 512x256 input size.

sion. It is not so surprising regarding BRAM, as they are mostly used for FIFOs (over 75% is used for them each time), and we did not change the configuration between the two configurations. However, we expected LUT utilization to increase much more. The reason behind this phenomenon remains to be found. Nonetheless, our exploration of neural network architecture design allowed us to find a more efficient network architecture than the one initially implemented, and reach much higher throughput. Thanks to its low complexity, it consumes quite a few resources, which gives a lot of freedom to explore different parallelization configurations and reach different throughputs.

Since there are several previous works that used CamVid as a dataset to train the neural network, we also implemented a version trained on CamVid using the same neural network architecture to see how our network compares with the literature of real-time semantic segmentation of CamVid. The network reaches 76.26% mIoU with INT4 quantization and achieves 175 FPS with 480x360 inputs. The implementation results of this CamVid version that we named P-ENet_C are also presented in Table 10.

Given that we have greatly exceeded the typical requirements for real-time embedded systems regarding both latency and throughput, it will now be intriguing to examine how our results measure up against the current leading technologies. This comparison forms the basis of the discussion in the following section.

4.3.1 Comparison against previous FPGA implementations

Drawing comparisons between different FPGA implementations of semantic segmentation models is not always easy, as evaluating the quality of a solution relies on many metrics (accuracy of the model, throughput, resources usage, etc), and it is unlikely that any given implementation would be better than the others on all metrics, making it rather unrealistic to pretend that a solution is the absolute best. Moreover, articles are often rather evasive regarding the way their solution was evaluated, and how the presented metrics were obtained. It is also questionable whether two implementations made on two different FPGAs or using different input image sizes can be directly compared.

Nonetheless, as identifying where our contributions stand within the literature is important, we need to rely on the available information that we have. Table 11 shows different evaluation metrics of our solutions and previous works for both datasets. In terms of mIoU on Cityscapes, our versions outperform other reported implementations

	Model	Arch. HW	mIoU (%)	Target	Res.	Lat. (ms)	FPS	Clock Freq. (MHz)	LUT (K)	BRAMs 18K	FFs (K)	DSPs
Cityscapes	P-ENet_S [Ours]	Pip.	70.33	ZU19EG	512x256	8	126	300	232.9	929	186.6	0
	P-ENet_M [Ours]	Pip.	70.33	ZU19EG	512x256	4.2	226	300	257	914	205.8	0
	EnetHQ [148]	Pip.	36.8	ZU9EG	240x152	3	-	140	76.7	499	87.1	450
	ENet [149]	DPU	63.1	Zynq 7035	1024x512	720	32.9	-	62.6	514	192.2	689
	DeepLabV3+ [150]	DPU	65.29	Arria 10 GX 1150	960x960	670	-	200	-	-	-	1,362
	FPN [151]	DPU	57.6	ZU9EG	512x256	-	128	333	150	1,120	292	-
	ResNet18-Unet[Ours]	Pip.	62.9	Alveo U250	256x256	44	22.65	152	490.3	4,260	569.5	1,043
	MobileNet-PSPNet [152]	Pip.	64.2	ZU9EG	1024x512	9.1	139	300	182	1,831	137	1,831
CamVid	P-ENet_C [Ours]	Pip.	76.26	ZU19EG	480x360	5.7	175	300	273.6	985	216.7	0
	UNet [145]	DPU	67.8	Alveo U200	480x352	-	123	300	453.8	1,206	685.6	882
	AlexNet-SFCN [143]	Pip.	44.90	ZU9EG	480x360	-	165	100	105.1	561.5	100.5	390
	SegNet [144]	DPU	41	Arria 10 GX 1150	480x360	2891	-	-	-	-	-	1,451
	BEDN [146]	Pip.	59.9	ZU7EV	480x360	-	25.89	-	160.1	312	124.6	92

Table 11: Comparison between different FPGA implementations of semantic segmentation models trained on Cityscapes and Camvid.

by at least 5%. Notably, our versions exceed [148] by around 34% in mIoU, while achieving similar latency. This highly compressed model achieves the shortest latency among the listed models but at the cost of a significant reduction in mIoU. Regarding throughput (measured in frames per second), our implementation P-ENet_M delivers the highest performance but generally consumes more resources than other implementations, except for our ResNet18-Unet implementation. Conversely, the P-ENet_S variant offers throughput comparable to [151] and [152], utilizing more LUTs but fewer BRAMs, and no DSP.

Model	P-ENet _T [Ours]	MobileNet-PSPNet [152]
Platform	ZU19EG	ZU9EG
Freq. (MHz)	375	300
Latency/tile (ms)	0.417	0.502
Latency/image (ms)	7.5	9.1
BRAM (36k)	369.5(37.6%)	1,368(75%)
DSP	0(0%)	1,831(72.7%)
FF	289,678(27.7%)	137,000(25%)
LUT	450,000(86.1%)	182,000 (66.5%)

Table 12: Comparison with [152] in the same setup, using 224x224 tiles of a 1024x512 input.

It should be noted that the authors' implementation in [152] operates on 224x224 tiles derived from a 1024x512 input through a tiling algorithm, complicating direct comparisons. For the sake of attempting to provide a more equitable comparison given that this was the previous state of the art, we place our solution on an equal footing and introduce a version of our network, named P-ENet_T, which functions with 224x224 inputs and high parallelism and can operate with a 375MHz clock. Table 12 illustrates the performance comparisons, indicating that our implementation would surpass this previous work if the same algorithm were used.

For the CamVid implementation, our model exceeds prior studies by at least 8.46% in mIoU and also demonstrates better throughput. Nonetheless, it requires more resources than earlier implementations, except [145]. All in all, our implementation of

ENet appears to be competitive with the literature, seemingly surpassing previous implementations on throughput and mIoU, but not on resource usage. These performance have been achieved thanks to our improvement of ENet’s accuracy through retraining, our modifications of the network’s architecture to make it implementable as a pipelined dataflow architecture that fits the network topology, and the network’s design exploration that allowed us to identify a network structure that led to a more efficient implementation. This exhibits that co-designing neural networks and hardware architectures is crucial to achieve great combined performance of throughput, latency, and accuracy.

4.4 Performance evaluation against GPU

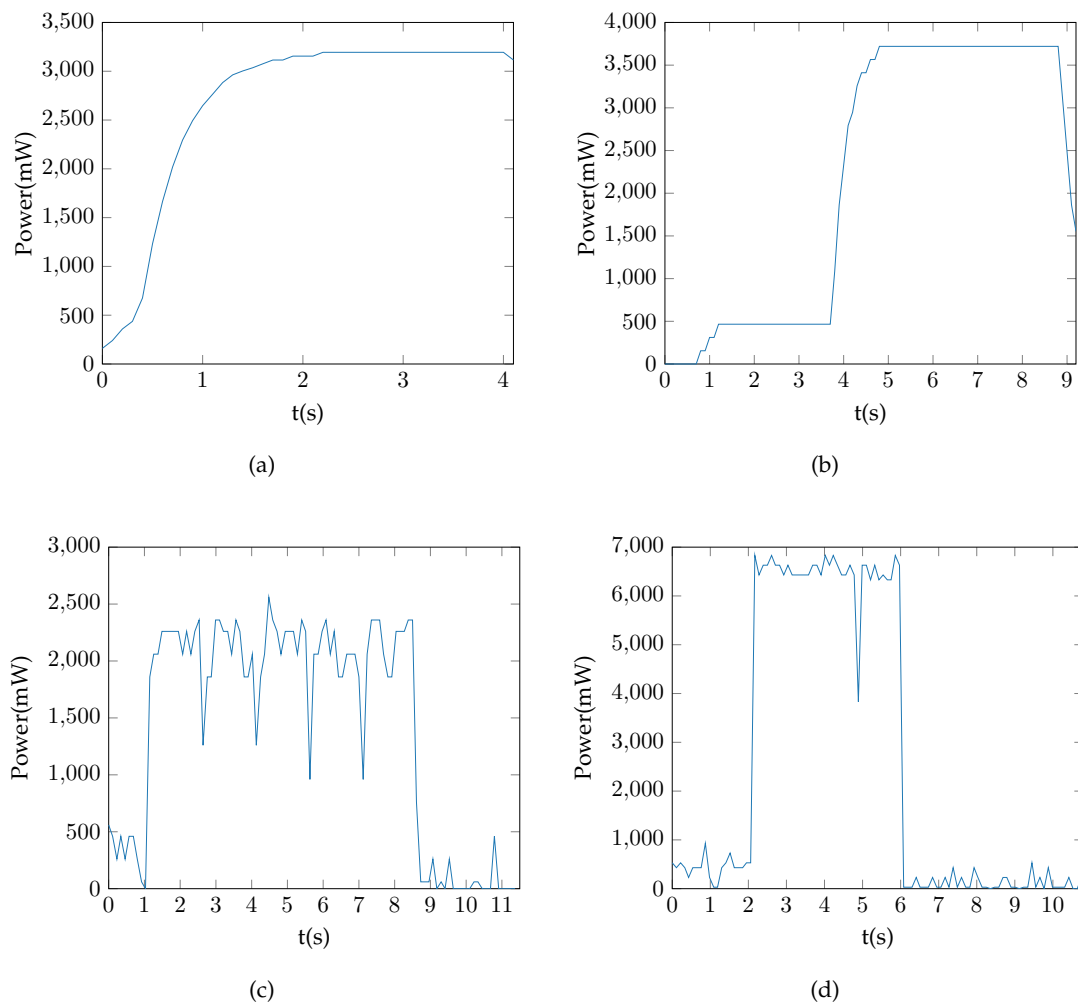


Figure 36: Dynamic power consumption over 1000 inferences on 512x256 inputs for different devices: Jetson Orin Nano GPU (a), Jetson AGX Xavier GPU (b), ZU19EG FPGA with P-ENet_S (c) and P-ENet_M (d) designs.

In Chapter 2, we discussed the role of FPGAs for embedded deep neural networks systems, and how they appeared to be promising alternatives to embedded GPUs in some contexts, especially by usually having a lower power consumption. Therefore, we thought it would be interesting to see if our FPGA design implementing ENet could compete with embedded GPUs.

We evaluate the model’s performance on the two Nvidia embedded GPUs available to

us, the Jetson AGX Xavier and the Jetson Orin Nano. These GPUs have distinct power modes: AGX Xavier supports 10W, 15W, and 30W, while Orin Nano supports 7W and 15W. Since 15W is the sole common power mode, both GPUs are tested under this condition to ensure fair comparison. Note that GPU performance scales with power mode, keeping power efficiency (throughput per watt) rather consistent. To maximize Jetson devices' performance, we use the Nvidia tool TensorRT to compile the model into optimized acceleration engines tailored to each GPU. For data types, FP16 is chosen as it is faster than FP32. Although Orin Nano's Ampere architecture supports INT8 format, AGX Xavier's Tegra architecture does not, so we use FP16 on both GPUs. We initially also considered doing measurements with INT8 on the Orin Nano, but the throughput was lower than with FP16, which appeared counter-intuitive. After some analysis, it appeared to be a problem related to how TensorRT handles quantized models which did not produce expected results with ENet. For precise measurements, we performed 1000 inferences on 512x256 inputs, calculating average throughput and latency for each case. Under these conditions, Jetson Orin Nano achieves 256 FPS with 4.7 ms latency, while Jetson AGX Xavier achieves 158 FPS with 7.2 ms latency. The latency measures correspond to the mean latency for each inference, taking account of the host-to-device, device-to-host and GPU compute times. Since raw throughput values do not give enough information for an accurate comparison, we also conducted comprehensive power consumption measurements.

Device	Techno.	FPS	Avg. Lat.	Peak Power
Jetson Orin Nano	7 nm	256	4.7 ms	3.2 W
Jetson AGX Xavier	12 nm	158	7.2 ms	3.7 W
ZU19EG (P-ENet _S)	16 nm	126	8 ms	2.56 W
ZU19EG (P-ENet _M)	16 nm	226	4.2 ms	6.83 W

Table 13: Performance and peak power consumptions of the evaluated devices and designs, measured on 1000 inferences on 512x256 inputs.

To ensure a fair comparison between these heterogeneous hardware architectures, our focus is on the dynamic power consumption of GPUs specifically for Jetson boards, and solely the FPGA for the TySOM-3A board, which means that we withdraw the average static power consumption (*i.e.* when no computations are performed on the device) from the total power measurements. This approach eliminates the influence of additional device-specific components, programs running in the background, or interfaces from the measures. The dynamic power measurements across 1000 inferences are depicted in Figure 36, and the throughput, latency, and power metrics are outlined in Table 13. As shown by the table and figures, the P-ENet_S FPGA design delivers performance comparable to the Jetson AGX Xavier while maintaining a lower power requirement than its GPU counterparts. Conversely, the P-ENet_M implementation achieves performance on par with the Jetson Orin Nano in its highest power mode, yet it experiences approximately double the power consumption.

For each device, we also computed the power efficiency as the ratio between the throughput and the peak power consumption, and the energy efficiency as the amount of energy per inference. Both metrics are interesting from an embedded systems point of view, as power measures give an idea of the power budget required by the semantic segmentation inference, while the energy measures inform about how many inferences could be performed before a battery with a certain capacity runs out. Figure 37 illustrates that P-ENet_S is 1.15x more power efficient and 1.28x more energy efficient than the Jetson AGX Xavier but less efficient than the Jetson Orin Nano on both metrics. Note that we measured the performance of the GPUs without batching, performing single-

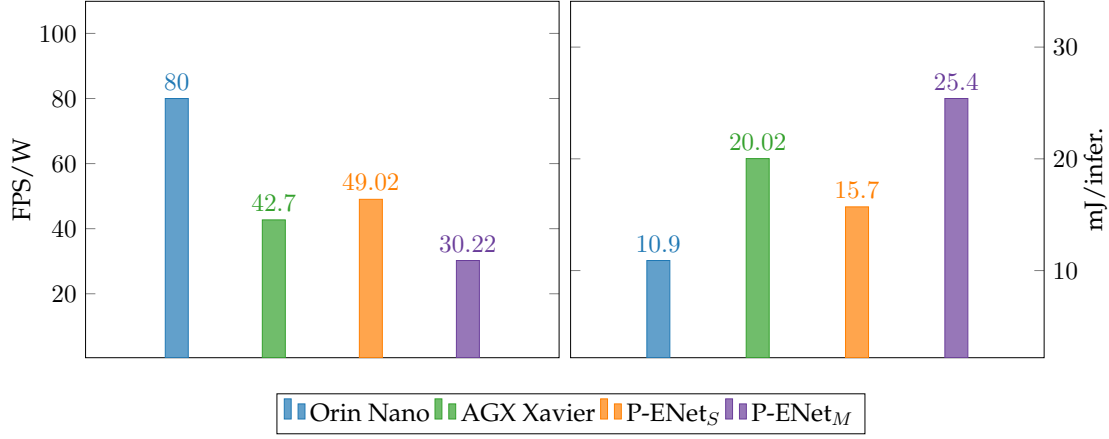


Figure 37: Power efficiency in FPS/W (left) and energy consumption in mJ/inference (right) of each device.

input inferences. Although batching is possible and slightly increases the throughput, it also increases the power consumption and leads to lower power efficiency as shown by Table 14, so we kept a batch size of 1 for the comparisons.

Device	FPS			Peak Power (W)			Power eff. (FPS/W)		
	b=1	b=10	b=100	b=1	b=10	b=100	b=1	b=10	b=100
Jetson Orin Nano	256	281	286	3.2	3.4	3.7	80	80.3	77.3
Jetson AGX Xavier	158	180	185	3.7	5.1	5.4	42.7	34.9	34.2

Table 14: Performance, peak power consumptions, and power efficiency of the evaluated GPU devices measured on 1000 512x256 inputs, by batches of 1, 10, or 100 inputs.

On the other hand, P-ENet_M is less power efficient, but can deliver higher raw throughput if necessary, matching the speed of the Orin Nano. These findings are promising, particularly given that both embedded GPUs are built using a more advanced transistor technology than the ZU19EG FPGA, which plays in their favor, as devices with lower transistor lengths can deliver higher throughput with the same chip area size. The comparisons in power efficiency do not seem as conclusive as the one obtained during the DAC-SDC challenge that we presented in Chapter 2, but the FPGA used in this challenge was much smaller than ours, and the Nvidia Jetson TX2 that was used for the GPU entries is from an older generation than the ones we used, having a 16 nm transistor technology (identical to the FPGA). If we had one, it would have been interesting to measure the power efficiency of our network on a Jetson TX2 as well to see if we would reach a $\times \frac{1}{2}$ to $\times \frac{1}{3}$ reduction in power consumption as well. Nonetheless, our results corroborate that FPGAs can surpass embedded GPUs in semantic segmentation scenarios where extremely high throughput is not essential, but low power consumption is critical. While GPUs are limited to a few power modes, pipelined dataflow architectures on FPGA give much more flexibility to explore the balance between accuracy, throughput, and resource usage, allowing to obtain implementations that, even if they may not be as power efficient as high-end embedded GPUs, can reach lower power consumption, which can be critical for some applications. Moreover, the measurements were conducted on a rather large FPGA device, but this large design space also offers the opportunity to find architectures that can be deployed on smaller and less power-consuming FPGAs.

4.5 Summary and discussions

In this chapter, we presented our implementation of two semantic segmentation deep neural networks as pipelined dataflow architectures using FINN. Our first proposal of a ResNet18-UNet was led by our initial understanding of the framework and its limitations as well as the will to start with a classical semantic segmentation network to explore the capabilities of FINN. Although this first implementation did not lead to sufficient results for us to claim that it would be suitable as a basis for a real-time system, it helped us acquire a deeper understanding of FINN, what to be mindful of when designing pipelined dataflow architecture, and the implications on the choice and design of the neural network. The continual improvements of FINN by the developers and also external contributors opened the possibility of implementing more efficient semantic segmentation networks, which led us to our second experiment based on ENet. We improved the initial accuracy of the model through retraining with better hyperparameters and extensive data augmentation and achieved a 12.2% increase on Cityscapes and a 26.3% increase on CamVid, proving that a lightweight model can compete with more complex models if properly trained. We presented our methodology to transform the model into a version implementable as a pipelined dataflow architecture with FINN, with only a small accuracy degradation with INT4 quantization, which meets the throughput and accuracy requirements of real-time inference. We further improved this implementation by conducting a hardware-oriented design space exploration of the neural network architecture and identified a version of the model that is more efficient than the initial version, reaching similar mIoU with a lower amount of required resources. This led to two implementations trained on Cityscapes with different parallelization parameters, and a version on CamVid. They show state-of-the-art results, displaying a better mIoU than previous works by at least 5% on Cityscapes and 8.4% on Camvid, with overall better throughput and latency. They also display a promising comparison with embedded GPUs, showing 1.15 times better power efficiency than a Jetson AGX Xavier, although not quite as efficient as last generation embedded GPUs like the Orin Nano, which can be explained by their much more advanced technology.

There are several leads for improvement of the design, starting with a more thorough study of quantization, as there is certainly a lot of room to find a more efficient quantization configuration for the model. Speaking of compression, we did not try to leverage other types like pruning, more precisely structured pruning. It could have been interesting for ResNet18-UNet as it is a rather large model, but we had the intuition that it would not lead to interesting results with ENet, since it already has very few parameters. However, it might have been interesting to study the question anyway. Regarding the hardware side, there are a lot of opportunities for improvement, especially regarding the placement and sizing of FIFOs. The first step would be to try to tune the parallelization parameters to balance the data consumption and production of each layer and obtain a pipeline with more evenly distributed latency, which would lower the buffer requirements. Unfortunately, part of the reason why ENet is able to achieve a high throughput it is very deep, but not very wide, *i.e.* it has a lot of rather small layers (and therefore fast to process), which is very beneficial in a pipelined architecture, as the throughput of the design is dictated by the throughput of its slowest element. However, this is also what makes a manual optimization of the pipeline especially difficult and time-consuming. Unfortunately, the current options for the automated determination of parallelization parameters of FINN are not of great help in that regard, but there is a high chance that this will be improved in future releases. Moreover, as we mentioned previously, the automatic FIFO sizing is usually very slow and often leads to inaccurate results, which requires manual adjustments that are difficult to optimize

properly. The developers have released a better version of FIFO sizing that relies on a characterization of each layer's dataflow rather than an RTL simulation which allows for obtaining faster and more accurate results but is currently incompatible with non-linear neural network architectures. Again, there is hope that the placement of FIFOs will be improved in the future, which would be extremely beneficial to the framework, as they play a major role in the performance and memory footprint of the designs.

Chapter 5

Upsampling method co-design

Contents

5.1	Context and motivations	112
5.2	Training setup, quantization, and results	113
5.3	Comparative implementation results	114
5.3.1	Inference transformations	114
5.3.2	Two-times upscaling	116
5.3.3	Higher upscaling ratios	117
5.4	Improving transposed convolution performance	118
5.5	Summary	120

In this last chapter, we present a project conducted in collaboration with researchers at AMD, namely Ian Colbert (Member of Technical Staff at AMD Radeon Technologies Group, San Diego, California), Jakoba Petri-Koenig (Research Engineer at AMD AECG Research in Dublin, Ireland) and Thomas Preußner (Principal Research Engineer at AMD Dresden, Germany). We study the hardware-software co-design of upsampling methods in deep neural networks, trying to identify the best technique based on accuracy and performance when implemented on FPGA using a streamlined dataflow architecture built with FINN. To do so, we chose to use the context of super-resolution neural network. It provides a much more appropriate context for this study, and the conclusions can easily be transferred to semantic segmentation networks with an encoder-decoder architecture.

5.1 Context and motivations

In the first chapter, we introduced different methods to perform upsampling in deep neural networks. When designing a neural network, the choice of upsampling layer is usually based on what it brings to the performance of the model, and is sometimes, if relevant, related to considerations of arithmetic complexity. The most natural way is to select the method before training, to evaluate it based on the obtained performances, to balance it with the number of parameters and computations, and to use the same for inference. However, previous studies have shown that the different techniques can have different interests during the training and inference phases. Specifically, prior research has indicated that subpixel convolution typically results in faster convergence and fewer errors during training compared to other upsampling methods [17]. On the other hand, transposed convolution is noted for its energy and memory efficiency during inference, provided it is executed in a manner that minimizes or completely avoids overlapping sums, as subpixel and resize convolution are prone to expensive high-dimensional processing [167]. Nevertheless, it tends to create checkerboard artifacts in the training phase [168]. This suggests a strategy of training with subpixels and switching to transposed convolution for inference, following a method similar to the one proposed by Colbert *et al.* [167]. These studies, however, were done in floating-point precisions, using analytical models for arithmetic intensities so we thought it would be interesting to see to what extent it transfers to the quantized domain since it could be influenced by elements like quantize/dequantize operators or straight-through estimators, as well as the context of pipelined dataflow architectures on FPGA built with FINN. To do so, we chose to use the task of super-resolution, as it is very suitable to evaluate the influence of upsampling methods on the quality of produced images, since super-resolution is, in essence, "just" image upsampling.

We carry out this study using the super-resolution network ESPCN [25] as a basis, which is a very small CNN originally introduced as a way to leverage sub-pixel convolution. Its simple and lightweight structure is very convenient for our study, as it will be fairly fast to train, and easy to deploy on FPGA with FINN, which makes it easy to try different configurations of upsampling methods, quantization, and upscaling factors. It is also very suitable for real-time embedded super-resolution and allows us to target very small FPGAs, such as the one in the Kria KV260 board from AMD. Although its performance on super-resolution tasks are largely outperformed by more recent and complex neural networks, it may still be interesting for some small-scale edge applications that require very small power consumption. Moreover, the goal here is not particularly to build the most efficient super-resolution system but to have a convenient and relevant baseline to study the co-design of upsampling methods in CNNs and FPGA

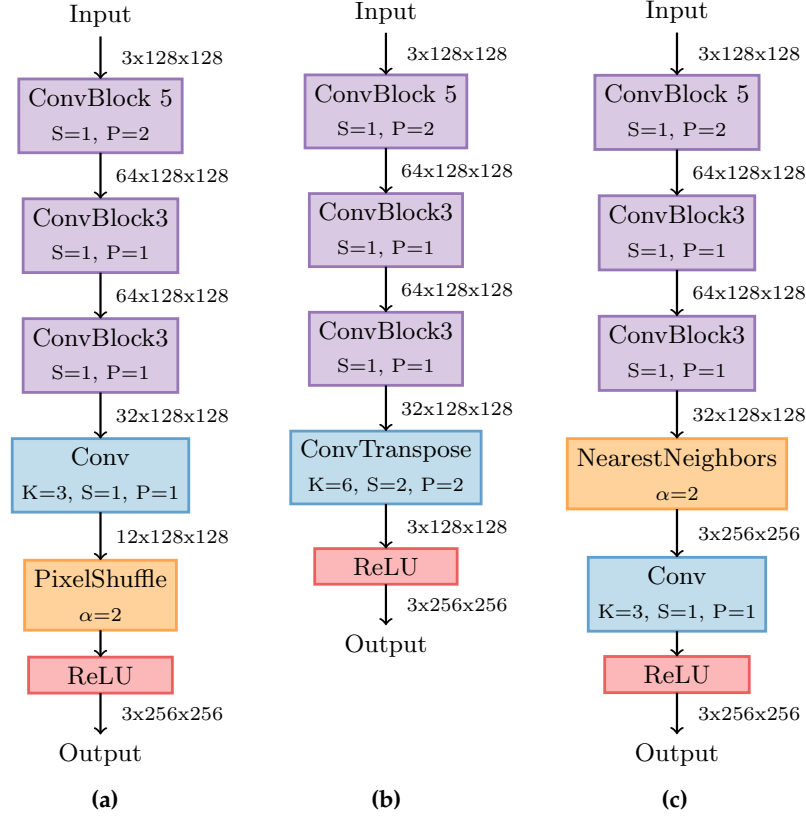


Figure 38: Illustration of the ESPCN neural network with the example of 128x128 inputs and an upscaling factor $\alpha = 2$. *ConvBlock* is the *Conv-BatchNorm-ReLU* association. The original network uses sub-pixel convolution (a), which can be replaced by other upsampling methods such as transposed convolution (b) or nearest-neighbor resize convolution (c).

architectures. The structure of the network is illustrated in Figure 38a, with the tensor sizes represented for a 128x128 resolution of input and two-times upsampling.

5.2 Training setup, quantization, and results

In an attempt to see if we could reproduce the results and conclusions obtained regarding the performance of different upsampling techniques and also see if they transfer to the quantized domain, we trained the ESPCN network on the BSD300 dataset using different configurations. For the upsampling methods, we considered nearest-neighbors resize convolution (mostly because nearest-neighbors is currently the only interpolation method supported by FINN), transposed convolution, and sub-pixel convolution. Switching from sub-pixel to the other two can be done by replacing the last *Conv* layer and *PixelShuffle* layer in the model with the appropriate equivalents: a transposed convolution layer with a kernel size of 3α , a padding of α and stride of α ; or a nearest-neighbors interpolation layer with a scale factor α followed by a convolution with a 3x3 kernel, padding and stride of 1, as shown in Figure 38. We also considered different upscaling factors α of 2, 3, and 4, to measure if increasing the factor leads to different behaviors. The different versions of the model were trained on 500 epochs, using the Adam optimizer, a MSE loss, an initial learning rate of $1e^{-3}$ that decayed by a factor of 0.999 every epoch, a weight decay of $1e^{-5}$ and a batch size of 8.

In terms of data augmentation, we used random cropping with a 512x512 window size and random horizontal and vertical flipping. Regarding quantization, we considered 4 different precisions: 3, 4, and 8-bit integers as well as 32-bit floating-point, and used QAT for the integer quantization. For the weights, uniform per-channel quantization was applied across all hidden layers, while the last layer utilized per-tensor quantization, both employing floating-point scaling factors optimized via gradient descent. For activation quantization, uniform per-tensor quantization was adopted for all layers and was refined using running average statistics. The network operates with UINT8 inputs and outputs, and a constant scaling factor of 1/255 was used for the final output layer to ensure direct correlation with RGB images. Table 15 highlights the best PSNRs observed over 5 training sessions for each configuration.

We note that no specific upsampling method consistently outperforms the others, regardless of quantization, which seems to contradict the results of previous studies. The upscaling factor appears to determine which technique excels, with Nearest-Neighbor Resize Convolution (NNRC) showing a slight advantage at 2x upsampling, while deconvolution yields better results for higher upscaling factors. Even though subpixel convolution never surpasses the other methods, it is never far behind, indicating that there is no one-size-fits-all solution for training and that the results vary based on the use case. This drives the interest in enabling transformations from NNRC and subpixel convolution to deconvolution during inference, to evaluate the interest of using a different algorithm for inference than training, based on the efficiency of the algorithm. We conduct these experiments in the context of inference performed with pipelined dataflow architectures on FPGA in the next section.

Method	Upscaling factor											
	2				3				4			
	INT3	INT4	INT8	FP32	INT3	INT4	INT8	FP32	INT3	INT4	INT8	FP32
NNRC	26.98	27.35	27.58	27.52	24.76	24.82	25.12	24.98	23.39	23.56	23.66	23.63
Subpixel	26.52	26.98	27.47	27.33	24.61	24.89	25.07	25.04	23.48	23.73	23.81	23.86
Transp. Conv.	26.44	26.86	27.60	27.45	24.57	24.92	25.13	25.10	23.55	23.89	24.01	23.98

Table 15: Best PSNRs (dB) over 5 trainings on BSD300 for different configurations of quantization, upsampling method, and upscaling factors of the ESPCN model. NNRC stands for Nearest-Neighbors Resize Convolution.

5.3 Comparative implementation results

5.3.1 Inference transformations

For the inference on FPGA, we identify four scenarios: nearest-neighbors resize convolution, transposed convolution, nearest-neighbors converted to transposed convolution, and sub-pixel convolution converted to transposed convolution. The transposed convolution is implemented as a fractionally-strided convolution, composed of pixel-padding followed by a regular convolution, as introduced in Chapter 3. The conversion from subpixel and nearest-neighbors to transposed convolution is done using the methods presented in [167].

Sub-pixel to deconvolution

For a given upscaling factor α , a sub-pixel convolution starts with a convolution of kernel size k_c and padding p_c that projects onto a space with $C'_c = \alpha^2 \times C'$ channels, where C' is the number of output channels, before applying the pixel shuffle algorithm over the output space. This can be converted into a functionally equivalent deconvolution

with a stride $s_d = \alpha$, a kernel size $k_d = \alpha k_c$ and padding $p_d = \alpha p_c$ using the weight shuffle algorithm [167] to transform the convolution kernel W_c into the deconvolution kernel W_d : This conversion is also a one-time cost done before deploying the model.

Algorithm 2 Weight Shuffle

```

procedure WEIGHT SHUFFLE
  for  $c = 0, c++$  while  $c < C$  do
    for  $c' = 0, c'++$  while  $c' < C'$  do
      for  $i_d = 0, i_d++$  while  $i_d < k_{h,d}$  do
        for  $j_d = 0, j_d++$  while  $j_d < k_{r,d}$  do
           $i_c \leftarrow k_{h,c} - \left\lfloor \frac{i_d}{\alpha} \right\rfloor - 1$ 
           $j_c \leftarrow k_{r,c} - \left\lfloor \frac{j_d}{\alpha} \right\rfloor - 1$ 
           $c'_c \leftarrow \alpha^2 c + \alpha \text{mod}(i_d, \alpha) + \text{mod}(j_d, \alpha)$ 
           $W_d[c, c', i_d, j_d] \leftarrow W_c[c, c'_c, i_c, j_c]$ 

```

It removes the necessity to apply pixel shuffling at each inference pass which greatly simplifies the FPGA integration, regardless of the framework used. However, in the case of quantized operations, it constrains the scaling factor to be per-tensor and not per-channel for the weights to be shuffled, which could influence the performance of the quantized model. However, in our experiments with ESPCN, it did not appear to be a downside.

Resize convolution to deconvolution

A nearest-neighbor resize convolution is composed of a nearest-neighbor interpolation step to upsampling the feature maps by a factor α , followed by a convolution with a kernel size k_c and padding $p_c = \frac{k-1}{2}$. It can be transformed into a transposed convolution with a kernel size $k_d = k_c + \alpha - 1$, the same padding, and a stride $s = \alpha$, using the weight convolution algorithm [167] to transform the weight tensor:

Algorithm 3 Weight Convolution

```

procedure WEIGHT CONVOLUTION
  for  $c = 0, c++$  while  $c < C$  do
    for  $c' = 0, c'++$  while  $c' < C'$  do
      for  $i = 0, i++$  while  $i < \alpha$  do
        for  $j = 0, j++$  while  $j < \alpha$  do
           $W_d[c, c', i : i + k_c, j : j + k_c] \leftarrow W_c[c, c', :, :]^T$ 

```

It acts as a transposition of the weight kernels followed by a convolution over the weight space. Similar to the weight shuffle, it can be performed before deploying the model on hardware but also has the advantage of reducing the theoretical number of required MAC operations by a ratio of $\frac{(\alpha+2)^2}{9\alpha^2}$. Regarding quantization, however, transforming the weights involves linear combinations that may end up requiring more bits for the representation of the obtained weights than the initial convolution, due to accumulation. From then, we can either expend the bit width to fit the new range of values at the cost of more hardware resources or clip the weights to fit the initial precision, which could degrade the accuracy of the model. This is once again a trade-off between performance and resource that can be studied. However, for now, we chose to not clip the weights in our experiments and present a "worst-case scenario" regarding resources consumption, and also to perform inferences that do not change the accuracy of the model and stay true to the performance measured after training.

5.3.2 Two-times upscaling

For this first round of experiments, we consider the case where $\alpha = 2$ and use FINN to build the network with the four types of upsampling, considering INT3 and INT4 quantization, targeting the FPGA of the Kria KV260 board. We chose the parallelization parameters to achieve the same throughput of around 40 FPS with a 200 MHz clock frequency for each configuration. For the FIFOs, we let the automatic placement and sizing tool operate, only limiting the maximum depth to 2^{14} to ensure the design could fit on the FPGA. The memory resources used for FIFOs were arbitrarily spread between LUTs, BRAMs, and URAM. All the designs' performances have been validated at least through RTL simulation, and some of them have been run on target, showing that the designs were able to achieve the throughput estimations. The resource utilization of all studied configurations is displayed in Table 16.

Resource	Method			
	TC		D-SP	
	INT3	INT4	INT3	INT4
LUT	67,332 (57.5%)	85,607 (73.1%)	67,499 (57.6%)	84,677 (72.3%)
FF	47,042 (20%)	55,431 (23.66%)	47,164 (20.1%)	55,272 (23.66%)
BRAM 36K	113 (78.5%)	141.5 (98.3%)	113 (78.5%)	141.5 (98.3%)
URAM	16 (25%)	16 (25%)	16 (25%)	16 (25%)

Resource	D-NN		NNRC	
	INT3	INT4	INT3	INT4
LUT	64,223 (54.8%)	81,372 (69.4%)	44,893 (38.3%)	54,498 (46.5%)
FF	45,323 (19.3%)	53,008 (22.6%)	30,990 (13.2%)	34,361 (14.7%)
BRAM 36K	61 (42.4%)	103.5 (71.8%)	54.5 (37.8%)	65 (45%)
URAM	20 (31.3%)	16 (25%)	2 (3.1%)	2 (3.1%)

Table 16: Resource utilization of the implementations of the ESPCN network with different upsampling methods and quantizations. "TC" stands for "Transposed Convolution", "D-SP" is sub-pixel convolution converted to deconvolution, and "D-NN" is NNRC converted to deconvolution.

These results show that transposed convolution and sub-pixel convolution converted to deconvolution (D-SP) use the same amount of resources, which was expected considering that the transposed convolution obtained from the sub-pixel convolution end up with the same parameters as the one in the model that was trained with it. NNRC converted to deconvolution (D-NN) uses less resources than transposed convolution, which is explained by the kernel size of the obtained transposed convolution layer being smaller, significantly reducing the amount of resources used. Meanwhile, a basic NNRC implementation uses the least resources, even less than D-NN, which seems counter-intuitive considering that D-NN is supposed to require less computations. This can be explained by the weights of the last layer needing higher bit widths due to the conversion. Although transposed convolutions are supposed to be faster and more efficient in software, it does not appear to be the case in FPGA hardware, at least in this context. To be fair, this could be greatly influenced by our hardware implementation of transposed convolutions. This will be discussed further in Section 5.4. Regarding quantization, results appear to remain proportional between INT3 and INT4 across the different methods.

5.3.3 Higher upscaling ratios

In this second round of experiments, we considered higher values of α , as higher upscaling is considered more challenging, which could lead to different comparison outcomes. We studied the cases of x4 and x8 upsampling with the same 4 algorithms as previously, and considered two scenarios each time, with the upsampling being done by a single layer, or split into multiple layers. Thus, the x4 upsampling can be done with one layer with $\alpha = 4$ or with two $\alpha = 2$ layers, and x8 upsampling can be done with one layer with $\alpha = 8$ or with three $\alpha = 2$ layers. We only considered INT4 quantization, to reduce the number of different versions, as we already have 32 implementations, and also because our previous experiment with x2 upsampling did not identify changing the quantization as particularly conclusive. Regarding the placement of FIFO, since the automatic placement usually takes a significant amount of computation time, we manually placed and sized them. We placed a 2^{14} -deep FIFO every time there was a different data width between two layers, which indicates that two subsequent layers have different parallelism and dataflow, which is where buffers are usually required. This happens often around MVAUs, padding layers, or nearest-neighbor upsampling layers. This is an arbitrary decision for convenience, and the actual FIFO placement would probably lead to lower depths, but we wanted to ensure that every architecture could reach the estimated throughput. As we noticed that our implementations of x2 upsampling were pushing the capacities of the Kria KV260 close to their limits in some cases, we switched to a larger FPGA, the AMD ZU7EV Ultrascale+ FPGA that is part of the ZCU104 board, to make sure all designs could fit. The implementation results are displayed in Table 17.

α	n	Method	FPS	LUT	FF	BRAM (36K)	LUT*	FF*	BRAM* (36K)
4	1	TC	21.2	54,643	30,931	146.5	53,321	29,646	32
		D-SP	21.2	53,405	30,342	147	51,918	29,845	34
		D-NN	42.3	45,024	29,621	126	43,617	28,475	28.5
		NNRC	42.3	39,644	28,568	178	37,970	27,048	19
	2	TC	42.3	39,350	32,755	152	34,900	30,244	19
		D-SP	42.3	39,191	32,768	142	34,746	30,217	19
		D-NN	42.3	34,314	28,220	118.5	32,708	26,947	20
		NNRC	42.2	33,985	28,365	153	32,085	26,606	19.5
8	1	TC	1.3	84,713	32,671	152	83,272	31,294	42.5
		D-SP	6.46	84,643	32,685	152	83,205	31,294	42.5
		D-NN	7.6	69,669	32,710	129.5	59,581	31,216	35.5
		NNRC	42.2	45,296	30,489	186	43,538	28,865	13
	3	TC	21.2	47,178	32,275	144.5	44,955	30,501	14.5
		D-SP	21.1	47,166	32,307	144.5	44,945	30,533	14.5
		D-NN	42.3	42,901	31,915	124	41,057	30,428	20
		NNRC	42.2	37,179	30,342	208	34,601	28,032	19.5

Table 17: Resource utilization of the ESPCN model on FPGA for different upscaling factors α , number of upsampling layers n , and upsampling methods. The columns marked with "*" contain the values when withdrawing the number of resources used for FIFOs from the total.

We tried to set the parallelization parameters to reach the same throughput of around 42 FPS each time. However, in a few cases, it was not possible to reach this throughput, in which case we configured the design to reach the largest achievable throughput. This happens when, due to large kernel sizes (or large input size for NNRC), even the maximum parallelism of the *ConvolutionInputGenerator* that creates the input is not enough to reach the desired throughput. This is due to the parallelization being performed

across the channel dimension, which limits the number of SIMD lanes to be equal to the number of channels in the input, at best. Since the depth does not change as we increase α , but the size of the matrix generated by the *ConvolutionInputGenerator* does, it becomes impossible to reach the desired throughput, even with a fully parallel architecture. This also explains why, due to hardware design, it is much more efficient to split the upsampling between multiple layers than to do it with just one, as illustrated by the throughput and resources values in the table, which is an interesting observation from a hardware-oriented neural network design perspective.

The obtained results lead us to draw similar conclusions as for x2 upsampling. As expected, the D-SP and transposed convolution implementations show equivalent performance and resource utilization and are outperformed by D-NN and NNRC. NNRC in particular, is the only implementation that is able to reach 42 FPS in every configuration and shows the lowest resource consumption in terms of LUTs and FFs but appears to use more BRAMs. However, this can be attributed to the FIFO placements, as well as different memory repartitions than other implementations, as shown by the resources usage when excluding the FIFOs (columns marked with a " * " in the table). Nevertheless, in configurations where the upsampling is split among multiple layers, D-NN is fairly close to NNRC in terms of resources and could be considered better when taking into account FIFOs. Real-case implementations with a FIFO placement optimized for the architectures would be necessary to tell which one is the actual best.

These results tend to show that NNRC is more interesting than transposed convolution in most cases. However, this conclusion is tied to the implementation of the transposed convolution that we chose, which is not the most optimal from a hardware perspective. A different implementation of transposed convolutions could lead to different results, as we will discuss in the next section.

5.4 Improving transposed convolution performance

As we discussed before, our implementation of transposed convolution was based on the fractionally-strided convolution algorithm. There are other ways to implement it, and the choice to use this algorithm was mainly motivated by the ease to integrate it into FINN. As it is just a pixel-padding layer followed by a regular convolution, we only had to implement this new special type of padding and take advantage of the already existing support of convolutions. However, this implementation is not extremely efficient, as it creates large feature maps containing a lot of zero-valued pixels, especially as α increases. Since most hardware implementations of convolutions (including the one in FINN) cannot leverage sparsity, it results in large amount of redundant and unnecessary computations. This could have skewed the results of our study and could be an argument as to why transposed convolution appears to be less efficient than NNRC for inference. This motivates the interest of trying to implement a better algorithm.

Zhang *et al.* [169] introduced an algorithm for transposed convolution that also solves the overlapping sum problem called Reverse Looping Deconvolution (REVD). The idea is to loop over the output space rather than the input space, by taking a s^2 -sized window (with s the deconvolution stride) of the output and determining which inputs are needed to compute the values in the window, each tile being independent and therefore compatible with a concurrent parallel processing. However, recasting the loops from the input to the output space requires to use a technique called stride-hole skipping to select correct index values for the input, which introduces several modulo operations as shown by Algorithm 4. These operations need to be computed at runtime and can be

Algorithm 4 Reverse Looping Deconvolution (REVD)

```

procedure REVERSE LOOPING DECONVOLUTION
  for  $c' = 0, c'++$  while  $c' < C'$  do
    for  $\hat{o}_r = 0, \hat{o}_r++ = s$  while  $\hat{o}_r < R$  do
      for  $\hat{o}_h = 0, \hat{o}_h++ = S$  while  $\hat{o}_h < H$  do
        for  $i = 0, i++$  while  $i < k_h$  do
          for  $j = 0, j++$  while  $j < k_r$  do
            for  $c = 0, c++$  while  $c < C$  do
               $o_h \leftarrow \hat{o}_h + \text{mod}(s - \text{mod}(p - k_h, s), s)$ 
               $o_r \leftarrow \hat{o}_r + \text{mod}(s - \text{mod}(p - k_r, s), s)$ 
               $i_h \leftarrow (o_h + p - k_h)/s$ 
               $i_r \leftarrow (o_r + p - k_r)/s$ 
               $y[o_h, o_r, c'] \leftarrow x[i_h, i_r, c] \times w[k_h, k_r, c, c']$ 

```

quite costly. Moreover, since the algorithm goes through the output space in s^2 tiles, it creates issues when the output size is not perfectly divisible by the stride, which often leads to a less efficient computation.

For these reasons, Colbert *et al.* [167] introduced an improved version of the reverse looping deconvolution algorithm called REVD2 which applies stride-hole skipping over the weight space instead of the output space, which reduces the cost of modulo operations and makes it compliant with more input sizes. This algorithm was proved to be

Algorithm 5 Improved Reverse Looping Deconvolution (REVD2)

```

procedure REVD2
  for  $c' = 0, c'++$  while  $c' < C'$  do
    for  $o_h = 0, o_h++$  while  $o_h < H$  do
      for  $o_r = 0, o_r++$  while  $o_r < R$  do
        for  $c = 0, c++$  while  $c < C$  do
          for  $\hat{i} = 0, \hat{i}++$  while  $\hat{i} < k_h$  do
            for  $\hat{j} = 0, \hat{j}++$  while  $\hat{j} < k_r$  do
               $i \leftarrow \hat{i} + \text{mod}(o_h + p, s)$ 
               $j \leftarrow \hat{j} + \text{mod}(o_r + p, s)$ 
               $i_h \leftarrow (o_h + p - k_h)/s$ 
               $i_r \leftarrow (o_r + p - k_r)/s$ 
               $y[o_h, o_r, c'] \leftarrow x[i_h, i_r, c] \times w[k_h, k_r, c, c']$ 

```

much more efficient than fractionally-strided convolution in terms of computation time and energy with GPU-based analytical cost models, bringing D-SP and D-NN above NNRC in terms of efficiency. Therefore, it appears as a much better algorithm to implement transposed convolutions. Obviously, there is no guarantee that the observed results would transfer to FPGA, as it would greatly depend on how the algorithm is implemented. Nevertheless, it will likely outperform our previous implementation. At the time of writing this manuscript, an implementation of REVD2 in FINN is on the way, and once finalized, it will allow us to conduct the same previous inference experiments but with this algorithm, which will be the final step to conclude this study.

5.5 Summary

In this chapter, we presented a study done in collaboration with researchers at AMD on the hardware-software co-design of upsampling methods in CNNs. Based on the observations from previous studies that the upsampling methods that lead to the best model accuracy are not necessarily the ones that perform the best regarding inference speed and energy cost, we studied if these findings would transfer to the domain of inference on FPGA. We used the super-resolution model ESPCN as a baseline and performed training with either sub-pixel convolution, transposed convolution, and nearest-neighbor resize convolution as the upsampling method used in the network, for different upscaling ratios and quantization. Our experiments highlighted that there are no clear winner, and the more adequate method to use for training is contextual. Therefore, it is interesting to explore the performance of the method during inference, to be able to determine which algorithm is the most efficient for joint considerations of hardware performance and accuracy. Regarding implementation on FPGA with FINN, we used previously published algorithms to convert nearest-neighbor resize convolutions and sub-pixel convolutions to transposed convolution, and studied the performance of each method or sub-method in terms of throughput and resources usage for different quantization parameters and upscaling factors. We found that in most scenarios, nearest-neighbor resize convolution performed better than other methods, with nearest-neighbor resize convolution converted to deconvolution being close behind. Interestingly, this corroborates our results of the previous chapter on semantic segmentation. These results differ from the ones obtained from prior studies of upsampling methods for inference on GPU, which further highlights the importance of rethinking the design of neural networks for inference on FPGA rather than deriving from results obtained on GPU. However, we suspect that these results are largely influenced by our implementation of transposed convolutions as fractionally-strided convolutions. Our future work for this study goes towards integrating in FINN a better algorithm, REVD2, and conducting experiments with it to see if it leads to different conclusions. Finally, our study exhibits that, when employing pipelined dataflow architectures, performing upsampling with high upscaling factors is more efficient when split between multiple upsampling layers than with just one, which is a valuable information from a co-design point of view.

With this final chapter done, we can move to the general conclusion of this thesis, wrapping up this manuscript.

Conclusion

Summary

Reminder of problem statement

Deep neural networks have become predominant solutions to many complex machine vision tasks. Some tasks, such as semantic segmentation, are essential to applications like autonomous driving or UAVs. However, these applications require the inference of the neural network to be performed within the constraints of low-latency, high-throughput processing, and low power consumption, making common GPUs inadequate hardware devices for neural network inference in this context and must rely on embedded hardware. In that regard, while embedded GPUs offer a straightforward solution and provide an easy deployment, FPGAs have proven to have the potential to be more efficient for embedded real-time machine vision tasks, especially in terms of power consumption [8]. Nonetheless, the complex design flows and low availability of computing and memory resources make the deployment of neural networks on these hardware devices particularly challenging. Therefore, the implementation of large and complex networks as the ones employed in the semantic segmentation literature often rely on high model compression rates, such as very low-bit quantization or high pruning rates, which leads to significant accuracy degradation. Thus, deploying complex models on resource-limited devices like FPGAs and achieving a high frame rate and low latency while maintaining the performance of semantic segmentation is a significant challenge due to the necessary balance between resources and accuracy. As a result, selecting an appropriate model is complicated and demands a combined evaluation of implementation complexity and task accuracy. In this thesis, we demonstrated that the design and structure of the neural network play a key role in obtaining efficient hardware architectures that can meet real-time constraints.

Contribution to an open-source framework

As handcrafting hardware architectures to implement neural networks is a long and tedious process, we reviewed the different frameworks that help with deploying deep neural networks on FPGA and selected FINN as the most adequate candidate for the purpose of this thesis, as it helps design low-latency hardware architectures that are adapted to the neural networks they implement. In the third chapter, we presented our contributions to this open-source tool, in the form of new streamlining transformations and the integration of transposed convolutions, which were necessary for our implementations, and participated in improving the support for semantic segmentation networks in the framework.

Investigations on efficiently implementing ResNet-based UNet networks on FPGA

In Chapter 4, we presented the main contributions of this thesis regarding the implementation of two encoder-decoder semantic segmentation neural networks on FPGA, starting with a ResNet18-UNet network. The obtained results of this experiment gave encouraging insights of the performance of pipelined dataflow architectures and led to interesting conclusions about them, particularly the roles of buffers and the potential hardware issues caused by seemingly trivial structural elements, such as adding a shortcut between the encoder and decoder. Thus, when working with this type of hardware architecture, some considerations extend beyond the framework's limitations and relate to the overall behavior of pipelined dataflow architectures.

Improvement of ENet's segmentation performances

Our second contribution within this chapter is the implementation of ENet, which has been designed specifically for real-time inference, and whose low-complexity design makes it an excellent candidate for implementation on FPGA as a pipelined dataflow architecture. The semantic segmentation performance of the model as reported by its authors was initially much lower than the state of the art, but we proposed better training parameters and extensive data augmentation, so that we were able to improve its score by over 12% on the Cityscapes dataset, and over 26% on CamVid. Thus, we proved that, with appropriate training, lighter and simpler networks can reach good performance and compete with more complex networks whose performances were solely attributed to their innovative architecture without considering the chosen training setup. Interestingly, this corroborates similar results found in the domain of image classification.

Hardware-oriented design space exploration

By applying several structural changes to the original network, we were able to implement it with FINN into an architecture able to meet the throughput and latency constraints of real-time inference by achieving 47 FPS with a 21 ms latency, without much compromising the accuracy of the model, even with INT4 quantization by remaining less than 1% behind the baseline. However, it introduced a significant increase of number of parameters and computational complexity of the model. Thanks to a hardware-oriented design space exploration of the architecture of the neural network, we identified design variations that led to a more efficient implementation in terms of accuracy per hardware resource usage, leading to a model with fewer parameters, requiring less MACs, and achieving 70.3% mIoU, only 0.2% less mIoU than the baseline with INT4 quantization. This improved design, which we named P-ENet, shows state-of-the-art performance compared to prior FPGA implementations of semantic segmentation networks trained on autonomous driving datasets, achieving 226 FPS and a low latency of 4.2 ms. These results highlight that when using a streamlined dataflow architecture, focusing on a design of the neural network in line with the hardware architecture design is essential in optimizing the trade-off between throughput and model accuracy to achieve efficient FPGA implementations. As such, it appears as a more adequate approach than simply selecting a high-performing neural network, and employing high compression rates to make its deployment on FPGA possible.

Evaluation of power efficiency compared to GPUs

The comparisons of power and energy consumption of this model on FPGA and embedded GPUs highlight that the question of which one of these two hardware devices is

the most suitable for embedded semantic segmentation applications has no straightforward answer and depends on the context and requirements of the system. Our FPGA implementation showed a better power efficiency than the Jetson AGX Xavier. However, it is not quite on par with the power efficiency of last-generation GPUs like the Jetson Orin Nano. The latter is built on a 7 nm transistor technology, much more advanced than the 16 nm FinFET of AMD's Ultrascaple+ devices. Nonetheless, the design space of neural network implementation on FPGA is much larger than with GPUs which are limited to a few power and quantization modes, which gives more opportunities to reach very low-power design points by, for instance, settling on 30 FPS to reduce the resources usage, or exploring the trade-offs introduced by low-bit quantization. Doing so also enables targeting FPGA of different sizes, to find the appropriate balance of throughput and power consumption. Furthermore, FPGAs provide greater compatibility for system integration purposes, as they allow for a direct connection to the sensors. This eliminates the need for multiple supplementary software layers or additional hardware interfaces, and reduces the total latency. On a side note, the interest of FPGAs over embedded GPUs is significant for applications that can employ tiny neural networks, such as radio signal processing [170].

Co-design of image upsampling methods and hardware architectures

The study of the co-design of image upsampling methods in the last chapter points out that, in the context of real-time super-resolution, nearest-neighbor resized convolution outperforms other techniques for inference on FPGA when using a pipelined dataflow architecture. These results seem to contradict prior studies made using inference on GPU, which suggests that observations on the performance of neural networks on GPUs do not necessarily transfer to FPGAs, which further justifies the interest of rethinking neural network design when considering FPGA or ASIC implementations.

Perspectives

Aside from the leads of improvements of the current implementation of ENet that we identified in Chapter 4, such as a more in-depth analysis of quantization or further optimizations of the FIFOs, the results of this thesis open several perspectives for further research.

Managing low-resolution inputs

An important element that we have not really mentioned so far is the implication of using a lower input resolution. As we discussed, using directly the full resolution of 2048x1024 pixels of Cityscapes for inference on FPGA is unrealistic considering the amount of computations and memory that would be required, especially with the type of architecture we used. This resolution which is used for benchmarking is therefore rarely used for inference, even on embedded GPUs, and replaced by a lower one, such as 1024x512, or even lower in most FPGA implementations, including our own. However, reducing the resolution for inference is not trivial, as it can significantly reduce the semantic segmentation accuracy depending on how it is performed. Naively down-sampling the input and up-sampling the output would be very detrimental to the performance of the system. A solution could be to use tiling, splitting the large input resolution into smaller tiles that could be processed faster before reconstructing the output. This would probably be the easier approach, at the cost of a lower general throughput, which might be acceptable depending on the constraints. Another method could be to

study different ways of compressing and decompressing the images to find techniques that preserve the accuracy better, or to study methods that can be employed at the neural network training phase to help the network adapt better to lower resolutions. To our knowledge, this has not been sufficiently studied, and would probably be the most interesting topic to focus on as further research in the field.

Better implementation of transposed convolution

As mentioned in Chapter 5, the integration of the REVD2 algorithm in FINN would greatly benefit to the performance of transposed convolutions in our FPGA designs. The results obtained in this chapter, as well as the design choices of P-ENet, are influenced by our choice of implementation of transposed convolutions, and the integration of a more efficient algorithm could change the conclusions drawn in favor of the use of transposed convolutions rather than nearest-neighbor resize convolutions.

Further design space exploration

Regarding the hardware-oriented design space exploration, it would be interesting to experiment with additional design changes to see if this could lead to a more efficient implementation. First of all, replacing ReLUs with other activation layers such as hyperbolic tangent is a potential way to alleviate for the loss in mIoU introduced by the placement of activation layers before elementwise additions, and between asymmetrical convolution layers. Second, our experiments with the projection ratio exhibited that although it allowed to considerably reduce the resources usage of the network, increasing it from 4 to 8 on all blocks led to a large drop in mIoU. Therefore introducing heterogeneous configurations to evaluate if there are blocks that are less sensitive to lower projection ratios than others could be a way to further explore the trade-off offered by this parameter. Finally, grouped convolutions offer an interesting method to reduce the complexity of convolutions, and introduce an additional trade-off between accuracy and complexity through the number of groups. As such, replacing some convolutions, such as the asymmetrical ones, with grouped convolutions and experimenting with different numbers of groups would be an interesting direction to follow. Their use is currently not supported by FINN, which incites to continue contributing to the open-source tool.

Towards linear topologies

Our experiments clearly exhibited that shortcuts were often difficult to deal with for streamlined dataflow architectures, and although the local shortcuts of P-ENet or similar networks are manageable, they still induce a memory overhead. Interestingly, Ding *et al.* [171] have proposed a methodology to convert CNNs with ResNet-like topologies after training into straightforward, VGG-like structures for inference without losing accuracy. This method has been evaluated for image classification and it would be interesting to see if it could be extended to other tasks, and possibly transform the complex structures of semantic segmentation networks into completely linear ones, which would be extremely beneficial for inference on FPGA.

Hardware-oriented neural architecture search

As NAS offers a solution to automatize the exploration of the neural network design space, it would be interesting to study if additional metrics like the hardware complexity could be inserted, by, for instance, using the resources estimations of FINN as part of

the evaluation strategy. This raises questions regarding the complexity overhead in the architecture search of adding such additional elements, as well as practical concerns as certain network architectures can be difficult to streamline properly, which could skew their evaluation. Nonetheless, exploring ways to automatize the co-design of neural networks and FPGA architectures is an interesting topic that, to our knowledge, has not been studied, and would benefit from some investigations.

Bibliography

- [1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012.
- [2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, *Imagenet large scale visual recognition challenge*, 2015. arXiv: 1409.0575 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1409.0575>.
- [3] R. Lafargue, L. Smith, F. Vermet, M. Löwe, I. Reid, V. Gripon, and J. Valmadre, *Oops, i sampled it again: reinterpreting confidence intervals in few-shot learning*, 2024. arXiv: 2409.02850 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/2409.02850>.
- [4] Y. Bendou, Y. Hu, R. Lafargue, G. Lioi, B. Pasdeloup, S. Pateux, and V. Gripon, "Easy—ensemble augmented-shot-y-shaped learning: state-of-the-art few-shot classification with simple components", *Journal of Imaging*, vol. 8, no. 7, 2022, ISSN: 2313-433X. DOI: 10.3390/jimaging8070179. [Online]. Available: <https://www.mdpi.com/2313-433X/8/7/179>.
- [5] C. Lassance, V. Gripon, and A. Ortega, "Laplacian networks: bounding indicator function smoothness for neural networks robustness", *APSIPA Transactions on Signal and Information Processing*, vol. 10, e2, 2021. DOI: 10.1017/ATSIP.2021.2.
- [6] L. G. Ribeiro, M. Leonardon, G. Muller, V. Fresse, and M. Arzel, *Flocora: federated learning compression with low-rank adaptation*, 2024. arXiv: 2406.14082 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/2406.14082>.
- [7] S. Chakraborty, R. Tomsett, R. Raghavendra, D. Harborne, M. Alzantot, F. Cerutti, M. Srivastava, A. Preece, S. Julier, R. M. Rao, T. D. Kelley, D. Braines, M. Sensoy, C. J. Willis, and P. Gurram, "Interpretability of deep learning models: a survey of results", in *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, 2017, pp. 1–6. DOI: 10.1109/UIC-ATC.2017.8397411.
- [8] X. Xu, X. Zhang, B. Yu, X. S. Hu, C. Rowen, J. Hu, and Y. Shi, "Dac-sdc low power object detection challenge for uav applications", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 2, pp. 392–403, 2021. DOI: 10.1109/TPAMI.2019.2932429.

- [9] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, “Finn: a framework for fast, scalable binarized neural network inference”, in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, Monterey, California, USA: Association for Computing Machinery, 2017, pp. 65–74, ISBN: 9781450343541. DOI: 10.1145/3020078.3021744. [Online]. Available: <https://doi.org/10.1145/3020078.3021744>.
- [10] M. Blott, T. Preusser, N. Fraser, G. Gambardella, K. O'Brien, and Y. Umuroglu, *Finn-r: an end-to-end deep-learning framework for fast exploration of quantized neural networks*, 2018. arXiv: 1809.04570 [cs.AR]. [Online]. Available: <https://arxiv.org/abs/1809.04570>.
- [11] H. Younes, H. Le Blevec, M. Léonardon, and V. Gripon, “Inter-operability of compression techniques for efficient deployment of cnns on microcontrollers”, in *Advances in System-Integrated Intelligence*, M. Valle, D. Lehmhus, C. Gianoglio, E. Ragusa, L. Seminara, S. Bosse, A. Ibrahim, and K.-D. Thoben, Eds., Cham: Springer International Publishing, 2023, pp. 543–552, ISBN: 978-3-031-16281-7.
- [12] H. Le Blevec, M. Léonardon, H. Tessier, and M. Arzel, “Pipelined architecture for a semantic segmentation neural network on fpga”, in *2023 30th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2023, pp. 1–4. DOI: 10.1109/ICECS58634.2023.10382715.
- [13] H. Le Blevec, M. Léonardon, H. Tessier, and M. Arzel, “Co-conception d’un réseau de neurones convolutif de segmentation sémantique et de son architecture matérielle”, in *29° Colloque sur le traitement du signal et des images*, GRETSI - Groupe de Recherche en Traitement du Signal et des Images, Grenoble, France, Aug. 2023. [Online]. Available: <https://hal.science/hal-04705594>.
- [14] H. Le Blevec, M. Léonardon, and M. Arzel, *Revisiting enet for more efficient real-time embedded semantic segmentation on fpga*, 2024. [Online]. Available: https://github.com/Xilinx/finn/blob/github-pages/docs/finn-fpl24/FPL24_poster_enet.pdf.
- [15] J. Kiefer and J. Wolfowitz, “Stochastic Estimation of the Maximum of a Regression Function”, *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 462–466, 1952. DOI: 10.1214/aoms/1177729392. [Online]. Available: <https://doi.org/10.1214/aoms/1177729392>.
- [16] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus, “Deconvolutional networks”, in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 2528–2535. DOI: 10.1109/CVPR.2010.5539957.
- [17] A. Aitken, C. Ledig, L. Theis, J. Caballero, Z. Wang, and W. Shi, *Checkerboard artifact free sub-pixel convolution: a note on sub-pixel convolution, resize convolution and convolution resize*, 2017. arXiv: 1707.02937 [cs.CV].
- [18] J.-W. Chang and S.-J. Kang, “Optimizing fpga-based convolutional neural networks accelerator for image super-resolution”, in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 343–348. DOI: 10.1109/ASPDAC.2018.8297347.
- [19] I. Colbert, J. Daly, K. Kreutz-Delgado, and S. Das, *A competitive edge: can fpgas beat gpus at dcnn inference acceleration in resource-limited edge computing applications?*, 2021. arXiv: 2102.00294 [cs.DC]. [Online]. Available: <https://arxiv.org/abs/2102.00294>.

- [20] S. Liu, C. Zeng, H. Fan, H.-C. Ng, J. Meng, Z. Que, X. Niu, and W. Luk, "Memory-efficient architecture for accelerating generative networks on fpga", in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 30–37. DOI: 10.1109/FPT.2018.00016.
- [21] X. Zhang, S. Das, O. Neopane, and K. Kreutz-Delgado, *A design methodology for efficient implementation of deconvolutional neural networks on an fpga*, 2017. arXiv: 1705.02583 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/1705.02583>.
- [22] V. Dumoulin and F. Visin, *A guide to convolution arithmetic for deep learning*, 2018. arXiv: 1603.07285 [stat.ML]. [Online]. Available: <https://arxiv.org/abs/1603.07285>.
- [23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *Pytorch: an imperative style, high-performance deep learning library*, 2019. arXiv: 1912.01703 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/1912.01703>.
- [24] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *Tensorflow: large-scale machine learning on heterogeneous distributed systems*, 2016. arXiv: 1603.04467 [cs.DC]. [Online]. Available: <https://arxiv.org/abs/1603.04467>.
- [25] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang, *Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network*, 2016. arXiv: 1609.05158 [cs.CV].
- [26] Z. Wang, J. Chen, and S. C. H. Hoi, "Deep learning for image super-resolution: a survey", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 43, no. 10, pp. 3365–3387, 2021. DOI: 10.1109/TPAMI.2020.2982166.
- [27] C. Dong, C. C. Loy, K. He, and X. Tang, *Image super-resolution using deep convolutional networks*, 2015. arXiv: 1501.00092 [cs.CV].
- [28] S. Ioffe and C. Szegedy, *Batch normalization: accelerating deep network training by reducing internal covariate shift*, 2015. arXiv: 1502.03167 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/1502.03167>.
- [29] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)", in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.
- [30] A. Finnerty and H. Ratigner, "Reduce power and cost by converting from floating point to fixed point", *WP491 (v1. 0)*, 2017.
- [31] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, *A survey of quantization methods for efficient neural network inference*, 2021. arXiv: 2103.13630 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2103.13630>.
- [32] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, *Integer quantization for deep learning inference: principles and empirical evaluation*, 2020. arXiv: 2004.09602 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/2004.09602>.

- [33] Z. Cai, X. He, J. Sun, and N. Vasconcelos, *Deep learning with low precision by half-wave gaussian quantization*, 2017. arXiv: 1702.00953 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1702.00953>.
- [34] Y. Choi, M. El-Khamy, and J. Lee, *Towards the limit of network quantization*, 2017. arXiv: 1612.01543 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1612.01543>.
- [35] J. Faraone, N. Fraser, M. Blott, and P. H. W. Leong, *Syq: learning symmetric quantization for efficient deep neural networks*, 2018. arXiv: 1807.00301 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1807.00301>.
- [36] Y. Gong, L. Liu, M. Yang, and L. Bourdev, *Compressing deep convolutional networks using vector quantization*, 2014. arXiv: 1412.6115 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1412.6115>.
- [37] C. Baskin, N. Liss, E. Schwartz, E. Zheltonozhskii, R. Giryes, A. M. Bronstein, and A. Mendelson, "Uniq: uniform noise injection for non-uniform quantization of neural networks", *ACM Transactions on Computer Systems*, vol. 37, no. 1–4, pp. 1–15, Nov. 2019, ISSN: 1557-7333. DOI: 10.1145/3444943. [Online]. Available: <http://dx.doi.org/10.1145/3444943>.
- [38] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, K. Gopalakrishnan, and L. Chang, "Biscaled-dnn: quantizing long-tailed datastructures with two scale factors for deep neural networks", in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19, Las Vegas, NV, USA: Association for Computing Machinery, 2019, ISBN: 9781450367257. DOI: 10.1145/3316781.3317783. [Online]. Available: <https://doi.org/10.1145/3316781.3317783>.
- [39] M. Courbariaux, Y. Bengio, and J.-P. David, *Binaryconnect: training deep neural networks with binary weights during propagations*, 2016. arXiv: 1511.00363 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/1511.00363>.
- [40] Y. Bengio, N. Léonard, and A. Courville, *Estimating or propagating gradients through stochastic neurons for conditional computation*, 2013. arXiv: 1308.3432 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/1308.3432>.
- [41] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, *Pact: parameterized clipping activation for quantized neural networks*, 2018. arXiv: 1805.06085 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1805.06085>.
- [42] S. Jung, C. Son, S. Lee, J. Son, Y. Kwak, J.-J. Han, S. J. Hwang, and C. Choi, *Learning to quantize deep networks by optimizing quantization intervals with task loss*, 2018. arXiv: 1808.05779 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1808.05779>.
- [43] S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha, *Learned step size quantization*, 2020. arXiv: 1902.08153 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/1902.08153>.
- [44] Y. Bhalgat, J. Lee, M. Nagel, T. Blankevoort, and N. Kwak, *Lsq+: improving low-bit quantization through learnable offsets and better initialization*, 2020. arXiv: 2004.09576 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2004.09576>.
- [45] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, *Mobilenets: efficient convolutional neural networks for mobile vision applications*, 2017. arXiv: 1704.04861 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1704.04861>.

- [46] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, *Aggregated residual transformations for deep neural networks*, 2017. arXiv: 1611.05431 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1611.05431>.
- [47] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, 2015. arXiv: 1512.03385 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1512.03385>.
- [48] G. Hinton, O. Vinyals, and J. Dean, *Distilling the knowledge in a neural network*, 2015. arXiv: 1503.02531 [stat.ML]. [Online]. Available: <https://arxiv.org/abs/1503.02531>.
- [49] X. Ma, S. Lin, S. Ye, Z. He, L. Zhang, G. Yuan, S. H. Tan, Z. Li, D. Fan, X. Qian, X. Lin, K. Ma, and Y. Wang, *Non-structured dnn weight pruning – is it beneficial in any platform?*, 2020. arXiv: 1907.02124 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/1907.02124>.
- [50] X. He, S. Pal, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, R. Dreslinski, and T. Mudge, “Sparse-tpu: adapting systolic arrays for sparse matrices”, in *Proceedings of the 34th ACM international conference on supercomputing*, 2020, pp. 1–12.
- [51] H. Tessier, G. B. Hacene, and V. Gripon, *Thinresnet: a new baseline for structured convolutional networks pruning*, 2023. arXiv: 2309.12854 [cs.NE]. [Online]. Available: <https://arxiv.org/abs/2309.12854>.
- [52] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, *Rethinking the value of network pruning*, 2019. arXiv: 1810.05270 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/1810.05270>.
- [53] T. Gale, E. Elsen, and S. Hooker, *The state of sparsity in deep neural networks*, 2019. arXiv: 1902.09574 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/1902.09574>.
- [54] C. White, M. Safari, R. Sukthankar, B. Ru, T. Elsken, A. Zela, D. Dey, and F. Hutter, *Neural architecture search: insights from 1000 papers*, 2023. arXiv: 2301.08727 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/2301.08727>.
- [55] J. Long, E. Shelhamer, and T. Darrell, *Fully convolutional networks for semantic segmentation*, 2015. arXiv: 1411.4038 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1411.4038>.
- [56] H. Zhang, A. Geiger, and R. Urtasun, “Understanding high-level semantics by modeling traffic patterns”, in *2013 IEEE International Conference on Computer Vision*, 2013, pp. 3056–3063. DOI: 10.1109/ICCV.2013.379.
- [57] A. Geiger, M. Lauer, C. Wojek, C. Stiller, and R. Urtasun, “3d traffic scene understanding from movable platforms”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 5, pp. 1012–1025, 2014. DOI: 10.1109/TPAMI.2013.185.
- [58] L. Bartolomei, L. Teixeira, and M. Chli, “Perception-aware path planning for uavs using semantic segmentation”, in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 5808–5815. DOI: 10.1109/IROS45743.2020.9341347.
- [59] M. Hua, Y. Nan, and S. Lian, “Small obstacle avoidance based on rgb-d semantic segmentation”, in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, 2019, pp. 886–894. DOI: 10.1109/ICCVW.2019.00117.

- [60] H.-Y. Luo, G. Xu, L. Chaofeng, L. He, L. Luo, Z. Wang, B. Jing, Y. Deng, Y. Jin, Y. Li, B. Li, W. Tan, C. He, S. Seeruttun, Q. Wu, J. Huang, D.-w. Huang, B. Chen, S.-b. Lin, and R.-h. Xu, "Real-time artificial intelligence for detection of upper gastrointestinal cancer by endoscopy: a multicentre, case-control, diagnostic study", *The Lancet Oncology*, vol. 20, Oct. 2019. DOI: 10.1016/S1470-2045(19)30637-0.
- [61] P. An, D. Yang, J. Wang, L. Wu, J. Zhou, Z. Zeng, X. Huang, Y. Xiao, S. Hu, Y. Chen, F. Yao, M. Guo, Q. Wu, Y. Yang, and H. Yu, "A deep learning method for delineating early gastric cancer resection margin under chromoendoscopy and white light endoscopy", *Gastric Cancer*, vol. 23, Apr. 2020. DOI: 10.1007/s10120-020-01071-7.
- [62] G. Piantadosi, M. Sansone, and C. Sansone, "Breast segmentation in mri via u-net deep convolutional neural networks", Aug. 2018, pp. 3917–3922. DOI: 10.1109/ICPR.2018.8545327.
- [63] L. Cai, J. Gao, and D. Zhao, "A review of the application of deep learning in medical image classification and segmentation", *Annals of Translational Medicine*, vol. 8, pp. 713–713, Jun. 2020. DOI: 10.21037/atm.2020.02.44.
- [64] K. m, M. Kampffmeyer, and R. Jenssen, "Uncertainty and interpretability in convolutional neural networks for semantic segmentation of colorectal polyps", *Medical Image Analysis*, vol. 60, p. 101619, Nov. 2019. DOI: 10.1016/j.media.2019.101619.
- [65] A. Milioto, P. Lottes, and C. Stachniss, "Real-time semantic segmentation of crop and weed for precision agriculture robots leveraging background knowledge in cnns", in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 2229–2235. DOI: 10.1109/ICRA.2018.8460962.
- [66] J. Jadhav and R. Singh, "Automatic semantic segmentation and classification of remote sensing data for agriculture", *Mathematical Models in Engineering*, vol. 4, pp. 112–137, Jun. 2018. DOI: 10.21595/mme.2018.19840.
- [67] C. Dechesne, C. Mallet, A. Le Bris, and V. Gouet-Brunet, "Semantic segmentation of forest stands of pure species as a global optimization problem", *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. IV-1/W1, pp. 141–148, May 2017. DOI: 10.5194/isprs-annals-IV-1-W1-141-2017.
- [68] C. Dechesne, C. Mallet, A. Le Bris, and V. Gouet-Brunet, "Semantic segmentation of forest stands of pure species as a global optimization problem", *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. IV-1/W1, pp. 141–148, May 2017. DOI: 10.5194/isprs-annals-IV-1-W1-141-2017.
- [69] T. Anand, S. Sinha, M. Mandal, V. Chamola, and F. R. Yu, "Agrisegnet: deep aerial semantic segmentation framework for iot-assisted precision agriculture", *IEEE Sensors Journal*, vol. 21, no. 16, pp. 17581–17590, 2021. DOI: 10.1109/JSEN.2021.3071290.
- [70] R. Hasimoto-Beltran, M. Canul-Ku, G. M. Díaz Méndez, F. J. Ocampo-Torres, and B. Esquivel-Trava, "Ocean oil spill detection from sar images based on multi-channel deep learning semantic segmentation", *Marine Pollution Bulletin*, vol. 188, p. 114651, 2023, ISSN: 0025-326X. DOI: <https://doi.org/10.1016/j.marpolbul.2023.114651>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0025326X23000826>.

- [71] M. M. dos Santos, G. G. De Giacomo, P. L. J. Drews, and S. S. Botelho, "Semantic segmentation of static and dynamic structures of marina satellite images using deep learning", in *2019 8th Brazilian Conference on Intelligent Systems (BRACIS)*, 2019, pp. 711–716. DOI: 10.1109/BRACIS.2019.00129.
- [72] S. D. Khan, L. Alarabi, and S. Basalamah, "Deep hybrid network for land cover semantic segmentation in high-spatial resolution satellite images", *Information*, vol. 12, no. 6, 2021, ISSN: 2078-2489. [Online]. Available: <https://www.mdpi.com/2078-2489/12/6/230>.
- [73] M. Wurm, T. Stark, X. X. Zhu, M. Weigand, and H. Taubenböck, "Semantic segmentation of slums in satellite images using transfer learning on fully convolutional neural networks", *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 150, pp. 59–69, 2019, ISSN: 0924-2716. DOI: <https://doi.org/10.1016/j.isprsjprs.2019.02.006>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0924271619300383>.
- [74] D. Hordiiuk, I. Oliinyk, V. Hnatushenko, and K. Maksymov, "Semantic segmentation for ships detection from satellite imagery", in *2019 IEEE 39th International Conference on Electronics and Nanotechnology (ELNANO)*, 2019, pp. 454–457. DOI: 10.1109/ELNANO.2019.8783822.
- [75] D. Teso-Fz-Betoño, E. Zulueta, A. Sánchez-Chica, U. Fernandez-Gamiz, and A. Saenz-Aguirre, "Semantic segmentation to develop an indoor navigation system for an autonomous mobile robot", *Mathematics*, vol. 8, no. 5, 2020, ISSN: 2227-7390. DOI: 10.3390/math8050855. [Online]. Available: <https://www.mdpi.com/2227-7390/8/5/855>.
- [76] Q. Yan, S. Li, C. Liu, M. Liu, and Q. Chen, "Roboseg: real-time semantic segmentation on computationally constrained robots", *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 52, no. 3, pp. 1567–1577, 2022. DOI: 10.1109/TSMC.2020.3032437.
- [77] W. Kim and J. Seok, "Indoor semantic segmentation for robot navigating on mobile", in *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2018, pp. 22–25. DOI: 10.1109/ICUFN.2018.8436956.
- [78] D. Wolf, J. Prankl, and M. Vincze, "Enhancing semantic segmentation for robotics: the power of 3-d entangled forests", *IEEE Robotics and Automation Letters*, vol. 1, no. 1, pp. 49–56, 2016. DOI: 10.1109/LRA.2015.2506118.
- [79] B. Zhou, H. Zhao, X. Puig, S. Fidler, A. Barriuso, and A. Torralba, "Scene parsing through ade20k dataset", in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5122–5130. DOI: 10.1109/CVPR.2017.544.
- [80] Z. Dong, K. Xu, Y. Yang, H. Bao, W. Xu, and R. Lau, "Location-aware single image reflection removal", Oct. 2021, pp. 4997–5006. DOI: 10.1109/ICCV48922.2021.00497.
- [81] S. Song, S. P. Lichtenberg, and J. Xiao, "Sun rgb-d: a rgb-d scene understanding benchmark suite", in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 567–576. DOI: 10.1109/CVPR.2015.7298655.
- [82] H. Caesar, J. Uijlings, and V. Ferrari, "Coco-stuff: thing and stuff classes in context", in *Computer vision and pattern recognition (CVPR), 2018 IEEE conference on*, IEEE, 2018.

- [83] G. J. Brostow, J. Fauqueur, and R. Cipolla, "Semantic object classes in video: a high-definition ground truth database", *Pattern Recognition Letters*, vol. 30, no. 2, pp. 88–97, 2009, Video-based Object and Event Analysis, ISSN: 0167-8655. DOI: <https://doi.org/10.1016/j.patrec.2008.04.005>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167865508001220>.
- [84] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding", in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 3213–3223. DOI: 10.1109/CVPR.2016.350.
- [85] O. Ronneberger, P. Fischer, and T. Brox, *U-net: convolutional networks for biomedical image segmentation*, 2015. arXiv: 1505.04597 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1505.04597>.
- [86] V. Badrinarayanan, A. Kendall, and R. Cipolla, *Segnet: a deep convolutional encoder-decoder architecture for image segmentation*, 2016. arXiv: 1511.00561 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1511.00561>.
- [87] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, *Deeplab: semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs*, 2017. arXiv: 1606.00915 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1606.00915>.
- [88] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, *Rethinking atrous convolution for semantic image segmentation*, 2017. arXiv: 1706.05587 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1706.05587>.
- [89] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation", in *Proceedings of the European Conference on Computer Vision (ECCV)*, Sep. 2018.
- [90] J. Wang, K. Sun, T. Cheng, B. Jiang, C. Deng, Y. Zhao, D. Liu, Y. Mu, M. Tan, X. Wang, W. Liu, and B. Xiao, *Deep high-resolution representation learning for visual recognition*, 2020. arXiv: 1908.07919 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1908.07919>.
- [91] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, *An image is worth 16x16 words: transformers for image recognition at scale*, 2021. arXiv: 2010.11929 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2010.11929>.
- [92] S. Zheng, J. Lu, H. Zhao, X. Zhu, Z. Luo, Y. Wang, Y. Fu, J. Feng, T. Xiang, P. H. Torr, and L. Zhang, "Rethinking semantic segmentation from a sequence-to-sequence perspective with transformers", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2021, pp. 6881–6890.
- [93] W. Wang, J. Dai, Z. Chen, Z. Huang, Z. Li, X. Zhu, X. Hu, T. Lu, L. Lu, H. Li, X. Wang, and Y. Qiao, "Internimage: exploring large-scale vision foundation models with deformable convolutions", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2023, pp. 14 408–14 419.
- [94] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo, P. Dollár, and R. Girshick, *Segment anything*, 2023. arXiv: 2304.02643 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2304.02643>.

- [95] D. C. Phillips, "Programming Real-Time Computer Systems", *The Computer Journal*, vol. 9, no. 3, pp. 249–249, Nov. 1966, ISSN: 0010-4620. DOI: 10.1093/comjnl/9.3.249. eprint: <https://academic.oup.com/comjnl/article-pdf/9/3/249/1133721/9-3-249.pdf>. [Online]. Available: <https://doi.org/10.1093/comjnl/9.3.249>.
- [96] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, *Enet: a deep neural network architecture for real-time semantic segmentation*, 2016. arXiv: 1606.02147 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1606.02147>.
- [97] E. Romera, J. M. Álvarez, L. M. Bergasa, and R. Arroyo, "Erfnet: efficient residual factorized convnet for real-time semantic segmentation", *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 1, pp. 263–272, 2018. DOI: 10.1109/TITS.2017.2750080.
- [98] A. Chaurasia and E. Culurciello, "Linknet: exploiting encoder representations for efficient semantic segmentation", in *2017 IEEE Visual Communications and Image Processing (VCIP)*, IEEE, Dec. 2017. DOI: 10.1109/vcip.2017.8305148. [Online]. Available: <http://dx.doi.org/10.1109/VCIP.2017.8305148>.
- [99] S. Mehta, M. Rastegari, A. Caspi, L. Shapiro, and H. Hajishirzi, *Espnet: efficient spatial pyramid of dilated convolutions for semantic segmentation*, 2018. arXiv: 1803.06815 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1803.06815>.
- [100] G. Li, I. Yun, J. Kim, and J. Kim, *Dabnet: depth-wise asymmetric bottleneck for real-time semantic segmentation*, 2019. arXiv: 1907.11357 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1907.11357>.
- [101] S.-Y. Lo, H.-M. Hang, S.-W. Chan, and J.-J. Lin, *Efficient dense modules of asymmetric convolution for real-time semantic segmentation*, 2019. arXiv: 1809.06323 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1809.06323>.
- [102] R. P. K. Poudel, U. Bonde, S. Liwicki, and C. Zach, *Contextnet: exploring context and detail for semantic segmentation in real-time*, 2018. arXiv: 1805.04554 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1805.04554>.
- [103] J. Zhuang, J. Yang, L. Gu, and N. Dvornek, *Shelfnet for fast semantic segmentation*, 2019. arXiv: 1811.11254 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1811.11254>.
- [104] M. Oršić, I. Krešo, P. Bevandić, and S. Šegvić, *In defense of pre-trained imagenet architectures for real-time semantic segmentation of road-driving images*, 2019. arXiv: 1903.08469 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1903.08469>.
- [105] Y. Wang, Q. Zhou, and X. Wu, *Esnet: an efficient symmetric network for real-time semantic segmentation*, 2019. arXiv: 1906.09826 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1906.09826>.
- [106] R. Gao, *Rethinking dilated convolution for real-time semantic segmentation*, 2023. arXiv: 2111.09957 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2111.09957>.
- [107] C. Yu, J. Wang, C. Peng, C. Gao, G. Yu, and N. Sang, *Bisenet: bilateral segmentation network for real-time semantic segmentation*, 2018. arXiv: 1808.00897 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1808.00897>.
- [108] H. Zhao, X. Qi, X. Shen, J. Shi, and J. Jia, *ICNet for real-time semantic segmentation on high-resolution images*, 2018. arXiv: 1704.08545 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1704.08545>.

- [109] C. Yu, C. Gao, J. Wang, G. Yu, C. Shen, and N. Sang, *Bisenet v2: bilateral network with guided aggregation for real-time semantic segmentation*, 2020. arXiv: 2004.02147 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2004.02147>.
- [110] X. Li, A. You, Z. Zhu, H. Zhao, M. Yang, K. Yang, and Y. Tong, *Semantic flow for fast and accurate scene parsing*, 2021. arXiv: 2002.10120 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2002.10120>.
- [111] Y. Hong, H. Pan, W. Sun, and Y. Jia, *Deep dual-resolution networks for real-time and accurate semantic segmentation of road scenes*, 2021. arXiv: 2101.06085 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2101.06085>.
- [112] J. Xu, Z. Xiong, and S. P. Bhattacharyya, *Pidnet: a real-time semantic segmentation network inspired by pid controllers*, 2023. arXiv: 2206.02066 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2206.02066>.
- [113] H. Li, P. Xiong, H. Fan, and J. Sun, *Dfanet: deep feature aggregation for real-time semantic segmentation*, 2019. arXiv: 1904.02216 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1904.02216>.
- [114] Y. Wang, Q. Zhou, J. Liu, J. Xiong, G. Gao, X. Wu, and L. J. Latecki, *Lednet: a lightweight encoder-decoder network for real-time semantic segmentation*, 2019. arXiv: 1905.02423 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1905.02423>.
- [115] P. Hu, F. Perazzi, F. C. Heilbron, O. Wang, Z. Lin, K. Saenko, and S. Sclaroff, *Real-time semantic segmentation with fast attention*, 2020. arXiv: 2007.03815 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2007.03815>.
- [116] J. Peng, Y. Liu, S. Tang, Y. Hao, L. Chu, G. Chen, Z. Wu, Z. Chen, Z. Yu, Y. Du, Q. Dang, B. Lai, Q. Liu, X. Hu, D. Yu, and Y. Ma, *Pp-liteseg: a superior real-time semantic segmentation model*, 2022. arXiv: 2204.02681 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2204.02681>.
- [117] W. Chen, X. Gong, X. Liu, Q. Zhang, Y. Li, and Z. Wang, *Fasterseg: searching for faster real-time semantic segmentation*, 2020. arXiv: 1912.10917 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1912.10917>.
- [118] Y. Nirkin, L. Wolf, and T. Hassner, *Hyperseg: patch-wise hypernetwork for real-time semantic segmentation*, 2021. arXiv: 2012.11582 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2012.11582>.
- [119] Y. Li, B. Sixou, and F. Peyrin, "A review of the deep learning methods for medical images super resolution problems", *IRBM*, vol. 42, no. 2, pp. 120–133, 2021, ISSN: 1959-0318. DOI: <https://doi.org/10.1016/j.irbm.2020.08.004>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1959031820301408>.
- [120] J. Cornebise, I. Oršolić, and F. Kalaitzis, "Open high-resolution satellite imagery: the worldstrat dataset –with application to super-resolution", in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., 2022, pp. 25979–25991. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/a6fe99561d9eb9c90b322afe664587fd-Paper-Datasets_and_Benchmarks.pdf.

- [121] J. Su, B. Xu, and H. Yin, "A survey of deep learning approaches to image restoration", *Neurocomputing*, vol. 487, pp. 46–65, 2022, ISSN: 0925-2312. DOI: <https://doi.org/10.1016/j.neucom.2022.02.046>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0925231222002089>.
- [122] S. I. Venieris, M. Almeida, R. Lee, and N. D. Lane, "Nawq-sr: a hybrid-precision npu engine for efficient on-device super-resolution", *IEEE Transactions on Mobile Computing*, vol. 23, no. 3, pp. 2367–2381, 2024. DOI: 10.1109/TMC.2023.3255822.
- [123] D. Martin, C. Fowlkes, D. Tal, and J. Malik, "A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics", in *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, vol. 2, 2001, 416–423 vol.2. DOI: 10.1109/ICCV.2001.937655.
- [124] J. Kim, J. K. Lee, and K. M. Lee, "Accurate image super-resolution using very deep convolutional networks", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016.
- [125] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, *Generative adversarial networks*, 2014. arXiv: 1406.2661 [stat.ML]. [Online]. Available: <https://arxiv.org/abs/1406.2661>.
- [126] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi, "Photo-realistic single image super-resolution using a generative adversarial network", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017.
- [127] X. Wang, K. Yu, S. Wu, J. Gu, Y. Liu, C. Dong, C. C. Loy, Y. Qiao, and X. Tang, *Esrgan: enhanced super-resolution generative adversarial networks*, 2018. arXiv: 1809.00219 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1809.00219>.
- [128] Y. Zhang, K. Li, K. Li, L. Wang, B. Zhong, and Y. Fu, *Image super-resolution using very deep residual channel attention networks*, 2018. arXiv: 1807.02758 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1807.02758>.
- [129] C.-C. Hsu, C.-M. Lee, and Y.-S. Chou, *Drct: saving image super-resolution away from information bottleneck*, 2024. arXiv: 2404.00722 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2404.00722>.
- [130] X. Chen, X. Wang, J. Zhou, Y. Qiao, and C. Dong, "Activating more pixels in image super-resolution transformer", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2023, pp. 22 367–22 377.
- [131] C. Korkmaz and A. M. Tekalp, *Training transformer models by wavelet losses improves quantitative and visual performance in single image super-resolution*, 2024. arXiv: 2404.11273 [eess.IV]. [Online]. Available: <https://arxiv.org/abs/2404.11273>.
- [132] M. Blott, L. Halder, M. Leeser, and L. Doyle, "Qutibench: benchmarking neural networks on heterogeneous hardware", *J. Emerg. Technol. Comput. Syst.*, vol. 15, no. 4, Dec. 2019, ISSN: 1550-4832. DOI: 10.1145/3358700. [Online]. Available: <https://doi.org/10.1145/3358700>.
- [133] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going deeper with convolutions*, 2014. arXiv: 1409.4842 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1409.4842>.

- [134] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks", in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16, Monterey, California, USA: Association for Computing Machinery, 2016, pp. 16–25, ISBN: 9781450338561. DOI: 10.1145/2847263.2847276. [Online]. Available: <https://doi.org/10.1145/2847263.2847276>.
- [135] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network", in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16, Monterey, California, USA: Association for Computing Machinery, 2016, pp. 26–35, ISBN: 9781450338561. DOI: 10.1145/2847263.2847265. [Online]. Available: <https://doi.org/10.1145/2847263.2847265>.
- [136] A. Montgomerie-Corcoran, P. Toupas, Z. Yu, and C.-S. Bouganis, *Satay: a streaming architecture toolflow for accelerating yolo models on fpga devices*, 2023. arXiv: 2309.01587 [cs.AR]. [Online]. Available: <https://arxiv.org/abs/2309.01587>.
- [137] *Nvidia tensorrt*, Accessed: 2024-09-25. [Online]. Available: <https://developer.nvidia.com/tensorrt>.
- [138] S. Y. Kung, *VLSI array processors*. USA: Prentice-Hall, Inc., 1987, ISBN: 013942749X.
- [139] P. Quinton and Y. Robert, *Algorithmes et architectures systoliques* (Etudes et Recherches en Informatique). Masson, 1989, 353 p. English translation: *Systolic algorithms and architectures*, Prentice Hall (1991). [Online]. Available: <https://inria.hal.science/hal-00857013>.
- [140] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: enabling systematic deep-learning architecture evaluation via full-stack integration", in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.
- [141] J. Duarte *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics", *JINST*, vol. 13, no. 07, P07027, 2018. DOI: 10.1088/1748-0221/13/07/P07027. arXiv: 1804.06913 [physics.ins-det].
- [142] S. I. Venieris and C.-S. Bouganis, "Fpgaconvnet: a framework for mapping convolutional neural networks on fpgas", in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 40–47. DOI: 10.1109/FCCM.2016.22.
- [143] M. Shimoda, Y. Sada, and H. Nakahara, "Filter-wise pruning approach to fpga implementation of fully convolutional network for semantic segmentation", in *Applied Reconfigurable Computing*, C. Hochberger, B. Nelson, A. Koch, R. Woods, and P. Diniz, Eds., Cham: Springer International Publishing, 2019, pp. 371–386, ISBN: 978-3-030-17227-5.
- [144] M. Yu, H. Huang, H. Liu, S. He, F. Qiao, L. Luo, F. Xie, X.-J. Liu, and H. Yang, "Optimizing fpga-based convolutional encoder-decoder architecture for semantic segmentation", in *2019 IEEE 9th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*, 2019, pp. 1436–1440. DOI: 10.1109/CYBER46603.2019.9066759.

- [145] M. Miyama, "Fpga implementation of 3-bit quantized cnn for semantic segmentation", *Journal of Physics: Conference Series*, vol. 1729, no. 1, p. 012004, Jan. 2021. DOI: 10.1088/1742-6596/1729/1/012004. [Online]. Available: <https://dx.doi.org/10.1088/1742-6596/1729/1/012004>.
- [146] H. Kim, J. Kim, J. Choi, J. Lee, and Y. H. Song, "Binarized encoder-decoder network and binarized deconvolution engine for semantic segmentation", *IEEE Access*, vol. 9, pp. 8006–8027, 2021. DOI: 10.1109/ACCESS.2020.3048375.
- [147] R. P. K. Poudel, S. Liwicki, and R. Cipolla, *Fast-scnn: fast semantic segmentation network*, 2019. arXiv: 1902.04502 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1902.04502>.
- [148] N. Ghielmetti, V. Loncar, M. Pierini, M. Roed, S. Summers, T. Aarrestad, C. Petersson, H. Linander, J. Ngadiuba, K. Lin, and P. Harris, *Real-time semantic segmentation on fpgas for autonomous vehicles with hls4ml*, Nov. 2022. DOI: 10.1088/2632-2153/ac9cb5. [Online]. Available: <https://dx.doi.org/10.1088/2632-2153/ac9cb5>.
- [149] W. Jia, J. Cui, X. Zheng, and Q. Wu, "Design and Implementation of Real-Time Semantic Segmentation Network Based on FPGA", in *2021 7th International Conference on Computing and Artificial Intelligence*, ser. ICCAI 2021, 2021, pp. 321–325, ISBN: 9781450389501. DOI: 10.1145/3467707.3467756.
- [150] P. Mori, M.-R. Vemparala, N. Fasfous, S. Mitra, S. Sarkar, A. Frickenstein, L. Frickenstein, D. Helms, N. S. Nagaraja, W. Stechele, and C. Passerone, "Accelerating and pruning cnns for semantic segmentation on fpga", in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22, San Francisco, California: Association for Computing Machinery, 2022, pp. 145–150, ISBN: 9781450391429. DOI: 10.1145/3489517.3530424. [Online]. Available: <https://doi.org/10.1145/3489517.3530424>.
- [151] S. Fang, L. Tian, J. Wang, S. Liang, D. Xie, Z. Chen, L. Sui, Q. Yu, X. Sun, Y. Shan, and Y. Wang, "Real-time object detection and semantic segmentation hardware system with deep learning networks", in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 389–392. DOI: 10.1109/FPT.2018.00081.
- [152] Y. Sada, M. Shimoda, A. Jinguji, and H. Nakahara, "A dataflow pipelining architecture for tile segmentation with a sparse mobilenet on an fpga", in *2019 International Conference on Field-Programmable Technology (ICFPT)*, Dec. 2019, pp. 267–270. DOI: 10.1109/ICFPT47387.2019.00044.
- [153] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, *Feature pyramid networks for object detection*, 2017. arXiv: 1612.03144 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1612.03144>.
- [154] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, *Pyramid scene parsing network*, 2017. arXiv: 1612.01105 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1612.01105>.
- [155] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing", in *Tenth International Workshop on Frontiers in Handwriting Recognition*, G. Lorette, Ed., <http://www.suvisoft.com>, Université de Rennes 1, La Baule (France): Suvisoft, Oct. 2006. [Online]. Available: <https://inria.hal.science/inria-00112631>.
- [156] Y. Umuroglu, H. Borrás, V. Loncar, S. Summers, and J. Duarte, *Fastmachinelearning/qonnx*, Jun. 2022. DOI: 10.5281/zenodo.7622236. [Online]. Available: <https://github.com/fastmachinelearning/qonnx>.

- [157] A. Pappalardo, Y. Umuroglu, M. Blott, J. Mitrevski, B. Hawks, N. Tran, V. Loncar, S. Summers, H. Borrás, J. Muhizi, M. Trahms, S.-C. H. Hsu, S. Hauck, and J. Duarte, “QONNX: Representing Arbitrary-Precision Quantized Neural Networks”, in *4th Workshop on Accelerated Machine Learning (AccML) at HiPEAC 2022 Conference*, Jun. 2022. arXiv: 2206.07527 [cs.LG]. [Online]. Available: [https://accml.dcs.gla.ac.uk/papers/2022/4thAccML_paper_1\(12\).pdf](https://accml.dcs.gla.ac.uk/papers/2022/4thAccML_paper_1(12).pdf).
- [158] Y. Umuroglu and M. Jahre, *Streamlined deployment for quantized neural networks*, 2018. arXiv: 1709.04060 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1709.04060>.
- [159] L. Petrica, T. Alonso, M. Kroes, N. Fraser, S. Cotozana, and M. Blott, *Memory-efficient dataflow inference for deep cnns on fpga*, 2020. arXiv: 2011.07317 [cs.AR]. [Online]. Available: <https://arxiv.org/abs/2011.07317>.
- [160] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: a large-scale hierarchical image database”, in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.
- [161] A. Pappalardo, *Xilinx/brevitas*, 2023. DOI: 10.5281/zenodo.3333552. [Online]. Available: <https://doi.org/10.5281/zenodo.3333552>.
- [162] W. Tang, G. Hua, and L. Wang, “How to train a compact binary neural network with high accuracy?”, *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, Feb. 2017. DOI: 10.1609/aaai.v31i1.10862. [Online]. Available: <https://ojs.aaai.org/index.php/AAAI/article/view/10862>.
- [163] M. Blott, N. J. Fraser, G. Gambardella, L. Halder, J. Kath, Z. Neveu, Y. Umuroglu, A. Vasilciuc, M. Leeser, and L. Doyle, “Evaluation of Optimized CNNs on Heterogeneous Accelerators Using a Novel Benchmarking Approach”, *IEEE Transactions on Computers*, vol. 70, no. 10, pp. 1654–1669, 2021. DOI: 10.1109/TC.2020.3022318.
- [164] E. D. Cubuk, B. Zoph, J. Shlens, and Q. V. Le, *Randaugment: practical automated data augmentation with a reduced search space*, 2019. arXiv: 1909.13719 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1909.13719>.
- [165] Y. Zhu, K. Sapra, F. A. Reda, K. J. Shih, S. Newsam, A. Tao, and B. Catanzaro, *Improving semantic segmentation via video propagation and label relaxation*, 2019. arXiv: 1812.01593 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/1812.01593>.
- [166] R. Wightman, H. Touvron, and H. Jegou, “Resnet strikes back: an improved training procedure in timm”, in *NeurIPS 2021 Workshop on ImageNet: Past, Present, and Future*, 2021. [Online]. Available: <https://openreview.net/forum?id=NG6MJnVl6M5>.
- [167] I. Colbert, K. Kreutz-Delgado, and S. Das, “An energy-efficient edge computing paradigm for convolution-based image upsampling”, *IEEE Access*, vol. 9, pp. 147 967–147 984, 2021. DOI: 10.1109/ACCESS.2021.3123938.
- [168] A. Odena, V. Dumoulin, and C. Olah, “Deconvolution and checkerboard artifacts”, *Distill*, 2016. [Online]. Available: <http://distill.pub/2016/deconv-checkerboard/>.
- [169] X. Zhang, S. Das, O. Neopane, and K. Kreutz-Delgado, *A design methodology for efficient implementation of deconvolutional neural networks on an fpga*, 2017. arXiv: 1705.02583 [cs.LG]. [Online]. Available: <https://arxiv.org/abs/1705.02583>.

- [170] F. Jentzsch, Y. Umuroglu, A. Pappalardo, M. Blott, and M. Platzner, “Radioml meets finn: enabling future rf applications with fpga streaming architectures”, *IEEE Micro*, vol. 42, no. 6, pp. 125–133, 2022. DOI: 10.1109/MM.2022.3202091.
- [171] X. Ding, X. Zhang, N. Ma, J. Han, G. Ding, and J. Sun, *Repvgg: making vgg-style convnets great again*, 2021. arXiv: 2101.03697 [cs.CV]. [Online]. Available: <https://arxiv.org/abs/2101.03697>.

Titre : Conception conjointe de réseaux de neurones profonds et d'architectures FPGA flux de données pour la segmentation sémantique dans les véhicules autonomes.

Mot clés : apprentissage profond, segmentation sémantique, véhicules autonomes, FPGA, co-conception

Résumé : Cette thèse examine le déploiement de modèles d'apprentissage profond, notamment de segmentation sémantique, sur des circuits reconfigurables (FPGAs), qui offrent des solutions efficaces en termes d'énergie et de faible latence pour des systèmes comme les véhicules autonomes ou les drones. Bien que les FPGAs soient avantageux pour ces applications, leurs ressources limitées en calcul et mémoire posent des défis. Les travaux présentés répondent à ces défis en s'intéressant à la conception conjointe de réseaux de neurones et d'architectures FPGA flux de données. Deux modèles sont déployés, ResNet18-UNet et ENet, étant opti-

misés pour l'inférence en temps réel. P-ENet, une version améliorée d'ENet grâce à une exploration d'architectures de réseaux de neurones, atteint des performances à l'état de l'art avec 70.3% de mIoU sur le jeu de données Cityscapes, 226 FPS, et une latence de 4,2 ms, démontrant que la co-conception des réseaux et du matériel offre de meilleurs compromis entre précision et performances matérielles que la simple compression. Les mesures comparatives effectuées indiquent que cette architecture sur FPGA est 1,15 fois plus efficace énergétiquement qu'un GPU embarqué.

Title: Joint design of deep neural networks and FPGA dataflow architectures for semantic segmentation in autonomous vehicles.

Keywords: deep learning, semantic segmentation, autonomous vehicles, FPGA, co-design

Abstract: This thesis explores the deployment of deep learning models, particularly semantic segmentation, on Field-Programmable Gate Arrays (FPGAs), which offer power-efficient, low-latency solutions for resource-constrained systems like autonomous vehicles and drones. While FPGAs have advantages for these applications, they present challenges due to limited computational and memory resources. This research addresses these challenges by focusing on the joint design of neural networks and dataflow architectures on FPGA. Two encoder-decoder models, ResNet18-UNet and ENet, were imple-

mented, and optimized for real-time inference. P-ENet an optimized version of ENet thanks to neural network architecture design space exploration, achieves state-of-the-art performance with 70.3% mIoU on the Cityscapes dataset, 226 FPS, and 4.2 ms latency. The study concludes that co-designing neural networks and hardware yields better trade-offs between accuracy and hardware performance than compression techniques alone. Power comparisons show this FPGA design offers 1.15 times better power efficiency than an embedded GPU.