High-level approach for the automatic generation of optimized hardware accelerators for deep neural networks

Nermine Ali

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Université Bretagne Sud

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE BRETAGNE SUD

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Electronique*

Par
Nermine ALI

High-level approach for the automatic generation of optimized hardware accelerators for deep neural networks

Thèse présentée et soutenue à CEA LIST, Nano-Innov, le 21 Mars 2022
Unité de recherche : CEA List, Saclay
UBS - Lab-STICC, Lorient
Thèse N° : 623

Rapporteurs avant soutenance :

Alix MUNIER — Professeur des Université Sorbonne Université, LIP6
François BERRY — Professeur des Université Université Clermond Auvergne, Institut Pascal

Composition du Jury :

Président :	Olivier SENTIEYS	Professeur des Université de Rennes 1, IRISA
Examinateurs :	Bernard GIRAU	Professeur des Université de Lorraine, Loria
Dir. de thèse :	Philippe COUSSY	Professeur des Université Bretagne Sud, Lab-STICC
Encadrants de thèse :	Jean-Marc PHILIPPE	Docteur Ingénieur CEA List/Thales

Invité(s) :

Benoit TAIN — Ingénieur chercheur CEA List (Co-encadrant)

Remerciements

Cette thèse est le fruit de trois années passés au Département Systèmes et Circuits Intégrés Numériques (DSCIN) du Commissariat à l'Energie Atomique et aux Energies Alternatives (CEA) (CEA LIST), au sein du Laboratoire Environnement de Conception et Architecture (LECA) sur le plateau de Saclay. Je remercie Fabien Clermidy, chef du DSCIN, et Thomas Dombek, Chef de Département Adjoint, pour leur accueil, la confiance qu'ils m'ont accordée et les moyens qu'ils m'ont fournis pour accomplir cette thèse dans les meilleures conditions.

Je tiens à remercier Tanguy Sassolas, chef du LECA et son prédécesseur Nicolas Ventroux pour la confiance et le soutien qu'ils m'ont accordés tout au long de ces trois ans.

Je remercie également mon directeur de thèse, Philippe Coussy, Directeur Adjoint du LAB-STICC (CNRS UMR 6285) à l'Université de Bretagne Sud. Merci pour tes conseils et tes recommandations qui étaient enrichissantes pour ce travail. Merci d'avoir accépté de diriger cette thèse et pour avoir participé à mon jury.

Je remercie du fond du cœur, Jean-Marc Philippe, ingénieur chercheur au CEA/Thales, pour avoir effectué l'encadrement de ma thèse. Tu es une personne magnifique, tant sur le plan professionnel que sur le plan humain. J'ai tellement apprécié de travailler avec toi. Merci d'avoir partagé avec moi tes connaissances scientifiques de grandes valeurs. Et merci pour la confiance que tu m'as accordée et pour la grande autonomie que tu m'as laissée. Nos échanges étaient très enrichissants, et m'ont permis de prendre du recule sur mes travaux. Merci pour tes encouragements durant les moments difficiles. Je te serai reconnaissante pour toujours.

Je remercie Benoit Tain, ingénieur chercheur au CEA, pour avoir effectué l'encadrement de ma thèse. Je te remercie pour ton enthousiasme et ton soutien sans faille tout au long de ma thèse. Je remercie également Lilia Zaourar pour son soutien et son encouragement tout au long de ma thèse.

Je remercie Alix Munier, Professeur à Sorbonne Université, et François Berry, Professeur à l'Université Clermond Auvergne, pour avoir accepté de juger mon travail en tant que rapporteur ainsi que pour leur participation à mon jury de thèse. Je remercie également Bernard Girau, Professeur à l'Université de Lorraine, et Olivier Sentieys, Professeur

à l'Université de Rennes 1, pour leur participation à mon jury.

Je remercie ensuite toutes les personnes que j'ai rencontrées au CEA tout au long de ma thèse, pour toutes les discussions intéressantes que nous avons pu avoir malgré la situation sanitaire qui nous a séparé.

Je remercie spécialement Oumaima Matoussi et Benjamin Binder qui ont su m'apporter écoute et confiance à tous les moments.

Un grand merci à ma formidable famille pour tout le soutien qu'elle m'a apportée. Ma mère Raoufa, mon frère Maxime et ma sœur Hanin, je ne pourrai jamais vous remercier assez pour tout ce que vous avez fait pour moi.

RÉSUMÉ

Context and motivations

De nos jours, de nombreuses tâches de la vie quotidienne font l'objet d'automatisation telles que la conduite automobile, la reconnaissance d'images et de la parole, la médecine, etc. Doter des machines ou des systèmes d'un processus décisionnel apparaît comme le germe d'applications de nouvelle génération. Cette automatisation essentielle nécessite des algorithmes intelligents et puissants pour modéliser le comportement des individus. Cependant, modéliser la connaissance d'un environnement ou d'un contexte pour permettre aux ordinateurs de prendre une décision est une tâche difficile. C'est là qu'entre en jeu l'intelligence artificielle (IA), qui est un vaste domaine de recherche qui propose des algorithmes pour résoudre des problèmes complexes en émulant le raisonnement, la prise de décision et l'acquisition de nouvelles connaissances, compétences et compréhensions chez l'homme par des machines. L'IA est un terme large qui inclut *Machine Learning* (ML) et *Deep Learning* (DL). ML, un sous-ensemble de l'IA, est un domaine d'étude qui utilise beaucoup de données pour rendre les ordinateurs plus intelligents et capables de résoudre des problèmes complexes sans être explicitement programmés. Il donne aux machines la capacité d'apprendre par elles-mêmes et d'accomplir des tâches de prise de décision. Le DL est un sous-ensemble de ML inspiré du comportement biologique des neurones (c'est-à-dire des cellules cérébrales humaines) qui utilisent un empilement de plusieurs couches, d'où le terme "profond". Ces couches sont formées d'éléments informatiques simples et interconnectés qui extraient progressivement des fonctionnalités de haut niveau à partir de données de niveau inférieur. Ce processus permet de dériver des connaissances de haut niveau (par exemple "un chien", "un visage", "une voiture") à partir de données brutes (par exemple un ensemble de pixels).

Les premiers réseaux de neurones artificiels (*Artificial Neural Networks* - ANN) sont apparus en 1950. On prétendait qu'ils étaient capables de simuler le comportement d'un cerveau très simple composé de neurones formels. La propriété intéressante de ces premiers RNA était leur capacité à être enseignés d'une manière ou d'une autre. Ce processus d'apprentissage a été effectué à l'aide d'un ensemble de données fourni par l'utilisateur

comprenant des données d'entrée et les données de sortie correspondantes. La rétropropagation des erreurs entre les données de sortie attendues et la sortie de l'ANN permet de modifier les paramètres de l'ANN, donc de lui apprendre à produire de bonnes sorties. En d'autres termes, cela lui permet de reconnaître des modèles spécifiques. L'un des premiers RNA (un seul élément) fut le Perceptron, inventé en 1957 par Franck Rosenblatt (McCulloch et Pitts ont décrit la base d'un neurone formel dans les années 1940). Comme exemples d'algorithmes de DL, les réseaux de neurones profonds (DNN) sont composés de neurones organisés en une structure en couches. En raison des multiples couches (profondeur) et du nombre potentiellement élevé d'éléments calculatoires dans chaque couche (largeur), les DNN présentent un nombre élevé de paramètres qui permettent de d'encoder une grande quantité de connaissances. Ceci explique leurs excellents résultats dans les tâches de reconnaissance complexes. Comme leurs ancêtres, les DNN peuvent acquérir les connaissances de manière supervisée en utilisant des données étiquetées dans une phase d'apprentissage. Ces connaissances acquises sont ensuite utilisées pour inférer des résultats dans la phase d'inférence, qui est la principale préoccupation des concepteurs d'architectures matérielles de systèmes embarqués, et donc le principal intérêt de cette thèse.

Parmi les DNNs, les réseaux de neurones convolutifs (CNN) sont l'exemple le plus célèbre d'une telle approche et le fondement des architectures modernes de réseaux de neurones. Les CNN sont appliqués dans de nombreuses applications telles que la classification d'images et la détection et la reconnaissance d'objets. En général, les premières couches de CNN sont basées sur des filtres convolutifs qui agissent comme des extracteurs de caractéristiques et les dernières couches sont basées sur des neurones entièrement connectés qui effectuent un processus de classification grâce aux caractéristiques extraites. Ils ont une organisation hiérarchique inspirée de l'architecture des neurones du cortex visuel. Ces différentes couches reposent sur des paramètres (coefficients de filtrage, pondérations, etc.) appris lors de la phase d'apprentissage de manière supervisée. De nos jours, les CNN sont l'une des principales technologies appliquées aux problèmes de classification. Ils sont utilisés par les géants mondiaux du numérique comme Google (c'est-à-dire Google Photos) et Facebook pour la détection et la reconnaissance des visages, etc. Ces algorithmes évoluent continuellement pour augmenter la précision de leur processus de reconnaissance et réduire les besoins en calcul et en mémoire.

Les CNN ont d'excellentes performances dans les défis de reconnaissance d'images populaires tels que le défi de reconnaissance visuelle à grande échelle ImageNet (*Ima-*

geNet Large Scale Visual Recognition Challenge - ILSVRC) [110]. Cependant, ces performances s'accompagnent d'une grande complexité de calcul et de besoins en mémoire. L'implémentation de ces algorithmes sur des appareils mobiles et edge suscite un grand intérêt et pousse les chercheurs en algorithmique à proposer de nouveaux modèles de DNN et des techniques d'optimisation pour réduire ces besoins (e.g. réduction de précision, pruning, etc.). Bien que de nombreux algorithmes et techniques d'optimisation efficaces aient été proposés au cours des dernières années, l'intégration de tels algorithmes dans les systèmes de périphérie reste un défi. En même temps, les concepteurs des accélérateurs matériels travaillent sur le développement d'architectures calculatoires et d'accélérateurs économes en énergie à cet effet. De nombreux accélérateurs matériels DNN proposés sont destinés à atteindre les objectifs classiques des systèmes embarqués : faible latence, faible consommation d'énergie, haute efficacité énergétique et capacité à exécuter différentes couches et formes CNN. Tout en introduisant diverses approches architecturales pour améliorer les calculs des DNN, ces propositions exploitent également des caractéristiques spécifiques des DNN telles que la sparsité pour réduire la consommation d'énergie et la latence. La Figure 1 montre la chronologie de l'évolution des architectures matérielles ciblant les CNN.

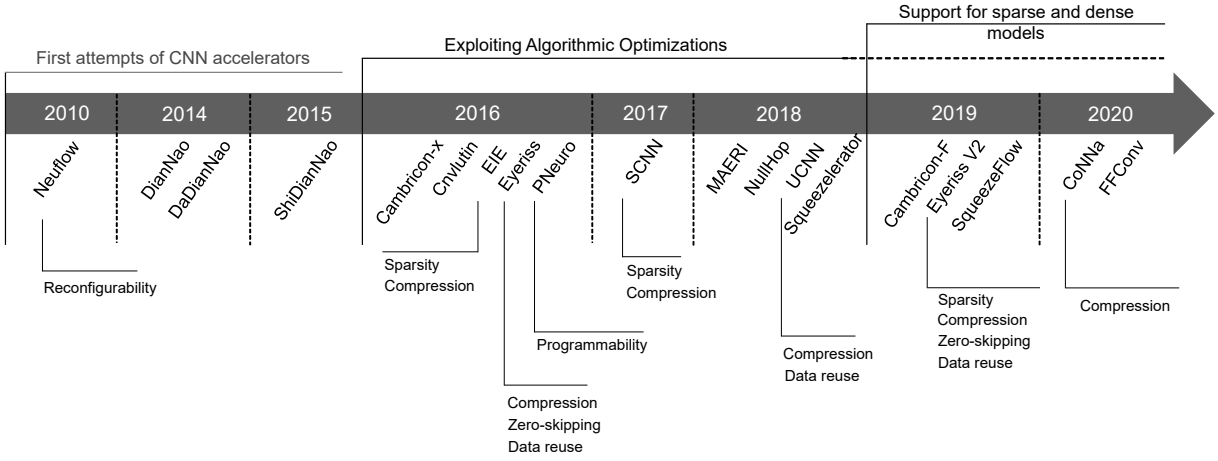


Figure 1 – Chronologie des architectures d'accélérateurs ciblant les réseaux de neurones. Au-dessus de la chronologie, trois époques principales d'accélérateurs matériels, compte tenu de diverses approches architecturales, sont mises en évidence. Sous la chronologie, les principales caractéristiques des accélérateurs sont mises en évidence.

Malheureusement, il existe un écart entre ces initiatives algorithmiques et architecturales. En fait, il est assez complexe pour les concepteurs d'architectures matérielles de trouver l'architecture appropriée ou au moins la configuration pertinente d'une ar-

chitecture spécifique pour les nouveaux réseaux de neurones (NN) ou pour un ensemble d'algorithmes basés sur les NN. De plus, la conception d'un accélérateur à l'aide de langages de description matérielle (HDL) tels que VHDL ou Verilog prend du temps, ce qui laisse les architectes matériels un pas en arrière, laissant l'écart se creuser entre l'évolution algorithmique et les accélérateurs matériels. Le processus de conception nécessite également de nombreux compromis de conception qui dépendent du résultat attendu par le concepteur et de contraintes spécifiques.

De nombreux efforts de recherche ont proposé d'utiliser des approches de génération de matériel pour combler cette lacune. Les frameworks qui en résultent s'appuient sur de nouvelles approches de conception telles que la synthèse de haut niveau (HLS) et/ou des modèles d'architecture informatique cible. Des outils HLS commerciaux peuvent être utilisés, tels que Siemens-EDA Catapult ou Xilinx Vivado-HLS, pour générer le niveau de transfert de registre (RTL) à partir d'un algorithme codé de haut niveau (par exemple C, C++). Le code source de haut niveau est optimisé en appliquant des transformations au code source, ce qui se traduit par une meilleure génération de matériel. De plus, des ensembles d'optimisations spécifiques à l'outil sont appliqués par le concepteur pour optimiser davantage la RTL et atteindre l'objectif de conception souhaité. Malgré les avantages de telles approches, des questions importantes demeurent : Quelles sont les clés pour développer des accélérateurs optimisés par rapport à un objectif de conception ? Comment concevoir efficacement des accélérateurs matériels ciblant les DNN tout en tenant compte de l'évolution continuelle de ces algorithmes? Quelles sont les techniques de conception qui permettent de diminuer le *time-to-market* des accélérateurs DNNs ?

Approche scientifique et résultats

Cette thèse présente d'abord l'état de l'art des réseaux de neurones profonds (*Deep Neural Networks* - DNN) et montre la différence entre apprentissage et inférence. La lumière est mise, en particulier, sur les algorithmes CNN qui sont efficaces et adaptés à l'accélération en raison de leurs structures parallèles intrinsèques, mais sont gourmands en calculs et ont des besoins en mémoire importants. De là vient le problème d'accélération matérielle des CNN en phase d'inférence, notamment dans les systèmes embarqués, qui est confronté à diverses solutions. Ces solutions reposent principalement sur des accélérateurs matériels ASIC qui offrent des solutions dédiées et fortement optimisées. En raison du manque de flexibilité des architectures ASIC dédiées, l'aspect reconfiguration des

FPGA est exploité. Les FPGA offrent la possibilité d'adapter les accélérateurs en termes de largeur de bit et de mémoire aux besoins exacts des algorithmes. Malheureusement, la conception d'accélérateurs DNN basés sur ASIC ou FPGA pour l'inférence nécessite du temps et de l'expertise si elle est effectuée manuellement ou même en utilisant des méthodes de conception de haut niveau, telles que la synthèse de haut niveau (HLS). Par conséquent, un autre ensemble de solutions, proposant des frameworks de génération de matériel qui exploitent des techniques de conception de haut niveau (par exemple HLS), est présenté qui vise à réduire l'écart entre l'application et l'architecture matérielle et ainsi réduire la complexité et le temps du processus de conception.

Ensuite, les approches de génération de matériel existantes sont comparées en fonction de différents critères. Ces critères permettent d'évaluer qualitativement chaque approche proposée en termes de niveau d'abstraction utilisé, les méthodes utilisées pour l'exploration de l'espace de conception, la méthode de génération de matériel et le processus global d'optimisation de l'architecture matérielle à générer. Cette comparaison montre que le HLS est l'un des plus utilisés pour sa facilité d'utilisation et sa flexibilité. En outre, elle montre que HLS n'est pas pleinement exploité au niveau algorithmique et que la plupart des approches sont basées sur des modèles ou des architectures existants. Concernant le processus d'optimisation, il repose largement sur une recherche exhaustive et il est également spécifique à l'architecture cible. De plus, le RTL produit, dans la plupart des cas, ne peut pas être optimisé une fois synthétisé. Sur la base de cette comparaison, des améliorations possibles au processus de conception automatisée sont identifiées.

Par conséquent, cette thèse présente une méthodologie automatisée de bout en bout pour concevoir des accélérateurs matériels pour les DNN embarqués. Cette méthodologie permet de réduire l'écart entre les descriptions abstraites et les architectures matérielles, car les concepteurs des architectures matérielles ont des difficultés à maîtriser le large espace des DNNs et leur évolution continue. Par conséquent, une première étape consiste à caractériser automatiquement et minutieusement le DNN pour en extraire des métriques pertinentes qui facilitent le processus de conception d'architectures. De plus, cette méthodologie suggère d'exploiter des techniques de conception de haut niveau pour réduire le temps de conception et augmenter la productivité, tout en permettant suffisamment de degrés de liberté pour l'exploration de l'espace de conception (*Design Space Exploration*- DSE). Malheureusement, le vaste espace de conception rend l'exploration de toutes les potentielles solutions très chronophage. Pour cette raison, la méthodologie proposée automatise cette exploration pour faciliter le processus de conception global.

Figure 2 présente le flot de la méthodologie proposée.

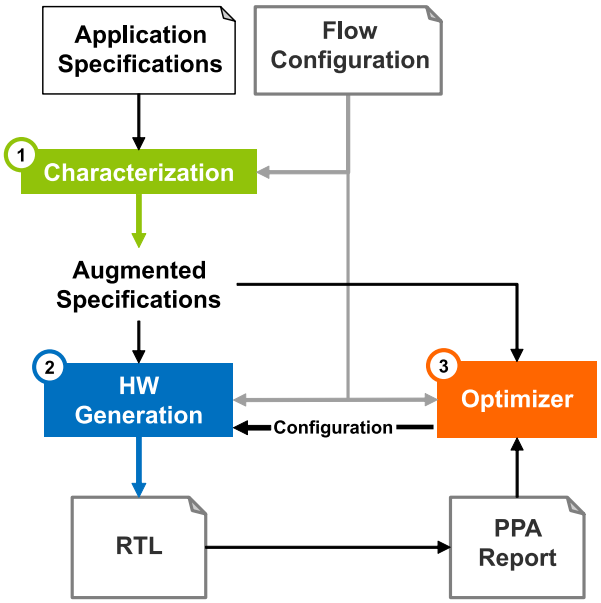


Figure 2 – Le flot de l'approche proposée avec ses trois étapes principales : Caractérisation, Génération Matérielle et Optimiseur.

Implementation et résultats

Cette méthodologie a été mise en œuvre sous la forme d'un framework, appelé SHEFTENN. Le framework cible les FPGA et ne dépend pas des architectures ou des modèles existants. Il consiste en trois modules interdépendants travaillant ensemble pour assurer une implémentation optimisée du RTL (cf. Figure 3). Chaque module a un rôle spécifique dans le processus de conception.

Le premier module, *Characterization*, réduit l'espace de conception grâce au calcul et à l'analyse de métriques et génère des spécifications augmentées qui fournissent un premier ensemble de pré-optimisations. Une caractérisation approfondie des applications DNN est essentielle pour acquérir les connaissances requises pour concevoir des accélérateurs matériels efficaces. Ce module caractérise et étudie le DNN d'entrée, puis produit des spécifications augmentées liées à l'application. Comme étape préliminaire, un script d'analyse implémenté en Python traite le fichier de description DNN. Le script Python est adaptable au format du fichier d'entrée (TensorflowLite, ONNX ou N2D2, incluant éventuellement les paramètres/poids du réseau). Les fichiers de description contiennent

principalement la topologie du DNN, l'ordre, les types ainsi que la configuration de chaque couche. Ils peuvent également contenir les paramètres du réseau de neurones, si le format de fichier prend en charge cette option. Pour prendre en charge cette variété de descriptions d'entrée, des bibliothèques d'analyse appropriées sont importées dans le script Python. La description DNN d'entrée est transformée en IR. Ensuite, un script Python utilise cet IR pour calculer des métriques indépendantes de la cible/conscientes du matériel pour chaque couche. Ensuite, il extrait les métriques liées aux données et les spécifications augmentées. L'IR est mis à jour avec les métriques nouvellement calculées. Le module de caractérisation vérifie également si le CNN est quantifié en vérifiant les largeurs de bits des paramètres pour définir la bonne optimisation liée à la précision des bits dans le module de génération de matériel. Ce module analyse le comportement de l'ensemble du CNN pour aider à piloter la génération de code C. De plus, le module Caractérisation dispose d'une étape de génération de code compatible C-HLS (il génère le code source de l'ensemble du CNN) qui sert de modèle de référence à la fois pour l'analyse dynamique et la validation de la génération matérielle par co-simulation.

Le deuxième module est le module de génération de matériel, *Hardware Generation*. Il exploite la HLS pour effectuer l'implémentation du matériel dans SHEFTENN. Ce module exploite un ensemble de données d'opérateurs HLS et de spécifications augmentées pour générer une représentation RTL optimisée de l'accélérateur DNN. Un module de génération de code (boîte bleue dans la Figure 3) génère un code source C optimisé et synthétisable (compatible HLS) basé sur les résultats de la caractérisation. Ces optimisations consistent en des transformations de boucle, telles que la réorganisation des boucles dans les premières couches, pour encourager la localisation et la réutilisation des données afin de réduire le temps de transfert des données. De plus, pour de meilleures optimisations, le code C-HLS intègre des directives de compilation associées aux opérateurs qui pourraient remplacer les boucles du noyau. En outre, le module Hardware Generation génère un fichier séparé comprenant des directives spécifiques à l'outil (pragmas) pour optimiser chaque couche ainsi que l'ensemble du DNN. Les paires pragma-valeur, les directives du compilateur, les facteurs de tuilage sont tous définis par le module *Optimizer*. L'outil HLS considéré dans l'implémentation SHEFTENN est Vivado-HLS puisque les FPGA Xilinx sont actuellement ciblés par le framework. Une fois toutes les optimisations définies, ce module synthétise le code C-HLS résultant, puis co-simule le code généré et compare les résultats avec ceux du modèle de référence pour vérifier le bon fonctionnement du design avant de générer le RTL.

Enfin, le module *Optimizer* prend en entrée les résultats de la Caractérisation, la configuration de chaque couche et les ressources du FPGA ciblé. Ensuite, il optimise le code source C-HLS (généré dans le module précédent). Il détermine les couples pragma-valeur à appliquer, les opérateurs à employer ainsi que les paramètres de tuilages. Ce module réduit le nombre d'exécutions de synthèse en utilisant des modèles de surface et de performance pour évaluer les différentes solutions lors de l'exploration de l'espace de conception. De plus, une véritable synthèse est établie à chaque *20* itérations pour s'assurer que les estimations de surface et de latence de l'ensemble du DNN ne s'écartent pas des valeurs de synthèse réelles. Ce nombre d'itérations a été choisi à des fins d'instanciation, et peut être remplacé par un nombre supérieur ou inférieur, permettant d'établir un compromis entre précision et rapidité. L'exploration de paramètres supplémentaires (e.g. nombre d'itérations) fait partie des perspectives, en particulier l'exploration du compromis entre vitesse et précision des modèles. L'algorithme génétique du module d'optimisation génère les meilleures configurations par couche uniquement en utilisant des métriques basées sur la couche et génère des configurations optimisées basées sur un compromis surface-performance. La deuxième étape du module Optimizer se concentre sur l'optimisation de l'implémentation du DNN complet. Il prend l'ensemble des configurations précédemment trouvées et recherche une implémentation satisfaisante qui intègre le DNN dans la cible FPGA. Précisément, il recherche une solution optimale de Pareto pour chaque couche. De plus, cette étape exploite les résultats de la caractérisation pour trouver des similitudes entre les couches en termes de formes de noyau, de nombre de paramètres et de volume de données d'entrée et de sortie. Par exemple, si deux couches ou plus ont les mêmes caractéristiques, une seule fonction C-HLS implémentant cette couche est conservée pour exécuter toutes les couches similaires. Par conséquent, cette étape sélectionne la meilleure configuration optimisée par le GA et l'utilise pour implémenter la couche sélectionnée. Ces couches sont ensuite allouées aux mêmes ressources RTL à l'aide du pragma d'allocation qui sert à gérer les ressources au niveau RTL.

Le framework est évalué sur deux DNN de la littérature : MobileNet-V1 [57] and SqueezeNet-V1.1 [58]. L'évaluation montre que l'approche proposée est fiable et permet de concevoir des accélérateurs matériels pour les DNN sans intervention de l'utilisateur dans le processus de conception. Les implémentations basées sur le framework ont montré de meilleures performances et un meilleur compromis surface-performance par rapport aux implémentations classiques.

La Table 1 résume les besoins en calcul et en ressources d'une implémentation clas-

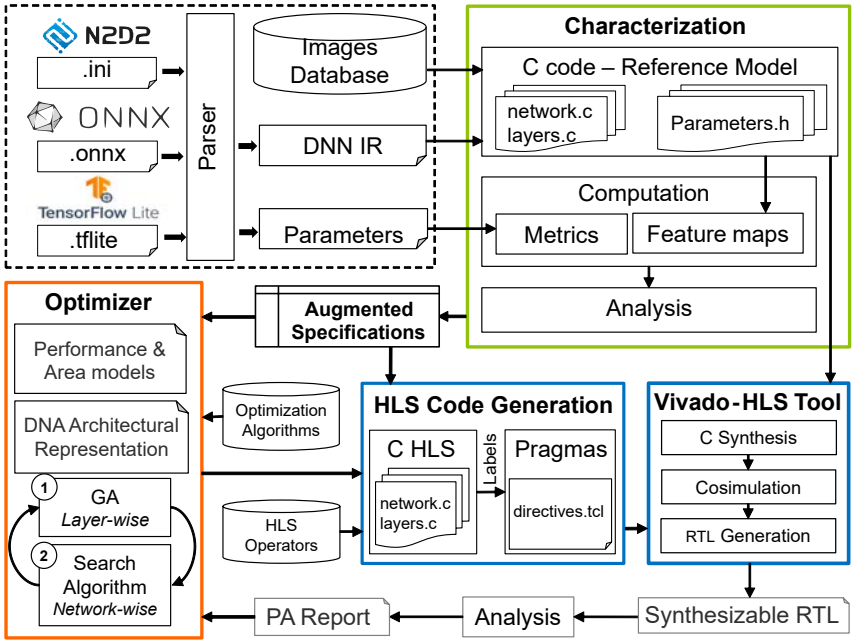


Figure 3 – Une description détaillée de chaque module du framework SHEFTENN instancié, où : le module de caractérisation calcule et analyse les métriques, la génération HW génère du code C-HLS et exploite une base de données d'opérateurs basés sur la HLS et utilise des outils HLS pour générer le RTL et l'optimiseur utilise un algorithme génétique pour optimiser le RTL en définissant les bonnes configurations pour le code C-HLS.

sique (non optimisée) de MobileNet-V1, qui n'inclut que les optimisations spécifiques à l'outil qui sont généralement appliquées automatiquement, et l'implémentation optimisée. Comme on peut le voir, la mise en œuvre de MobiletNet-V1 est optimisée en termes de ressources et de performance. L'utilisation des ressources a considérablement diminué par rapport à une implémentation classique. Par exemple, la mise en œuvre optimisée de MobileNet-V1 utilise *63.69*% moins de LUTs, *31.84*% moins de FF, *10.15*% moins de BRAM and *80.95*% moins de DSP, ce qui est dû aux optimisations employées, notamment le pragma Allocation et l'opérateur choisi dans certaines couches. En ce qui concerne la latence, la version optimisée de MobileNet-V1 est *22.5*% plus rapide que la version non optimisée, ce qui peut s'expliquer par le fait que certaines optimisations appliquées incluent souvent des calculs simultanés et un accès parallèle aux données, en particulier dans les opérateurs choisis. Il est à noter qu'une seule synthèse a été nécessaire dans ce cas, car les valeurs estimées de latence et d'utilisation des ressources sont proches des valeurs réelles, avec une erreur de *1.6*% pour la latence et une erreur moyenne de *15.63*% pour les estimations de ressources.

Ressources (%)	Non-Optimisée	Optimisée	Gain (%)
LUT	7,8	2,8	63,69
FF	2,7	1,8	31.84
DSP	26,3	5,0	80,95
BRAM	87,4	78,5	10,15
Latence (cycles) $\times 1000$	29 972	23 227	22,5

Table 1 – Utilisation des ressources (%) et latence (cycles) des implémentations optimisée et non-optimisée de MobileNet-V1.

En ce qui concerne le CNN SqueezeNet-V1.1, la Figure 4 montre l'utilisation des ressources et la latence des deux implémentations optimisées et non optimisées. À partir de cette figure, on peut voir que la version optimisée de SqueezeNet-V1.1 a une latence plus faible mais une utilisation des ressources plus élevée par rapport à la version non optimisée. Précisément, les LUT, FF et DSP ont augmenté de 18.20%, 0.62% et 3.2% respectivement par rapport à l'implémentation non optimisée. En revanche, les BRAM ont légèrement diminué de $0,05\%$. En ce qui concerne la latence, la version optimisée de SqueezeNet-V1.1 est 39.24% plus rapide que l'implémentation classique. On peut voir que l'outil a optimisé la performance de l'implémentation de SqueezeNet-V1.1 sans sacrifier l'utilisation des ressources, puisque la plupart des ressources utilisées ne dépassent pas 50% de celles disponibles sur le FPGA ciblé.

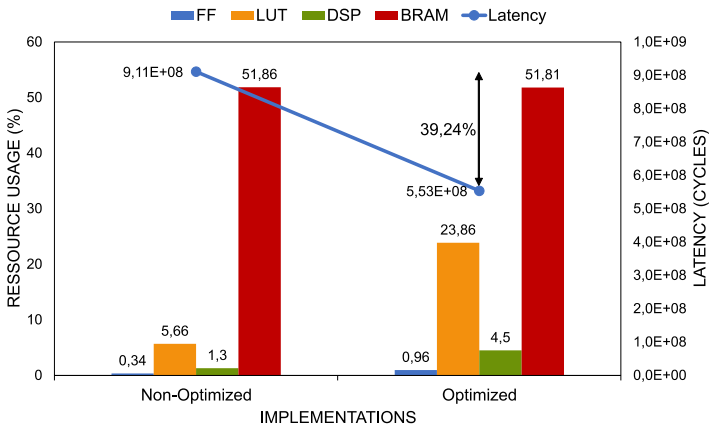


Figure 4 – Utilisation des ressources (%) et latence (cycles) des implémentations optimisée et non-optimisée de SqueezeNet-V1.1.

Ces résultats expérimentaux ont montré que le framework SHEFTENN est capable de produire des solutions viables avec un compromis intéressant entre les performances de

la surface et de générer des implémentations optimisées d'accélérateurs pour les réseaux de neurones de l'état de l'art, en utilisant un algorithme génétique qui encode l'ordre des boucles et les optimise.

Comparaison entre SHEFTENN et la littérature

Cette section présente une première comparaison entre le présent travail et certains des frameworks les plus pertinents de l'état de l'art, résumés dans la Table 2.

Frameworks	Contributions	Input	Template	HLS-tool	Optimizations	Ref.
hls4ml	Hardware Generator Quantization-aware training quantization-aware pruning Targets FPGAs & ASICs	Trained DNN	No	Vivado-HLS, Catapult	No feedback after RTL synthesis; Relies on internal feedback loops of HLS-tools; optimize PPA	[43]
MAGNet	Template-based Hardware Generator; Targets ASICs	Design goal Hardware constraints DNN Specifications	Yes	Catapult	Bayesian optimization strategy for DSE; Feedback after RTL synthesis; Optimize PPA	[127]
Rivera-Acosta et al.	Direct Hardware Generator	CNN configuration Database	Yes	Quartus II	No feedback loops; No exploitation of parallelism in FPGAs	[107]
fpgaConvNet	Exploits SDF for efficient DSE; Targets FPGAs	Trained DNN model	No	Vivado-HLS	Optimization of performance & throughput only; No feedback loop after RTL synthesis	[126]
FP-DNN	Instantiate hybrid RTL-HLS template; Topology analysis	Trained DNN	Yes	Catapult	No feedback loop after RTL synthesis	[50]
FINN	Targets BNN; Exploits a predefined library for RTL generation	BNN model	No	Vivado-HLS	No feedback loop after RTL synthesis; Performance optimization	[125]
Solazzo et al.	Direct Hardware generator; Empirical estimation models for FPGA resources	CNN configuration & Weights	No	Vivado-HLS	Optimization of FPGA resources based on estimation models; No feedback loop after RTL synthesis	[120]
SHEFTENN	Hardware generator; estimation models for FPGA resources; Quantization-aware hardware generation	Trained CNN	No	Vivado-HLS	2-steps optimization process of FPGA resources and latency based on estimation models & real synthesis	-

Table 2 – Comparaison avec les frameworks de génération d'accélérateurs matériels de la littérature.

Ce tableau montre que la plupart des approches proposées, contrairement à SHEFTENN, offrent des frameworks de génération de matériel directs basés sur HLS sans boucles de rétroaction après la synthèse RTL pour optimiser divers aspects de conception (c'est-à-dire la puissance, les performances et la surface) en utilisant des valeurs réelles. MAGNet offre une boucle de rétroaction sur le PPA uniquement après la synthèse. Cependant,

il utilise la HLS pour générer des composants architecturaux individuels décrits à haut niveau mais pas au niveau algorithmique.

Certains frameworks de génération de matériel incluent une étape d'optimisation avant la synthèse RTL, mais ne se concentrent que sur certains aspects de la conception, ce qui n'est pas le cas de SHEFTENN qui vise à optimiser le PPA (*power performance et area*) de la conception en utilisant un processus d'optimisation en 2 étapes. Par exemple, FINN optimise l'implémentation en termes de performances uniquement et fpgaConvNet se concentre sur l'optimisation des performances et du débit en utilisant un DSE qui exploite le flux de données synchrones (*Synchronous Dataflow* - SDF). Le framework proposé par Solazzo et al. optimise uniquement l'utilisation des ressources. Il convient de noter que hls4ml s'appuie sur les boucles de rétroaction internes de l'outil HLS utilisé, tel que Xilinx Vivado-HLS et Siemens Catapult, pour optimiser le PPA de la mise en œuvre de l'accélérateur.

D'autre part, presque tous ces frameworks ne s'attaquent pas au temps d'évaluation et de conception important des accélérateurs DNN, qui est l'une des contributions SHEFTENN qui introduit des modèles d'estimation pour PPA afin de réduire le temps d'évaluation des solutions d'optimisation potentielles, et donc le temps total de conception. Seuls Solazzo et al. ont présenté des modèles d'estimation empiriques pour l'utilisation des ressources sur les cibles FPGA, tout en ignorant les performances et la puissance. Cependant, ces estimateurs sont limités à quelques configurations CNN et ne peuvent pas inclure les DNN de la littérature.

La majorité des frameworks de génération de matériel proposés, contrairement à SHEFTENN, sont limités à certains types ou configurations de DNN et ne prennent pas en compte les optimisations algorithmiques sensibles au matériel. Les frameworks introduits par Rivera-Acosta et al. et Solazzo et al. ne peuvent pas prendre en charge que quelques configurations de couches. FINN n'accepte que les DNN binarisés et ne peut donc pas prendre en compte les DNN standard. Certaines approches, telles que MAGNet, fpgaConvNet et FP-DNN, considèrent les DNN de la littérature ayant de grandes topologies. Cependant, ils n'explorent pas la génération de matériel sensible à la quantification pour les DNN quantifiés.

SHEFTENN, contrairement à d'autres approches dans l'état de l'art, est un outil automatisé ciblant les FPGA, et est indépendant d'un modèle ou d'une architecture cible. Il permet de concevoir rapidement des accélérateurs DNN grâce à la combinaison d'une étape de caractérisation qui réduit l'espace de conception et d'une étape de génération RTL qui

exploite la HLS. Une étape d'optimisation optimise l'implémentation de l'accélérateur DNN grâce à une exploration automatique de l'espace de conception, qui utilise un processus d'optimisation hybride de génération de code source RTL basé sur un code source C-HLS, comme expliqué précédemment. Ce processus repose sur deux algorithmes distincts, le premier optimise chaque couche individuellement et le second optimise l'ensemble du CNN. Un tel processus offre un compromis entre la qualité et la rapidité de l'exploration à l'aide de modèles d'estimation (pour les performances et l'utilisation des ressources) et la synthèse réelle. De plus, contrairement aux autres approches, SHEFTENN supporte les réseaux quantifiés grâce à la HLS.

Ce travail doit être considéré comme une première étape de la mise en œuvre de la méthodologie proposée, et de nombreuses perspectives ont été identifiées pour l'améliorer. La section suivante discute des résultats et présente ces perspectives.

Conclusion

La HLS offrait un moyen rapide de concevoir des puces IA, sans avoir besoin d'une expertise matérielle avancée, tout en exploitant les FPGA. En même temps, de nombreux nouveaux défis et perspectives intéressantes ont émergé. L'un des principaux défis est le DSE en raison de la variété des paramètres à modifier. Ces paramètres se présentent sous la forme d'un style de codage algorithmique et d'optimisations de haut niveau, qui nécessitent de nombreuses interventions de l'utilisateur pour atteindre l'objectif de conception souhaité, ce qui augmente le temps de conception. Par conséquent, les outils HLS doivent évoluer pour automatiser cette exploration, en particulier pour les accélérateurs d'IA, afin de réduire le délai de mise sur le marché (*time-to-market*) des nouveaux accélérateurs. Cette automatisation doit tenir compte de l'évolution rapide des DNN ainsi que des contraintes des systèmes embarqués.

Cette thèse a présenté une approche de conception de haut niveau qui exploite HLS. Le processus de conception de haut niveau est abordé à trois niveaux différents. Dans l'implémentation actuelle de la méthodologie proposé, l'espace de conception est d'abord réduit en considérant plusieurs options d'optimisation déduites de la phase de caractérisation, puis augmenté en raison de la combinaison de pragmas et d'opérateurs HLS. La combinaison de ces trois niveaux semble essentielle pour assurer un flot de conception efficace et automatisé et pour réduire le temps de conception.

Discussion et perspectives

Bien que ce travail ait exploité les avantages de la HLS et des FPGA offrant respectivement la vitesse de conception matérielle et l'aspect de reconfiguration, divers aspects devraient encore être améliorés pour rendre l'approche proposée plus générique et efficace. Certaines de ces perspectives sont discutées ci-dessous :

— Afin de faire face aux évolutions algorithmiques, le framework devrait être capable de prendre en considération les nouvelles techniques d'optimisation des DNN, telles que le *pruning* et l'entraînement *quantization-aware* employées dans hls4ml [43]. La prise en compte de ces techniques dans l'implémentation matérielle améliore les performances et réduit l'utilisation des ressources et la consommation d'énergie.

— La consommation d'énergie doit être prise en compte dans le processus d'optimisation afin d'optimiser tous les aspects de la conception. Par conséquent, un modèle de puissance doit être créé et intégré dans le module *Optimizer*. Un exemple de puissance de modélisation consiste à suivre la même approche présentée dans ce manuscrit qui consiste à créer un dataset de microbenchmarks de différentes couches. Ces benchmarks seront écrits en C, synthétisés et simulés puis alimentés à un outil d'estimation de puissance RTL. Les données collectées seront transmises au logiciel NCSS pour concevoir le modèle approprié. D'autres solutions existent également et peuvent être utilisées pour estimer la consommation électrique comme présenté dans [92].

— Les modèles adoptés dans le module d'optimisation (*Optimizer*) sont liés aux configurations de couche. Par conséquent, ces modèles doivent être adaptés à chaque nouvelle configuration disponible pour obtenir une estimation acceptable avec une faible erreur quadratique moyenne. De plus, les données collectées après la simulation des microbenchmarks dépendent des FPGA employés, puisque seuls deux FPGA ont été utilisés pour synthétiser toutes les couches.

— Exploration d'autres paramètres du framework, en particulier l'exploration du compromis entre la vitesse de recherche d'une solution et la précision des modèles d'estimation utilisés.

— Les travaux futurs porteront également sur le partitionnement DNN pour permettre la mise en œuvre de grands DNN sur des FPGA plus petits. L'algorithme de partitionnement s'appuiera sur la représentation algorithmique de l'architecture du réseau (code source C-HLS) pour trouver le bon partitionnement, c'est-à-dire

un partitionnement qui réduit l'utilisation des ressources sans sacrifier les performances. Cet algorithme fera partie de l'algorithme de recherche, son rôle principal sera de trouver des couches avec des besoins en mémoire et en informatique similaires pour les allouer dans la même instance RTL. Le partitionnement peut également être au niveau de la couche, c'est-à-dire un partitionnement intra-couche, pour augmenter le parallélisme au niveau de la couche. De plus, des approches de type fpgaConvNet qui utilisent SDF peuvent également être utilisées pour trouver un partitionnement approprié pour le DNN.

— Une autre perspective intéressante est de trouver d'autres algorithmes d'optimisation capables d'explorer rapidement l'espace de conception et de fournir des solutions précises, tels que les algorithmes de DL.

— Exploration des effets de nouveaux pragmas pour étendre davantage l'espace de conception. De plus, cibler les DNN *sparse*, d'un point de vue matériel, en introduisant des techniques de compression basées sur la HLS enrichira le framework proposé.

— Différentes options possibles peuvent être utilisées pour abstraire la surface en termes de ressources FPGA, telles que des solutions pondérées, qui peuvent être considérées comme favorisant les designs nécessitant moins de LUT ou moins de BRAM en fonction du système global à mettre en œuvre, ou en tant que valeurs ajoutées des ressources sous une forme de combinaison linéaire de leur usage.

— Comparaison du framework SHEFTENN quantitativement avec les frameworks existants dans la littérature en implémentant les mêmes réseaux que ceux utilisés dans la littérature ou en expérimentant avec des frameworks open source existants.

Contributions

Cette thèse fait l'objet de plusieurs contributions cités ci-dessous. Les publications sont listées dans l'ordre chronologique inverse et classées parmi des revues, des conférences et des brevets.

Soumises dans des journaux:

1. Ali, N.; Philippe, J. M.; Tain, B. & Coussy, P., "SHEFTENN : Software and Hardware Exploration Framework for Efficient Implementation on Embedded Systems". ACM Transactions on Design Automation of Embedded Systems (TOADES), 2022
2. Ali, N.; Philippe, J. M.; Tain, B. & Coussy, P., "Exploration and Generation of

Efficient FPGA-based Deep Neural Network Accelerators". Journal of Signal Processing Systems (JSPS), 2022

Soumises dans des conférences:

1. Ali, N.; Philippe, J. M.; Tain, B. & Coussy, P. "An Integrated Design Space Exploration and Hardware Generation Tool Flow for Next-Generation Deep Neural Networks Accelerators", Field Programmable Custom Computing Machines (FCCM), 2022.

Publiées:

1. Ali, N.; Philippe, J.-M.; Tain, B. & Coussy, P., "Exploration and Generation of Efficient FPGA-based Deep Neural Network Accelerators", 2021 IEEE Workshop on Signal Processing Systems (SiPS), 2021
2. Ali, N.; Philippe, J. M.; Tain, B.; Peyret, T. & Coussy, P.,"Deep Neural Networks Characterization Framework for Efficient Implementation on Embedded Systems", 2020 IEEE Workshop on Signal Processing Systems (SiPS), 2020, 1-6

Submitted patents:

1. High-level design flow for artificial intelligence accelerator (2021)

Table of Contents

Résumé **5**

Introduction **25**

1 Background on Deep Neural Networks and Hardware Accelerators **31**
1.1 Deep Neural Networks - DNNs . . . 32
1.1.1 Training vs. Inference . . . 34
1.1.2 Convolutional Neural Networks - CNNs . . . 35
1.2 Hardware Accelerators for DNNs . . . 43
1.2.1 Dataflows Taxonomy . . . 45
1.2.2 Dedicated ASIC Architectures . . . 46
1.2.3 Programmable ASIC architectures . . . 48
1.2.4 FPGA Architectures . . . 48
1.3 Conclusion . . . 50

2 State of the Art of Design Methodologies and Proposed Approach **53**
2.1 Design Flows and Tools . . . 54
2.1.1 High-Level Hardware Description Languages . . . 54
2.1.2 High-Level Synthesis - HLS . . . 55
2.2 Hardware Generation Frameworks . . . 58
2.3 Proposed Methodology Overview . . . 64
2.4 Comparison between SHEFTENN and the State of the Art . . . 66
2.5 Conclusion . . . 68

3 Characterization and Metrics Analysis **69**
3.1 Characterization step Overview . . . 70
3.1.1 Metrics Computation . . . 72
3.1.2 Metrics Analysis . . . 74
3.2 Characterization and Analysis of different State-of-the-Art Networks . . . 82
3.3 Conclusion . . . 83

4 Flexible Hardware Generation 85
4.1 Hardware Generation step Overview . . . 86
4.1.1 Non-optimized C-HLS implementations of DNN layers . . . 87
4.1.2 C-Code transformation example . . . 88
4.2 High-Level Optimizations . . . 90
4.3 Library of HLS operators . . . 95
4.3.1 Introduction . . . 95
4.3.2 HLS operators overview and early results . . . 97
4.4 Early Results using the Hardware Generation step . . . 99
4.5 Conclusion . . . 105

5 Optimizing hardware through Design Space Exploration 107
5.1 Related Works on DSE . . . 108
5.1.1 Model-based DSE approaches . . . 109
5.1.2 Black-box-based DSE approaches . . . 110
5.1.3 Discussion . . . 112
5.2 DSE Algorithm . . . 112
5.2.1 Introduction to Genetic Algorithm . . . 113
5.2.2 Implementation of the DSE module . . . 115
5.3 Model-based estimations . . . 118
5.3.1 Performance Model . . . 118
5.3.2 Models for resource utilization . . . 119
5.3.3 Discussion on obtained models . . . 122
5.4 Early results for DNN layer implementation optimization . . . 123
5.5 Conclusion . . . 126

6 Implementation of the SHEFTENN Framework and Assessment 127
6.1 Implementation of the SHEFTENN framework . . . 128
6.1.1 Characterization module . . . 128
6.1.2 Hardware Generation module . . . 130
6.1.3 Optimizer module . . . 130
6.2 Experiments and Results . . . 132
6.2.1 Experimental Setup . . . 132
6.2.2 SHEFTENN Evaluation using MobileNet-V1 . . . 133
6.2.3 SHEFTENN Evaluation using SqueezeNet-V1.1 . . . 143

6.3 Conclusion 144

Conclusion **147**

Bibliography **153**

INTRODUCTION

Nowadays, many tasks of daily life are subjects of automation such as car driving, image and speech recognition, medicine, etc. Providing machines or systems with a decision-making process is seen to be the seed of next-generation applications. The needed automation requires smart and powerful algorithms to model the behavior of individuals. However, modeling the knowledge of an environment or a context to enable computers to take a decision is a hard task. This is where artificial intelligence (AI) comes into play, which is a large field of research that proposes algorithms to solve complex problems by emulating the reasoning, the decision-making and the acquiring of new knowledge, skills and understandings in humans by machines. AI is a wide term which includes Machine Learning (ML) and Deep Learning (DL). ML, a subset of AI, is a field of study that uses a lot of data to make computers smarter and capable of solving complex problems without being explicitly programmed. It gives machines the ability to learn on their own and complete decision making tasks. DL is a subset of ML inspired by the biological behavior of neurons (i.e. human brain cells) that use a stack of multiple layers, hence the term "deep". These layers are formed by simple, interconnected computing elements that progressively extract high-level features from lower-level data. This process enables to derive high-level knowledge (e.g. "a dog", "a face", "a car") from raw data (e.g. a set of pixels). Figure 5 shows the chronological evolution of AI with a brief description of AI, ML and DL.

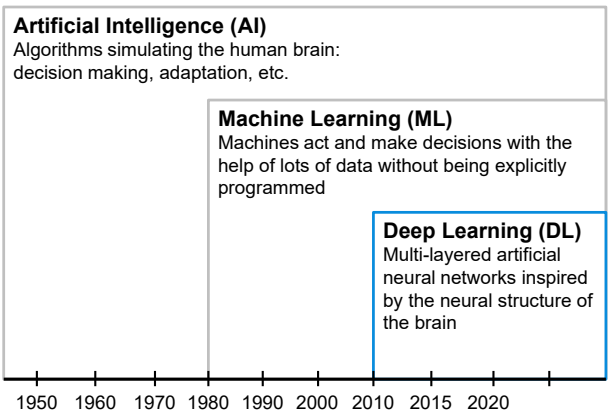


Figure 5 – Artificial Intelligence evolution and subsets.

First Artificial Neural Networks (ANNs) appeared in 1950. They were claimed to be capable of simulating the behavior of a very simple brain composed of formal neurons. The interesting property of these first ANNs was their ability to be taught in some way. This learning process was performed using a user-provided dataset comprising input data and the corresponding output data. Back-propagating the errors between the expected output data and the output of the ANN enables to modify the parameters of the ANN, thus to teach it the way to produce good outputs. In other words, it enables it to recognize specific patterns. One of the first ANN (one single element) was the Perceptron, invented in 1957 by Franck Rosenblatt (McCulloch and Pitts described the basis of a formal neuron in the 1940s). As examples of DL algorithms, Deep Neural Networks (DNNs) are composed of neurons organized in a layered structure. Due to the multiple layers (depth) and the potentially high number of computing elements in each layer (width), DNNs exhibit a high number of parameters which enable to store or encode a high amount of knowledge. This explains their excellent results in complex recognition tasks. As their ancestors, DNNs can acquire the knowledge in a supervised manner using labeled data in a training phase. This acquired knowledge is later used to infer results in the inference phase, which is the main concern of hardware engineers of embedded system architectures, and therefore the principle interest in this thesis.

In the landscape of DNN algorithms, Convolutional Neural Networks (CNNs) are certainly the most famous example of such an approach and the foundation of modern architectures of neural networks. CNNs are applied in many applications such as image classification and object detection and recognition. In general, first layers of CNNs are based on convolutional filters that act as feature extractors and the last layers are based on fully connected neurons that perform a classification process thanks to the extracted features. They have a hierarchical organization inspired by the architecture of the neurons in the visual cortex. Those different layers rely on parameters (filter coefficients, weights, etc.) learned during the training phase in a supervised manner. Nowadays, CNNs are one of the leading technologies applied for classification problems. They are used by the world digital giants like Google (i.e. Google Photos) and Facebook for face detection and recognition, etc. These algorithms are continuously evolving to increase the accuracy of their recognition process and to reduce computing and memory requirements.

CNNs have excellent performance in popular image recognition challenges such as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [110]. However, these performances come with high computing complexity and memory requirements. Imple-

menting these algorithms on mobile and edge devices raises great interest and push algorithmic researchers to propose new DNN architectures and optimization techniques to reduce these needs (e.g. precision reduction, pruning, etc.). Although many efficient algorithms and optimization techniques were proposed in the last years, embedding such algorithms into edge systems remains challenging. At the same time, hardware architects work to develop energy-efficient computing architectures and accelerators for this purpose. Numerous proposed DNNs hardware accelerators are intended to achieve the classical objectives of embedded systems: low latency, low energy consumption, high energy efficiency and ability to execute different CNN layers and shapes. While introducing various architectural approaches to improve DNN computations, these proposals also exploit specific features of DNNs such as sparsity to reduce energy consumption and latency. Figure 6 shows the timeline of the evolution of the hardware architectures targeting CNNs. Unfortunately, there is a gap between these algorithmic and architectural initiatives. In fact, it is quite complex for hardware designers to find the appropriate accelerator architecture or at least the relevant configuration of specific architectures for new neural networks (NN) or for a set of NN-based algorithms. In addition, designing an accelerator using Hardware Description Languages (HDLs) such as VHDL or Verilog is time consuming which leaves the hardware architects one step behind, letting the gap grow wider between the algorithmic evolution and the hardware accelerators. The design process also requires a lot of design compromises which depend on the output expected by the designer and specific constraints.

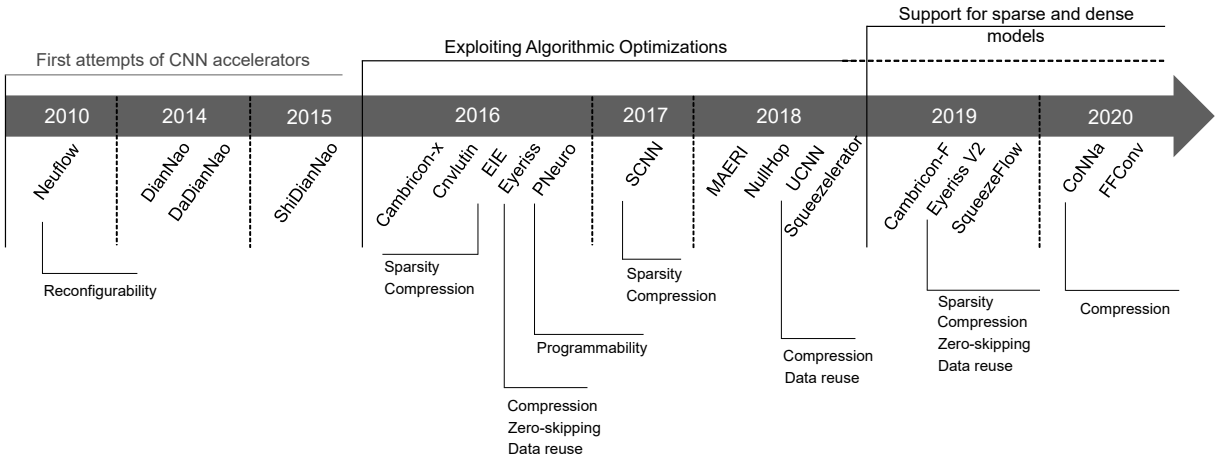


Figure 6 – Timeline of accelerator architectures targeting neural networks. Above the timeline, three main epochs of hardware accelerators, considering various architectural approaches, are highlighted. Below the timeline, the main features of accelerators are highlighted.

Numerous research efforts proposed to use hardware generation approaches to fill this gap. The resulting frameworks leverage new design approaches such as high-level synthesis (HLS) and/or target computing architectural templates. Commercial HLS tools can be employed, such as Siemens-EDA Catapult or Xilinx Vivado-HLS, to generate the Register Transfer Level (RTL) from a high-level coded algorithm (e.g. C, C++). The high-level source code is optimized by applying transformations to the source code, hence resulting in better hardware generation. In addition, tool-specific sets of optimizations are applied by the designer to further optimize the RTL and achieve the desired design goal. Despite the advantages of such approaches, important questions remain: What are the keys to develop optimized accelerators with respect to a design goal? How to efficiently design hardware accelerators targeting DNNs while also considering the changing landscape of such algorithms? What are the design techniques that allow to decrease the time-to-market of DNNs accelerators?

Aim of This Work

This thesis proposes a design approach and its implementation in the form of a framework, called SHEFTENN (Software-Hardware Exploration Framework Targeting Embedded Neural Networks), an end-to-end framework for generating efficient hardware accelerators from high-level algorithmic descriptions. The goal of this work is to reduce the gap between the software (SW) and the hardware (HW) by relying on an automatic design flow for CNN hardware accelerator architectures. SHEFTENN comprises three modules which interact with each other to bring together the algorithmic descriptions and the hardware architectures. The first module, *Characterization*, performs an in-depth study of the CNN algorithm with a hardware perspective. It computes metrics to derive augmented specifications to help the two other modules. The second module, *Hardware Generation*, uses the characterization results to generate an RTL (Register Transfer Level) source code of the accelerator using a high-level description (e.g. C/C++ source code if using high-level synthesis - HLS) and a database of operators. The third module, *Optimizer*, exploits the results of the other two steps to control the hardware generation block according to different objectives. Its goal is to optimize the generation of the RTL source code of the accelerator based on a high-level model of the architecture. This module reduces the number of synthesis runs (long exploration loop) by leveraging area and performance models to quickly evaluate each design point (short exploration loop).

To summarize, the contributions are as follows:

— The hardware generation approach based on three interdependent modules cooperating to generate optimized hardware accelerators. Its main purpose is to reduce the gap between the application and the hardware architecture.
— The SHEFTENN framework which generates optimized accelerators from a CNN description file. Side contributions also consist of its three modules: SHEFTENN analyses the input algorithm and generates augmented specifications. Then, it produces a HLS-friendly code to feed it to a third-party HLS tool that generates the RTL source code of the accelerator. The optimization phase of SHEFTENN works on the high-level algorithmic architecture of the network to optimize the RTL source code through generation.
— A database of HLS-based operators with different areas and latencies. These operators are meant to replace the kernel loops (innermost loops) in convolution and pooling layers to reduce the surface of the hardware representation of the layer or its latency or both. Only kernel shapes bigger than 1×1 have substitute operators.
— A two-steps optimization process which optimizes the high-level description of the architecture (i.e. C source code). The first step uses a genetic algorithm to optimize each layer of the network. The second step uses a search algorithm to optimize the whole network while being able to fit it on the target FPGA architecture.
— An optimizer module which optimizes the RTL source code generation based on an algorithmic representation of the accelerator architecture (e.g. C/C++). In other words, it optimizes the high-level representation of the accelerator architecture to generate an optimized RTL source code, based on high-level latency and area models and real synthesis runs. SHEFTENN offers a trade-off between exploration precision and speed by controlling the ratio between these two exploration loops.
— A quantization-aware C code generation that translates into a bitwidth optimized accelerator thanks to HLS. Accelerators for homogeneous and heterogeneous quantized DNN can be easily generated by the proposed approach, since it exploits the ability of HLS to tailor bitwidths to the exact needs of the DNN weights by using specific libraries.

This manuscript is organized as follows. The first chapter describes the context of this dissertation, including a background on DNN algorithms and hardware accelerators targeting these DNNs. The second chapter starts with a state-of-the-art on design flows and

hardware generation tools/frameworks, which their main features and discusses both their advantages and inconveniences, to later introduce the proposed approach and compare it with existing work. The third chapter presents the first building block of the proposed approach, Characterization module, while presenting technical details and partial results. The fourth chapter details the second building block, Hardware generation module, including its technical implementation, the adopted tools for high-level synthesis, the possible optimizations (pragmas) as well as their impact on the generated architecture. In addition, it also introduces HLS-based operators as a new layer of optimization. The fifth chapter details the last building block, Optimizer module, including the used optimization algorithms as well as their working flow. The sixth chapter presents the flow of the software and hardware exploration framework targeting embedded neural networks, SHEFTENN, which uses all the modules detailed in the previous chapters. The sixth chapter sketches the results of the framework used on state-of-the-art DNNs. Finally, a conclusion summarizes the work in this PhD and sketches future works.

Chapter 1

Background on Deep Neural Networks and Hardware Accelerators

Designing efficient hardware accelerators for Deep Neural Networks (DNNs) requires a thorough understanding of such applications. This chapter provides a background on both the application and the hardware accelerators, which allows gaining a deeper understanding of the goal of the present dissertation.

The first section presents a background on DNNs including a brief history of these algorithms. It shows the different steps that are employed to teach (learning phase) an DNN and to later use it to infer results (inference phase). This section also puts the light on Convolutional Neural Networks (CNNs) that are targeted in the scope of this thesis. It presents the different types of layers that form a CNN, and finishes with an overview of existing deep learning frameworks. Furthermore, this section highlights the extensive computational and memory requirements of CNNs that makes them difficult to embed on mobile and edge systems.

The second section introduces the challenges related to designing efficient hardware accelerators for DNNs in the inference phase, while emphasizing on the difficulty to find a suitable hardware architecture for various DNNs topologies. This section also presents the various employed approaches by hardware architects to achieve certain design goals (e.g. high performance, high energy-efficiency) by employing various dataflows that exploit the intrinsic parallelism offered by CNNs as well as the locality and reuse of different data types. In addition, it sheds the light on the complexity and the expensive design of dedicated ASIC architectures and the desire to reduce that cost by devising more flexible targets, such as programmable ASIC or reconfigurable FPGAs. Some of the most relevant accelerators in each type of target are briefly discussed.

Finally, a conclusion concludes the chapter by recalling its main sections and summa-

rizing the challenges of hardware designs, while briefly introducing the content of the next one.

1.1 Deep Neural Networks - DNNs

As briefly evoked in the introduction of this document, ANNs really started when Frank Rosenblatt came up with the Perceptron in 1957. It is the first model that shed the light on the learning mechanics. It is a mono-layer linear binary classifier, formed of one formal neuron, designed to classify visual inputs [108]. A formal neuron (also called artificial neuron) is a mathematical representation of a biological neuron with multiple inputs and a single output. It is a computational rule that associates an output with a number of inputs having real values. The proposed model by McCulloch and Pitts, presented in Figure 1.1, associates one synaptic weight (W) per input (X). In this model, the first operation consists of a sum the weighted inputs (e.g. $W_1.X_1+W_2.X_2+...+W_n.X_n$). A threshold value is added to this sum. The result of this summation is later transformed by a non-linear activation function (ϕ) to obtain the output Y.

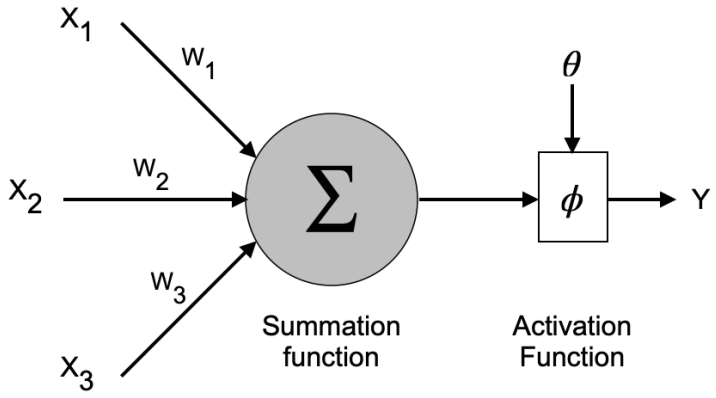


Figure 1.1 – Illustration of a formal neuron having three inputs and one output. X_1, X_2 and X_3 represent the inputs. W_1, W_2 and W_3 represent the synaptic weights. Y is the output.

An ANN can be created by stacking a set of computational layers, see Figure 1.2. Each of these layers consist of multiple artificial neurons. They are organized as follows: one input layer, several hidden layers, and one output layer. The layers are interconnected via artificial neurons (with weighted connections) with each layer using, as input, the output of the preceding layer.

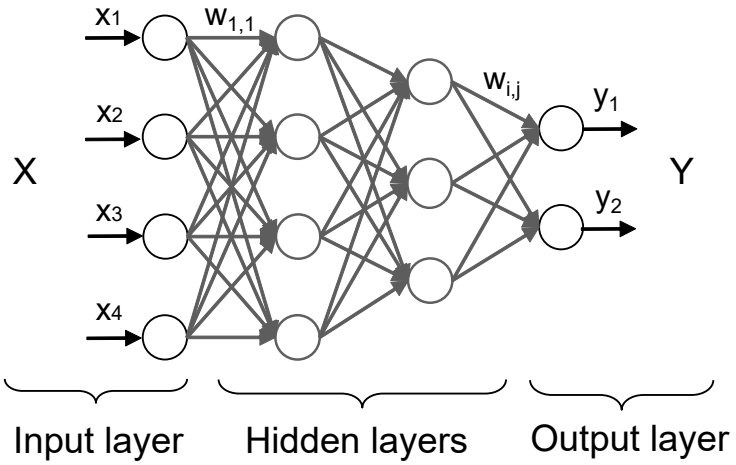


Figure 1.2 – Structure of a classical Deep Neural Network. X represents the inputs, W_{ij} represent the connections (i.e. weights), Y represents the outputs.

While trying to understand the functioning of neurons in the visual cortex, the conducted experiments by Hubel and Wiesel revealed two types of cells, simple cells and complex cells [36]. Simple cells respond to stimuli with specific orientations, thus serve for pattern or feature extraction. Complex cells respond to features (gathered from simple cells) regardless to variations in position unlike simple cells. Thereby, the experiments introduced the concepts of pooling and receptive fields. By means of these two cells, Hubel and Wiesel introduced a model for pattern recognition having a hierarchical organization similar to the visual nervous system.

Inspired by this model, the Neocognitron, a model with multiple layers organized in a hierarchical structure, was introduced by Kunihiko Fukushima [46]. Each layer consists of simple cells linked to complex cells with a fixed connection. This model was used for various pattern recognition tasks. The Neocognition was later revisited by applying the back-propagation algorithm to the learning system (a process of fine-tuning weights based on error rate from previous epoch) to adapt the weights of the model [109]. After years of research, the first Convolutional Neural Network (CNN) capable of classifying handwritten digits, LeNet-5, was created by Yann LeCun et al. in 1998 [72]. LeNet-5, represented in Figure 1.3, consists of a stack of five layers based on convolutional filters, sub-sampling and fully connected layers associated with a back-propagation algorithm.

Inspired by all these research efforts, Deep Neural Networks (DNNs) use multiple processing layers. Figure 1.4 shows, AlexNet [67], a classical example of a DNN. AlexNet became one of the most influential networks that spurred a significant AI wave, since it

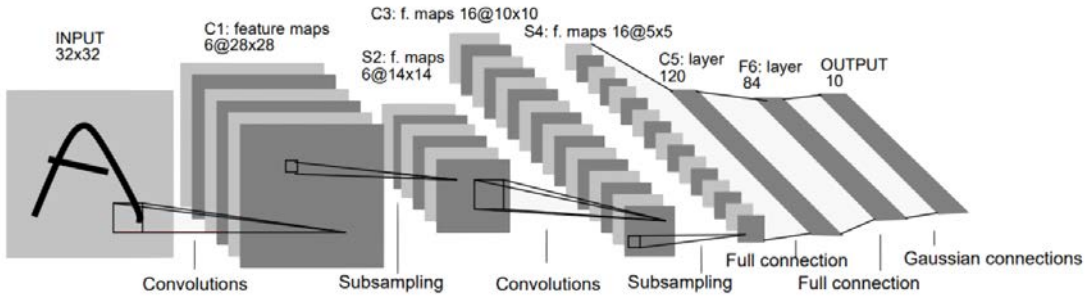


Figure 1.3 – Structure of LeNet-5 (from [72]).

won the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) [110] in 2012.

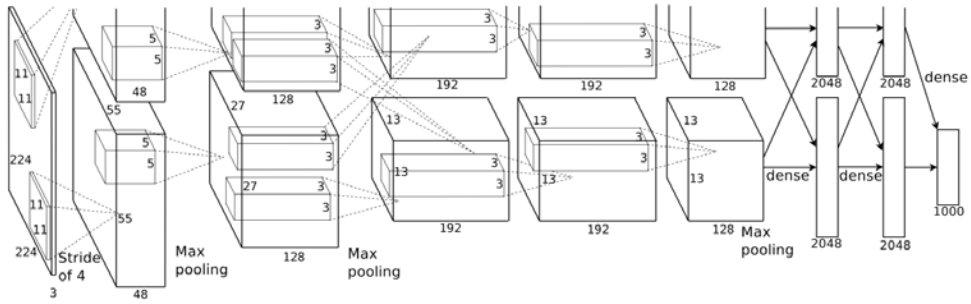


Figure 1.4 – Architecture of AlexNet (retrieved from [67]).

Having detailed the origin of DNNs and their history development, the next sections explain how a DNN learns, what are the steps of the leaning process and what is the difference between training and inference.

1.1.1 Training vs. Inference

DNN algorithms, as explained earlier, are inspired by the biological nervous system. Therefore, they require a learning process which allows them to learn a specific task (e.g. image recognition, classification, etc.). The goal of the so-called training phase is to modify and adapt the parameters of the DNN to enable it perform the required task with a reliable accuracy. After this phase, the DNN is meant to infer results about unknown data in the so-called inference phase. Figure 1.5 illustrates the two phases in which a DNN passes through.

Training The training is an essential step in which a model acquire the ability to perform a certain task, such as making predictions on data. It could have three different types: supervised, unsupervised and semi-supervised. Supervised learning, the most common approach, provides the neural network with labeled training dataset. In contrast, unsupervised learning uses an unlabeled training dataset. Semi-supervised learning falls between supervised and unsupervised learning, since it uses a small amount of labeled data and a large amount of unlabeled data. Only DNNs with supervised learning are used in the scope of this thesis.

This phase requires an important training dataset and a training algorithm to learn relevant features and fine-tune the weights of the network. In this phase the model learns in epochs (i.e. iterations). Back-propagation is employed to propagate the error backward through the layers of the network and tune the weights based on error rate (i.e. error between the actual label and the predicted one) obtained in previous iteration. For instance, the network in Figure 1.5 learns to classify images into two categories ("car" and "truck"). The network takes the training dataset as input, adjusts its weights and makes its prediction on each input whether it is a "car" or "truck" or neither. Based on the prediction and the actual answer, the training algorithm receives true or false in response.

Inference The inference takes place after the training is done and a reliable model is obtained (i.e. satisfying accuracy is reached on a validation dataset). The training infrastructure is no more required: the weights are fixed and no longer need adjustments. The neural network is capable of processing new data and propagates it forward only. It is therefore employed to perform the previously learned task.

1.1.2 Convolutional Neural Networks - CNNs

This section presents the building blocks of a CNN and clarifies some notations used throughout the manuscript. It also shows the types of layers, their hyperparameters, the mathematical operations they perform, as well as the computational complexity and memory requirements.

CNNs are one of the most popular examples of DNNs and one of the cutting edge technology in the computer vision field. As detailed earlier, CNNs have a hierarchical organization inspired by the biological architecture of the visual cortex. They use concepts similar to pooling and receptive fields, inspired by simple and complex cells introduced by Hubel and Wiesel. In addition, weights sharing is employed to reduce the number

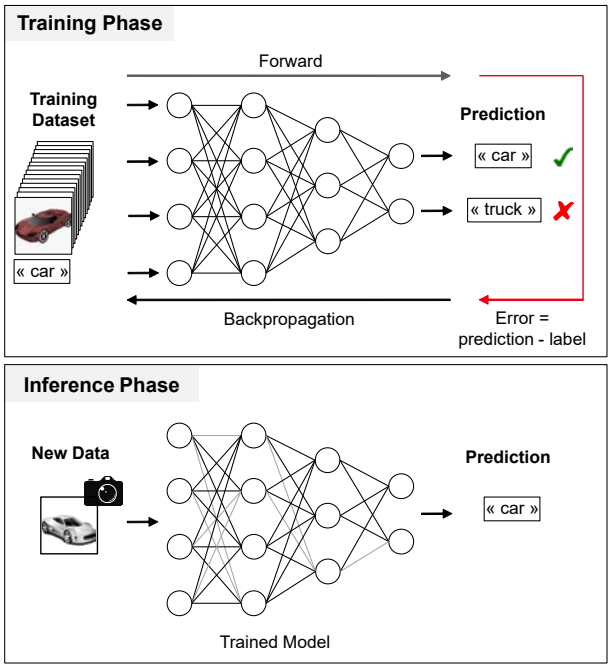


Figure 1.5 – Training vs Inference of a Deep Neural Network.

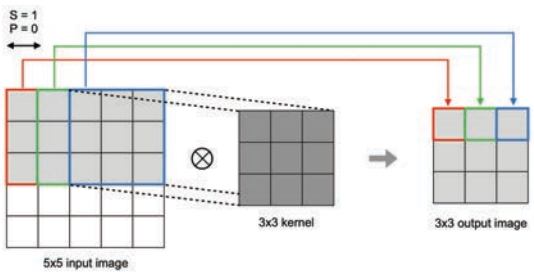
of weights and make feature recognition indifferent to the location of the feature. First layers are based on convolutional filters which extract key features from the input data. Latter layers perform the classification process to classify input data by using the previously extracted features. These layers rely on parameters (filter coefficients, weights, etc.) learned during a training phase on a specific dataset. More specifically, hidden layers perform mathematical operations such as convolutions, pooling and activation functions. In general, the core building blocks of a CNN are convolution, pooling and fully connected layers. Most popular types of CNN layers are detailed below.

Convolutional layer - Conv

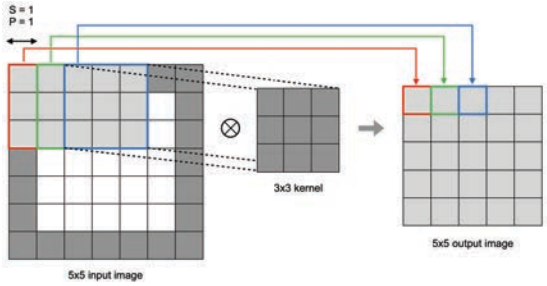
The main role of this layer is to extract features from input data. It is a computational layer which performs mathematical operation called convolution. Convolutional layers are presented as a stack of 2D matrices (i.e. 3D matrix). Each element of the matrix is the output of a neuron looking at a local region (e.g. a small window with certain dimensions) in the input data and sharing parameters with left and right neurons. Weighted filters (i.e. parameters) carry out connections between the input and the output of a layer.

In the image processing field, a convolutional operation is represented as a sliding

window, known as filter, over an input image (i.e input feature map). Each area covered by the sliding window is called the receptive field. The idea of the sliding window is interpreted as shared weights between the output neurons. However, the position of each receptive field depends on the stride S (i.e. step) and the zero padding P (the most common type of padding) of the input. Figure 1.6 illustrates a convolutional operation with a stride of 1 without padding in Figure 1.6(a), with a padding of 1 in Figure 1.6(b).



(a) 5x5 image convolved with a 3x3 filter with a stride $s = 1$ and padding $p = 0$.



(b) 5x5 image convolved with a 3x3 filter with a stride $s = 1$ and padding $p = 1$.

Figure 1.6 –
Illustration of a convolutional operation. The receptive field is represented as a shaded area on top of the input image.

In this operation, relevant features and characteristics are extracted from the input image. Output feature maps (also called activations map) are generated by accumulating partial sums. It is basically a Multiply ACcumulate (MAC) operation. MAC is used as a unit to express the computational complexity of the layer. Considering a batch size of one, the dimension of the input feature map is a 3D structure composed of N 2D planes, known as channels. Output feature maps are also 3D. As for filter weights, the dimension is 4D consisting of 3D structure (height K_x, width K_y and depth N matching the number of channels of the input feature map), and a 1D structure representing the number of filters that determines the number of output feature maps M. In a given layer, a 3D input feature map is processed by M 3D filters (i.e. 4D filters). Furthermore, there is a 1D bias that is added to the result of the filtering operation to help the layer produce non-zero valued results if the input features were null. The mathematical representation of a three dimensional convolutional layer is represented in Equation 1.1:

$$O[m][x][y] = f(\sum_{m=1}^{M}\sum_{i=1}^{K_x}\sum_{j=1}^{K_y}(W[m][n][i][j]\times I[n][ix][iy]) + B[m])$$
$$ix = x * S_x - P_x + i \;\; ; \;\; iy = y * S_y - P_y + j \tag{1.1}$$

Where $0 \leq n < N$ (N is the number of channels) and:

— O is the output feature maps (output matrix)

— I is the input feature maps (input matrix)

— W is the weights matrix

— B is the biases matrix

— S is a given stride

— P is a given padding

— K_x & K_y are the width and depth of the weights array

— f is the Activation Function, which will be described later

The stride, padding along with the size of the filter are the hyperparameters of a convolutional layer. Figure 1.7 illustrates the computation of a convolutional layer, omitting biases. The height and width of the output feature map are computed as in Equation 1.2.

$$R = \frac{H - K_x + 2P_x + S_x}{S_x} \;\; Output\ Height$$
$$C = \frac{W - K_y + 2P_y + S_y}{S_y} \;\; Output\ Width \tag{1.2}$$

Using the details above, parameters characterizing a convolutional layer can be calculated using Equation 1.3. These parameters include the number of parameters Nb_{Param}, the number of pixels in the input and output feature maps, also called input activations $Nb_{input\ activations}$ and output activations $Nb_{output\ activations}$ respectively.

$$Nb_{Param} = K_x \times K_y \times N \times M$$
$$Nb_{input\ activations} = H \times W \times N \tag{1.3}$$
$$Nb_{output\ activations} = R \times C \times M$$

The memory requirements of a layer can be determined using Equation 1.3, given a specific bit precision, by multiplying the number of parameters, and the number of input

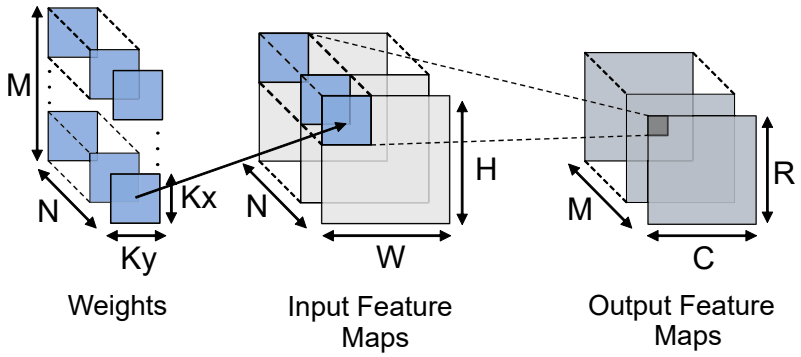


Figure 1.7 – Illustration of a convolutional layer, M being the number of filters/outputs, N the number of channels, R the output width, C the output height, W the input width, H the input height, Kx the filter width and finally Ky the filter height.

and output activations by the number of bits. The computing complexity (in MAC) of a convolutional layer can be calculated using Equation 1.4.

$$Nb_{std\ conv_{MAC}} = R \times C \times K_x \times K_y \times M \times N \tag{1.4}$$

Activation Function

Activation functions are inspired by the activation process of the biological neuron. They usually follow convolutions to introduce non-linearity and help the network learn complex features. Activation functions define the state of each neuron. In other words, it determines if a neuron is activated or not.

Several activation functions with different properties have been studied and evaluated in the literature [4, 39]. One of the most popular and modern activation function is the rectifier linear unit, ReLU [89]. It is efficient from an application performance perspective, it is also computationally efficient, especially in hardware. ReLU is not zero-centered, it clamps negative inputs to zero while positive inputs are transmitted as outputs as follows: $f(x) = max(0, x)$.

Pooling Layers - Pool

Pooling layers are standard layers in CNNs. They do not have parameters and thus are not affected by the training phase. These layers perform downsampling operations to maintain robust features only and are usually applied after convolutions. The process of a pooling layer is the same as the sliding window in a convolution without weights to share,

thus depends on two hyperparameters: stride, size of the sliding window (K_x & K_y). The most commonly used types of pooling are max and average. In max pooling the maximum value is taken out of a window of values $y = max(input\ window)$, and in average pooling, the output corresponds to the average of values in a window is computed and taken out $y = avg(input\ window)$. Figure 1.8 sketches the operations of max and average pooling on a 4×4 input feature map, with a 2×2 filter size and a stride of 2.

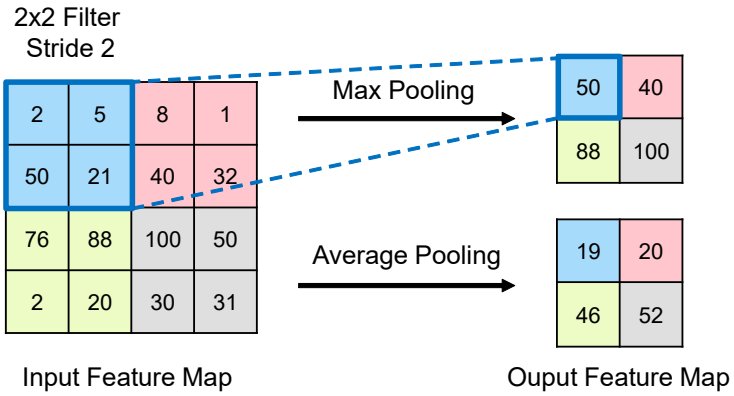


Figure 1.8 – Illustration of max and average pooling operations.

Fully Connected Layers - FC

Fully connected layers are usually the last layers in a CNN. Each output neuron is connected to every input neuron from previous layer, hence the name "Fully Connected". These layers have parameters, thus they are affected by the training phase. FC layers perform the classification process using flattened feature maps. The number of parameters as well as the input and output activations are computed using Equation 1.5. It is worth noting that the computational complexity of FC layers is equivalent to the number of parameters.

$$\begin{aligned} Nb_{Param} &= K_x \times K_y \times N \times M \\ Nb_{input\ activations} &= H \times W \times N \\ Nb_{output\ activations} &= M \end{aligned} \tag{1.5}$$

The previous sections described the basic building blocks of common CNNs. The given equations and notations will be used throughout the manuscript to compute the

computational complexity and the memory requirements which will be used to compare state-of-the-art CNNs.

Numerous CNNs were designed using the building blocks above. In particular, the most famous CNNs were competing and won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in object detection and image classification tasks [110]. Researchers continuously put efforts to improve the accuracy of the CNNs (i.e. top-1 and top-5 scores on the ILSVRC challenge) or to reduce the computational complexity and the memory needs. Therefore, the comparison between DNNs is mainly based on their accuracy and their execution time, during the inference phase, on GPU platforms as provided by some DNN frameworks like Tensorflow.

The majority of artificial neural network designers focused on increasing the number of layers to reach higher accuracy, such as GoogleNet [122], ResNet-50 [53], etc. Table 1.1 shows the main characteristics of a variety of neural networks. It lists a few state-of-the-art CNNs and presents some parameters that characterize each network: size of the input, number of conv. and fc layers, input data, number of parameters, and the number of MACs. The volume of parameters and the input data of each networks are reported in Bytes considering an 8-bit precision. These networks (except LeNet) are part of ILSVRC and take similar input dimensions $3 \times 224 \times 224$. Changing the size (decreasing or increasing the size) of the input only affects the number of input and output activations (i.e. memory requirements) in proportional ways. However, modifying the input dimension does not affect the number of parameters, since the dimensions of each layer in a network are fixed before training. Another interesting point is the huge number of convolutional layers, more than 90% of the layers, which concentrate most of CNNs computation.

CNNs Models	Input Size	Total Layers	Conv. Layers	FC Layers	Total Parameters (B)	Total Data (B)	Total Operations (MACs)
LeNet-5 [72]	28x28x1	5	2	2	60K	8.1K	0.33M
0.25-MobileNet [57]	224x224x3	28	27	1	466K	1.26M	41.03M
0.25-MobileNet [57]	128x128x3	28	27	1	466K	0.46M	13.58M
SqueezeNet-v1.1 [58]	227x227x3	28	10	1	1.2M	3M	352.54M
EfficientNet-b0 [123]	224x224x3	99	81	1	5.3M	13.9M	387.78M
VGG-16 [119]	224x224x3	16	13	3	138M	15.1M	15.47B
GoogLeNet [122]	224x224x3	22	57	1	7M	4.7M	1.68B
ResNet-50 [53]	224x224x3	50	53	1	25.5M	16.6M	3.83B

Table 1.1 – State-of-the-art CNNs and some characterizing metrics.

Many deep learning frameworks are developed to create reliable models. They provide, through APIs, building blocks for designing, training and validating deep learning models.

An overview of existing deep leaning frameworks is presented next.

Deep Learning Frameworks

With the growing popularity of DL algorithms in various fields, academic (e.g. University of California, Berkeley) and industrial groups (e.g. Google) showed lots of interest in developing the frameworks for designing, training and validating DNNs [8]. Most of the existing frameworks are open source (e.g. PyTorch [45], TensorFlow [1], etc.) and offer the possibility to design and train neural networks and even execute pretrained ones. Most of these frameworks strongly rely on hardware accelerators such as Graphical Processing Units (GPUs) and the related software APIs (e.g. CUDA) to speed-up the compute-intensive training phase. Table 1.2 presents a few Deep Learning Frameworks and compares some of their properties.

For instance, TensorFlow, one of the most used tools in DL, is an end-to-end open source DL framework developed by Google. It provides stable Python and C++ APIs and offers a complete and flexible ecosystem of tools, libraries and community resources that allow researchers and developers to easily train and deploy neural networks. It allows to deploy processing on multiple CPUs and/or GPUs. In addition, it provides a specific framework called TensorFlow Lite to deploy DL models on mobile and edge devices. Keras is an open source library developed "for humans" [66]. It offers consistent and simple APIs, it is user-friendly and extensible since it is written in Python. PyTorch, developed by FaceBook, is a powerful DL tool based on Python. It is easy to use and integrates acceleration libraries (such as Nvidia cuDNN [94]) to maximize development and deployment speed. N2D2 (Neural Network Design and Deployment) is another open source framework for designing and deploying DNN-based applications developed at CEA (*Commissariat à l'Energie Atomique et aux Energies Alternatives*) [87]. It provides a robust and efficient source code generation backend for several targets, and includes specific features targeting embedded systems such as advanced precision reduction techniques. The Nvidia Caffe, known as NVCaffe, is developed by the Berkeley Vision and Learning Center and community contributors [63]. It supports acceleration libraries for CPUs and GPUs such as cuDNN.

As it can be seen, from Table 1.1, these networks have extensive memory and computation needs which make them hard to implement into embedded systems. Fortunately, they exhibit a significant intrinsic parallelism, which make them the prime target for hardware acceleration. Although these networks are based on convolutional filters, the

Software	Caffe	Keras	TensorFlow	PyTorch	N2D2
Open source	Yes	Yes	Yes	Yes	Yes
Written in	C++	Python	C++, Python, CUDA	Python, C, C++, CUDA	C++
CUDA support	Yes	Yes	Yes	Yes	Yes
Initial release	2013	2015	2015	2016	-
Software license	BSD	MIT license	Apache 2.0	BSD	CeCILL-C license
Actively developed	No	Yes	Yes	Yes	Yes

Table 1.2 – Overview of few Deep Learning frameworks.

networks topologies as well as the types and dimensions of layers make them behave very differently. Therefore, finding a suitable implementation on a target architecture for CNNs is challenging due to their respective needs in memories and processing operators. The following section sets out the literature of the existing hardware CNN accelerators.

1.2 Hardware Accelerators for DNNs

Researchers in the past decade put efforts in designing hardware accelerators for DNNs algorithms [12]. The main focus was on accelerating the inference phase to target embedded systems, since the training is less critical and can be accelerated offline using GPUs. Designers have exploited all types of parallelism, offered by the data structure and the mathematical operations of DNNs, to design accelerators while focusing on performance or energy-efficiency or flexibility depending on the target technology (ASICs - Application Specific Integrated Circuits, or FPGAs - Field-Programmable Gate Arrays) or application. It is worth noting that FPGAs are often used for prototyping ASIC architectures. A typical high-level representation of spatial DNN accelerators can be illustrated as in Figure 1.9.

More specifically, they usually consist in an array of Processing Elements (PEs) for computation, which is mostly Multiply-ACcumulate (MAC) operations. PEs have internal control and are interconnected using a Network-on-Chip (NoC) to interchange data to support a specific dataflow. Regarding memories, a three-level memory hierarchy is often implemented to fulfill the extensive memory needs of DNNs and overcome memory accesses bottleneck. The first memory structure is at PE level, where each PE has a Register File (RF) to store intermediate results or accumulations. The second memory structure is an on-chip global buffer (GB) which is often implemented to provide the computation array with enough data. The third one is the off-chip memory, DRAM in most cases, where no longer used data is sent, and new data is retrieved from off-chip memory to

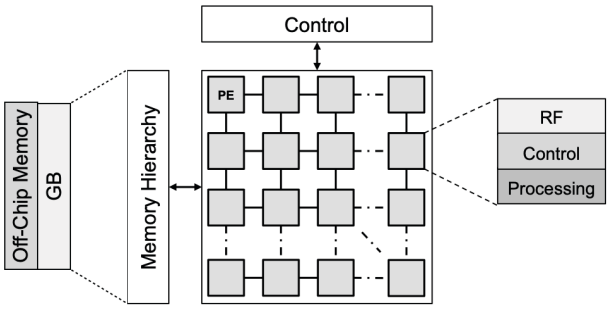


Figure 1.9 – Illustration of a spatial architecture of a DNN accelerator. Each processing element (PE) consists in 3 main parts: Register File, Control and the Processing part. The memory hierarchy of the overall architecture consists in a Global buffer and an off-chip memory.

fill the GB. RF and GB are usually implemented in such architectures to encourage data locality and reuse and ensure fast load/store operations. In addition, they are smaller, faster and have lower energy cost compared to DRAMs.

Moving CNNs to embedded systems is, till now, one of the main goals of hardware designers. Most research efforts focused on achieving classical embedded systems design objectives: high throughput, limited accesses to off-chip memory, high performance, high energy-efficiency. Choosing ASIC or FPGA targets, designers focused on dataflow-based architectures to reduce memory footprints of DNN algorithms due to their large memory requirements. They investigated different scenarios of data reuse provided by the internal structure of CNNs, which allows using data close to the PE and avoiding time and energy consuming read operations. Three reuse scenarios can be found: weights reuse in which a filter is reused $E \times F$ times to compute one output feature map; input reuse in which the input feature map is reused M times to compute the output feature maps; convolutional reuse when input pixels (i.e. activations) are reused to compute more than two output activations. Most accelerators are essentially designed to compute convolutional layers, which are computationally intensive, to explore various dataflows, reuse and parallelism opportunities.

While employing optimized dataflows or using different memory levels for different types of data were not enough to either overcome energy-efficiency issues or to improve performance, designers exploited application-related optimizations. For example, one can cite the use of specific approaches exploiting sparsity, in which a significant number of weights and/or input activations are zeros, and pruning [52], in which less useful connections in a CNN are removed (i.e. set to zero). According to [12], the number of zero values

can go up to 80% for weights and 70% for activations. Researchers leveraged this sparsity in hardware, to avoid zero-valued multiplications. Sparsity or pruning can be exploited using compression techniques, such as Compressed Sparse Row (CSR), Compressed Sparse Column (CSC) and Run Length Coding (RLC), to store data in a compressed format and thus reduce memory usage and allow direct access to the "next" non-zero valued data. This section shows a taxonomy of existing dataflows, then presents the state of the art of dedicated and programmable ASIC architectures as well as FPGA-based accelerators architectures.

1.2.1 Dataflows Taxonomy

The idea behind specific dataflows is to improve the energy of the system by articulating data movement on-chip. Dataflows usually reflect the type of data reuse employed in an accelerator. Researchers in [20] classified dataflows into four different categories as follows:

— Weight Stationary (WS) dataflow encourages weights reuse by keeping weights on-chip, in the RF of each PE. Input pixels are broadcasted to all PEs and partial sums, once generated, are accumulated spatially across the PE array. This maximizes weight and convolutional reuse. Works employing this dataflow are presented in [16], [48], and [64].

— Output Stationary (OS) dataflow encourages the locality of partial sums on-chip, where they are accumulated in the RF of each PE, thus maximizes the reuse of partial sums. Filter weights are broadcasted to all PEs while input pixels are distributed spatially across the computing array. Works presented in [51], [41], and [99] adopted the OS dataflow.

— Row Stationary (RS) dataflow encourages all types of data reuse (i.e. input pixels, filter weights and partial sums) in each PE. In RS dataflow, each row of the convolutional operation is mapped to a PE, thus each PE processes one row of the convolution. MIT researchers, who introduced this dataflow, employed it in the Eyeriss architecture [19].

— No Local Reuse (NLR) dataflow, as indicated by its name, does not leverage any data reuse. Therefore, no data is kept on-chip and the RFs are exchanged with a large global buffer for the whole PE array. For instance, DianNao [136] and the work in [22] use this dataflow.

1.2.2 Dedicated ASIC Architectures

ASIC stands for Application-Specific Integrated Circuits. Every designed dataflow-based ASIC chip is customized for a specific application and thus can no longer be modified or adjusted once fabricated. ASICs can be classified into various categories, such as Full Custom cells and Semi-Custom cells (such as Standard Cells). Full Custom cells imply designing a chip from scratch where most logic cells, circuits and the overall chip layout must be designed. This allows to design customized circuits optimized in terms of area and performance. The main drawback is the significant design effort and the high cost. On the other hand, Standard Cells involve pre-designed, pre-characterized and pre-tested logic cells as building blocks from a library of standard cells, which allows saving design time and reduce the risk compared to Full Custom. However, purchasing such libraries is expensive and designing standard cells is time extensive, which imply long manufacturing time.

The design flow of such architectures is complex and the production cost is quite extensive which means a significant time-to-market. Although the design flow of ASIC chips is pretty complex, these architectures have been one of the best targets to execute DL algorithms due to the resulting optimized and energy-efficient chip.

Due to the efficiency of data reuse in the implementation of DNNs, dataflow-based accelerators consist of the majority of existing DNN accelerators in the literature. Neuflow [100] and DianNao [22] are good examples of early dataflow architectures targeting DNN acceleration. They consist in computational units performing each an arithmetic operation (i.e. multiplication or accumulation) or a non-linear operation and connected by a reconfigurable interconnect system. These approaches reduce latency since intermediate data do not need to be stored, which also reduces the required on-chip memory. The main issue in such approaches is flexibility, since the implementation of newly emerging CNNs with different topologies or CNNs leveraging algorithmic optimizations cannot be easily implemented due to the fixed structure of such architectures. Other approaches allow more flexibility by implementing reconfigurable arrays, they also leverage algorithmic optimization, such as sparsity in specific dataflows (e.g. RS, OS, etc.).

Architectures exploiting sparsity These architectures are also dataflow-based and consist in an array of PEs. In general, PEs perform MAC operations and, in most cases, non-linear operations (e.g. ReLU). PEs are interconnected to exchange data with respect to the adopted dataflow. For instance, Eyeriss is an RS-dataflow-based architecture, de-

signed to accelerate convolutional layers [21]. It consists of an array of PEs with four levels of memories: external DRAM, On-chip GB, Inter-PE communication network and scratch pads. It leverages the sparsity in activations (due to the application of ReLU) on two levels to improve energy. The first level consists in skipping the computation of zero-valued activations, and the second one consists in compressing data using RLC. The Unique Weight CNN accelerator (UCNN) exploits sparsity in weights [54]. Unlike Eyeriss, it also leverages weights repetition (similar to weights sharing) to improve performance and reduce memory accesses. An improved version of Eyeriss, Eyeriss V2, runs compact sparse CNNs [23]. It exploits the sparsity in weights and activations. It uses the CSC compression technique to keep both weights and activations in a compressed format during computation and in memory. In addition, it presents more flexibility due to the hierarchical mesh-based NoC used to connect the PEs. SCNN targets sparse CNNs and uses CSC to compress both weights and activations [98]. It is optimized for convolutional layers and employs a dedicated dataflow to keep both weights and activations in a compressed format throughout the execution of the layers. Data is sent to a computing array (mainly multipliers), and the results are accumulated by means of a dedicated mesh-based NoC. Researchers revisiting SCNN designed the SparTen architecture to tackle the load imbalance problem found in SCNN [49]. The authors in [75] designed SqueezeFlow, an accelerator which compresses weights only using the RLC technique. They proposed the concise convolution rules to reduce the gap between sparse and dense DNN accelerators and to eliminate the processing of zero-valued weights and activations. EIE, presented in [52], in an accelerator for sparse networks and optimized for FC layers. It exploits dynamic sparsity in input pixels by skipping zero-valued activations. It also uses the CSC compression scheme to compress weights and reduces energy consumption by avoiding DRAM accesses.

Other approaches The work in [132], FixyNN, proposed a deeply optimized fixed structure for feature extraction to improve energy-efficiency while sacrificing flexibility. In this approach the Register Transfer Level (RTL) source code of the spatial architecture of the CNN topology is generated using fixed weight-optimized operators. To manage a performance-flexibility trade-off, the generation can be applied on a set of early layers of the input CNN. The last part of the architecture is more flexible and new applications can be implemented using the transfer learning technique. The authors also studied how to reuse generated partial structures for other deep learning algorithms using retraining.

Another array-based accelerator, called Squeezelerator, is designed with the idea of co-designing the algorithm and architecture to reduce energy consumption [69]. It alternates between two dataflows, Output Stationary and Weight Stationary, to suit the needs of each layers with no switching overhead.

Optimized ASIC architectures, exploiting various algorithmic optimizations, mostly depend on the employed dataflow. Updating the dataflow of the architecture is necessary to execute new DNN applications. However, changing or adapting the dataflow to the application requires redesigning the overall chip, which is pretty high-cost. Therefore, some approaches would rather design programmable and reconfigurable architectures to reduce this cost and enable more flexibility.

1.2.3 Programmable ASIC architectures

Most of the previously analyzed approaches suffer from a common problem which is flexibility. They are either dedicated to one feature, either a dataflow or a type of network (dense or sparse) since supporting more features makes the hardware more complex which makes the control more difficult. Therefore, more flexible (i.e. less dedicated) approaches can be found in the literature which are basically programmable DSP-like architectures (for Digital Signal Processing). These approaches rely on less dedicated hardware to be more flexible. For instance, PNeuro is a homogeneous accelerator formed by clusters of identical programmable processing elements working using the Single Instruction Multiple Data (SIMD) paradigm [13]. Orlando, a heterogeneous approach presented in [38], is composed of two parts: a dedicated one and a programmable one. The dedicated part consists of accelerators for convolutions and other image processing algorithms, while the programmable part consists of programmable DSP clusters which are in charge of other layers (such as pooling or fully connected ones). The common goal of these proposals is to efficiently implement variations of existing layers or to be future-proof in case new layers are introduced.

1.2.4 FPGA Architectures

FPGAs, for Field Programmable Gate Array, are characterized by their fine-grain reconfigurablity which facilitates the implementation of any design while tailoring each of its components (e.g. PEs, memory infrastructure and interconnect) to the exact requirements of the application. One of the most appealing property of FPGAs is their reconfigurabilty

which allows to modify the structure of the accelerator even on-the-fly. This extend FPGAs life cycle as technology and application requirements evolve. Another interesting aspect of FPGAs is their computation structure suited to application with intrinsic spatial parallelism as well as to dataflow algorithms, found in deep learning, such as CNNs. Compared to ASICs, FPGAs have lower performances and higher energy-consumption. Unlike ASICs, FPGAs have a less complex design flow and are cost-effective, thus they have a short time-to-market. Consequently, FPGAs are one of the first targets to implement accelerators for deep learning algorithms and are often used for ASIC prototyping. FPGA vendors offer their own tools and dedicated deep learning architectures to facilitate the use of these platforms, such as Vitis AI from Xilinx [133] or OpenVINO from Intel [62].

FPGAs Exploiting Sparsity

Similarly to the ASIC-based dedicated implementations, some FPGA-based accelerators exploits sparsity in CNNs to reduce energy consumption. For example, NullHop targets the acceleration of convolutional layers while leveraging sparsity in activations in computational and memory aspects [3]. Computational-wise, the sparsity is exploited to skip zero-valued computation with no overhead. Memory-wise, a compression technique is applied to reduce accesses. Researchers in [81] propose an accelerator exploiting sparsity in weights to improve performance. They employ a weight-oriented dataflow, in which each weight is processed separately to leverage the mathematical element-matrix multiplication.

FPGA for Dense CNNs

In addition to all these works, some researchers addressed the problem of designing hardware accelerators on FPGAs for CNNs differently. They employ a high-level template (i.e. written in C/C++ or SystemC) of the loop nest of convolutional layer. Most of these approaches leverage the High-Level-Synthesis (HLS) technique to generate an accelerator from a high-level code. The main focus is to optimize the loop nest by applying loop transformations which include tiling, unrolling and ordering techniques to improve throughput and resource usage.

The work in [136] proposes an optimized accelerator template for convolutional layers in which certain loops are tiled and reordered and some are unrolled to reduce data transfer and increase throughput. The template uses the same unrolling factors for different

convolutional layers applying the concept of one size fits all. Contrary to this approach, the work in [117] focuses on resource partitioning among a number of CLP (for Convolutional Layer Processor) tailored differently to maximize efficiency. The authors emphasis on the size of the CLP which is determined by the dimensions of the convolutional layer and the tiling parameters. In this approach layers are pipelined, different layers with similar dimensions can be assigned to the same CLP. Adjacent layers must be assigned to different CLPs to avoid idle time. The FPGA-based accelerator proposed in [82] aims at improving performance, reducing accesses to memories (on-chip and off-chip) and maximizing resource usage. The three loop transformations are deeply analyzed to find the right configuration with respect to the design goal (e.g. reduce memory requirements). [130] introduces a pipeline architecture with tiling techniques to improve computation.

Other approaches aims at implementing all the layers of a given CNN on-chip. For instance, the authors in [74] proposed an accelerator in which all the CNN layers are pipelined to allow a concurrent execution and increase throughput. A similar approach is presented in [10]. The layers of a given CNN are seen as building blocks, they are implemented separately and then assembled to build the accelerator. These building blocks are pipelined to increase throughput and improve performance.

Table 1.3 summarizes the most relevant CNN accelerators in the literature. It highlights the contributions of each work, the tested networks with the supported layers, and the adopted bit-precision. As can be seen, 16-bit fixed point is the trending bit-precision, employed for weights and activations, among most accelerators. The focus is on accelerating convolutional layers in very demanding networks in terms on memory and computation (such as AlexNet, VGG, etc.) while exploiting sparsity in activations or weights or both. Some other accelerators are more dedicated to embedded systems and thus target less complex networks, as PNeuro. Table 1.4 completes Table 1.3 with area and performance comparison. The reported performances and energy consumptions are not constant. They vary with respect to the deployed networks, more specifically to the deployed layers since the workload shifts from layer to layer according to the number of parameters and the number of pixels to be processed.

1.3 Conclusion

This chapter presented an overview of deep learning algorithms, with a focus on CNNs, and the embedded hardware architectures targeting their acceleration. From this chapter,

Architecture	Contributions	Tested Networks	Supported Layers	Bit-Precision (W & A)	Ref.
Eyeriss v2	Sparsity in Weights and activations: CSC compression format kept during processing	AlexNet, GoogleNet, MobileNet	Conv.	8-bit Fixed-Point	[23]
SqueezeFlow	Sparsity in Weights: RLC compression. Concise Conv. rules to support dense and sparse	AlexNet, VGG16, GoogleNet	Conv.	16-bit Fixed-Point	[75]
SparTen	Enhanced SCNN Tackle the load imbalance	AlexNet, VGG16, GoogleNet	Conv.	16-bit Fixed-Point	[49]
NullHop	Sparsity in activations: zero-skipping Compression.	VGG16-19, GigalNet, RoshamboNet, Face Detector	Conv., FC, Pool	16-bit Fixed-Point	[3]
UCNN	Sparsity in weights	LeNet, AlexNet, ResNet-50	Conv.	16-bit Fixed-Point	[54]
Squeezelarator	Alternates between OS and WS	SqueezeNet NN family	Conv., FC	16-bit Integer	[69]
PNeuro	SIMD programmable homogeneous architecture	Small test CNN, LeNet-5, Mnist,MobileNet-V1	Conv., FC, Pool	8-bit & 16-bit Integer	[13]
Orlando	Heterogeneous programmable SoC	AlexNet, VGG16	Conv., FC, Pool	16-bit Fixed-Point	[38]
Eyeriss	Sparsity in activations: RLC compression, Zero Skipping. Dedicated NoC, for RS dataflow	AlexNet, VGG16	Conv.	16-bit Fixed-Point	[19]
SCNN	Sparsity in weights and activations: CSC compression format kept during execution.	AlexNet, VGG16, GoogleNet	Conv.	16-bit Fixed-Point	[98]
EIE	Sparsity in weights and activations: CSC compression for weights zero-skipping for activations.	AlexNet, NeuralTalk, VGG16	FC	16-bit Fixed-Point	[52]

Table 1.3 – Comparison of state-of-the-art accelerators. The main contributions of each accelerator are highlighted as well as the tested networks, the supported layers and the bit-precision of weights (W) and activations (A).

Architecture and target technology	Platform	Area(mm^2)	Power (mW)	GOps/s	GMACS/W	Ref.
Eyeriss v2 - 65nm	ASIC	-	-	-	153.6	[23]
SqueezeFlow - 65nm	ASIC	4.8	536.09	-	-	[75]
SparTen - 45nm	ASIC	0.766	118.3	-	-	[49]
NullHop - 28nm	FPGA	8.1	2300	17.19	28.8	[3]
UCNN - 32nm	ASIC	-	-	-	-	[54]
Squeezelarator	ASIC	-	-	-	-	[69]
PNeuro - 28nm	ASIC	0.93	73	102.2	700	[13]
Orlando - 28nm	ASIC	34	39 (Dedicated part only)	2930	1465	[38]
Eyeriss - 65nm	ASIC	12.25	278	16.8 - 42	122.8	[19]
SCNN - 16nm	ASIC	7.9	-	-	-	[98]
EIE - 45nm	ASIC	40.8	590	1.6	85	[52]

Table 1.4 – Comparison of state-of-the-art accelerators in terms of area and performance. It highlights the target technology and platform as well as the resulting power (mW), performance (GOps/s), energy efficiency (GMACS/W) and area (mm^2).

one can conclude that designing accelerators for deep learning algorithms is a laborious work in which plenty of factors come into play. First, the design choices such as the choice of the target (e.g. FPGA, ASIC, etc.) and the trade-offs (e.g. flexibility-performance, area-performance, power-performance, etc.) which depend on the design goal, are hard to make. Second, designing an architecture is a time-consuming process, in which the

functioning of the design is not guaranteed. In addition, depending on the chosen target, it might be mandatory to develop a specific compiler to deploy the application to the designed architecture. Third, the quickly changing landscape of algorithms, especially deep learning algorithms, adds a thick layer of difficulty to the design process. Finally, the various existing neural network topologies and the constant evolution of these algorithms is making an existing gap between the hardware architectures and the evolving algorithms grow wider. In the next chapter, the state of the art of design approaches tackling these challenges are introduced, and we propose in this PhD thesis report an approach and compare it to existing work.

Chapter 2

State of the Art of Design Methodologies and Proposed Approach

Research efforts are put to reduce the gap between the applications and the hardware architectures to ease the design of hardware accelerators for DNNs. Instead of using hardware description languages such as VHDL (Very high speed integrated circuits Hardware Description Language) or Verilog, some new approaches focus on raising the level of abstraction of hardware descriptions by using high-level general purpose languages, such as Python, to describe hardware designs at RTL level. Other efforts are put on the High-Level Synthesis (HLS) approach to directly generate the RTL source code from the algorithmic description. Researchers pushed the level of abstraction further by developing end-to-end hardware generation frameworks based on these tools to automate and accelerate the design process. These frameworks are part of the EDA (Electronic Design Automation) field.

This chapter is organized as follows. Section 2.1 presents the state of the art of existing HDLs and HLS tools with a comparison between them. It also sketches a comparison between a standard design process and an automated one. Section 2.2 details the state of the art of existing accelerator generation frameworks while discussing the most popular and relevant approaches. Based on the challenges to tackle and the analysis of the state of the art of design methodologies for DNN accelerators, a novel approach is proposed in Section 2.3. A first comparison between the proposed approach and the existing work is introduced in Section 2.4. Finally, Section 2.5 concludes the chapter by recalling its main content and giving an opening on the following chapters.

2.1 Design Flows and Tools

To raise the level of abstraction of hardware descriptions, boost the productivity and reduce time-to-market of the hardware design processes, EDA (for Electronic Design Automation) and CAD (for Computer Aided Design) tools were introduced. The layer of abstraction of the design process is raised by including simulation and verification of the design functionality as well as synthesis, which automate the manual tasks. These tools rely on HDLs, such as VHDL and Verilog, which the designers use to specify the detailed structure of the architecture.

Even using these description languages, the design is still at a very low level, since the designer has to tightly model the behavior of the target logic circuit. The design process is thus error-prone and requires an advanced expertise in the hardware design field. To speed-up and simplify the design process, new design methodologies and relevant tools were introduced. For example, using high level programming languages (i.e. C/C++, Python, Scala, etc.), to reduce design time and errors was suggested by a study in [129].

Many researchers and developers tried to leverage the advantages of high-level programming languages in two different ways. The first one consists in adding hardware primitives to these languages to describe hardware components, turning them into hardware description languages (HDLs). The second one consists in inputting algorithmic descriptions and automatically generating a hardware implementation that respects design constraint and optimize design objectives, known as High-Level Synthesis (HLS). These two approaches are presented below.

2.1.1 High-Level Hardware Description Languages

Plenty of high-level HDLs were developed to ease the design task by adding the ability to some high-level programming languages to describe hardware components and structures. The main benefit of such approaches is the ability to leverage the advantages of high-level programming languages especially the object-oriented aspect which includes the reusability and modularity features.

For instance, Chisel is an open-source HDL based on Scala (a programming language characterized by its object-oriented and functional programming features) that outputs a synthesizable Verilog [7]. It allows to describe complex and configurable circuit generators with modern high-level programming language. The Flexible Internal Representation for RTL (FIRRTL) hardware compiler is employed in the design flow to optimize the Chisel-

generated circuits. FIRRTL also supports custom user-defined circuit transformations. The main flaws of this HDL is that it does not support directly Verilog logic values and the instantiations of a module must be wrapped in a module. Another open-source Scala-based hardware description language is SpinalHDL [121]. It is characterized by its interoperability with standard VHDL/Verilog-based EDA tools, its reduced code size and the clock domain safety. A generic design can be created and can also incorporate IPs (based on VHDL and Verilog) as blackboxes in the design.

Other open-source HDLs leverage the simplicity of the Python language to enable rapid prototyping of complex circuits. PyRTL, for example, concentrates on the entire toolchain and does not include any non-synthesizable hardware primitives [27]. It also supports imports and exports from and to Verilog. On the other hand, PyMTL is a unified framework for hardware generation, simulation and verification [80]. It leverages the Python runtime to simulate and co-simulate the design. However, it does not support a full system simulation with real OS support. As for MyHDL, it is suitable for IP block development and allows modeling hardware concurrency [86]. However, it is not well suited for accurate timing simulation.

The provided details above rely on what was mentioned by the developers. No more details can be given, since no experiments were done, in the scope of this work, to further characterize these tools. After all, these languages are interesting, but generating an architecture (Template) requires to precisely describe it.

2.1.2 High-Level Synthesis - HLS

HLS, known as behavioral synthesis, is a viable solution to RTL-design flows [34, 84, 31, 78]. It links the high-level application and the hardware architecture with the use of high-level languages, e.g. C/C++ or SystemC, to a synthesizable RTL which describes the design functionality using HDLs. This high-level of abstraction allows devising hardware accelerator without the need of advanced hardware skills [90]. Technology-wise, HLS offers the possibility to target both ASICs and FPGAs by easily synthesizing the generated RTL with both ASIC and FPGA toolchains.

The use of high-level languages in HLS significantly enhances design productivity. It also allows to easily reuse behavioural intellectual properties (BIPs), since they are not limited to fixed architectural and interface protocols. Consequently, re-targeting these IPs to different technologies can be rather smoothly done. Furthermore, the HLS community showed a specific interest in FPGA targets due to their reconfiguration aspect. This latter

fits well the advantages provided by HLS tools, especially the fast generation of IPs having different functionalities. Another interesting aspect of FPGAs is the efficient estimation of the design costs and performances. Therefore, combining reconfigurable architectures and HLS tools tackles the time-to-market problem [33].

In a recent study, different aspects of the HLS and traditional RTL design flows are analyzed and compared [70]. The compared aspects involve the quality of results as well as the productivity. According to this study, although the quality of results of traditional design flows is consistently better than that of academic and commercial HLS tools, the average development time is strongly reduced with HLS tools achieving 4× more productivity. The work in [70] illustrated these differences via a graph of box-plots presented in Figure 2.1. This figure shows the required time to perform different steps of both HLS-based and RTL-based design flows. From this figure, one can clearly notice that the HLS design flow requires less time than a traditional design process.

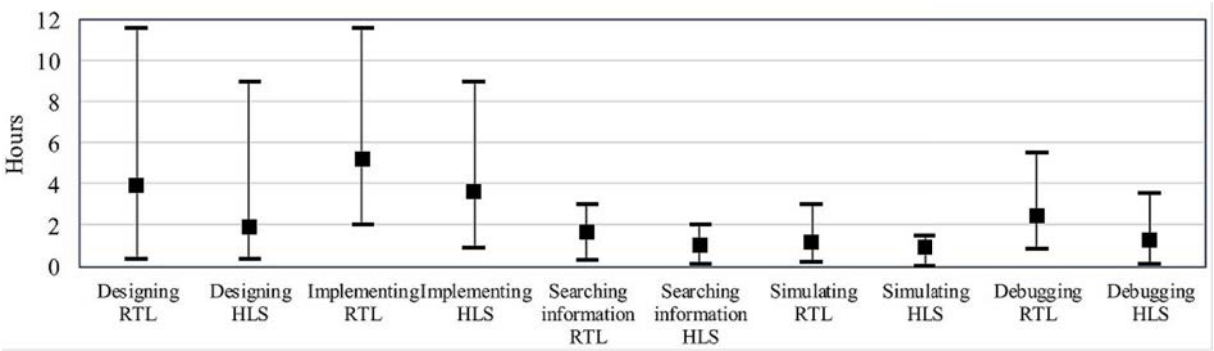


Figure 2.1 – Comparison of HLS and RTL design flows across different steps of the design process. The box-plots show the maximum, minimum and average time usage for the different evaluated features. Figure retrieved from [70].

Design flow of HLS tools The transformation from high-level source code to RTL representation requires certain steps. Therefore, the synthesis process relies on the following phases: code transformation, allocation, operation scheduling, datapath allocation, binding and controller generation. Figure 2.2 sketches the steps of HLS process. The code transformation step transforms the input source code into an equivalent representation of it (formal model in Figure 2.2). This representation exposes features of the code and enables simple optimizations. Once all operations and variables are extracted in the compilation step, the allocation specifies the number as well as the type of the required

hardware resources (e.g. functional units, storage components, connectivity elements) to output the hardware implementation. The allocation step selects from a library of RTL components to implement operations present in the code, such as multiplications, accumulations, etc. It is worth noting that this library comprises characteristics and metrics for each component to later estimate power, performance and area in synthesis processes. Then, each operation is assigned to a specific cycle in the operation scheduling step. Next, required hardware resources are defined in the datapath allocation phase to meet the design constraints. The binding step allows to share hardware components to satisfy design constraints. For instance, various variables can be mapped to the same storage unit if their lifecycles do not overlap. Finally, design decisions are applied in the controller generation step to produce a synthesizable RTL model.

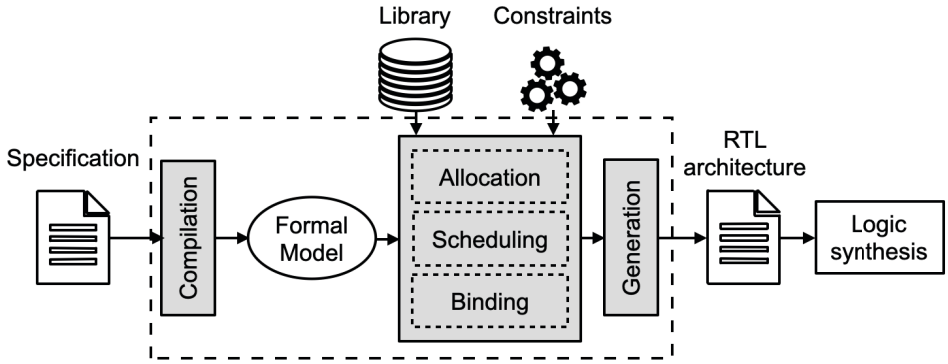


Figure 2.2 – Steps of the HLS design process (retrieved from [34] and modified).

HLS tools The growth in the FPGA market is one of the main reasons that encouraged the researchers and private enterprises to take particular interest in HLS tools and expand their usages [124]. Hence, myriad of academic and commercial HLS tools exists in the state of the art [90]. These tools are improved to provide efficient techniques to automatically devise customized hardware accelerators. On the academic side, BAMBU [102], developed at the Politecnico di Milano, takes as input a C code. It exploits the GNU compiler to support compiler-based optimizations and build new memory architecture to support high-level features of the C language, such as pointers and *structs* [103]. BAMBU supports various data types and is capable of generating Pareto-optimal implementations (regarding latency and resource usage). GAUT, developed at the University of South Brittany, is a free HLS tool [35]. This tool is capable of automatically

generating, from a C/C++ input description and a set of synthesis options, the hardware architecture of the accelerator while also generating the controller and the memory as well as the communication interfaces. DWARV, another academic tool, is based on CoSy commercial compiler [42] and developed at the Delft University of Technology [91]. The tool is scalable and can be easily extended with new optimizations. LEGUP, developed at the university of Toronto, takes as input a C code [11]. The tool is capable of synthesizing the input source code to hardware or to a hybrid system including a processor and a hardware accelerator. It exploits the open-source low level virtual machine (LLVM) compiler framework [71]. LEGUP allows an automated bitwidth reduction and is capable of synthesizing software threads into parallel hardware. On the commercial side, many HLS tools are widely used. For instance, Xilinx Vivado-HLS uses LLVM and accepts as inputs three different high-level languages, which are C, C++ and SystemC [128]. It has a complete design environment and includes various optimization options to refine the generation process. Vivado-HLS targets FPGAs and can generate hardware modules in VHDL, Verilog and SystemC. Siemens Catapult-C also accepts C/C++ and SystemC as input [15]. It targets both FPGAs and ASICs and offers flexibility in selecting external libraries and various design options. Differently from other tools, Bluespec takes as input Bluespec System Verilog (BSV) [60]. BSV is a Verilog-based HDL inspired by Haskell, where Verilog syntax is used to describe a set of rules to implement hardware modules. Thus, using this tool requires specific expertise.

Having introduced the literature of DNN accelerators as well as hardware generation tools and targeted technologies, the next section presents state-of-the-art approaches that aim at accelerating the design process of inference accelerators by leveraging high-level design techniques.

2.2 Hardware Generation Frameworks

Several approaches sought to simplify the design process of DNN inference accelerators by high-level design techniques, where most of the works target FPGAs. These approaches fall into three main categories: C-HLS-based templates, direct hardware generation frameworks and integrated hardware generation frameworks.

Regarding the first category, researchers build HLS-friendly convolutional templates by using C, C++ or SystemC programming languages to express the functionality. They

focus on tiling loops having a large bound and unrolling them when necessary or even when the bound is small, as well as loops reordering. These optimizations are employed to improve performance and optimize data transfer and memory accesses of the resulting convolutional accelerator. Some works adopt a *one size fits all* approach to deploy different convolutional layers on the same template like the work in [136]. The authors of this paper employed roofline model to optimize computation and memory accesses. They used the roofline model to identify all possible optimizations, including loop tiling parameters and unrolling pragmas, and then set the best solution for each layer. The main drawback of this approach is the use of uniform unrolling parameters for all layers which underutilizes hardware resources on the target FPGA. In addition, the employed model depends on the target template written in C++. A similar approach developed a template allowing the exploitation of various parallelism options to mainly improve performance [85]. This latter employed analytical performance and memory models, inspired by the roofline model in [136], which are used to enumerate all possible solutions. Solutions having the best communication to computation ratio will be picked. [104], like previous works, considers various code transformation options such as loop tiling and unrolling. It proposes an analytical model to evaluate each possible design point in terms of computation time, data transfer, DSP usage and bandwidth. Another interesting approach, based on a C++ template, is proposed in [117]. It focuses on resource partitioning on FPGAs to enhance efficiency and throughput. The same FPGA is partitioned to accommodate all layers of a CNN. The design space is explored via an iterative optimization process, based on two steps, which takes as input a performance target. The first step searches for the best partitioning, the tiling parameters of loops which are close to the computing part and the layers assignments. The second one partitions BRAM and minimizes the peak memory bandwidth by finding the right tiling parameters of memory-related loops. The exploration and the evaluations are facilitated via performance, BRAM and DSP models that depend on the template.

The second category, direct hardware generation frameworks, comprises some works aiming to generate inference accelerators by using as input a DNN configuration or description. For instance, [107] takes as input a custom configuration of a CNN and a training database chosen by the user. The configured CNN is then trained using MATLAB and the Verilog files are produced to later generate the RTL, which will be synthesized using Quartus II [61]. The main drawback of such approach is the absence of optimizations to exploit the parallelism in FPGA. In addition, neither evaluation models are used nor

feedback loops exist to optimize the design after synthesis. Furthermore, all CNN layers (e.g. convolution, pooling, fully connected) are hand-coded templates written in Verilog, which require adjustments when new layers become available. Few other works, unlike [107], propose direct hardware generation frameworks which generates a synthesizable high-level code from a DNN model description. The generated code is then fed to commercial high-level synthesis tool, such as Vivado-HLS and Catapult. For instance, [120] proposes a web-based framework which takes as input a CNN configuration along with its weights. Based on the network configuration, it provide an empirical estimation of resource usage in an optimization step. Next, it generates a synthesizable C-HLS code with Tcl files containing directives specific to Vivado-HLS. Then, using Vivado-HLS, the RTL is generated from the C-HLS code to further produce the bitstream. The optimization in this framework relies solely on resource estimation. Thus, it does not have any feedback loop to the optimization phase with the actual values of performance, energy consumption and resource utilization after the generation of the C code and even after the generation of the RTL. This can lead to less optimized solutions and put quite a lot of pressure on the estimation of computational resources since the optimization is performed only on these estimates. Furthermore, no compatibility with other hardware generation techniques is indicated. FP-DNN [50] is another framework for DNN accelerators generation. It takes as input a DNN model description in a protobuf format generated by TensorFlow and generates the corresponding RTL. FP-DNN is another approach presented in Figure 2.3. Unlike other approaches, it analyzes the topological structure of the input DNN and has an optimization step which is used to generate part of the C++ source code. Once the source code is compiled, a hardware generator module instantiates RTL-HLS hybrid templates which are then transformed into a hardware implementation using Catapult. However, it does not have any feedback on the actual performance of the produced architecture, thus doing a thorough optimization after generating the RTL is not possible. Figure 2.3 presents the framework flow.

The authors in [126] propose an automated design flow, fpgaConvnet, which generates a hardware implementation from a trained DNN model, in Torch or Caffe format. The structure of the input DNN is analyzed to be later transformed into a synchronous dataflow (SDF) model to efficiently explore the design space. The resulting SDF model is equivalent to a set of connected hardware building blocks (nodes represent hardware building blocks and arcs represent connections). It allows optimizations and transformations to be applied to explore various mapping strategies with respect to available resources

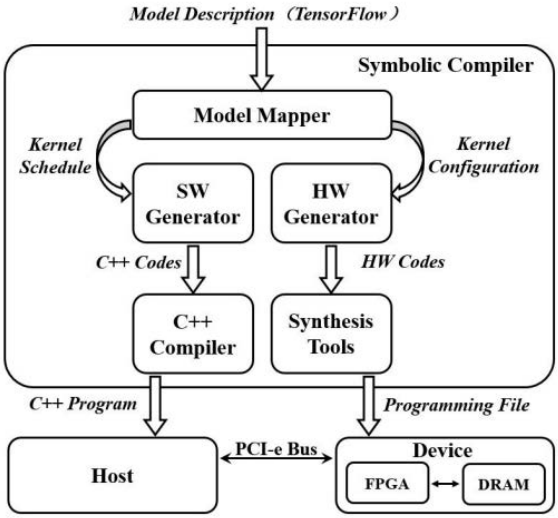


Figure 2.3 – Flow of the FP-DNN framework representing a direct hardware generation approach. Figure retrieved from [50].

on a target FPGA. The framework leverages the reconfigurability of FPGAs to partition the DNN model into a number of sub-networks mapped on different bitstreams. Once the mapping strategy is set, the hardware implementation is generated using Vivado-HLS. The optimization process targets latency and throughput only, resource usage is not optimized. In addition, once the FPGA is configured, the implementation can no longer be optimized.

Regarding the third and the last category, frameworks present a hardware generation step based on a thorough optimization process. For example, FINN [125], illustrated in Figure 2.4, accepts as input binary neural networks (BNN) and targets FPGAs. It produces a synthesizable C++ source code of a streaming architecture, optimized performance-wise. The source code is then fed to Vivado-HLS to generate the RTL based on a predefined library of basic blocks. FINN shows low modularity in terms of employed tools and targeted applications. In addition, the lack of optimization and feedback on actual performance makes it difficult to subsequently optimize the generated RTL. A more flexible open-source approach, hls4ml [43] sketched in Figure 2.5, automatically transforms a trained DNN model into a synthesizable representation adequate for HLS tools and targets FPGAs and ASICs; NN models can be trained using Pytorch, Keras, etc. Trained models can be optimized by using compression techniques before hardware generation. Unlike existing frameworks, hls4ml offers to use quantization-aware training

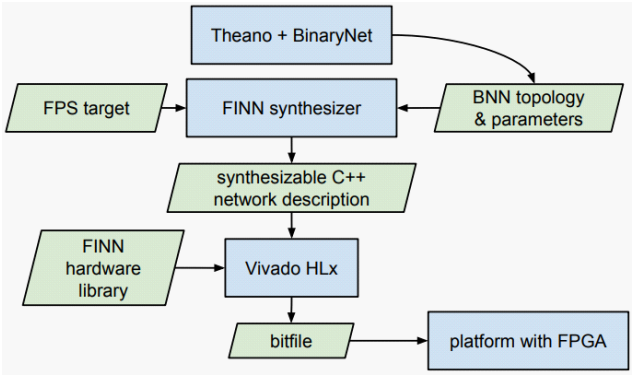


Figure 2.4 – Flow of the FINN framework representing a hardware generation approach for binarized networks. Figure retrieved from [125].

and quantization-aware pruning before generating the hardware in order to exploit the bit-level structure of FPGAs or even to target custom ASIC implementations. The DNN model is translated into an HLS-friendly high-level code, written in C/C++, which is then optimized to improve performance and reduce resource usage. Performance can be optimized via pipelining and resource usage via bit-width tailoring of each layer. The framework then leverages HLS tools to generate an IP core representing the entire NN. Unfortunately, both FINN and hls4ml rely on internal feedback loops of the employed HLS-tools, such as Vivado-HLS and Catapult, to optimize the hardware implementation of the DNN model. Thus, no further optimizations can be done after synthesizing the RTL. Another interesting framework targeting ASIC implementations is MAGNet [127]

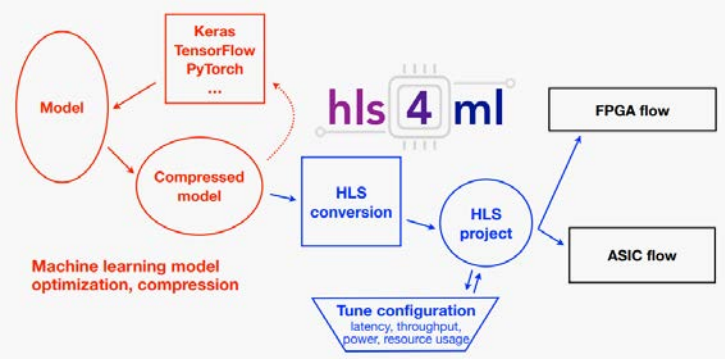


Figure 2.5 – Flow of the hls4ml framework which uses quantization-aware training and quantization-aware pruning on a higher level. After a DNN optimization phase (in red), the DNN model is transformed into an HLS code and both FPGA and ASIC architectures can be targeted. Figure retrieved from [43].

presented in Figure 2.6. It proposes a generator of DNN accelerators based on a specific architectural template. It allows a co-optimization between hardware parameters, tiling strategies and the chosen DNN. Contrary to existing state-of-the-art frameworks, MAGNet offer an optimization step based on real power, performance and area (PPA) values of the generated architecture. Unfortunately, since the flow depends on the target template, the design space exploration (DSE) also depends on it. In addition, this approach uses HLS to only generate the RTL of the architecture using elementary low-level components described at high-level. It does not leverage any algorithmic representation to fully exploit the HLS benefits. Moreover, the exploration is based on systematic feedback loop on PPA after RTL synthesis, which lengthen optimization time as well as the overall design time of an optimized architecture.

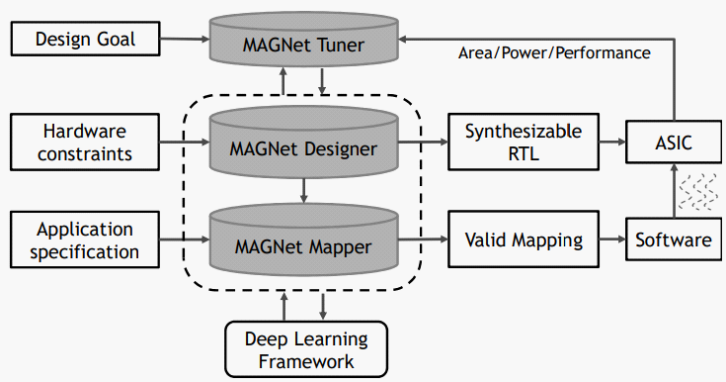


Figure 2.6 – Flow of the MAGNet framework that consists in Designer that generates the architecture, a Mapper that maps the application to the architecture and a Tuner that optimizes the architecture based on a design goal. Figure retrieved from [127].

In summary, most of the proposed hardware generation flows are direct flows with little or no optimization, and no feedback loop connected to the real PPA values of the resulting architecture. These flows lack modularity either in terms of generation tools or in terms of targeted DNN applications. Thereafter, the following section presents the proposed approach, SHEFTENN, a Software and Hardware Exploration Framework Targeting Embedded Neural Networks that partially addresses the weak points of the state of the art. It is an approach for hardware generation of efficient neural accelerators based on high-level algorithmic descriptions, which aims at reducing the gap between the software and the hardware.

2.3 Proposed Methodology Overview

In this section, a novel methodology is proposed to overcome some hardware design process issues identified in the state of the art. Designing an accelerator for DNNs requires a deep understanding of the application itself. Unfortunately, DNN algorithms are quite different in various ways. Each DNN model has its unique topology, which consists of a stacking of various layers. These layers have different types (e.g. standard and depthwise convolutions, max and average pooling, etc.) and shapes (3x3, 5x5, etc.), which makes the computational and memory requirements inconsistent from one DNN to another as well as from one layer to another in the same model. In addition, applying high-level optimizations on DNNs, such as quantization and pruning, have an impact on the design choices to make while devising a DNN accelerator. Unfortunately, the wide landscape of DNN applications is hard to study and master by hardware architects, since a lot of design choices depend on the DNN algorithm itself. Therefore, a first step to design a DNN accelerator is to automatically and meticulously characterize the algorithm to ease the design process.

Furthermore, conceiving a DNN accelerator by using VHDL-like hardware description languages is complex and time consuming, which lengthen the time-to-market. Hence, high-level design methodologies are needed to facilitate the design process and automatically generate the hardware architecture, while allowing enough degrees of liberty to explore the design space. However, the resulting design space is relatively wide, which makes the exploration of all potential solutions quite complex and laborious.

Consequently, automating the exploration is required to facilitate the decision-making that requires lots of compromises to reach the desired design goal(s).

All these discussed steps are put in the proposed methodology which is implemented in a framework called SHEFTENN (Software and Hardware Exploration Framework Targeting Embedded Neural Networks). Figure 2.7 sketches a high-level representation of the SHEFTENN flow.

This methodology, as it can be seen in Figure 2.7, is based on three interdependent steps cooperating to generate optimized hardware accelerators.

The first step of the approach is the *Characterization* that processes the DNN specifications to characterize the DNN attributes and behavior. This step aims at producing relevant metrics and augmented specifications. The Characterization step is in charge of bringing the DNN algorithm closer to the hardware part, by calculating target-agnostic

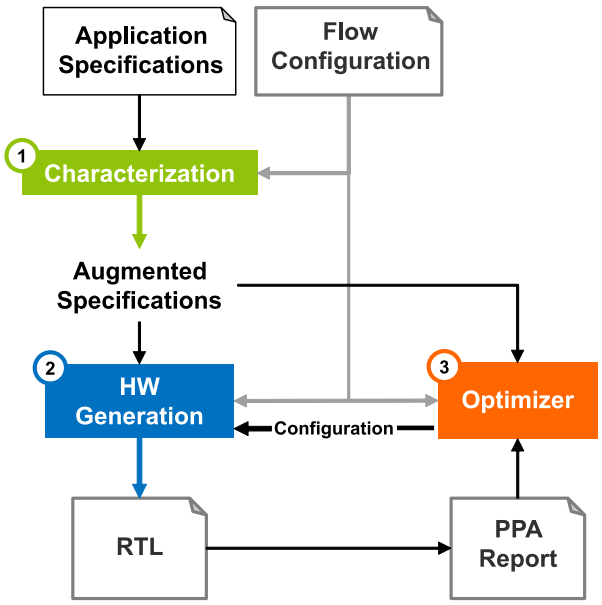


Figure 2.7 – The flow of the proposed approach with its three main steps: Characterization, HW Generation, Optimizer.

metrics, which are also hardware-aware, such as the number of MAC operations and the memory requirements of weights and input feature maps. Additional studies can be carried out, such as determining the reuse ratio per layer, or unfolding the internal graph structure of the DNN algorithm in a simplified way [5] to outline the communications between the various types of layers forming this DNN, and therefore, between the accelerator hardware operators in case available.

The second step of the approach is the *Hardware Generation*. This step abstracts the hardware by employing high-level design techniques and RTL source code generation. It exploits augmented specifications, computed in the previous step, and generates a synthesizable RTL starting from high-level network description. After synthesizing the RTL, the figures of merit of the designed accelerator are obtained and analyzed. In this context, the traditional PPA (Performance, Power, Area) metrics are designated by these figures of merit. For FPGA targets, the area can be substituted by the resources (LUTs, BRAMs, etc.) utilized for the implementation of the accelerator. Regarding the PPA report, it is generated by classical synthesis tools, whether the target is an ASIC (e.g. Design Compiler from Synopsys) or an FPGA (e.g. Vivado from Xilinx).

The third and last step of the proposed methodology is the *Optimizer*. This step performs the exploration of the accelerator design space. It relies on one or more opti-

mization algorithms that exploit the characterization results to optimize the high-level representation of the DNN algorithm, and thus the RTL generation. It identifies various configurations to be generated by the Hardware Generation step. The optimizer computes different configurations, which are fed to the Hardware Generation step to be generated, as it can be seen in Figure 2.7. Two different methods are used to generate these configurations: the first one uses simple augmented-specifications-based hints, the second one depends on the optimization algorithm itself. As a matter of fact, common and efficient code transformation strategies can be deduced from the specifications' analysis. These strategies can be applied by default to generate relevant configurations. The remaining design space can be explored by using the optimization algorithm to find more efficient mixture of code transformations or tool-specific directives. The Optimizer step can rely on two exploration loops to avoid the long exploration time: a long loop that uses results derived from the synthesis step, and a short loop that relies on internal PPA high-level estimation models. An exploration speed and precision tradeoff can be defined from the ratio between the long exploration loop and the short one.

2.4 Comparison between SHEFTENN and the State of the Art

This section presents a first comparison between the present work and some of the most relevant frameworks in the state of the art, summarized in Table 2.1.

This table shows that most of the proposed approaches, unlike SHEFTENN, offer direct HLS-based hardware generation frameworks with no feedback loops after RTL synthesis to optimize various design aspects (i.e. power, performance and area) by using real values. MAGNet offers a feedback loop on PPA only after synthesis. However, it employs HLS to generate individual architectural components described at high-level but not the algorithm.

Some hardware generation frameworks include an optimization step before the RTL synthesis, but only focus on some aspects of the design, which is not the case for SHEFTENN that aims at optimizing the PPA of the design by using a 2-steps optimization process. For instance, FINN optimizes the implementation performance-wise only and fpgaConvNet focus on optimizing performance and throughput by using a DSE that leverages SDF. The framework proposed by Solazzo et al. only optimizes resource usage. It is worth noting that hls4ml rely on the internal feedback loops of the employed HLS tool, such

Frameworks	Contributions	Input	Template	HLS-tool	Optimizations	Ref.
hls4ml	Hardware Generator Quantization-aware training quantization-aware pruning Targets FPGAs & ASICs	Trained DNN	No	Vivado-HLS, Catapult	No feedback after RTL synthesis; Relies on internal feedback loops of HLS-tools; optimize PPA	[43]
MAGNet	Template-based Hardware Generator; Targets ASICs	Design goal Hardware constraints DNN Specifications	Yes	Catapult	Bayesian optimization strategy for DSE; Feedback after RTL synthesis; Optimize PPA	[127]
Rivera-Acosta et al.	Direct Hardware Generator	CNN configuration Database	Yes	Quartus II	No feedback loops; No exploitation of parallelism in FPGAs	[107]
fpgaConvNet	Exploits SDF for efficient DSE; Targets FPGAs	Trained DNN model	No	Vivado-HLS	Optimization of performance & throughput only; No feedback loop after RTL synthesis	[126]
FP-DNN	Instantiate hybrid RTL-HLS template; Topology analysis	Trained DNN	Yes	Catapult	No feedback loop after RTL synthesis	[50]
FINN	Targets BNN; Exploits a predefined library for RTL generation	BNN model	No	Vivado-HLS	No feedback loop after RTL synthesis; Performance optimization	[125]
Solazzo et al.	Direct Hardware generator; Empirical estimation models for FPGA resources	CNN configuration & Weights	No	Vivado-HLS	Optimization of FPGA resources based on estimation models; No feedback loop after RTL synthesis	[120]
SHEFTENN	Hardware generator; estimation models for FPGA resources; Quantization-aware hardware generation	Trained CNN	No	Vivado-HLS	2-steps optimization process of FPGA resources and latency based on estimation models & real synthesis	-

Table 2.1 – Comparison with state-of-the-art hardware generation frameworks.

as Xilinx Vivado-HLS and Siemens Catapult, to optimize the PPA of the accelerator implementation.

On the other hand, almost all of these frameworks do not tackle the large evaluation and design time of DNN accelerators, which is one of the SHEFTENN contributions that introduces estimation models for PPA to reduce the evaluation time of potential optimization solutions, and thus the overall design time. Only the Solazzo et al. work presented empirical estimation models for resource usage on FPGA targets, while ignoring performance and power. However, these estimators are limited to few CNN configurations and cannot encompass literature DNNs.

The majority of the proposed hardware generation frameworks, contrary to SHEFTENN, are limited to a certain DNN types or configurations and do not consider hardware-aware algorithmic optimizations. The frameworks introduced by Rivera-Acosta et al. and Solazzo et al. can only support few layers configurations. FINN accepts binarized DNNs only, and therefore cannot take into account standard DNNs. Some approaches, such as MAGNet, fpgaConvNet and FP-DNN, consider state-of-the-art DNNs having large

topologies. However, they do not explore the quantization-aware hardware generation for quantified DNNs.

SHEFTENN, unlike other state-of-the-art approaches, is an automated tool targeting FPGAs, and is independent from a target template or architecture. It allows to quickly design DNN accelerators through the combination of a characterization step that reduces the design space, and an RTL generation step that leverages HLS. An optimizer step optimizes the implementation of the DNN accelerator thanks to an automatic exploration of the design space, which uses a hybrid optimization process of RTL source code generation based on a C-HLS source code, as explained earlier. This process relies on two separate algorithms, the first one optimizes each layer individually, and the second one optimizes the whole CNN. Such process offers a trade-off between quality and rapidity of exploration using estimation models (for both performance and resource usage) and real synthesis. In addition, unlike other approaches, SHEFTENN supports quantified networks thanks to HLS.

2.5 Conclusion

This chapter presented the state of the art of the EDA field while providing a comparison between standard and high-level hardware design processes. Furthermore, high-level hardware generation frameworks were detailed discussed. Then, the proposed approach, subject of this thesis, was introduced. It offers a solution to cope with the rapidly evolving DNN applications and to reduce the time and complexity of the overall design process. A qualitative comparison was then provided to position the suggested approach with respect to existing work. The three following chapters will provide a detailed description of each step of the proposed approach as well as early results assessing their efficiency. A last chapter will provide complete detailed results of the approach implementation.

Chapter 3

Characterization and Metrics Analysis

One of the important idea presented in the previous chapters is that the design of an efficient DNN accelerator requires a deep knowledge of the algorithm. This is critical in design flow relying on algorithm/architecture matching. Automatically obtaining this knowledge is the purpose of the first step of the proposed methodology introduced in Section 2.3. The aim of the so-called characterization step is to extract reliable metrics from neural network descriptions, and analyze them to provide relevant information on their structures and requirements to drive efficient implementation strategies.

The purpose of this chapter is to detail the characterization step of the proposed methodology (see Figure 3.1) and to present some early results.

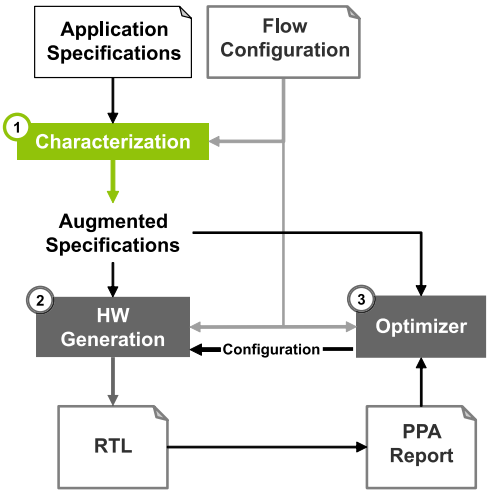


Figure 3.1 – The flow of the proposed approach, with a focus on the Characterization module.

This chapter is organized as follows. Section 3.1 presents the characterization flow. Section 3.2 details the metrics computation phase. Section 3.3 explains the metrics analysis

phase. Section 3.4 shows the results of the characterization module on various state-of-the-Art neural networks. And finally section 3.5 sketches a conclusion of the chapter.

3.1 Characterization step Overview

The characterization step represents the first step of the proposed methodology [5]. Based on a modular implementation, it can be adapted and enhanced to characterize newly available DNN algorithms. This step characterizes and analyzes deep learning algorithms on several levels and outputs DNN specifications augmented with metrics and related analysis, called *Augmented Specifications*. An overview of the characterization module is presented in Figure 3.2.

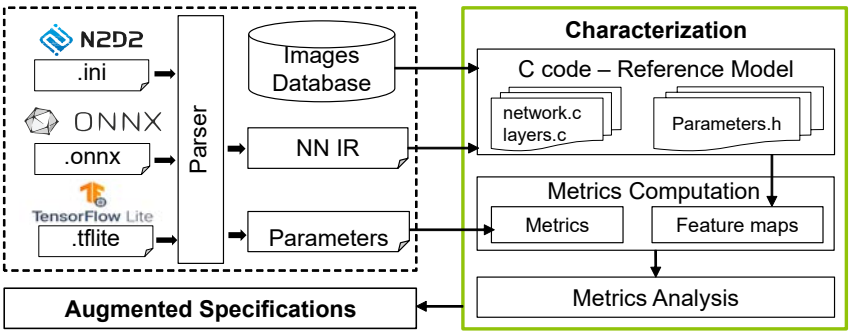


Figure 3.2 – Overview of the characterization flow.

A preliminary step first parses the input CNN description, such as a *.tflite* TensorFlowLite framework enabling deep learning models on mobile and edge devices [1], an *.onnx* from ONNX (Open Neural Network Exchange) the open defacto standard for representing machine learning models [9] or an *.ini* from N2D2 (Neural Network Design and Deployment) framework [87]. Listing 3.1 presents a description of two convolutional layers.

```
[conv1] conv_def
Input = sp        ;#input image
KernelHeight = 3
KernelWidth = 3
NbOutputs = 8
Stride= 2
Padding = 1

[conv1_3x3] conv_def
Input = conv1
KernelHeight = 3
KernelWidth = 3
NbOutputs = 16
Stride = 1
Padding= 0
```

Listing 3.1 – Configuration of two convolutional layers in the N2D2 *.ini* file.

The step also transforms the input description to an Internal Representation (IR), including layers types and configurations, and optionally extracts the weights of the CNN and exports them into arrays for advanced analysis related to their distribution and their sparsity. The IR is a list of objects, where each object represents a layer with its configuration and all its characteristics. By means of this IR, the characterization step generates a C-based implementation of the DNN to dynamically characterize and analyze the sparsity of the input feature maps using a test or validation dataset. This C-based implementation includes the C-code of each layer composing the input DNN, such as convolutional (Conv.) layers, pooling (Pool.) layers and even fully connected (FC) layers. This implementation is the C reference (or "golden") model of the DNN for the framework. This C model is HLS-friendly and can be used as it is for hardware generation as a reference implementation.

The characterization step later computes target-agnostic/hardware-aware metrics such as the number of MAC operations and the memory requirements and aggregates this information with the DNN parameters- and feature maps-related data metrics (if available in the description). Then, it performs an advanced analysis to generate application-dependent **Augmented Specifications** such as potential mapping strategies and dataflows for dataflow-based accelerators. Figure 3.3 illustrates a detailed representation of the characterization flow while highlighting the metrics computation and analysis results. More details on these two points are presented below.

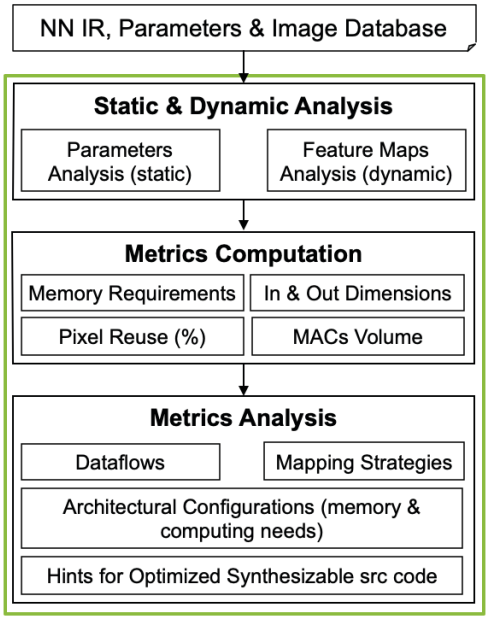


Figure 3.3 – Detailed flow of the characterization step focusing on the metrics computation and analysis.

3.1.1 Metrics Computation

Metrics Computation highlighted in Figure 3.3, is the first phase to gain thorough knowledge about the behavior of the DNN by analyzing (optionally) weights and feature maps. A second step takes the IR of the given CNN which holds the dimension of the input data (if available), the topology of the network, the type of each layer as well as its corresponding configuration (i.e. hyper-parameters).

Figure 3.4 sketches the function of the this step. One can see that the IR is used to compute hardware-aware/target-agnostic metrics. The the three main tasks are performed concurrently. The first one (on the left) checks if the parameters (i.e. weights) are available to simply count them on a layer by layer basis. If they are not available, the the number of weights is computed per layer based on the configuration of each layer, called hyper-parameters, which consist in the dimensions of the layers (kernel width, height and depth). The number of parameters is calculated using Equation 1.3 in Section 1.1.2.

The task in the middle checks if the input and output dimensions are included and extracts them. Else, computing the dimensions is a must using Equation 1.2 in Section 1.1.2. Once dimensions are obtained, the width and depth are extracted and the amount of activations is computed. In addition to pure metrics computation, which can be provided

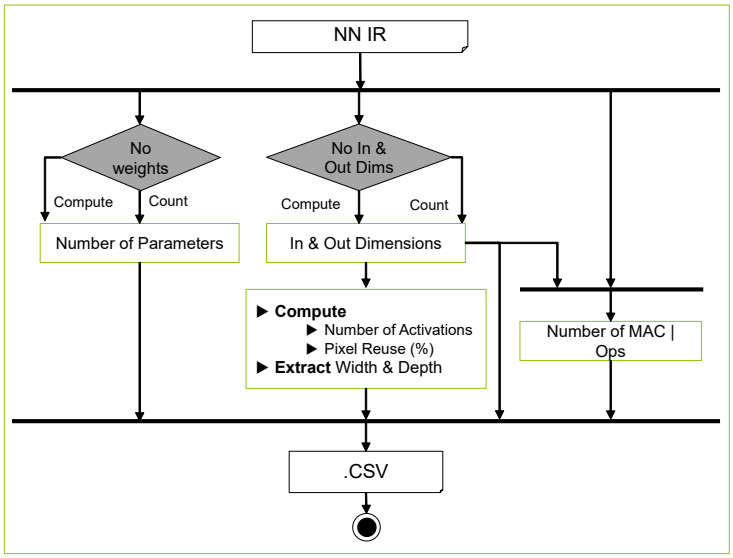


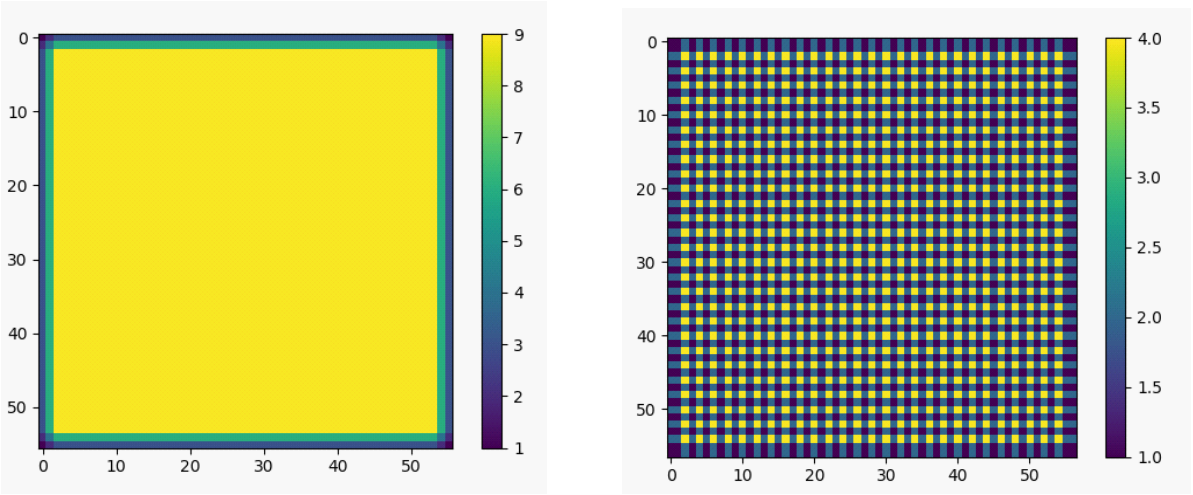
Figure 3.4 – Overview of the Metrics Computation.

by numerous machine learning frameworks, more sophisticated metrics related to data reuse such as the percentage of pixel reuse in the input feature map of convolutional and pooling layers can be computed. The pixel reuse in a feature map is computed based on the fact that a convolution is a sliding window over that map. Figure 3.5 shows two different reuse maps, having a 56x56 size, generated using a stride of 1 in Figure 3.5(a) and a stride of 2 in Figure 3.5(b). As it can be seen, the pixel reuse is impacted by the stride, a hyper-parameter expressing the step of the filter over the input image.

The third task on the right of Figure 3.4 consists in computing the number of operations with respect to the type of each layer, expressed in MACs (Multiply-ACcumulate) for convolutional layers, and Ops (Operations) for pooling layers. The computation depends on the input and output dimensions. It is worth mentioning that the computation of the number of MAC differs according to the type of the convolutional layer. For instance, the number of MAC for a standard convolution can be computed using Equation 1.4. A depth-wise (dw) convolution applies a filter on each channel of the image. Therefore, the number of MAC is $N\times$ lower compared to a standard convolution. In the dw convolution, the number of MACs is therefore expressed by Equation 3.1.

$$MAC_{dw} = R \times C \times K_x \times K_y \times M \tag{3.1}$$

This information is stored in a *.csv file* and the appropriate charts are then generated



(a) 56x56 image convolved with a 3x3 filter and a stride $s = 1$

(b) 56x56 image convolved with a 3x3 filter and a stride $s = 2$

Figure 3.5 –
Reuse maps with different stride values.

to be later analyzed in the Metrics Analysis phase.

3.1.2 Metrics Analysis

The analysis phase, represented by the green blocks in Figure 3.4, takes the resulted metrics as input to drive implementations strategies and provides hardware-based hints for efficient implementation. This phase highlights the potential use of each metric, listed earlier, in designing an optimized hardware accelerator. It is worth noting that each metric could be beneficial on several levels in designing or configuring an architecture or even a template-based architecture. Below a detailed description of potential utilization of each of the previously computed metrics.

Bit-precision It is an important metric to consider, especially when dealing with quantized networks. It allows to tailor the hardware architecture to the exact needs of the DNN by setting the required number of bits for weights and features. It has an significant impact on various design aspect: area, performance and power. Quantized DNNs with small bit-width allow to reduce the memory required to store the model, the size of the operators and thus, the area of the resulting accelerator.

Input/output feature maps and kernels dimensions This metric, in addition to the kernels dimensions, is the backbone for computing the number of MACs operations. Besides, it helps estimating the bandwidth and the memory requirements, in bytes, for each layer with respect to the used bit-precision for configurable architectures and templates. On a higher level, it helps computing the pixel-reuse percentage in each layer. In addition, it helps setting threshold values for tiling parameters for C-based convolutional templates especially in HLS approaches. For instance, a C-based convolution consists in six nested loops, represented in Listing 3.2, where each loop has its own loop bound which is a dimension in a convolutional layer. Each loop iterates over a dimension to complete the convolution process. Usually loops with large bounds are optimized to reduce the required on-chip memory by applying loop transformations such as tiling.

```
ifmap[N][(R-1)*S+K][(C-1)*S+K]  //input feature maps
outfmap[M][R][C]  //output feature maps
weights[M][N][Kx][Ky]  //filter weights
l0: for(r=0; r<R; r++)  //output X
  l1: for(c=0; c<C; c++)  //output Y
    l2: for(m=0; m<M; m++)  //nb outputs
      l3: for(n=0; n<N; n++)  //nb channels
        l4: for(kx=0; kx<Kx; kx++)  // kernel X
          l5: for(ky=0; ky<Ky; ky++)  // kernel Y
            wx=weights[m][n][kx][ky]
            ix=ifmap[n][S*r+kx][S*c+ky]
            outfmap[m][r][c]+=wx*ix
```

Listing 3.2 – Pseudo-code of a convolutional layer.

Setting threshold values, as in Equations (3.2), is essential to limit the search space of tiling parameters T_r for $l0$, T_c for $l1$, T_n for $l3$ and T_m for $l2$. The minimum threshold values for the width T_r and the height T_c are set to the kernel height Ky and width Kx respectively. And their maximum threshold values are set to their respective loop bounds.

$$\begin{aligned} Kx \leq Tr \leq R \\ Ky \leq Tc \leq C \\ tn \leq Tn \leq N \\ tm \leq Tm \leq M \end{aligned} \tag{3.2}$$

Memory requirements This metric helps to quantify memory needs of a DNN. It gives an estimation of the required memory when designing a hardware accelerator. This metric includes both weights and input activations, where each one is computed based on different dimensions. The required weights' memory can be computed for each layer by using the convolutional kernel. The activations' memory can be computed using the input feature maps dimensions. Equation 1.3 and 1.5 presented in Section 1.1.2 sketches memory requirements for both weights and activations for different types of layers. Figure 3.6 and Figure 3.7 present the number of parameters (in byte) for each layer of MobileNet-V1 and SqueezeNet-V1.1 respectively, considering an 8-bit precision.

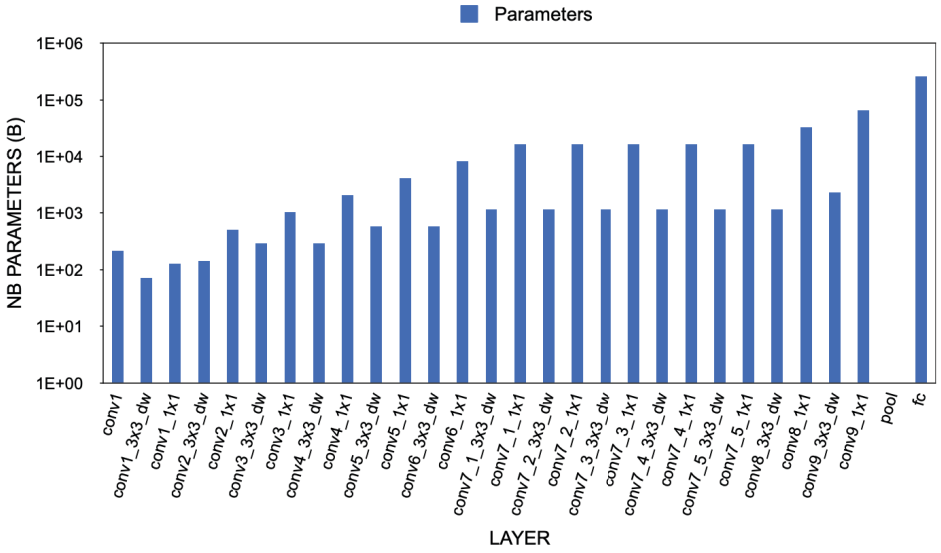


Figure 3.6 – Number of weights (parameters) of each layer of MobileNet-V1.

Number of MACs Its computation depends on the input/output feature maps dimensions, kernels dimensions, as well as on the type of the layer. This metric gives an estimation of the required number of hardware MAC operators based on a target rate which could represent a number of DSPs, for DSP-like architectures, such as Orlando [38], or Processing Elements found in Eyeriss [19], SqueezeFlow [75], etc. Furthermore, since the computation of this metric relies on the input/output dimensions and on the configuration of the layer, the number of MACs can be employed to create analytical models to estimate the latency of C-based templates of convolutional accelerators on FPGAs like the works in [136] and [117].

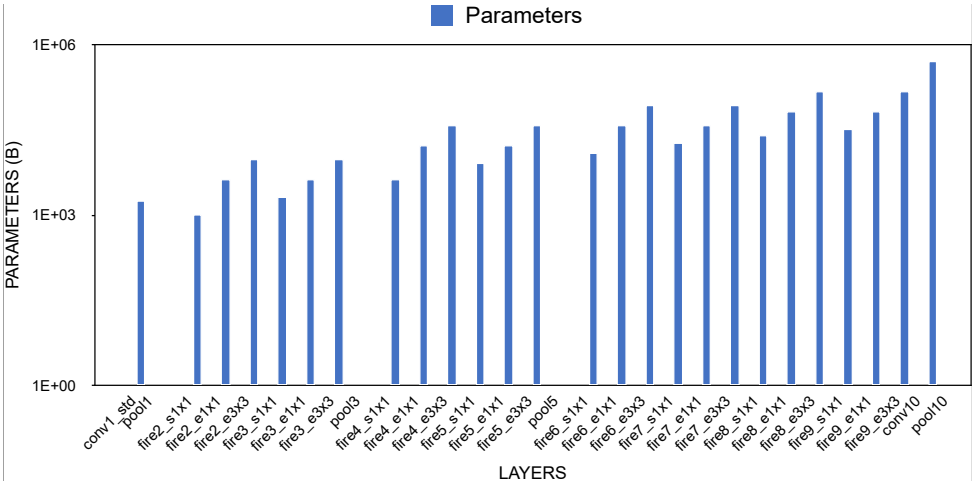


Figure 3.7 – Number of weights (parameters) of each layer of SqueezeNet-V1.1.

Data-Reuse percentage par layer It is an interesting metric from which several dataflows can be derived, such as the input stationary (IS) dataflow used in [17]. Thus, it can be leveraged in dataflow-based architectures. This metric is key to determine which dataflow to employ in each layer of the input network and thus the suitable mapping strategy to implement. It can also be combined with other metrics as explained next.

Width and Depth comparison Monitoring the width and depth evolution throughout the CNN, such as Figure 3.8 that presents a layer by layer characterization of the width and depth of the input feature maps in MobileNet-V1, gives some hints on possible parallelism and mapping opportunities in each layer.

For example Figure 3.9 illustrates potential mapping scenarios derived from this metric. The 3D feature map (at the top) having its width greater than its depth implies a 2D mapping to promote data locality and pixel reuse. In the opposite scenario (at the bottom), a depthwise mapping is more likely to encourage channel-wise parallelism. However, this metric alone is incomplete to confirm those mapping choices.

Connection graph It is an essential and complex result, obtained by analyzing the DNN metrics, for an optimized mapping. For instance, Figure 3.10 represents the connection graph of MobileNet-V1 in terms of the layers, the shape of their kernels and their stride value. For example, the number of connections between each type of layer is presented. It also highlights, in green, the layers having the same input and output volume of

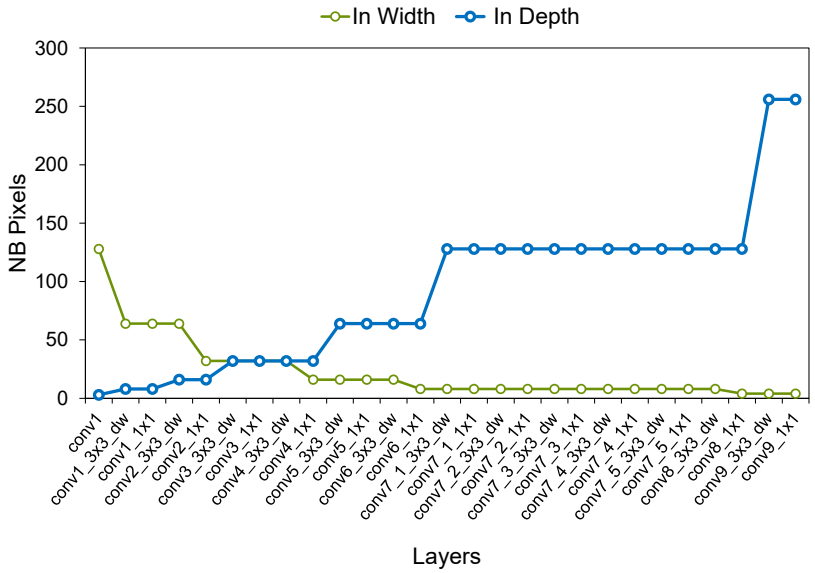


Figure 3.8 – A per-layer characterization of the width (In Width) and depth (In Depth) of the input feature maps in MobileNet-V1.

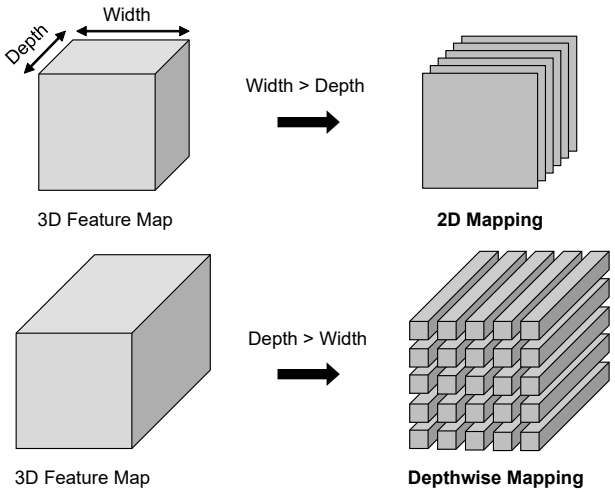


Figure 3.9 – Mapping opportunities derived from the shape of the input feature map.

data in terms of number of pixels. A connection graph can be used to map similar layers on the same hardware resources to reduce area. However, its usage depends on the adopted approach. For instance, some HLS-based approaches tend to design a monolithic accelerator capable of hosting all the layers of a CNN on the target FPGA, creating thereby

an RTL module for each layer and resulting in a large area like the work in [10]. This metric can help limit the resource usage by assigning identical layers or even similar ones, in terms of memory requirements and kernel dimensions, to the same hardware resources.

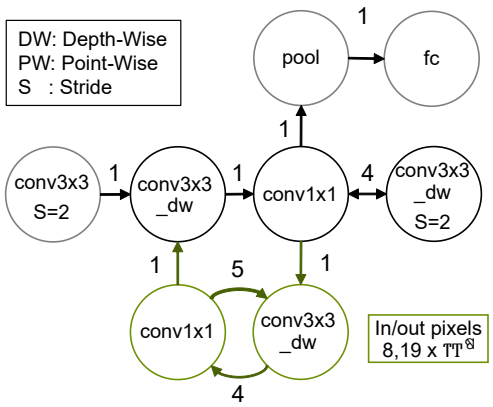


Figure 3.10 – Overview of the connections in MobileNet-V1 organized according to the types of layers.

Parameters sparsity Shows the weights distribution in each layer, the number of positive and negative values as well as the number of zero weights. For example, Figure 3.11 presents the weights distribution of the *conv9_1×1* convolutional layer of MobileNet-V1. One can see that the number of zero-valued weights is significant compared to weights with negative and positive values. It consists of 30.31% of the overall weights.

The number of zeros is a key information to help identify layers in which memory requirements can be reduced by applying compression techniques. In addition, computational needs can be reduced by skipping zero-valued computations as done in EIE [52], Eyeriss-V2 [18] and NullHop [3]. Figure 3.12 shows the percentage of zeros in each layer of MobileNet-V1. It can be seen that the percentage of zeros varies throughout the DNN due to the variety of layers shapes. In addition, 1x1 convolutions have higher percentage of sparsity compared to 3x3 convolutions, which is related to the significant number of weights in those layers.

Dynamic activations sparsity The sparsity of activations comes from the use of some activation function (e.g. ReLU) after a convolutional layer. It clamps negative values to zero. The use of this metric is the same as the previous one, however the only difference is that sparse weights are fixed after the training phase. Therefore, a dynamic sparsity study, using the learning dataset (used to train the network), the validation one or a

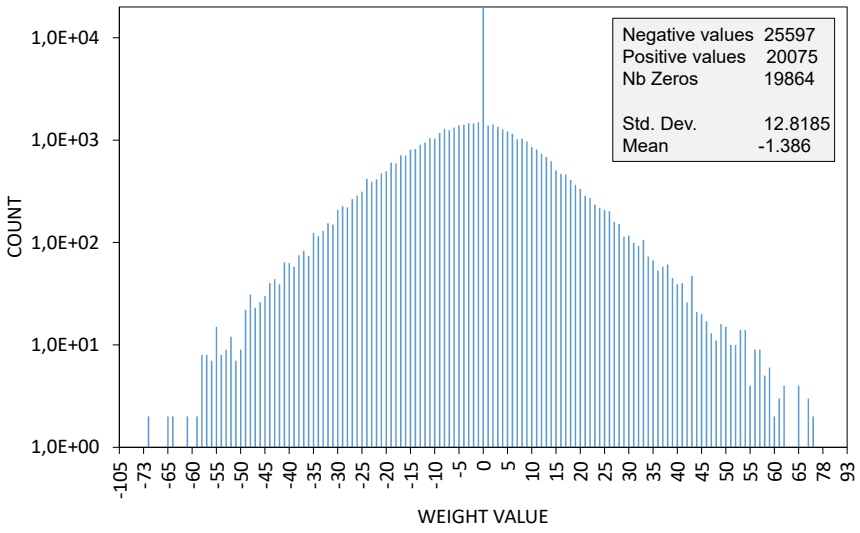


Figure 3.11 – Weights distribution in the *conv9_1×1* convolutional layer of MobileNet-V1.

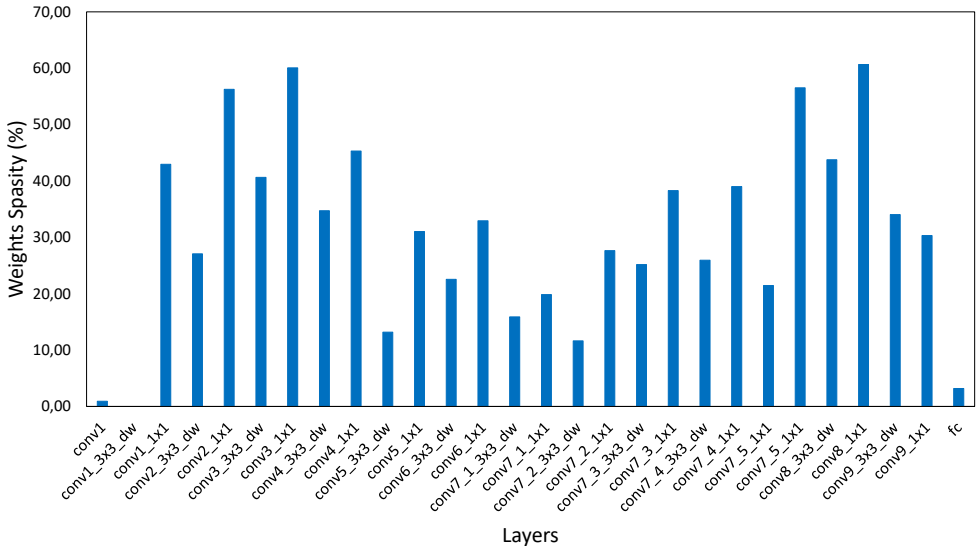


Figure 3.12 – A per-layer characterization of the percentage of zero in the parameters of MobileNet-V1.

set of real images, is essential to choose the right techniques to decrease computational and memory requirements. This analysis uses the generated C reference model evoked in Section 3.1 to dynamically characterized the sparsity profile of activation maps. For instance, the generated C code of the CNN can use the learning database or the validation one to analyze the dynamic sparsity of the input DNN when executed. For example, the average sparsity of 5% of the training dataset of MobileNet-V1 is 0.46%. Those images

are analyzed by being an input of the first layer in the DNN. On the other hand, the layer per layer analysis of the output feature maps in MobileNet-V1, presented in Figure 3.13, shows a significant sparsity in the majority of the layers (e.g. 83.88% for *conv9_1×1*). The average sparsity of the input features of all layers is 51.9%.

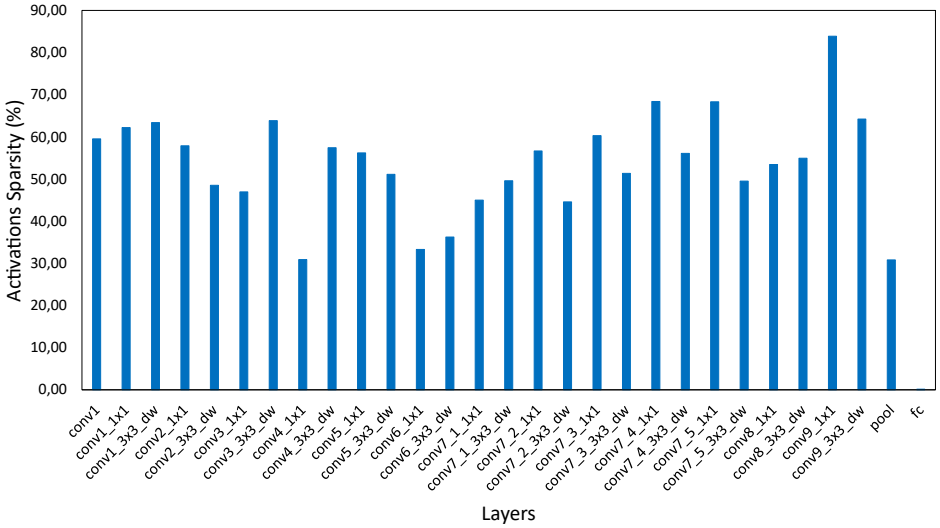


Figure 3.13 – A per-layer characterization of the percentage of zero of activations in MobileNet-V1.

Width/Depth and Data-reuse Combining width and depth evolution of the feature maps throughout the network and the pixel-reuse percentage emphasizes the use of some previously presented metrics and generates new ones. For instance, it could be used to determine the achievable degree of parallelism and what to priories in reuse, i.e. pixel reuse or weight reuse. It could also help choose the right dataflow for dataflow-based architectures and refine memory requirements according to the chosen dataflow. On a higher level, it hints to efficient algorithmic implementations that have a great impact on the required on-chip memory and amount of data transfer. Examples of transformations are loop tiling along with their possible values (for algorithmic architectural representation in C/C++, SystemC, etc.) as well as loop ordering.

The main goal of the metrics analysis is to facilitate the implementation of DNN algorithms on embedded systems in a short time by providing target-agnostic and hardware-aware metrics which gives hints on applicable methods and strategies to implement op-

timized CNN accelerators. The idea is to have a systematic, automatic and modular approach to gather the required knowledge to ease the next generation and optimisation phases.

3.2 Characterization and Analysis of different State-of-the-Art Networks

In the literature, DNNs are trained on the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) dataset [110]. They are usually compared based on their accuracy and their execution time on particular platforms such as GPUs as provided by most DNN training and inference frameworks like TensorFlow [1]. Unlike the accuracy-based comparison, this section presents the characterization results of some of the popular state-of-the-art networks and compares them in terms of computational and memory needs. In addition, an analysis of these metrics is also sketched to provide potential implementation strategies for DNNs on embedded systems which, at the same time, reduces design time.

Various state-of-the-art networks were used to assess the interest of the characterization and analysis phase. The evaluation was performed on small neural networks such as LeNet-5 [72], as well as different and complex topologies such as GoogleNet (2014) [122], EfficientNet-b0 (2019) [123], VGG16 (2014) [119], ResNet-50 (2015) and [53] SqueezeNet (2016) [58]. These networks take similar input dimensions, as presented in Table 1.1 in Section 1.1.2, and have won the ILSVRC challenge, except for LeNet-5. It is worth mentioning that the input image has a significant impact on the memory requirement of the neural network, especially the activation memory needs. However, increasing the size of the input image has no impact on the number of weights, since it depends on the fixed structure of the network. As an illustration, for a neural network with a 48x48 input image (2304 input activations) and a total of 5512 activations, if the size of its input image is doubled (96x96), then the total number of input activations becomes 11024 (which is 2×5512).

For instance, considering a neural network with a 48x48 input image (2304 input activations) and a total of 5512 activations. If the size of the input image is doubled (96x96), the total number of input activations becomes 11024 (which is 2×5512).

The chosen networks are characterized, first, in terms of computational complexity expressed in MAC which is considered as an elementary unit. Second, the networks are

dissected to study their internal structure, i.e., the types and dimensions of layers performing MAC operations as well as the percentage of performed MACs. Table 3.1 shows the total computational complexity of various DNNs and highlights the most computationally intensive layers in each of the considered networks. For instance, LeNet-5 consists of 5x5 kernels were 96.66% of the computation happens, the remaining 3.35% are located in FC layers. MobileNet-V1.0, a network with a more complex structure, has 80 to 90% of MAC operations performed by 1x1 point-wise convolutions, 5 to 10% are performed by depth-wise convolutions and the remaining are performed by FC layers. Implementing an accelerator capable of hosting these different types of layers is challenging even though the performed operation is always a MAC in a convolution. The main obstacle is that these kernels have different data access patterns and might require different dataflows. Therefore, hosting those three layers necessitate a flexible microarchitecture with operators performing the elementary MAC operation. More challenges are faced with modern networks having three different kernel shapes 1x1, 3x3 and 5x5 like EfficientNet-b0. In such case, the use of several accelerators might be a solution to avoid sacrificing performance due to load imbalance, at the cost of flexibility.

In case a set of networks is targeted, the challenge becomes even harder. Designing an accelerator for EfficientNet-b0, MobileNet and SqueezeNet is a very hard task due to the variety of kernel shapes as well as the irregular computational partitioning: 3x3 convolutions occupy 9.52% of the computation in EfficientNet, 60% in SqueezeNet and 10.65% of 3x3 depth-wise in 0.25-MobileNet-V1, which means that data access patterns are not the same and a significant load imbalance can take place due to this diversity. Additionally, memory needs of these networks are widely different due to the variety of the employed layers in each one. For example, considering an 8-bit precision, 0.25-MobileNet-V1 has $0.46E+06$B of activations, EfficientNet has $13.97E+06$B and SqueezeNet has $3E+06$B as presented in Table 1.1 (Chapter 1).

3.3 Conclusion

In this chapter, the characterization of an application is presented as a way to gain knowledge about the algorithm(s) to implement to reduce the gap between the application and the hardware architecture. The proposed characterization step studies the application to provide key features and characteristics of CNNs algorithms for an efficient implementation. It can also take into consideration the newly available DNNs. It provides relevant

CNNs	**Total MACs**	**MACs % of each kernel**						
		First Layer	**1x1**	**3x3**	**5x5**	**7x7**	**DW-3x3**	**FC**
LeNet-5	3.26E+05	36.02	0	0	96.66	0.0	0.0	3.35
0.25-MobileNet V1.0	41.03E+06	6.64	82.64	6.64	0	0	10.65	0.06
0.5-MobileNet v1.0	149.49E+06	3.62	90.22	3.62	0	0	5.82	0.34
SqueezeNet V1.1	352.54E+06	6.25	40	60	0	0	0	0
EfficientNet-b0	387.78E+06	0.68	78.57	9.52	10.71	0.0	0.0	1.19
VGG16	15.47E+09	0.5	0	99.2	0	0	0	0.8
ResNet-50	3.83E+09	3.078	48.95	47.92	0	3.08	0	0.052
GoogleNet	1.69E+09	6.992	28.102	57.042	7.8	6.992	0	0.06

Table 3.1 – Computational complexity of different state-of-the-art CNNs.

metrics and data visualization techniques based on a static analysis with no execution of the CNN. In addition, it offers optional information regarding weights sparsity as well as the dynamic activations sparsity. The utility of each metric is presented to suggest optimization strategies regarding memory and computational requirements as well as mapping techniques. In addition, it can be employed to drive the configuration of target architectures by setting the required on-chip memory per layer, identifying the required number of MACs (i.e. computational elements) and setting up the interconnect.

In the next chapter, the Hardware Generation step of the proposed approach, introduced in Section 2.3, will be presented. This step exploits the high-level synthesis (HLS) to generate the RTL of the architecture. Furthermore, it will be illustrated how the presented layer-based metrics as well as their analysis are used to help HLS-based hardware generation.

Chapter 4

Flexible Hardware Generation

Designing a hardware accelerator is a time consuming process, especially when using traditional Hardware Description Languages (HDLs), such as VHDL or Verilog. To overcome RTL design limitations, researchers felt the need to raise the level of abstraction to improve the comprehension about a system and increase productivity. Therefore, High Level Synthesis (HLS), which enables the Electronic System-Level (ESL) design automation, has been introduced.

The second step of the proposed methodology (blue box in Figure 4.1), presented in Section 2.3, leverages this high level of abstraction through the use of HLS tools to increase productivity.

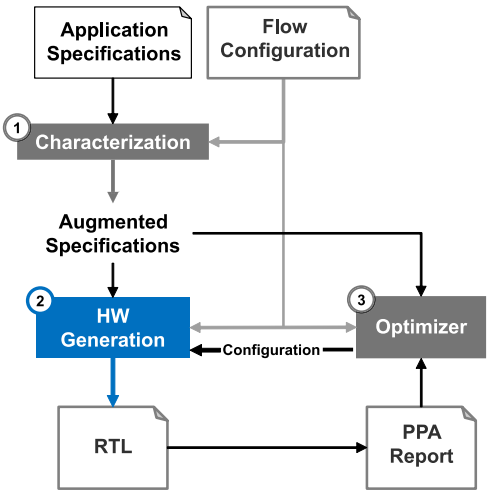


Figure 4.1 – The flow of the proposed approach, with a focus on the Hardware Generation step.

The main goal of this Hardware Generation step is to generate an optimized synthesizable RTL for an input DNN. This step takes as input the results of the Characterization step to drive the generation of the HLS-friendly source code. Hence, the produced source code includes transformations that intend to optimize the RTL that will be generated.

The purpose of this chapter is to detail the Hardware Generation step of the proposed methodology and to present some early results on DNN layers generation.

This chapter is organized as follows. Section 4.1 presents an overview about the Hardware Generation step as well as a code transformation example that motivates the need for high-level optimizations and a library of operators. Section 4.2 presents the possible high-level optimizations of the C source code. Section 4.3 presents the implementation of a library of C-HLS oprators. Section 4.4 details the hardware generation of different C code implementations, and finally section 4.5 sketches the conclusion.

4.1 Hardware Generation step Overview

The Hardware Generation step is composed of two stages. The first one generates a high-level source code of the different layers composing the input DNN. These source codes are then combined to build the source code of the entire DNN. The second step takes the resulting source code and effectively generates the RTL representation of the accelerator using an HLS approach. More precisely, the hardware generation step uses the IR of the DNN (cf. Section 3.1) to generate an optimized RTL representation of each layer of the input DNN using an HLS tool. Figure 4.2 illustrates (in blue) the HLS-based hardware generation step. The generated C-HLS source code is optimized by leveraging the Characterization step results. The functionality of the generated C-HLS code is verified using co-simulation with respect to the C reference model generated in the Characterization step, detailed in Section 3.1. As explained in the previous Chapter, this reference model (also called golden model) is HLS-friendly and can be synthesized and co-simulated using the employed HLS tool in the Hardware Generation step. Consequently, the RTL of the optimized C-HLS source code can be generated. In addition, the RTL of the C golden model can be generated to obtain a reference hardware implementation.

It is worth noting that some layers in the DNNs might remain non-optimized if the Characterization metrics did not imply any relevant changes that could positively impact the RTL. Hereafter, non-optimized C-HLS implementations of the different layers are presented, which are the same found in a C reference model.

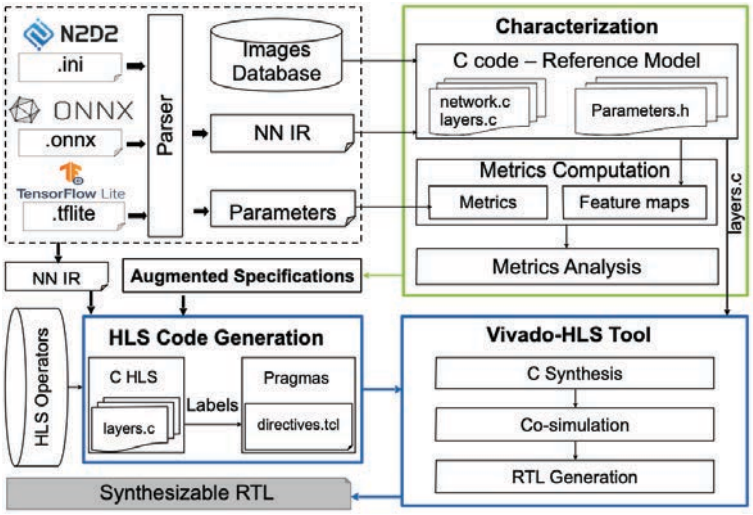


Figure 4.2 – The HLS-based generation flow, in blue, of each layer of an input DNN algorithm. It consists of two main steps: the C-HLS code generation and the RTL generation using Vivado-HLS.

4.1.1 Non-optimized C-HLS implementations of DNN layers

The most common C implementation of convolutional layers can be found in [136] [117] [104] and many others. It consists in a nest of six loops, where each loop represents one dimension of the layer. The pseudo-code is presented in Listing 4.1.

Pooling layers have a similar structure to convolutional layers. The pseudo-code of such layers consists in a nest of five loops as presented in Listing 4.2. Only max-pooling layers are considered due to their high efficiency compared to average pooling layers [114].

Regarding FC layers, the C implementation consists of four nested loops as presented

```
ifmap[N][(R-1)*S+K][(C-1)*S+K]  //input feature maps
outfmap[M][R][C]  //output feature maps
weights[M][N][Kx][Ky]  //filter weights
10: for(r=0; r<R; r++)  //output X
  11: for(c=0; c<C; c++)  //output Y
    12: for(m=0; m<M; m++)  //nb outputs
      13: for(n=0; n<N; n++)  //nb channels
        14: for(kx=0; kx<Kx; kx++)  // kernel X
          15: for(ky=0; ky<Ky; ky++)  // kernel Y
            wx=weights[m][n][kx][ky]
            ix=ifmap[n][S*r+kx][S*c+ky]
            outfmap[m][r][c]+=wx*ix
```

Listing 4.1 – Pseudo-code of a convolutional layer.

```
l0 : for (r=0; r<R; r++)  //output X
  l1 : for (c=0; c<C; c++)  //output Y
    l2 : for (m=0; m<M; m++)  //nb outputs
      tmp = 0
      l3 : for (kx=0; kx<Kx; kx++)  // kernel X
        l4 : for (ky=0; ky<Ky; ky++)  // kernel Y
          ix=ifmap[n][S*r+kx][S*c+ky]
          tmp = max(ix, tmp)
          outfmap[m][r][c]=tmp
```

Listing 4.2 – Pseudo-code of a max-pooling layer.

```
  l0 : for (m=0; m<M; m++)  //nb outputs
    l1 : for (n=0; n<N; n++)  //nb channels
      l2 : for (i=0; i<H; i++)  // input X
        l3 : for (j=0; j<W; j++) // input Y
          wx=weights[m][n]
          ix=ifmap[i][j]
          outfmap[m]+=wx*ix
```

Listing 4.3 – Pseudo-code of a fully-connected layer.

in Listing 4.3.

4.1.2 C-Code transformation example

The coding style has a significant impact on area and performance of the final hardware design. For example, a C code written in many different ways, which have the exact same functionality, results in different area and performance for each implementation after synthesis. To illustrate this fact, an experiment presented in Listing 4.4 presents a C implementation of standard 3×3 convolution, and Listing 4.5 presents a modified C code having the same functionality. The body of the loop nest in the first implementation consists in multiply-accumulate (MAC) operation of the weights and input feature maps. This MAC operation is separated in the second implementation (Listing 4.5), changing the order of the operations from MAC at each iteration to a set of multiplications followed by a set of accumulations.

Both C-code implementations are synthesized and simulated to verify that both implementations have the same behavior. Then the RTL is generated using Vivado-HLS 2020.1. The Xilinx Zynq7000 xc7z030 FPGA is set as a target to implement the two convolutions assuming a fixed frequency of 100MHz. Table 4.1 presents a comparison in terms of resource usage and performance. As it can be seen, the standard implementa-

```
//ifmap: input maps
//w: filter weights
void conv3x3(w[Ky][Kx], ifmap[Ky][Kx], *outpix) // Kx = 3; Ky = 3
  *out_pix = 0;
  for (int i = 0; i < Ky; ++i)
    for(int j = 0; j < Kx; ++j)
      *outpix += w[i][j] * ifmap[i][j]
```

Listing 4.4 – Pseudo-code of a 3x3 standard convolution.

```
void conv3_modif(w[Kx*Ky], ifmap[Kx*Ky], *outpix)
  tmp_mult[Kx*Ky];
  *out_pix = 0;
  for (int i = 0; i < KDIM; ++i) {  // multiplication loop
  #pragma HLS ALLOCATION instances=mul limit=9 operation
    tmp_mult[i] = coeff[i] * in_pix[i]

  for(int j = 0; j < KDIM; ++j){ // adding loop
  #pragma HLS ALLOCATION instances=add limit=9 operation
    *outpix += tmp_mult[j]
```

Listing 4.5 – Pseudo-code of a 3x3 modified convolution.

Implementation	**Std. Conv.**	**Modif. Conv.**
Latency (cycles)	25	38
LUT	32	142
FF	16	61
DSP	1	0
BRAM	0	0

Table 4.1 – Comparison of RTL implementations of the standard listing and the modified listing 3x3 convolution.

tion is faster than the modified one. However, this can be explained by the fact that the first implementation employs a DSP (DSP48E) having pipeline registers for both multiply and accumulate operations, which enables it to run faster. Regarding resource utilization, the modified implementation uses more LUTs and FFs compared to the standard one. However, it does not use any DSP, which can be the reason why it has a higher latency compared to the standard implementation.

From this experiment, one can see that modifying the C code could add other possibilities. Improving some aspects of a design requires high-level optimizations in addition to the coding style. Furthermore, combining different C-code transformations and pragmas results in numerous design possibilities, which allows extending the design space for DNN

accelerators to enable new potential optimizations. Hence, a library is required to store the resulting set of combinations, each called an "operator". These operators replace, when convenient, the computing part in the DNN layers, precisely the two inner-most loops (also called kernel loops) in convolutional and pooling layers, in order to improve certain aspects of the hardware implementation of each layer. These kernel loops can represent different shapes such as 3×3, 5×5, etc. More details are provided in the following sections.

4.2 High-Level Optimizations

The Characterization step, detailed in Chapter 3, generates augmented specifications (cf. section 3.1) which enable the generation of optimized C-HLS code of each layer (presented in Section 4.1.1) of the given DNN algorithm. The generated metrics are employed to enhance the reference C-code to produce optimized RTL representations of hardware accelerators. Since all layers are based on a set of nested loops, important high-level optimizations to consider are related to loop transformations. The most common loop optimizations are loop unrolling, pipelining, ordering and tiling, which apply transformations on a loop or a loop nest to improve data locality or parallelism. Applying transformations on high-level descriptions has an important impact on the RTL to be generated [73, 29].

Despite the possibilities to enhance the performance of the C/C++ code, and thus the RTL to be generated, more optimization opportunities are available. C and C++ are sequential languages that describe the software behavior, they do not have the clock concept and do not offer the possibility to set variable bit-width, especially using a low number of bits such as int3_t, int4_t, etc. Therefore, researchers proposed solutions via libraries and extensions [47] [59] [68]. For instance, the use of directives (tool-specific or not), and the use of a subset of ANSI C/C++ are one of the most common solutions. The directives are, in general, used to apply transformations for RTL generation, like improving resource usage. Such directives play a significant role in reducing the gap between the software and the hardware and makes the design space rich and flexible. These high-level optimizations are detailed in the following subsections.

Unrolling

This type of optimization introduces spatial parallelism which aims at improving latency and throughput of a loop. It allows a concurrent execution of a number of iterations, set by a given factor N which enables partial or full unroll. If the loop is fully unrolled,

then all the iterations are executed in parallel and the latency is reduced to the latency of a single iteration of the loop, in this case N may not be specified. Otherwise, in partial unrolling N must be specified and the tools checks if the partially unrolled loop and the original one are functionally the same. From a hardware perspective, unrolling a loop implies instantiating necessary RTL copies of the loop body to allow a concurrent execution of the loop iterations. However, despite the significant decrease in execution time an important amount of resources is needed.

Figure 4.3 sketches an unrolling example of a loop. The pragma completely unrolls the loop allowing to start all iteration at the same cycle. Therefore, since there are only 2 loop iterations in this example, the functional units in the loop's body are doubled, and the total number of iterations is halved.

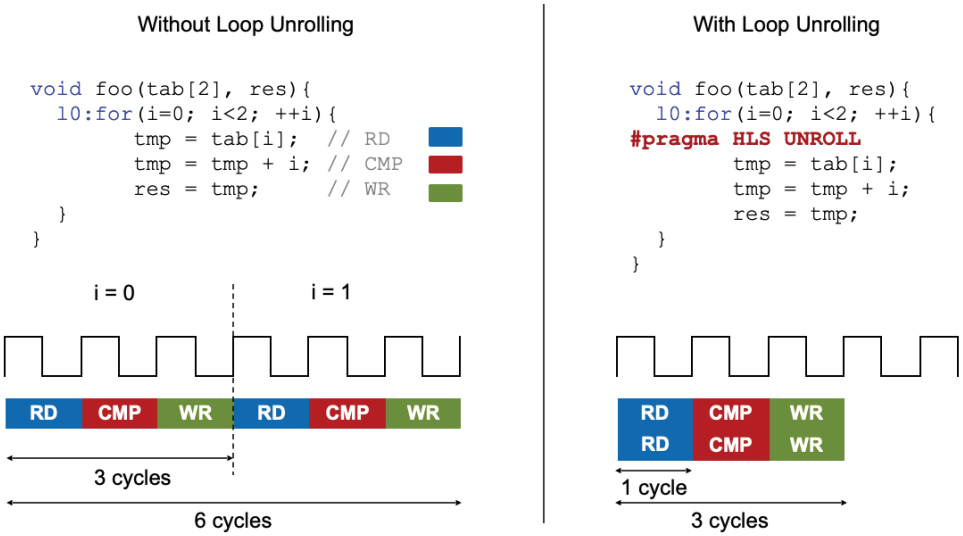


Figure 4.3 – Illustration of a loop unrolling example. On the left, the loop is not unrolled, no unroll directive in the loop body. Performing two iteration takes 6 clock cycles. On the right, the loop is totally unrolled. Performing the two iterations of the loop takes 3 cycles.

Pipelining

Pipelining is an optimization that allows concurrent execution of operations inside a loop or a function to reduce its Initiation Interval (II). II is the number of clock cycles between the processing of consecutive inputs or loop iterations. With this optimization, the processing of the second input or loop iteration can start before finishing the first one. A new process starts every II cycles, if II is set to 1 then a new input or loop iteration is

processed every cycle. Pipelining can highly improve latency, however, resource constraints as well as loop carried dependencies can restrict the attainable II. From a hardware viewpoint, this optimization requires more resources to concurrently process inputs or loops iterations.

Figure 4.4 sketches a pipelining example of a loop. The pragma is applied with an II of 1 allowing to start a new iteration every cycle.

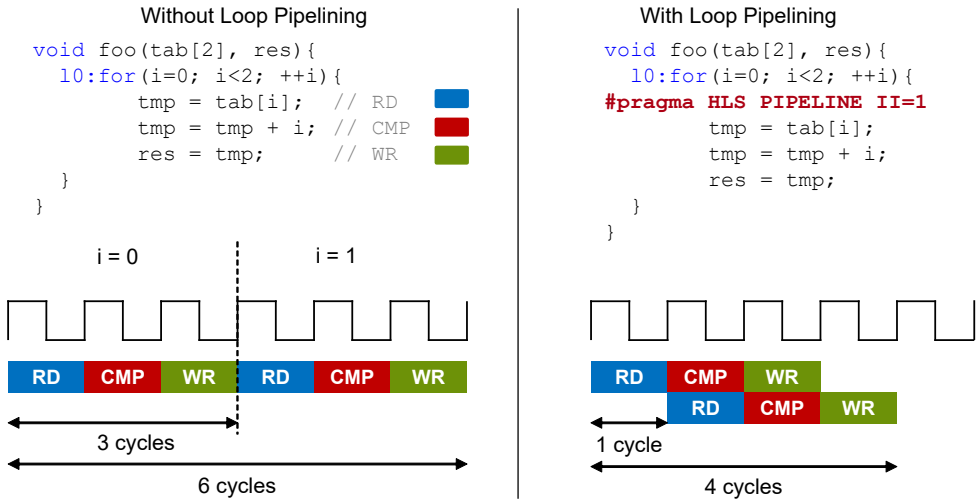


Figure 4.4 – Illustration of a loop pipelining example. On the left, the loop is not pipelined, no pipeline directive in the loop body. Performing two iteration takes 6 clock cycles. On the right, the loop is pipelined with an II= 1. Performing the two iterations of the loop takes 4 cycles.

Loop Reordering

Loop reordering is another loop nest optimization technique related to data management. It is exploited to make sure that data is accessed in the same pattern in which it is present in memory. This improves data locality as well as memory accesses. Deciding which loop levels to interchange, relies on the same metrics used to set the loop levels to tile and their tiling parameters. For instance, in Listing 4.1, instead of being tiled, loops $l0$ and $l1$ (suppose they have high loop bounds) can be swapped with $l2$ to bring the data closer to the computation part and, therefore, to improve performance.

Loop Tiling

Loop tiling is a commonly used optimization, especially in HLS approaches for CNNs [136] [117] [104]. It encourages data locality to limit expensive memory accesses and

transfers. In addition, it improves performance if the tile size is correctly set. When tiling is applied, the source code incorporates new loop levels, depending on the number of tiled loops in the loop nest. For instance, tiling loop $l0$ and $l1$ in Listing 4.1 with Tr and Tc factors respectively, introduces two new loops, $l0.1$ and $l1.1$, as presented in listing 4.6. The intra-tile loops ($l0.1$ and $l1.1$) drives the computation inside a tile and the inter-tile loop nest that traverses different tiles. Hence, it allows working on smaller chucks of the input. Figure 4.5 illustrates the tiling of two nested loops. These loops are tiled into (Ti, Tj) with four nested loops consisting of two inter-tile loops and two intra-tile loops.

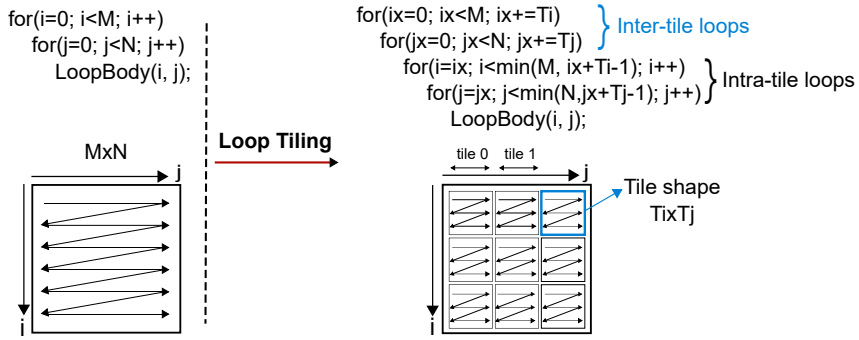


Figure 4.5 – Illustration of the tiling of two nested loops.

However, it is challenging to determine the loops to tile and their tiling factors. Therefore, the metrics extracted in the characterization, especially the **Input/output feature maps and kernels dimensions** and **Width/Depth and Data-reuse**, can be employed. The first metric sets the threshold value of each tiling factor, as explained in Section 3.1.2, to limit the design space. The second one highlights the loop levels to be tiled, since it takes into account the width and depth of the input feature map as well as the reuse percentage. For instance, a layer with a large input width and a high reuse percentage (e.g. loop $l0$) will be tiled as data locality can be exploited to also encourage data reuse.

With the help of the characterization results, the C-HLS source code can be generated with a first optimization level involving loop transformations, which can be directly applied at C-HLS code generation.

Dataflow

The main purpose of this directive is to increase the concurrency and the throughput of the design at RTL level by reducing the II. Therefore, after a dataflow analysis between

```
ifmap[N][(R-1)*S+K][(C-1)*S+K] //input maps
outfmap[M][R][C]   //output maps
weights[M][N][K][K]
l0 : for(r=0; r<R; r+=Tr)      //output X
 l1 : for(c=0; c<C; c+=Tc)     //output Y
  l2 : for(m=0; m<M; m++)      //nb outputs
   l3 : for(n=0; n<N; n++)     //nb channels
    l0.1 : for(tr=r; r<min(R,r+Tr); tr++)
     l1.1 for(tc=c; c<min(C,c+Tc); tc++)
      l4 : for(kx=0; kx<Kx; kx++)      // kernel X
       l5 : for(ky=0; ky<Ky; ky++)     // kernel Y
        wx=weights[m][n][kx][ky]
        ix=ifmap[n][S*tr+kx][S*tc+ky]
        outfmap[m][tr][tc]+=wx*ix
```

Listing 4.6 – Pseudo-code of a tiled convolutional layer.

the components of the sequential high-level code (i.e. functions or loops), the tools create channels based on FIFOs or Ping Pong RAMs, which enable functions and loops to overlap, or in other words to be pipelined.

Hierarchical Optimization - Inlining

The inlining directive has an impact on the generated RTL hierarchy, especially the amount of the generated control logic after synthesis. A function usually appears as a separate RTL module. Inlining dissolves the target function into the calling one, so it does no longer appear as an isolated level of hierarchy in the RTL code. Therefore, the logic related to function calling is no longer required and will not be generated. In contrast, the code of the called function is replicated in the calling one. Sometimes, operations and resources within the inlined function can be efficiently shared and used in the surrounding RTL code. Once a function is inlined, sharing it and reusing its RTL module is no longer possible. This can result in an increase of the required area for the RTL implementation.

Bit-Precision Optimization

Optimizing bit-widths has a significant impact on the area, the power and the performance of a design as well as the quality of the produced hardware. HLS tools offer the ability to generate hardware with arbitrary bit-widths, using only the required number of bits for operators and variables. This can be done by including the "*ap_cint.h*" HLS C Library ("*ap_int.h*" for C++). An pseudo-code using this library is presented in Listing 4.7.

```
#include <ap_cint.h>
int4 w;
```

Listing 4.7 – Pseudo-code using the "ap_cint.h" library.

It is an important optimization especially for memory and computationally demanding applications, such as CNNs. It reduces the required area to implement the CNN model, if the bit-widths are tailored to the exact number of bits of weights and activations of the trained CNN. In addition, quantization can significantly reduce bit-precision of a DNN, and thus HLS can leverage this optimization to reduce design resources.

Table 4.2 summarizes the optimizations detailed above and shows their configuration and a brief description.

Optimization	**Configuration**	**Description**
Unrolling	Unrolling factors	Creates multiple copies of the loop body & increases parallelism
Pipeling	Enabled/Disabled & II	Reduces II; & Allows a concurrent execution of operations
Inline	Yes/No	Dissolves a function into the calling one
Bit-Width	Bit-Width	Specifies the exact amount of required bits

Table 4.2 – Summary of the used optimizations.

4.3 Library of HLS operators

4.3.1 Introduction

Manipulating the computational part in the loop nest of a DNN layer is a high-level optimization that offers an additional degree of liberty. This manipulation consists in taking C-HLS code of kernel loops ($l4$ and $l5$) and transforming it by changing the coding style and applying high-level optimizations (Section 4.2) while guaranteeing the same functionality. Each transformed set of kernel loops is then designated as an HLS operator. This helps building a library of C-HLS operators with figures of merit. Since the C HLS source code has a great impact on the RTL, each kernel shape is coded in several different ways and optimized, performance-wise or latency-wise or a trade-off of both, using pragmas. These kernels are aggregated into a library of HLS operators.

```
void multAddTree(ifmap[Kx*Ky], w[Kx * Ky], *outpix)
    tmp[Kx*Ky]
    // multiply loop
    for(i=0; i < Ky * Kx; ++i)
#pragma HLS UNROLL
        tmp[i] = w[i] * ifmap[i]
    // add loop
    for(j=0; j < Ky * Kx; ++j)
#pragma HLS UNROLL
        *outpix += tmp[j]
```

Listing 4.8 – Pseudo-code of a 3x3 convolutional operator with multiplication and accumulation implemented in separate loops - Adder tree.

This library of operators presents a new optimization option for DNN algorithms, which is directly related to the computational part, and a diversification of the design space of each layer as well as the whole network.

Therefore, the convolutional kernel presented in Listing 4.4 can be implemented by separating the multiplication and the accumulation into two distinct loops, as in Listing 4.5. Such implementation resulted in greater area and poorer performance (cf. Table 4.1). To improve this implementation, each loop can be optimized using different types of pragmas to obtain different operators in terms of area and performance. For instance, as a code transformation, both loops can be unrolled, which results in a vector of multiplications followed by a parallel adder tree. The pseudo-code of the resulting operator is presented in Listing 4.8. Other optimizations can be applied to create different operators with the same functionality. For example, input arrays can be partitioned to improve throughput and performance.

Another C-HLS code transformation consists in modifying the optimization of one of the loops. For instance, the multiplication loop can be kept in unrolled state, and the accumulation loop can be replaced by a single instruction accumulating all the values at once. This approach results in a serial adder instead of an adder tree. The pseudo-code is presented in Listing 4.9.

The same approach is applied on pooling layers, i.e., replacing kernel loops ($l3$, $l4$) by an operator, especially *max-pool* layers. An example of a substitute max-pool operator is presented Listing 4.10, where a *max* operation finds the maximum value of three inputs. The loop embodying this operation is unrolled to accelerate the max-pool operation. The applicable code transformations are limited in this type of layers due to the simplicity of the operation (finding the max value).

```
void multSerialAdd(ifmap[Kx*Ky], w[Kx * Ky], *outpix)
    tmp[Kx*Ky]
    for(i=0; i < Ky * Kx; ++i)   // multiply loop
#pragma HLS UNROLL
        tmp[i] = w[i] * ifmap[i]
    *outpix = tmp[0] + tmp[1] + tmp[2]   // serial adder
            + tmp[3] + tmp[4] + tmp[5]
            + tmp[6] + tmp[7] + tmp[8]
```

Listing 4.9 – Pseudo-code of a 3x3 convolutional operator with separate operations replacing the accumulation loop by a single instruction - Serial Adder.

```
#define max(a, b, c) (((a > b) && (a > c)) ? a : ((b > c) ? b : c))
void maxpool(ifmap[Kx*Ky], *outpix)
    j=0
    tmp[Kx*Ky]
    for(i=0; i < Ky * Kx; ++i)
#pragma HLS UNROLL
        tmp[j] = max(ifmap[i], ifmap[i+1], ifmap[i+2])
        i+=3
        j++
  *outpix = max(tmp[0], tmp[1], tmp[2])
```

Listing 4.10 – Pseudo-code of a 3x3 pooling operator.

4.3.2 HLS operators overview and early results

Following the same approach of combining C-HLS code transformations and high-level optimizations, a library of operators comprising different shapes of convolutions and max-poolings is created.

Each implemented operator of this library is synthesized (place and route), excluding infrastructure such as memory controllers and crossbars, to get resource usage and performance details. Assuming a fixed frequency of 100MHz, the latency is expressed in cycles. Contrary to ASIC designs where area is measured in terms of square millimeters or gate equivalent, area is expressed in terms of used resources (i.e. FF, LUT, DSP, BRAM), since the target in here is FPGA.

In the context of this thesis, the chosen solution is to express area as aggregated values of *FF, LUT, DSP* and *BRAM* as in Equation (4.1).

$$PseudoArea = \frac{FF}{FF_{Available}} + \frac{LUT}{LUT_{Available}} + \frac{DSP}{DSP_{Available}} + \frac{BRAM}{BRAM_{Available}} \quad (4.1)$$

For instance, various implementations of a 3×3 convolution operator, having different code transformations and configurations, result in different area and latency values as illustrated in Figure 4.6 in which each point represents an operator. In this figure, one can see that some of these points form a shape of an area-latency Pareto curve. The x-axis and y-axis represent the latency and area respectively. Only operators on the Pareto curve will be selected for future optimizations.

The same is valid for the 5×5 and the 7×7 convolutional operators in Figure 4.7 and Figure 4.8 respectively, and the 3×3 pooling operators in Figure 4.9.

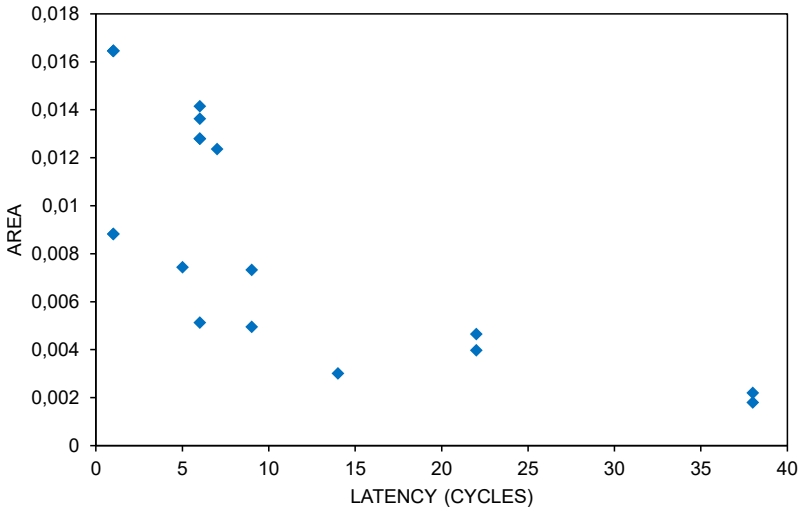


Figure 4.6 – Distribution of the 3×3 convolutional operators according to their area and latency. Some of these points are on the Pareto curve.

The proposed operators diversify the design space and enable a higher degree of liberty to optimize the implementation of each layer of the network, and to later optimize the whole network. Area and latency details will be later used in specific estimation models that will be detailed in the next chapter. It is worth noting that power consumption was not considered in this first implementation of the SHEFTENN framework. This explains why *PPA Report* in Figure 4.1 is replaced by *PA Report* in Figure 4.2. Introducing power consumption as another optimization parameter is clearly a perspective of this work.

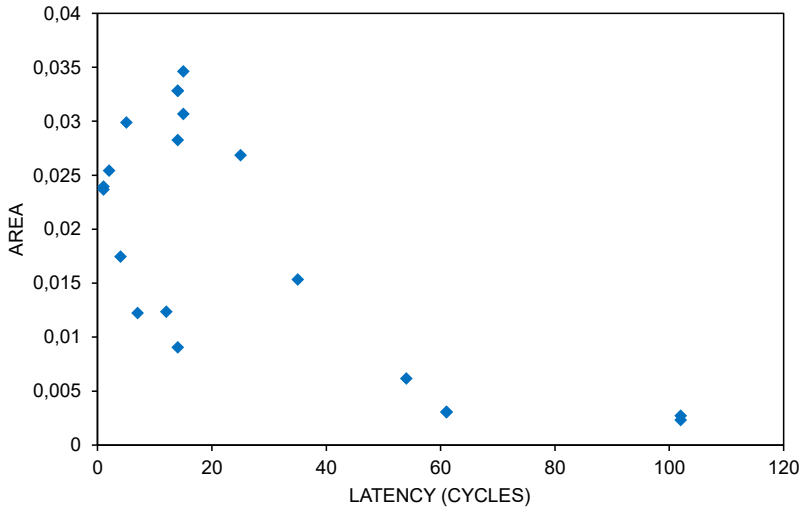


Figure 4.7 – Distribution of the 5x5 convolutional operators according to their area and latency. Some of these points are on the Pareto curve.

4.4 Early Results using the Hardware Generation step

This section shows few examples of some layers' implementations to illustrate the use of C-HLS source code, while considering possible optimizations related to loop transformations, pragmas and HLS operators. Regarding the hardware implementation, Vivado-HLS 2020.1 is employed to generate the RTL of the layers. The chosen target is the Xilinx Zynq7000 xc7z030 FPGA supposing a fixed frequency of 100MHz. Each layer is implemented separately without memory controller or crossbars.

Three example layers were chosen to illustrate the generation step. Table 4.3 shows the type of each layer, its order of loops and its dimensions. For comparison purposes, each layer is implemented without optimization, which serves as a reference implementation. It is worth noting that these layers are extracted from trained networks, where L1 is the first layer of MobileNet-V2, L2 is the sixth layer of MobileNet-V1 and L3 is the second layer of an evaluation network employed in [6].

Regarding the applied optimizations, some are derived from the characterization step, especially the ones related to loop re-ordering and high-level optimizations thanks to the characterization results especially the *Width and Depth comparison* and the *Data-Reuse percentage* (detailed in Section 3.1.2). The ones related to the use of operators cannot be derived from the characterization' results, since the characterization has not the operators

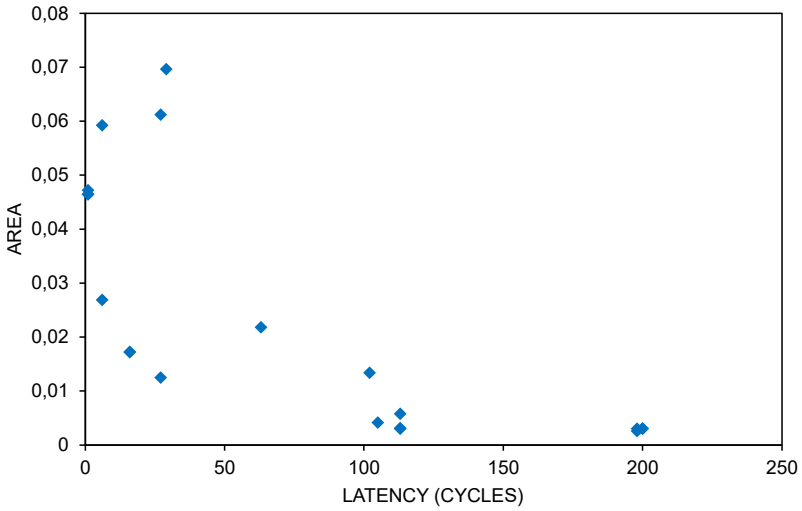


Figure 4.8 – Distribution of the 7x7 convolutional operators according to their area and latency. Some of these points are on the Pareto curve.

Layers	Type	Loops Order	Values	Stride
L1	Std. Conv.	R, C, M, N,Kx, Ky	128, 128, 8, 3, 3, 3	2
L2	Depth-Wise Conv.	R, C, M, N, Kx, Ky	16, 16, 32, 1, 3, 3	1
L3	Max Pooling	R, C, M, N, Kx, Ky	16, 16, 2, 2, 3, 3	3

Table 4.3 – Type and configuration of the implemented layers. Where, R: *Output Height*, C: *Output Width*, M: *Number of Outputs*, N: *Number of Channels*, Kx: *Kernel Height*, Ky: *Kernel Width*. Values correspond to loops order variables.

details (e.g. latency and resource usage), and can neither estimate the performance nor the resource utilization. All layers' implementation, optimized and non-optimized, are implemented with an 8-bit precision for weights and activations. In addition, bit-widths of employed indexes are tailored to their exact needs.

Layer L1 this layer has the width R greater than the depth N, as it can be seen in Table 4.3. According to the characterization analysis, the maximum reuse percentage in this layer is 23.85%, which is quite low for a convolution layer. A typical reuse factor for such a layer is above 80%. This low value can be explained by the employed stride value (stride = 2). While the reuse factor is not very high, encouraging data locality will still impact positively the performance, since the convolutional reuse is not negligible. Therefore, the first applied optimization is loop re-ordering (*Reorder*) which is applied

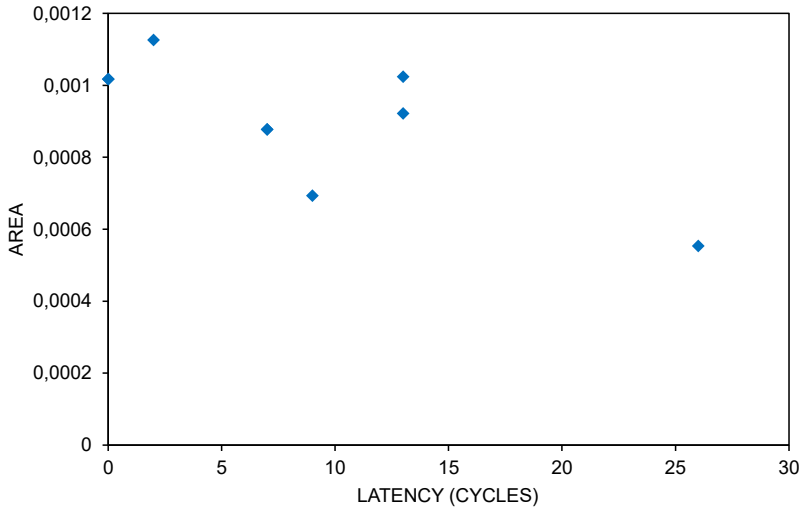


Figure 4.9 – Distribution of the 3×3 pooling operators according to their area and latency. Some of these points are on the Pareto curve.

automatically thanks to the results of the Characterization phase; loops are re-ordered to move data closer to the computing part, i.e. kernel loops (Kx, Ky), and limit unnecessary transfers. The resulting loop order is M, R, C, N, Kx, Ky. Reordering the loop improved the latency of 1.5% compared to the non-optimized implementation, while having little impact on resource usage.

The second optimization is related to the computing part. Since the kernel width and height are very small ($[Kx, Ky] = [3, 3]$) compared to the remaining dimensions, the loops are fully unrolled to improve parallelism and accelerate the processing (*Unroll KL*). Unrolling kernel loops improved performance by 70.15% but increased significantly resource usage, especially the DSP and the LUT, using 3.7× more LUTs and 9× more DSPs compared to the reference (non-optimized) implementation.

The third potential optimization is to employ a 3×3 convolutional operator (*Operator*); the operator is chosen from the Pareto curve. Two additional configurations are applied to the reordered version: unrolling kernel loops (*Reorder + Unroll KL*) and replacing kernel loops by a convolutional operator (*Reorder + Operator*). Compared to only unrolling the loops, the implementation combining reordering and unrolling improves the latency by 5%, while having no obvious impact on resources. Regarding the implementations including operators, the improvement in latency is similar to previous solutions, around 70% compared to the non-optimized implementation. In addition, resource uti-

lization significantly increased, 2× more LUTs, 9× more DSPs and 3× more BRAM in comparison to the non-optimized implementation. Figure 4.10 shows the resource usage (in percentage) and the latency (in cycles) of the different implementations.

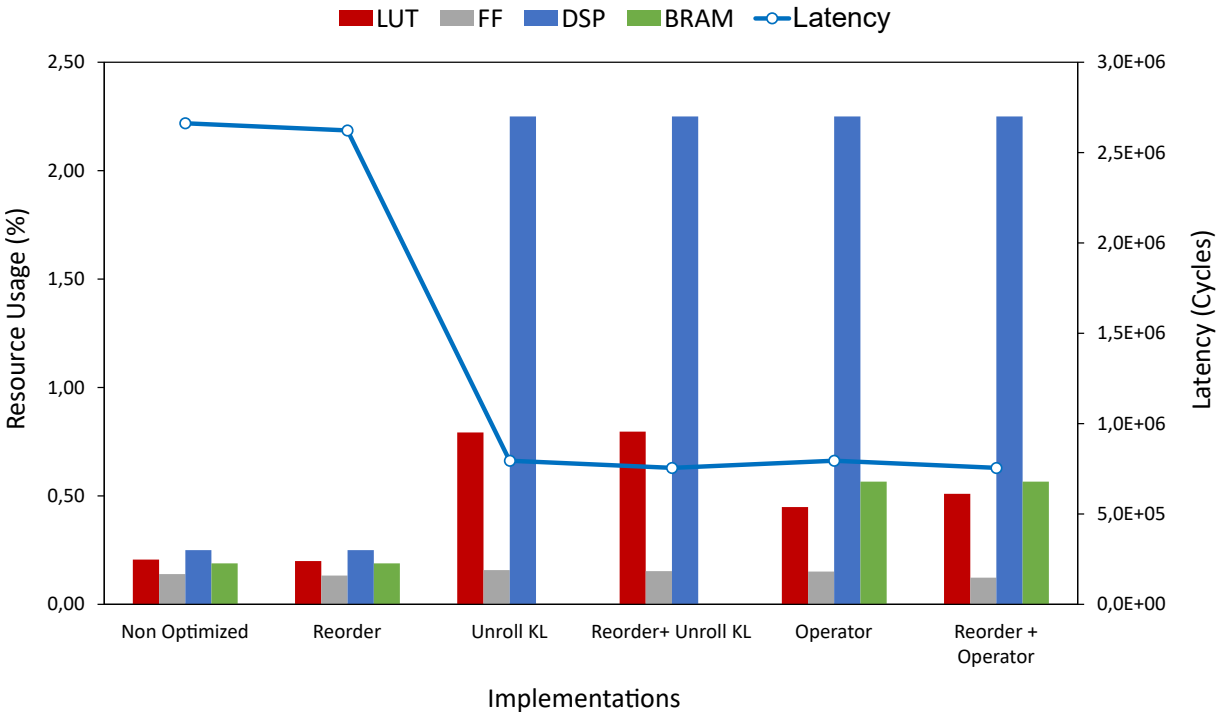


Figure 4.10 – Different implementations of the convolutional layer L1. *KL*: Kernel loops. *Operator*: 3×3 convolutional operator, pseudo-code in Listing 4.8.

Layer L2 this layer is a depth-wise convolutional layer. Contrary to the first one, its depth M is greater than its width R, as can be seen in Table 4.3. Additionally, the reuse percentage is low, 20.66%, due to the employed stride (stride = 2). As for the previous example, and despite the low reuse percentage, reordering will slightly enhance the performance, since the reuse percentage is not negligible. The same configurations applied on L1 are also applied on L2. Figure 4.11 shows the resource usage (in percentage) and the latency (in cycles) of the different implementations.

Reordering the loops improved the latency of 3.43% compared to the non optimized implementation, while having no impact on resources. Unrolling kernel loops improved performance by 74%, eliminated BRAM usage but increased significantly the DSP usage by 9× compared to the reference implementation. The implementation combining reordering and unrolling does not improve performance, compared to unrolling only. It

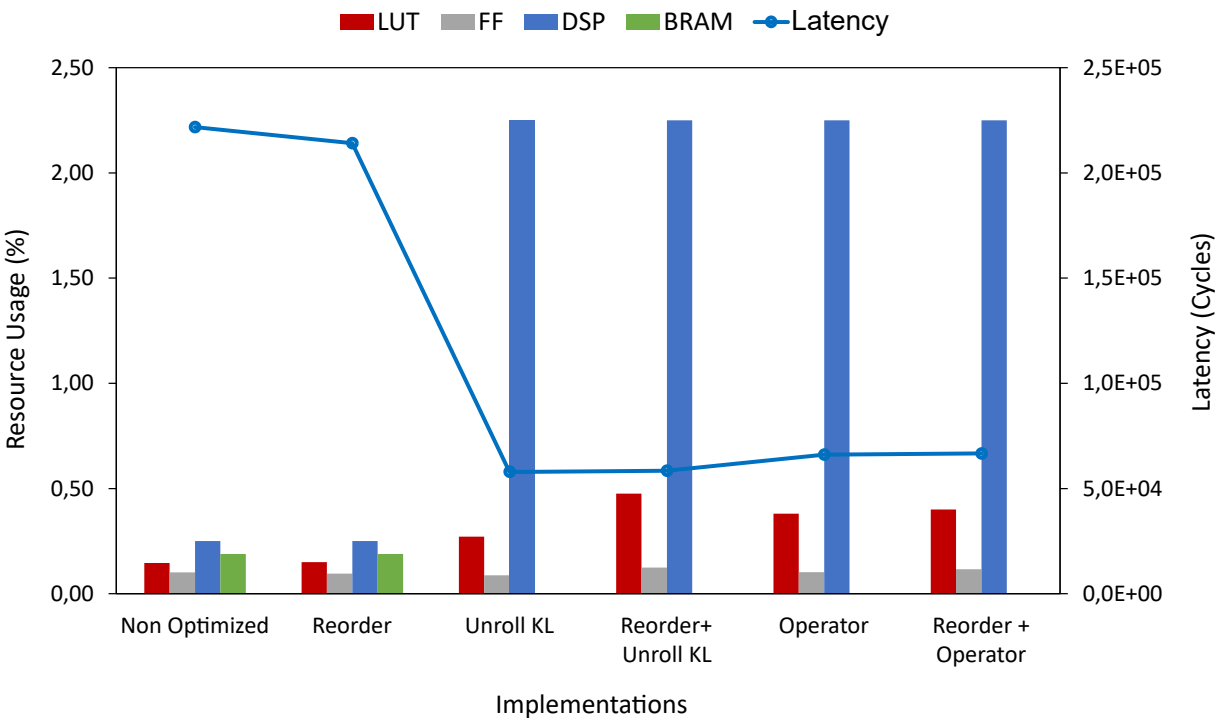


Figure 4.11 – Different implementations of the convolutional layer L2. *KL*: Kernel loops. *Operator*: 3×3 convolutional operator, pseudo-code in Listing 4.8.

also impacts negatively resource usage, especially the number of LUTs and FFs. Regarding the implementations in which operators are used as substitutes for kernel loops, the improvement in latency is similar to previous solutions, around 70% compared to the reference implementation. In addition, resource utilization significantly increased, 2.67× more LUTs, 9× more DSPs in comparison to the non-optimized implementation, but eliminated the use of BRAMs. It is worth noting that using the proposed operators as an optimization is not the best option between the proposed optimizations, since it degrades latency by 11% compared to implementations with unrolling and reordering + unrolling.

Layer L3 it is a pooling layer in which the width R greater than the depth N (cf. Table 4.3). The reuse percentage in input feature maps is zero, since the values of the stride and the kernel size are the same. Therefore, reordering should have little to no impact on area and performance. The same configurations applied on L1 and L2 are also applied on L3. Figure 4.12 shows the resource usage as well as the latency of each implementation.

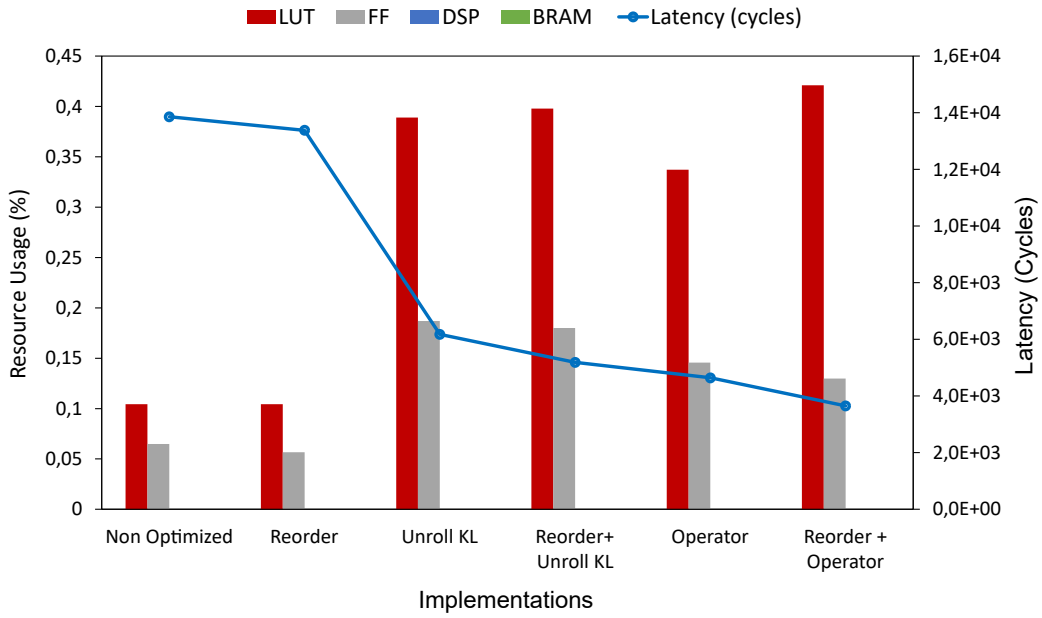


Figure 4.12 – Different implementations of the pooling layer L3. *KL*: Kernel loops. *Operator*: 3×3 max-pooling operator, pseudo-code in Listing 4.9.

Reordering loops improved performance by 3.44%, while having insignificant impact on used resources. The latency improved greatly with the unrolling of kernel loops, 52.4% faster than the reference implementation. On the other hand, the use of LUTs and FFs increased by 3.72× and 2.8× respectively. Applying reordering and unrolling simultaneously

enhanced further the performance by 62%, but kept the high use of FFs and LUTs. Using a pooling operator reduced greatly latency compared to all other implementations. The implementation with operator is 66.5% faster but uses more resources. As for resources, obviously this implementation requires more resources, 3.23× and 2.45× more LUTs and FFs respectively.

Many other implementations, which can improve various aspect of the design, should be explored. However, the large design space make this mission tricky especially when manually done.

4.5 Conclusion

This chapter presented the hardware generation step which, based on the characterization results, generates a semi-optimized C-HLS code including loop transformations, such as loop re-ordering and tiling. In addition, it leverages other high-level optimizations as well as a library of HLS operators, also exploiting these optimizations, to further optimize the high-level code. The generated metrics can determine the order of the loops and the loops to be tiled. However, it cannot (yet) determine the tiling parameters, which operators or pragmas to employ, since it has no way to explore the design space and evaluate each solution.

At this point, the hardware designer must test all the configurations himself to find an appropriate implementation, which is a very laborious task due to the resulting large design space. Therefore, the proposed methodology introduces an automated optimization step to perform this tedious task.

The following chapter presents an optimizer step to automate and facilitate the search for optimized configurations, for each layer, in the design space. This optimizer step will also be used to ensure the generation of an optimized accelerator for an entire DNN.

Chapter 5

Optimizing hardware through Design Space Exploration

In the previous chapter, a flexible Hardware Generation module was presented. This module allows exploiting numerous optimizations of high-level code transformations and pragmas of HLS tools. Unfortunately, new issues appeared similar to the ones of a standard design process but at a higher level. The first challenge is related to the number of high-level optimizations that can be applied, which increases the number of possible implementations of a design.

The second issue is the time-consuming syntheses to check the latency and resource usage of the design, which lengthen the exploration time and the overall design process. Therefore, design exploration strategies are required to obtain Pareto-optimal implementations while reducing the number of required syntheses.

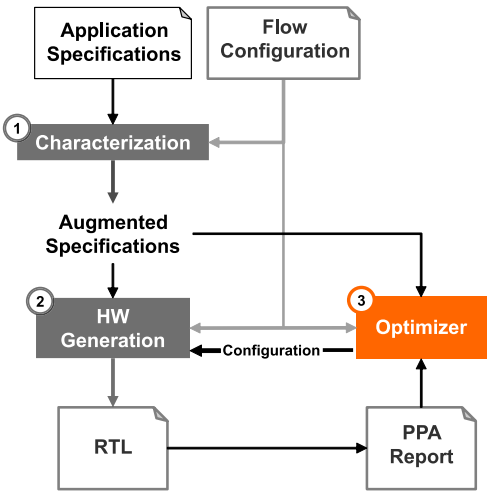


Figure 5.1 – The flow of the proposed methodology with a focus on the Optimizer module.

In the scope of this thesis, a specific DSE methodology, based on Genetic Algorithm, is

developed to speed-up the exploration as well as the whole design process. The proposed method aims at optimizing each layer of a DNN individually based on its dimensions. It uses area (i.e. resource utilization for FPGAs) and performance models to ensure a faster evaluation of the implementation and to decrease the number of synthesis runs. These models are based on a statistical analysis, which are developed for the purpose of this work. Section 5.1 presents some state-of-the-art DSE approaches: model-based and black-box-based. Section 5.2 presents the rationale for using a GA algorithm, the algorithm itself and finally, its current implementation. Section 5.3 presents the employed area and performance models. Section 5.4 presents early results assessing the interest of the GA algorithm for automatically optimizing the implementations of some DNN layers. Finally, section 5.5 sketches the conclusion of the chapter.

5.1 Related Works on DSE

The literature unveils different Design Space Exploration (DSE) techniques to reach a set of Pareto-optimal solutions of the hardware design problem, while diminishing the number of synthesis runs. Some of the proposed approaches are mainly HLS-driven and intend to mimic the HLS-tools' behavior in order to predict the effect of applied pragmas on area and performance and to limit the required number of syntheses. The main drawback of such strategies is that they cannot catch the impact of all optimizations, especially the inter-dependant ones. Recently, some papers have reviewed HLS-driven DSE strategies and classified them into different categories like Schafer et al. [113], Bulnes et al. [106] and Shathanaa [116]. They presented the current progress of these strategies while proposing different classifications to categorize existing approaches. HLS-driven DSE are classified into:

— *synthesis-based and model-based* as in [113]. The synthesis-based ones, such as meta-heuristics and dedicated heuristics, invoke the HLS-tool to evaluate every implementation. The model-based ones, such as supervised learning and graph-based, rely on performance and cost models for fast evaluation to avoid invoking the HLS tool. Schafer et al. imply that supervised learning methodologies belong to both categories since they require the synthesis of numerous configurations to build their knowledge. Once the model has learned the required information to mimic the behavior of the HLS tool, it then uses this knowledge to estimate the design costs and performance.

— *exact and approximate methods* as in [106] which focus on heuristic and meta-heuristic approaches. Exact methods (based on other forms of the branch and bound algorithm) have few appearances in the literature according to [44]. On the other hand; approximate ones are more popular among researchers.
— *learning-based methodologies*, exploration-based types, evolutionary algorithm, and population-based stochastic optimization methods as in [116]. This classification slightly matches the first one.

Following the classification used in [44], inspired by [113], the works in the literature are classified into two categories: model-based and black-box-based methodologies. Model-based strategies uses estimation models to mimic the behavior of a given HLS-tool. In contrast, black-box strategies deduce the behavior of the used HLS-tool.

5.1.1 Model-based DSE approaches

These approaches focus on estimating the behavior of the synthesis tool, while using various tool-specific directives and high-level code transformations. These approaches do not directly invoke the HLS-tool and rely on cost and performance models, usually analytical models, to evaluate the resulting implementations. However, they are often limited to a small set of tool-specific directives due to the complexity in predicting the effect of various and simultaneously applied pragmas on different steps of the HLS-based design process like allocation, scheduling and binding.

For example, the work proposed in [25] investigates three types of pragmas only, which are array partitioning, loop unrolling and loop pipelining. In order to estimate different design aspects, analytical resource and performance models are devised. While also using analytical models for performance and resource utilization, the works in [24], [28] and [30] aim at reducing the complexity of these models in various ways. For instance, [24] focuses on a distinct types of applications, especially on stencils. The work in [28] targets systolic architectures, and [28] aims at reducing the design space, i.e., configuration space, by suggesting the use of architectural templates. The authors in [76] also use architectural templates to limit the design space and conceive analytical models specific to these templates, and thus reduce the complexity of the design problem. Furthermore, using static graph analysis strategies have been also suggested, like the work in [137] in which a graph representation of the design is generated from its behavioral model using compiler methods. Using these graphs, processing elements (PEs) and their communication can be analyzed in order to respectively devise a model of the PEs and evaluate the

communication cost.

Other works have chosen another research path to accurately predict cost and performance. In particular, [115] presents a Pre-RTL, Power-Performance accelerator simulator, called Aladdin. It takes a high-level description of the architecture (written in C/C++) along with the applied directives to accurately estimate power, performance, and area. [26] proposes FLASH, an HLS simulation flow capable of extracting scheduling information from the HLS tool. Then, while maintaining C semantics, the tool automatically builds an analogous cycle-accurate simulation model. Such approaches substitute the synthesis process by accurate costs and performance prediction models, but do not reduce the design space.

The above cited approaches tackled the design space exploration problem by reducing the complexity of the design. They proposed various solutions, by using analytical models (tied to the HLS tool) , or by targeting specific applications, or also by developing accurate simulators for costs and performance estimations. Despite the fine results that could be obtained, such approaches are hard to generalize. For instance, analytical models must be updated with every new release of the employed commercial HLS tool (e.g. Vivado-HLS). These HLS tool updates are essential to tackle the market needs.

On the other side, Black-box-based techniques are agnostic to the number of considered pragmas and to the chosen HLS tool, which is an advantage compared to model-based approaches. Some of these approaches are presented in the following subsection.

5.1.2 Black-box-based DSE approaches

Black-box-based approaches, unlike model-based approaches, have no prior knowledge of the problem. These approaches are independent of the used pragmas and the HLS tool. Therefore, they need to acquire their knowledge throughout exploration, online or offline. These techniques require a large number of synthesis until they can provide high-quality and accurate results. Black-box-based approaches can be divided into different categories: learning-based approaches and refinement-based approaches. The learning-based ones gain their knowledge through a training phase in which a model of the synthesis process is learned, such as in supervised learning. The acquired knowledge is then used to drive the exploration process or to estimate costs and performance. Regarding refinement-based approaches, their knowledge is refined with new acquired data throughout the exploration process, such as in meta-heuristics and dedicated heuristics. It is worth noting that their initial knowledge can be acquired via a learning step.

Refinement-based strategies refine their knowledge during the exploration process to optimize the search for Pareto-optimal solutions. Therefore, the search will only seek promising regions of the design space, discovered online. Thus, poor solutions are ignored by adjusting the internal model. The model is refined using syntheses feed-backs, to boost its knowledge, while searching for Pareto-optimal solutions. Such approaches offer promising results especially when handling multi-objective optimization problems. For instance, the paper in [111] proposes a tool to explore the design space by generating a set of designs using an adaptive simulated annealing approach. Another approach proposes a probabilistic model [14] to accelerate the design space exploration. The probabilistic model aims at predicting Pareto-dominant solution with respect to the selected pragmas. The work in [112] suggests exploring each loop individually then merging the exploration results.

The authors in [77] derive a model of the used HLS tool by employing the Random Forest algorithm. The model is refined at every new synthesis. Some other approaches investigated the usage of Genetic Algorithms to explore the design space while tackling a small set of pragmas, like the works in [2], [97] and [56]. Few other works aim at refining the simulation-based exploration and producing a synthesis model by employing Response Surface Model, like the works in [134], [83], [118], [96]. The authors in [101] suggest an automated design space exploration technique which simultaneously coordinates pragmas and memory optimizations. The authors in [40] accentuate the importance of exploiting previously acquired data to efficiently diminish the DSE problems' complexity.

The work in [37] uses synthesis results in order to enhance the accuracy of HLS estimations. At the same time, the work in [79] uses an ASIC synthesis report to estimate the performance of the targeted FPGA. The work in [131] performs offline pre-characterization of micro-kernels and then builds predictive models of these kernels. The main purpose is to accelerate the HLS-driven DSE process.

Regarding Learning-based strategies, various approaches have been proposed in the literature. For instance, [95] addresses the DSE problem by using a neural network to predict performance and cost of a given processor. It also predicts memory, and bus parameters of a new architecture. Other approaches sought to gain high-quality results by employing various learning models while reducing the need for syntheses. In particular, [138] proposes to predict area and latency by employing a regression model, which relies on a Gaussian process, trained on a given dataset. [112] uses pattern matching techniques to implement local search methods. The approach in [135] predicts the optimal unrolling factors of given loops by combining a compiler pass analysis and Random Forest classifier.

These learning models use a training dataset to learn a model of the synthesis process, they use the acquired knowledge to infer results on the configurations to explore.

5.1.3 Discussion

The above-detailed state of the art presented various DSE methods that aims at finding Pareto-optimal implementations in a short time. Hence, those techniques tackle two main problems: the exploration of the design space and the evaluation time of each solution. Model-based approaches proposed using analytical models or accurate simulators developed to estimate area and latency to reduce the evaluation time. Even though such approaches outputs quality results, they are not easily generalizable, since models and simulators must be updated to take into account HLS tool' updates. On the other hand, black-box-based approaches are independent of both HLS tools and considered pragmas. Such techniques require a learning step either online (i.e. during the exploration) or offline (i.e. using neural network trained offline).

The work in this thesis tackle the same DSE issues of the literature stated earlier since it aims at facilitating the design of a DNN accelerator, and thus on easing laborious steps in the HLS design process, especially the synthesis part. Regarding the design space exploration problem, black-box-based approaches sound promising and suitable since they acquire their knowledge online through exploration and are not dependent on an HLS tool nor pragmas. Among the state-of-the-art black-box-based techniques, Genetic Algorithm (GA) seems well adapted to the DSE problem in this thesis since it learns online through exploration and does not require an offline learning step. In addition, it is scalable, and thus capable of dealing with small to large sets of pragmas. On the speed of evaluation side, estimation models for area and performance are developed to reduce the number of syntheses, and thus to rapidly evaluate each configuration. These two aspects are detailed in the following sections.

5.2 DSE Algorithm

The design space exploration problem, in the context of this work, consists in finding the right configuration, of a given layer, based on area-latency tradeoff. The configuration comprises loops order, pragmas and their corresponding values as well as the operators

detailed in the previous chapter (cf. Section 4.3). The main purpose is to find the right configuration of the high-level representation of a given layer, i.e., C-HLS code.

Herein, a method to solve this problem is presented. This method is based on a genetic algorithm (GA) that uses estimation models to rapidly evaluate each configuration. The main considered design criteria are: area and latency. Other design aspects can also be considered, such as power consumption. First, an introduction including the motivations behind the choice of GA as well as some basic notions will be presented. Then, the optimization method will be exposed.

5.2.1 Introduction to Genetic Algorithm

GAs are search algorithms that fall within the scope of Evolutionary Algorithms. GAs are inspired by Darwin's theory of evolution, and were firstly developed by John Holland in 1975 [55]. They were later used to solve various optimization problems, such as DNN hardware mapping [65], multi-objective optimizations (power, area and delay) [105], scheduling multiprocessor tasks [32], etc. Such algorithms mimic the mechanism of natural selection in which the fittest individuals are chosen for reproduction; selected individuals reproduce children.

These are powerful algorithms for solving multi-optimization problems, in which optimization aspects are often conflicting, and even for problems with a minimal amount of information. Multi-optimization problem is the most common problem in hardware design architectures, especially in embedded systems. Hardware designers often need to concurrently optimize several design aspects, such as performance and area, memory accesses and power, power and performance, or a combination of power, area and performance, etc. Therefore, a scalable algorithm is needed to rapidly adapt to the size of the problem and to be able to easily integrate new features (new pragmas or operators in this work). Such features characterize the GA, hence the choice of such algorithm for the DSE problem.

In this section, common GA terms and core principles of the Darwinian evolution are clarified, which are required to devise a specific genetic algorithm for the DSE problem of this thesis. To ensure that the natural selection takes place, three main elements are crucial: heredity, variation and selection. Heredity is a process in which parents, who survive long enough, pass down their genetic traits to their children. Variation must be introduced to obtain a diversified population, i.e., a population with a variety of traits, to enable new combinations and ensure evolution. Finally, selection is a process in which the fittest members of a population are selected for reproduction to produce children, referred

to as *survival of the fittest*. The term *fittest* is related to the adaptability of the individual to his environment, especially the traits of this individual. Hence, such individuals have higher probability of survival and reproduction.

This phenomenon inspired John Holland to develop the principle of GAs. The DNA, short for deoxyribonucleic acid, encodes the genetic information of every living organisms. Similarly, in GA a solution is an individual represented by its genotype and its environment is the search space. The individuals' evolution in an artificial environment (search space) leads to the improvement of their performance. Therefore, GA can be used to optimize any problem for which an encoding of the solutions and an evaluation function can be established. Some common terms are employed in this work and should be clarified before diving deeper into the proposed implementation. First, a gene encodes a trait of a solution. Second, an allele is a potential form of a gene. The genome is a set of genes of an individual, i.e., a design point in the solution space. The genotype is the genetic information, and finally the phenotype is the physical information of that individual.

A GA, as represented in Figure 5.2, is an iterative process that works as follows. First, solutions should be encoded in the form of chromosomes where each solution is represented by its genotype in the search space; it is usually represented by a binary or real valued vector of fixed or variable size. Once the solutions' encoding is set, the population of N individuals can be initialized by generating random solutions. The second step is the *Evaluation* where each individual (genome) of the generated populated is evaluated by means of a fitness function. The third step is the *Selection* process, in which genomes are selected for reproduction, and thus create the new genomes of the next generation. Various selection strategies are proposed:

— Random Selection, where a random number of genomes is selected for reproduction regardless of their fitness.
— Elitist Selection, where a small set of the fittest individuals are selected for reproduction. Unfortunately, this method cannot produce optimal outputs since all individuals, top and low -scoring, have the same probability to be selected for reproduction.
— Selection by using the Wheel of Fortune, also known as the Roulette wheel, which is a probability-based selection. In this method, each individual is assigned a sector of the roulette wheel proportional to his fitness and then a random draw is made to select the parents for reproduction.
— Selection by tournaments, in which a fixed number of genomes is randomly selected

and individuals having the best fitness score will be selected for reproduction.

The fourth step is the *Crossover*, where a new child genome is created by mixing the parents genes. The fifth and the last step is the *Mutation*, it is an alteration of the genetic information, where certain genes are modified based on a mutation rate, and produces new genomes. This step allows to produce new genomes and ensures the exploration of new areas in the search space. This allows reducing the probability of arriving in a local minimum. The GA then iterates on *Selection*, *Crossover*, *Mutation* and *Evaluation*, until a stop condition is reached, such as a solution is satisfied or a maximum number of generations is reached, etc. It is worth noting that elitist politics of population renewal is often used, in which a set of the best individuals (also called elite set) of the current population replaces the rest of the population with the new generation.

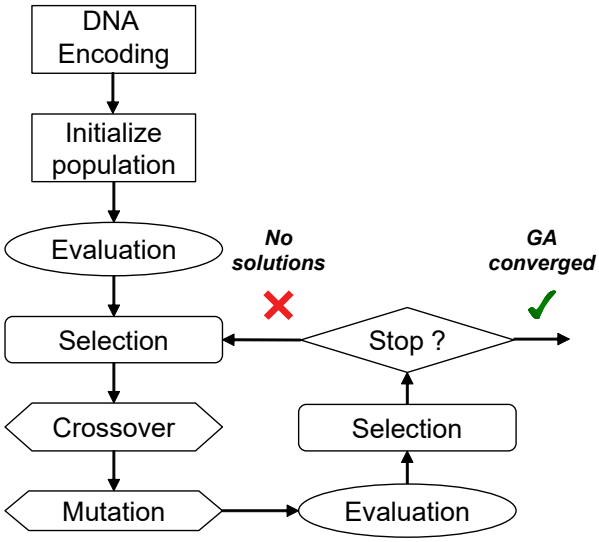


Figure 5.2 – Workflow of a Genetic Algorithm.

5.2.2 Implementation of the DSE module

In the context of this work, a custom GA is developed for this hardware generation problem to optimize the implementation of each layer of a given DNN. This implementation is inspired by the work in [65] where layers' dimensions are encoded into a set of genes to optimize the mapping of DNNs on a given target. However, in this approach the GA is used to search for the right pragmas and tiling parameters to optimize, in the first place, the high-level implementation (C-HLS source code) of a layer and thus to generate

an optimized RTL representation.

DNA encoding

In the current GA implementation, the DNA is first encoded by employing the N dimensions of a layer, which also represent the number of loops of the layer. Those N dimensions are encoded into N pairs of genes. Each dimension is encoded into a gene which has four optimization types: tiling factor (if loop is tiled), pragma-value pair, operators (for innermost loops only, i.e., kernel loops), and Null to express the absence of optimizations. Figure 5.3 sketches the adopted encoding of the DNA. This latter is illustrated as a pair of chromosomes carrying those genes. As it can be seen, the first chromosome, on the left, encodes the loops' order and the tiling parameters. The one on the right, encodes the potential optimizations, such as Pipelining, Unrolling and the use of Operators. Hereinafter the used abbreviations: *Tx* denotes the tiling factor, *P-Val* denotes a certain pragma and its value, *Opx* indicates the Operator to use and *N* for Null and indicates that no optimization is applied. It is worth mentioning that the loops' order is guided by the results of the Characterization step, especially the dimensionality analysis, and fed as it is to the optimizer module. For example, if a given layer has its width bigger than its depth, it means that this layer has higher reuse opportunities. Therefore, shifting a part of the input closer to the computational part by changing the loop' order of the layer to *M, R, C, N, Kx, Ky* (see Listing 4.1), favors data locality and reuse and enhances the overall latency. In contrast, if a given layer has its depth bigger than its width is more likely to have lower reuse opportunities. Consequently, the loops' order can be kept as *R, C, M, N, Kx, Ky*. A layer with a width equivalent to depth can be implemented in both previously discussed loops orders.

The genotype, described in Figure 5.3, is the employed genetic encoding, and the Area-Latency is the physical behavior of the identified solution, called phenotype.

GA evaluation

Once the population is initialized, each individual (solution) in this population is evaluated based on its phenotype. This evaluation consists in attributing, for each individual, a fitness score. The computation of this latter depends on two parameters: the latency and the resource utilization. The individual resources are compared to the ones on the targeted FPGA, while his latency is compared to the one of a non-optimized (classical)

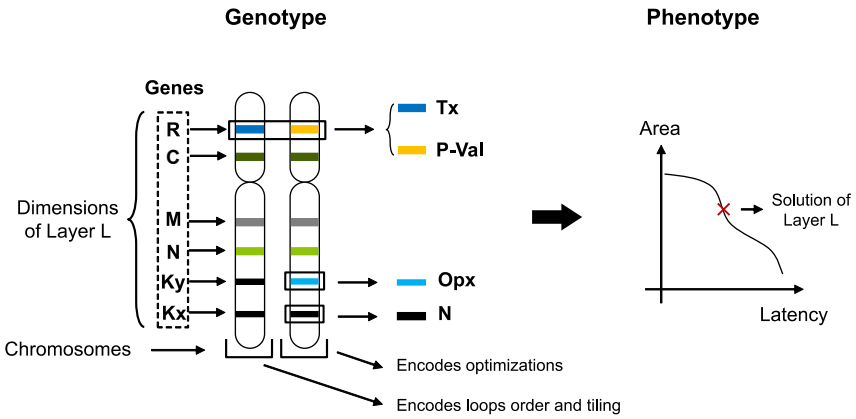


Figure 5.3 – Encoding scheme of the proposed GA' implementation. *Tx*: tiling factor, *P-Val*: pragma and its value, *Opx*: Operator to use and *N*: Null for non-optimization.

implementation. The fitness score is set as follows: if the estimated latency exceeds the latency of the classical implementation, and if the estimated resources surpass the available one on the targeted FPGA regardless of routing, then the fitness is given a -1 value. Else, area and latency of each individual are normalized and then multiplied to compute the fitness score. Equation 5.1 illustrates the fitness function employed in this work, where A and L are respectively the normalized area and latency.

$$Fitness = \begin{cases} L \times A > 0, \ \textit{if constraints met} \\ -1, \ \textit{if constraints not met} \end{cases} \tag{5.1}$$

Therefore, all individuals having a positive fitness are kept. Only Pareto font solutions (individuals) are considered as viable solutions. These latter are also used for the optimization of the entire neural network, which will be detailed in the next chapter.

The resource utilization and the latency of each individual can be obtained by simulating and synthesizing (place and route) each obtained configuration. However, this lengthen the exploration of the design space, since real synthesis is time-consuming. Therefore, resource utilization and latency estimation models are introduced to speed up the exploration phase. These models are detailed in section 5.3.

Regarding the selection process, a probability-based selection is used, the so-called Roulette Wheel. This method ensures that the highest-scoring individual will be most likely to reproduce. It also guarantees the diversity of the population since it does not entirely eliminate any variation.

5.3 Model-based estimations

To reduce design time of hardware implementation, the evaluation time of each proposed solution must be reduced. Hence, instead of synthesizing every proposed solution, resource utilization (DSP, BRAM, LUT, FF) and latency are modeled to imitate the behavior of the employed HLS tool. This allows a faster evaluation as well as a faster convergence towards a suitable solution.

5.3.1 Performance Model

In the context of modeling latency for HLS-based designs, many researchers assume a fixed clock frequency for the design and thus use the number of cycles as a measurement unit, such as the works in [117, 104]. These works built algorithmic convolutional templates suitable for multiple convolutional layers, and built their performance models based on these high-level templates. Thus, the performance is expressed in terms of the dimensions of a given layer. Since the proposed approach in this thesis does not rely on existing templates or architectures to use fixed performance models, an adaptive model had to be developed that takes into consideration possible loop optimizations, such as tiling, unrolling, pipelining and the use of an operator. This model is built based on layers' dimensions, i.e., loop bounds as in Listing 4.1. Hence, it depends on the latency of the loop nest. A performance model for non-optimized layers is used as a building block for this model, in which the latency of the loop is computed recursively starting from the innermost loop (having the highest index) as in Equation 5.2. This model is inspired by the work in [137]. It is worth noting that if an operator is employed, the latency of kernel loops is replaced by the latency of the operator, especially in convolutional and pooling layers.

$$\begin{aligned} IL_l &= IL_{l+1} \times LB_{l+1} + P \\ LL_l &= IL_l \times LB_l \end{aligned} \tag{5.2}$$

In Equation 5.2, IL_l, LB_l and LL_l are respectively the *Iteration Latency*, the *Loop Bound* and the *Loop Latency* of loop l. IL_{l+1} and LB_{l+1} are the *Iteration Latency* and *Loop Bound* of the nested loop. P is the pipeline depth of the innermost loop.

The proposed latency model can compute the latency of a nest of multiple loops in a recursive way. It takes into account various optimizations: UNROLL, PIPELINE and

selected operators performances.

5.3.2 Models for resource utilization

Unlike ASICs where area is expressed in mm^2, area on FPGAs is expressed in terms of resource utilization. Resources are four types: *DSP, FF, LUT* and *BRAM*. Therefore, FPGA resources are modeled separately to estimate the resource utilization of each layer of a given CNN; The resources of the entire CNN can be later computed by aggregating the resources of all its layers. Various layer-based high-level parameters and configurations are employed to model resource utilization, especially the layers loop bounds, the number of parameters, the volume of data to be processed and the input/output depths. In addition, these models take into consideration the presence of an operator (from the dataset of HLS-based operators) in a layer.

In our work, statistical analysis is used to model resource utilization by trying to find viable links between the parameters stated above. For this purpose, a set of 70 microbenchmarks of various layers coded in C-HLS were simulated. This set includes 20 non-optimized layers having different dimensions and types, which comprises the most common layers used in state-of-the-art DNNs. It also includes 30 optimized layers (having the same dimensions as the non-optimized layers), using pipelining and unrolling as optimizations. The remaining 15 layers are optimized using operators. These microbenchmarks were then synthesized for different FPGA targets. These layers have different types and dimensions. Syntheses results of every layer were gathered in a database.

Estimating DSP utilization

In the first place, modeling *DSP* usage relies on the innermost loop in convolutional and fully connected layers, since it is where the MAC operation resides. Performed microbenchmarks have shown that a MAC operation is mapped onto one DSP, even if a pipelining directive is employed. In a convolutional loop nest, if the two innermost loops are substituted with a specific HLS operator, then the number of DSP is the same as that of the chosen operator. If unrolling directives are used to optimize the loop nest, then the number of DSP is equal to the product of unrolling factors of unrolled loops. The number of DSP is dictated by Equation 5.3 where U_l is the unrolling factor of loop level l, and DSP_{Op} is the number of DSP of the substitute operator.

$$Number\ of\ DSPs = \begin{cases} 1, & \textit{if no optimization OR loop } l \textit{ is pipelined only} \\ DSP_{Op}, & \textit{if operator only} \\ DSP_{Op}, & \textit{if operator AND loop } l \textit{ is pipelined} \\ \prod_{l=0}^{L-1} U_l \times DSP_{Op}, & \textit{if loop } l \textit{ unrolled AND operator} \end{cases} \tag{5.3}$$

It is worth noting that pooling layers do not use DSPs since the computational part is based on comparisons only. Therefore, the tool does not use DSP resources to implement a pooling layer.

Estimating BRAM, LUT and FF utilization

The collected data of the 70 microbenchmarks have shown a dependency on layers properties, such as dimensions, number of parameters, volume of input/output data as well as the input depth. It first showed that the number of BRAM depends on the input and output depths of a layer, and that the numbers of LUT and FF are affected by the number of BRAM.

Due to these dependencies, it is quite challenging to model BRAM, LUT and FF utilization. Therefore, statistical methods are chosen as a way to model those resources. These methods are tested to check if they are capable of outputting reliable models that accurately predicts the value of each resource. Thus, various statistical tools are tested, Among these tools is the NCSS software (Number Cruncher Statistical System) [93]. NCSS is selected to accomplish this task, since it is one of the best tools that provides a complete collection of statistics, graphics tools and model fitting functions.

Therefore, NCSS is used to process and analyze the collected data to find reliable resource models. The NCSS-based analysis showed, along with layer-related configurations, a chain dependency between the number of BRAM, LUT and FF.

Some techniques offered by the NCSS software were tested to find reliable models. The Robust Multiple Regression (RMR) technique was a good candidate, since it found a relationship between the number of BRAM and the input and output depths of the layers.

It is worth noting that the mean square error (MSE) of the BRAM model is *6.69%*.

The resulting BRAM model is represented in Equation 5.4, where I is the input depth and O is the output depth. The coefficients of this equation as well as all the following

ones in this chapter are provided by the NCSS tool.

$$\begin{aligned} N_{BRAM} =& 0.87 - 0.058\times I + 0.0169\times O - 0.0017\times I^2 + 0.0043\times I\times O - \\ & 0.0014\times O^2 \ + 4,26\times 10^{-5}\times I^3 - 8,14\times 10^{-5}\times O\times I^2 + \\ & 4,03\times 10^{-5}\times I\times O^2 - 5.12\times 10^{-6}\times O^3 \end{aligned} \tag{5.4}$$

RMR was not able to find a viable relationship between layer-based configurations and the number of LUT and FF. Therefore, the Polynomial Regression (PR) method is employed as a substitute, since it lead to better FF and LUT models. Based on the analysis of the NCSS software, a relationship between the number of LUT and the number of BRAM is identified. In addition, the analysis also showed that the number of LUT depends on the number of parameters, the number of input pixels and the input depth. The output LUT model is a ratio of polynomials of order *3* taking as input four variables, as presented in Eaquation 5.5. The MSE of the resulting LUT model is *1.81*%.

$$\begin{aligned} N_{LUT} =& (10.45 - 0.094\times BRAM + 0.19\times BRAM^2 + 0.0012\times BRAM^3 + 2.88\times 10^{-5} \\ & \times NB\_PIX - 2.12\times 10^{-6}\times BRAM\times NB\_PIX + 1.21\times 10^{-6}\times BRAM^2 \\ & \times NB\_PIX + 8.94\times 10^{-12}\times NB\_PIX^2 - 1.04\times 10^{-10}\times BRAM\times NB\_PIX^2 \\ & - 7.0\times 10^{-18}\times NB\_PIX^3 + 9.16\times 10^{-5}\times NB\_PARAM - 4.74\times 10^{-5}\times BRAM \\ & \times NB\_PARAM - 3.27\times 10^{1}0{-}6\times BRAM^2\times NB\_PARAM - 4.44\times 10^{-9} \\ & \times NB\_PIX\times NB\_PARAM + 2.53\times 10^{-10}\times BRAM\times NB\_PIX\times NB\_PARAM \\ & + 6.28\times 10^{-14}\times NB\_PIX^2\times NB\_PARAM - 2.87\times 10^{-8}\times NB\_PARAM^2 \\ & + 1.56\times 10^{-9}\times BRAM\times NB\_PARAM^2 - 1.78\times 10^{-14}\times NB\_PIX\times NB\_PARAM^2 \\ & - 6.93\times 10^{-14}\times NB\_PARAM^3 - 0.072\times I - 0.022\times BRAM\times I - 0.002\times BRAM^2\times I \\ & + 2.42\times 10^{-7}\times NB\_PIX\times I - 1.32\times 10^{-7}\times BRAM\times\times NB\_PIX\times I \\ & - 7.75\times 10^{-13}\times NB\_PIX^2\times I + 1.35\times 10^{-5}\times NB\_PARAM\times I \\ & + 5.54\times 10^{-7}\times BRAM\times NB\_PARAM\times I + 6.07\times 10^{-11}\times NB\_PIX\times NB\_PARAM \\ & \times I + 2.74\times 10^{-10} * \times NB\_PARAM^2\times I + 0.0002\times I^2 + 0.0001\times BRAM\times\times I^2 \\ & - 6.79\times 10^{-10}\times NB\_PIX\times I^2 - 8.96\times 10^{-8}\times NB\_PARAM\times I^2 \\ & - 9.34\times 10^{-8}\times I^3) \ /(1 + 6.43\times 10^{-22}\times BRAM\times NB\_PIX\times NB\_PARAM^2\times I^2) \end{aligned} \tag{5.5}$$

Regarding the number of FF, it depends on the number of LUT, since it is a chain dependence. Additionally, the number of LUT also depends on the number of parameters, the number of input pixels and the input depth. The resulting FF model is also a ratio of polynomials of order *2* taking as input four distinct variables, as presented in Equation 5.6. Its MSE is *8*, *83*%.

$$
\begin{aligned}
N_{FF} =& (82.49 - 0.97 \times LUT + 0.0014 \times LUT^2 + 0.0016 \times NB\_PIX \\
& - 5.88 \times 10^{-6} \times LUT \times NB\_PIX - 4.69 \times 10^{-9} \times NB\_PIX^2 \\
& + 0.0015 \times NB\_PARAM - 2.51 \times 10^{-6} \times LUT \times NB\_PARAM \\
& - 1.13 \times 10^{-8} \times NB\_PIX \times NB\_PARAM + 4.62 \times 10^{-9} \times NB\_PARAM^2 \\
& - 1.23 \times I + 0.011 \times LUT \times I - 7.30 \times 10^{-7} \times NB\_PIX \times I \\
& - 1.7 \times 10^{-5} \times NB\_PARAM \times I + 0.0026 \times I^2) \; / \\
& (1 - 0.013 \times LUT + 3.35 \times 10^{-5} \times LUT^2 + 1.16 \times 10^{-5} \times NB\_PIX \\
& - 4.6 \times 10^{-8} \times LUT * NB\_PIX - 3.057 \times 10^{-11} \times NB\_PIX^2 \\
& + 1.65 \times 10^{-5} \times NB\_PARAM - 7.35 \times 10^{-9} \times LUT \times NB\_PARAM \\
& - 3.66 \times 10^{-10} \times NB\_PIX \times NB\_PARAM + 4.45 \times 10^{-11} \times NB\_PARAM^2 \\
& - 0.013 \times I + 0.0001 \times LUT \times I + 4.23 \times 10^{-8} \times NB\_PIX \times I \\
& - 1.74 \times 10^{-7} \times NB\_PARAM \times I + 3.27 \times 10^{-5} \times I^2)
\end{aligned}
\tag{5.6}
$$

5.3.3 Discussion on obtained models

Such modeling approach, especially the modeling of BRAM, FF and LUT, was not previously used in the state of the art, at least in the analyzed literature. It uses statistical analysis to find a solid relationship between the resource utilization and the characteristics of the application (i.e. the layers in this case), which is also a way to bring closer the hardware and the software parts.

Although the proposed models showed a low MSE value, some parameters must be considered when devising them. For instance, considering optimized and non-optimized layers in the microbenchmarks does not mean that the model is capable of predicting accurate value of latency and used resources, since it does not have the information if a layer is optimized or not, which make them more suited for non-optimized layers mostly.

Therefore, gathering additional data related to the employed optimization, especially its type (e.g. unrolling, operator, etc.) and its location (loop index), is mandatory. The added data must thoroughly be analyzed to tighten the links between the layers' properties and the hardware resources. In addition, microbenchmarks layers do not cover all the layers in the literature, which means more layers and syntheses are required to expand the usage of such models. However, this remains a promising approach since no training is needed to create the estimation models. Such models can provide accurate predictions once they are correctly created. In addition, the estimation process is quite fast.

Alternative approaches like Aladdin [115] can be used to provide latency and area estimates without the need to generate the RTL. On the other hand, deep learning approaches can be used to create a predictive model or even models for area and performance. However, such approach requires a larger database to have enough data for learning, testing and validation and to prevent overfitting. In this case, more syntheses are required to gather the needed data, which is time-consuming. Despite that, once the DNN model is trained, inferring results does not require a lot of time.

5.4 Early results for DNN layer implementation optimization

This section presents few results on the Optimizer module to assess the interest of using GA for this optimization problem. It also show the use of the performance and area models to estimate each solution in the solution space. Therefore, the GA is applied to find suitable solutions for a few layers, as an illustration for the proposed approach. The chosen layers are the same used in the previous chapter in Section 4.4, in which Table 4.3 shows the type of each layer, its default loops order and its dimensions

First of all, the mutation rate and the population size of the GA are properly set to ensure an appropriate functioning of the algorithm. In here, the mutation rate is set to 0.1, which means for each gene, there is a 0.1% chance that it will mutate. Regarding the number of individuals in the population, it is set to 300 to allow diversification.

Before searching for the right optimizations, the DNA of each layer must be encoded based on its dimensions. However, the loops order is dictated by the Characterization module. This latter provides a dimensionality analysis that allows to set the appropriate loops order in a loop nest. For instance, layer *L1* has its width (128) bigger than its depth 3, this means that this layer has higher reuse opportunities in the 2-D plane. Therefore,

the loop order that is more likely to encourage reuse is the M, R, C, N, Kx, Ky loop order. This allows to move a great part of the image to be closer to the computational part and consequently enhances performance and memory accesses. Hence, the DNA of layer *L1* is encoded using the M, R, C, N, Kx, Ky loop order.

As for layer *L2*, its depth (32) is larger and its width (16), which means that less reuse opportunities are available in the 2-D plane. However, in this case, filter reuse could be promoted by processing the input feature maps depth-wise, which improves the weights locality. Hence, the loops order should remain R, C, M, N, Kx, Ky.

Regarding the last layer in the table, layer *L3*, the analysis is similar to layer *L1*, since its width (16) is greater than its depth (2). Consequently, the adopted loop order that will be encoded into the DNA is the M, R, C, N, Kx, Ky.

After set the order of loops and encoding the DNA, the GA can now search for appropriate solutions to implement those layers. The GA found various solutions for each layer, which are the Pareto front solutions. Figure 5.4 shows the found solutions of layer *L1*, where the blue points are the Pareto front. The same goes for layers *L2* and *L3*, their respective solution spaces are presented in Figure 5.5 and Figure 5.6. It is worth noting that the area in these figures is normalized using Equation (4.1) in Chapter 4.

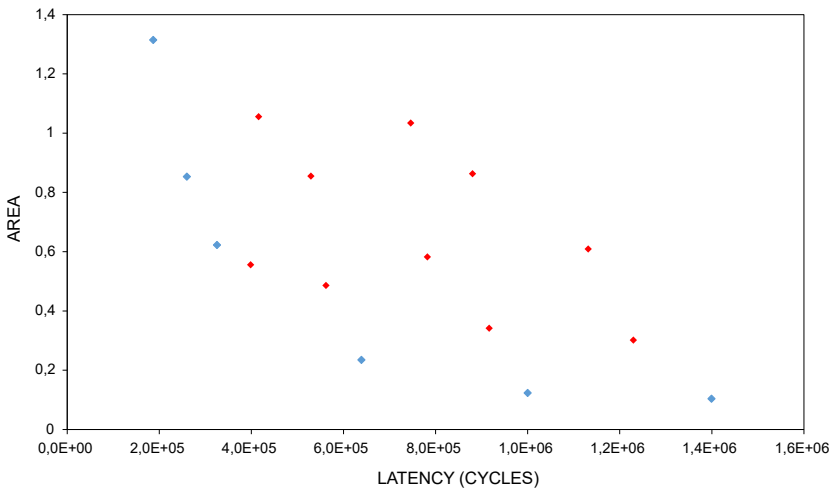


Figure 5.4 – The GA search space of layer L1.

Although various solutions are identified by the GA, in the final implementation only Pareto front solutions are considered. The set of results found by the GA is a finite solution space, and thus selecting one solution depends on the design goal, which can be to either

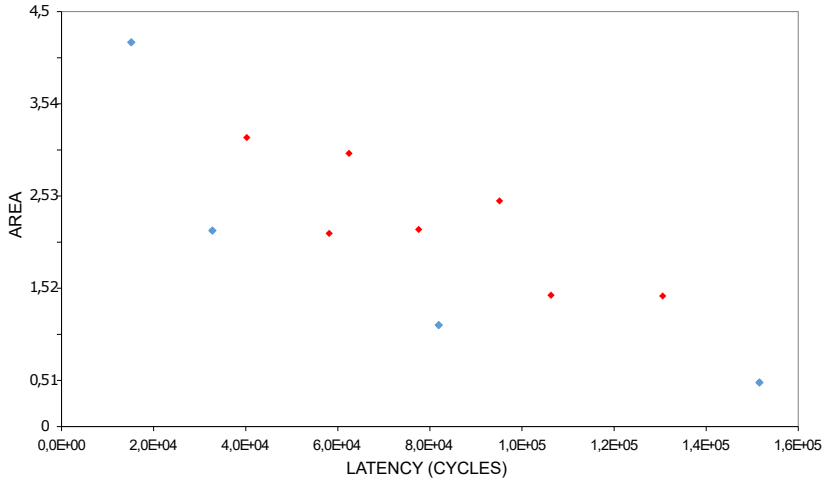


Figure 5.5 – The GA search space of layer L2.

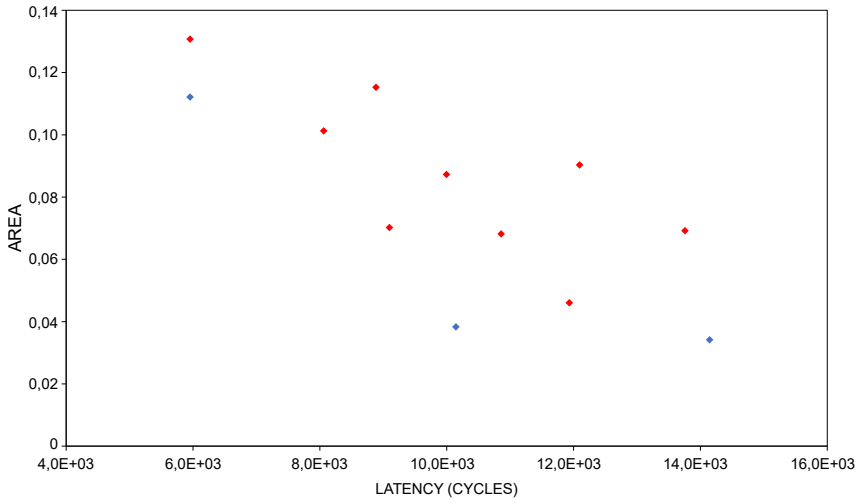


Figure 5.6 – The GA search space of layer L3.

optimize performance-wise or area-wise or to have a tradeoff of both. However, the main purpose of the proposed methodology is to implement an entire DNN, instead of a single layer, thus the solution of each layer should be chosen carefully to fit the whole CNN on the target FPGA. This issue will be the tackled in the next chapter.

5.5 Conclusion

This chapter presented the layer-based optimization part of the Optimizer step. This module uses the characterization results, especially layers dimensions and data reuse features, to optimize the C-HLS source code of each layer of a given DNN. It can take into account different types of optimizations, such as pragmas, tiling factors and HLS-based operators. A specific genetic algorithm was developed to find the Pareto-optimal configurations for each layer. However, selecting a relevant solution from a Pareto-optimal set of solutions is challenging. In addition, finding an optimized implementation for the entire CNN cannot be achieved by only using a per layer GA-based optimization. Therefore, another algorithmic layer is needed to optimize the implementation of the whole network on a FPGA target.

The following chapter will present the complete implementation of the purposed methodology, called SHEFTENN, along with the missing piece that allows generating an accelerator for the entire CNN by using layer-based optimization. Furthermore, it will detail implementations of various state-of-the-art DNNs to validate the proposed approach.

Chapter 6

Implementation of the SHEFTENN Framework and Assessment

As illustrated in previous chapters, SHEFTENN is an end-to-end automated framework that consists of three main modules working together to bring the hardware architecture and the high-level application closer and reduce design time. Chapter 3 detailed the first step of the proposed methodology, called Characterization, which performs a thorough investigation of the CNN algorithm with a hardware perspective, and computes metrics to derive augmented specifications (see Figure 3.2). Following, Chapter 4 described the second step of SHEFTENN, called Hardware Generation, that generates from high-level descriptions (e.g. C/C++ source code) an RTL source code of the CNN accelerator (see Figure 4.2). It also exploits augmented specifications to incorporate loop transformations and set the right bit-widths in the high-level description, while also using a database of HLS-based operators in the generation process. Finally, Chapter 5 presented the last step of the proposed methodology, called Optimizer, which aims at optimizing the RTL source code generation by relying on a high-level model of the architecture. The Optimizer module uses a hybrid algorithm that selects HLS tool-specific pragmas and operators to further optimize the implementation with respect to latency and resource utilization.

Previous chapters thoroughly introduced the different steps of the proposed methodology, presented possible implementations and validated them by providing per-layer results based on trained CNNs. This chapter presents the overall implementation of the proposed methodology while exposing each of its modules. It also introduces an additional method to the previous chapter that limits the solution space of each layer and provides an implementation for the entire CNN. In addition, it presents complete results showing how the modules work together to generate hardware accelerators for complete CNNs instead of a layer by layer hardware generation. These results enable to assess the relevance of the SHEFTENN framework to efficiently generate CNN accelerators.

This chapter is organized as follows. Section 6.1 presents the whole implementation of the SHEFTENN framework and introduces the additional step in the Optimizer that allows generating a full DNN accelerator. Section 6.2 details the results of the implementation of state-of-the-art networks. Finally, Section 6.3 sketches the conclusion of the chapter.

6.1 Implementation of the SHEFTENN framework

This section provides a description of each module of the framework and the communication between them to generate optimized CNN hardware accelerators. Relying on the proposed methodology depicted in Section 2.3, SHEFTENN is instantiated and presented with an increased level of details in Figure 6.1. The main elements of the Characterization (in green), the Hardware Generation (in blue) and the Optimizer (in orange) modules are represented, as well as supporting databases of components (operators, algorithms). More details in the following subsections.

6.1.1 Characterization module

As explained in Chapter 2, an in-depth characterization of DNN applications is essential to get the required knowledge to design efficient hardware accelerators. This module characterizes and investigates the input DNN, and then produces application-related augmented specifications. As a preliminary step, a parsing script implemented in Python processes the DNN description file. The Python script is adaptable to the input file format (TensorflowLite, ONNX or N2D2, optionally including network parameters/weights). Description files mainly contain the topology of the DNN, the order, types as well as the configuration of each layer. They may also contain the parameters of the neural network, if the file format supports this option. To support this variety of the input descriptions, suitable parsing libraries are imported to the Python script. The N2D2 *.ini* DNN configuration files are employed in here, since the N2D2 open source framework is used as a front-end in the experiments. A sample of a *.ini* file, describing two convolutional layers, is presented in Listing 6.1 below.

The input DNN description is transformed into an IR. Then, a Python script uses this IR to compute target-agnostic/hardware-aware metrics for each layer. Then, it extracts data-related metrics and augmented specifications (cf. Section 3.1.1). The IR is updated

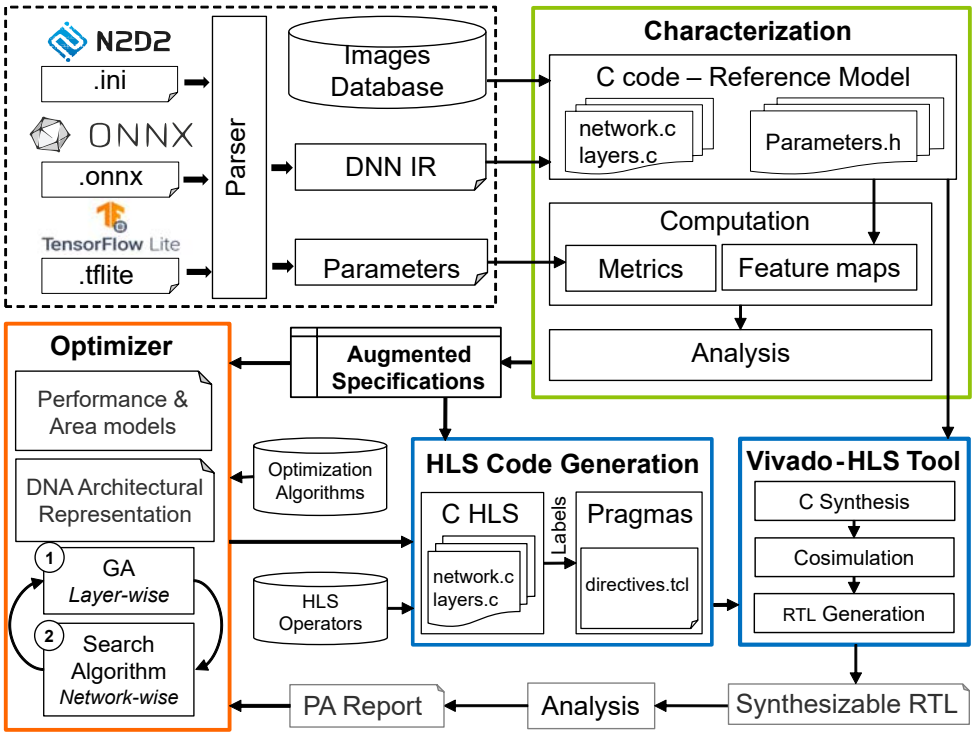


Figure 6.1 – A detailed description of each module of the instantiated SHEFTENN framework, where: the characterization module computes and analyses metrics, the HW generation generates C-HLS code and leverages a database of HLS-based operators and uses HLS tools to generate the RTL, and optimizer uses a genetic algorithm to optimize the RTL by setting the right configurations for the C-HLS code.

```
[conv1] conv_def
Input = sp        ;#input image
KernelHeight = 3
KernelWidth = 3
NbOutputs = $(int(32 * ${ALPHA}))
Stride= 2
Padding = 1

[conv1_3x3_dw] conv_def
Input = conv1
KernelHeight = 3
KernelWidth = 3
NbOutputs = $(int(32 * ${ALPHA}))
Stride = 1
Padding= 1
```

Listing 6.1 – Configuration of two convolutional layers in a N2D2 *.ini* file.

with newly computed metrics. The Characterization module also verifies if the CNN is quantized by checking parameters bit-widths to set the right bit-precision-related optimization in the Hardware Generation module. This module analyzes the behavior of the entire CNN to help drive the C-code generation. Moreover, the Characterization module has a C-HLS-friendly code generation step (it generates the source code of the entire CNN) that serves as a reference model for both dynamic analysis and validation of the hardware generation using co-simulation.

6.1.2 Hardware Generation module

The Hardware Generation module leverages HLS to abstract the implementation of the hardware in SHEFTENN. This module exploits a dataset of HLS operators and augmented specifications to generate an optimized RTL representation of the entire DNN accelerator. As explained in section 4.1, a code generation module (blue box in Figure 6.1) generates an optimized and synthesizable (HLS-friendly) C source code based on the characterization results. These optimizations consist of loop transformations, such as loop reordering in early layers, to encourage data locality and reuse to reduce data transfer time. In addition, for better optimizations, the C-HLS code incorporates compiler directives associated to the operators that could replace kernel loops (more details in Section 4.3). Besides, the Hardware Generation module generates a separate file including tool-specific directives (pragmas) to optimize each layer as well as the whole DNN. Pragma-value pairs, compiler directives, tiling factors are all set by the Optimizer module. Table 4.2 in section 4.2 summarizes the employed optimizations. The considered HLS tool in the SHEFTENN implementation is Vivado-HLS since Xilinx FPGAs are currently targeted by the framework. Once all optimizations are set, this module synthesizes the resulting C-HLS code, then co-simulates the generated code and compares the results with the ones of the reference model to check the right functioning of the design before generating the RTL.

6.1.3 Optimizer module

The Optimizer module takes as input the Characterization' results, the configuration of each layer and the resources of the targeted FPGA. Then, it optimizes the C-HLS source code (generated in the previous module). It determines the pragma-value pairs to be applied, the operators to be employed as well as the tiling parameters. This module

reduces the number of synthesis runs by using area and performance models to evaluate the different solutions during design space exploration, as detailed in section 5.3. Furthermore, a real synthesis is established every *20* iterations to ensure that area and latency estimations of the whole DNN are not deriving far from real synthesis values. This number of iterations was chosen for instantiation purposes, and can be replaced by a higher or lower number, enabling to set up a tradeoff between accuracy and speed. The exploration of additional parameters (e.g. number of iterations) is part of the perspectives, in particular the exploration of the tradeoff between speed and models precision.

The GA of optimizer module outputs the best configurations per layer only by using layer-based metrics and outputs optimized configurations based on an area-performance trade-off (cf. Chapter 4, Section 5.2.2). However, at this stage the entire neural network is not optimized. Hence, another algorithm is required to optimize the whole DNN by using the previously obtained layer-based configurations. The next subsection presents this algorithm.

Network-wise Optimizations

The second step of the Optimizer module, *optimizeNetwork* (in Listing 6.2), focuses on optimizing the implementation of the entire DNN. It takes the set of the previously found configurations and searches for a satisfying implementation that fits the DNN into the FPGA target. Precisely, it searches for a Pareto-optimal solution for every layer. In addition, this step exploits the characterization results to find similarities between layers in terms of kernel shapes, number of parameters and the volume of input and output data. For instance, if two or more layers have the same features, only one C-HLS function implementing this layer is kept to execute all the similar layers. Consequently, this step picks the best GA-optimized configuration and uses it to implement the selected layer. These layers are later allocated to the same RTL resources by using the allocation pragma.

Allocation It is a synthesis directive employed inside a function, a loop or a region of code, to manage resource usage at RTL level. Precisely, it allows the user to set the number of RTL instances to limit the resources implementing particular loops, functions, cores or operations. For instance, calling the same function n times in a C code implies instantiating n RTL modules. Using this pragma allows implementing all instances of the high-level function using the same RTL resources, or limiting the RTL modules to be instantiated to a certain number. Despite reducing resource usage, the performance is negatively affected.

```
optimizeImplementation(layersConf, hwResources)
    // A: Area, L: Latency
    while !A and !L
        // GA algorithm
        network = optimizeLayers(layers)
        // Search algorithm
        optimizeNetwork(network)
        A, L = evaluateNetwork(network)
    return network
```

Listing 6.2 – Pseudo-code of the multi-objective optimization process.

Consequently, a two-step iterative optimization process is performed by the Optimizer module until at least one suitable solution for implementing the whole network is found. The optimizer module sets a standard implementation if no solution is found after *100* iterations. Listing 6.2 summarizes the two-step process. The first step, *optimizeLayers*, uses a GA to optimize each layer individually. The next step optimizes the implementation of the whole network.

The next section presents the experimental results that allowed to evaluate the performance of SHEFTENN in the generation of efficient CNN hardware accelerators.

6.2 Experiments and Results

6.2.1 Experimental Setup

To evaluate and validate the SHEFTENN framework, two state-of-the-art networks were employed as workloads: MobileNet-V1 [57] and SqueezeNet-V1.1 [58]. Both networks were trained on the ImageNet dataset by using the open-source N2D2 framework. After training, these networks were quantified to *8*-bit using Post-training quantization, explained in [88].

GA configuration

To correctly set the GA parameters, numerous simulations of various layers from MobileNet-V1 and SqueezeNet-V1.1, varying the population size (from *100* to *400*) and the mutation rate (from *0.1* to *0.4*), were performed. The maximum number of generations is set to *100* in those simulations. The impact of modifying the population size and

the mutation rate of different layers, on the number of generations and thus the speed for obtaining viable solutions, is illustrated in Figure 6.2. For instance, in Figure 6.2(a) representing a mutation rate of *0.1*, it can be seen that the majority of the layers require less than *20* generations to find appropriate solutions. Additionally, increasing the number of individuals in a population reduces the number of required generations for almost all layers. However, for a population size of *400*, the GA converges slowly for two layers, since the number of individuals in the population is large and the mutation rate is small, which limits the variety in the population. Raising the mutation rate from *0.2* to *0.4*, Figure 6.2(b) - 6.2(d), increased the number of required generations for the GA to converge, since the population is less stable. Based on this analysis and the fact that a few generations means fast convergence, the mutation rate is set to *0.1*, for which the GA converges faster, and the population size is set to *300*, since the algorithm found fit solutions for mostly all layers with this configuration.

Hardware implementations

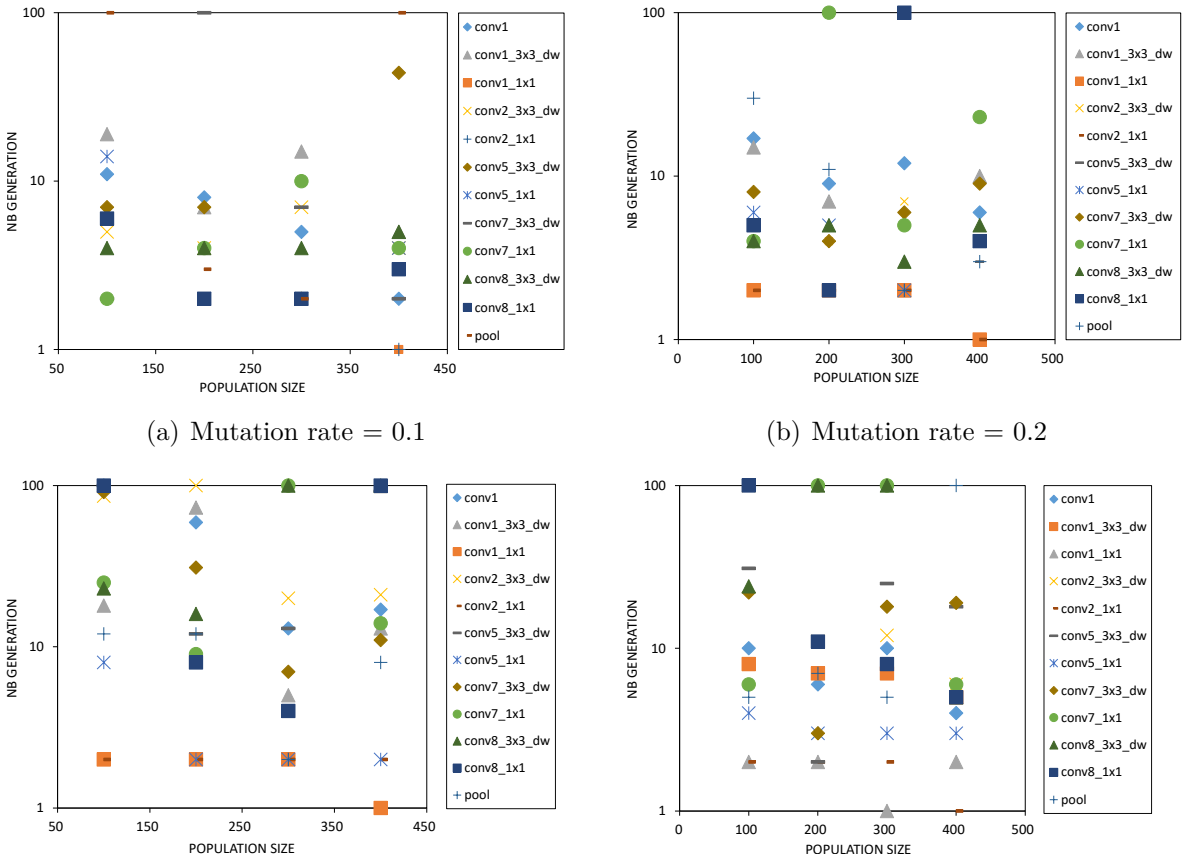
To generate the RTL source code of the resulting high-level optimized implementation fo the CNN accelerator, Vivado-HLS 2020.1 is used. All implementations are performed by using Xilinx FPGAs as targets, assuming a fixed frequency of 100MHz. Besides, both networks are implemented (place and route), omitting memory controllers and crossbars, since there is no intermediate memory control. Furthermore, the main focus is on the computational part. Memory footprint is handled in the Characterization module, which performs a memory analysis on a layer by layer basis.

6.2.2 SHEFTENN Evaluation using MobileNet-V1

MobileNet-V1 is an adjustable CNN thanks to its hyper-parameter α. Herein, MobileNet-V1 is configured, before training, as follows: α is set to *0.25* (to reduce the depth of the network) and the dimensions of the input image are $3 \times 128 \times 128$.

Characterization results exploited by the Optimizer module

First, the DNN is analyzed, in the Characterization module to extract relevant information, such as computation needs, memory requirements and reuse opportunities. For instance, Figure 6.3 shows the evolution of the input feature maps in terms of number of input activations, as well as the input and output depths across the network; the number



(a) Mutation rate = 0.1

(b) Mutation rate = 0.2

(c) Mutation rate = 0.3

(d) Mutation rate = 0.4

Figure 6.2 – The impact of varying population size (from 100 to 400) and mutation rate (from 0.1 to 0.4) on the number of generations.

of pixels is used as a measurement unit. As it can be seen, the number of input activations decreases in the last few layers where the size of input feature maps diminishes due to the use of convolutional filters. In contrast, the input and output depths increases throughout the DNN due to the large number of filters in deeper layers. The optimizer module exploits this information, especially in the area and performance models, to estimate the resource usage of each layer based on its configuration.

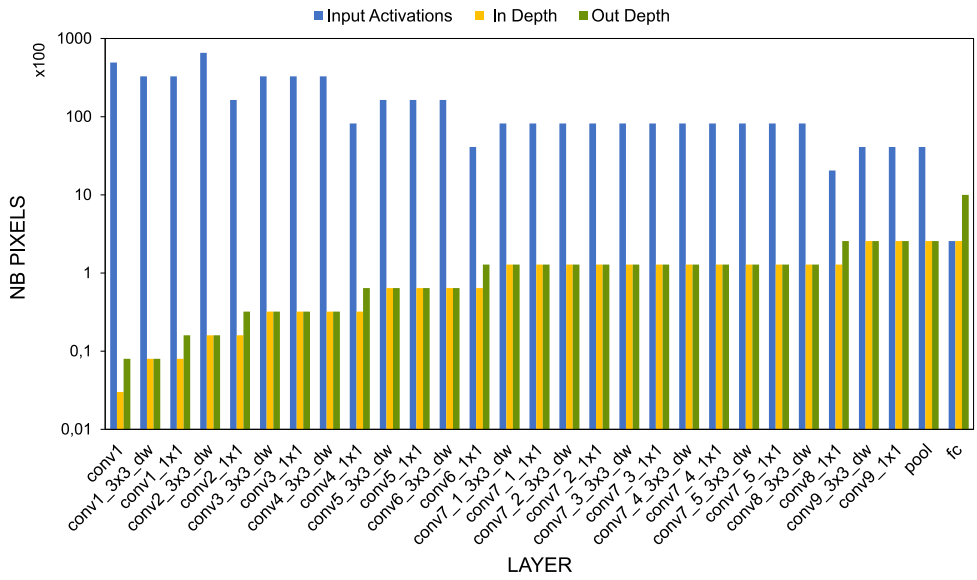


Figure 6.3 – Variation of the input pixels of each layer along the MobileNet-V1 network.

A different way to exploit the width and depth metric is by combining it to the reuse percentage metric and using this combination to determine the loops order, which will be later employed to encode the DNA for the GA implementation. Figure 6.4 sketches both metrics, the widths and depths evolution of the input feature maps and the reuse percentage for each layer of MobileNet-V1. The reuse percentage varies according to the width and height of the image and the stride. As it can be seen in Figure 6.4, the reuse ratio is lower than *30%* in early layers, in which the stride has a value of *2*, such as *conv1*, *conv2_3×3_dw* and *conv4_3×3_dw*. In addition, layers in which the width of the input feature map is less than or equal to its depth have a low reuse ratio. These layers are implemented in the R, C, M, N, Kx, Ky loop order (see Listing 4.1 in Section 4.1.1), since the reuse percentage is small. Every *conv1×1* in this CNN has a high reuse ratio, since all input activations (pixels) are equally used. These layers are implemented in the M, R, C, N, Kx, Ky loop order to promote data reuse and reduce expensive mem-

ory transfers. A layer with a width equivalent to its depth can be implemented in both previously discussed orders. An additional aspect is the depth of the feature map that helps identifying parallelism opportunities where appropriate loop optimizations could be applied. Hence, layers having a depth larger than a width could be tiled along the M and N dimensions to encourage parallelism along outputs and channels respectively.

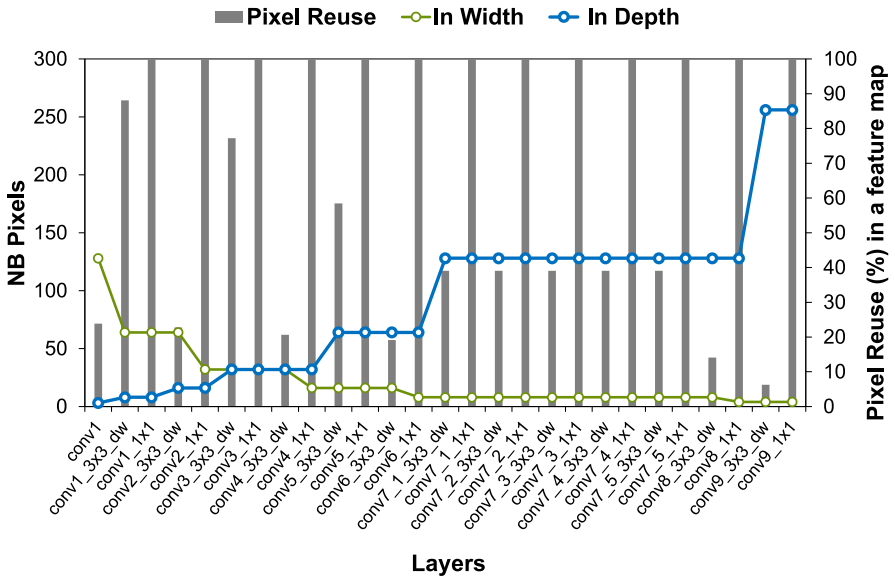


Figure 6.4 – Variation of the input pixels of each layer along the MobileNet-V1 network.

Other exploited information by the Optimizer module are the number of parameters (weights) and the number of operations (MACs for convolutional and fully connected layers and Ops for pooling layers). The number of weights is employed for area models to estimate resource usage, especially LUT and FF models as detailed in Section 5.3. The number of operations is used to dictate the computational latency of non-optimized layers, since the product of all bounds in a loop nest presents the number of MAC operations, where *1* MAC operation takes *1* clock cycle. Multiplying all loop bounds in a loop nest yields the number of MAC operations, especially in convolutional and fully connected layers. As it can be seen in Figure 6.5, the number of parameters increases throughout the network, since the number of filters increases in the deepest layers of the DNN. On the other hand, the number of operations does not have a consistent evolution due to the types of layers and the employed stride values. For instance, MobileNet-V1 uses *depth-wise* convolutions, a channel-wise convolutional computation, to reduce the computational

complexity and accelerate the inference phase [57].

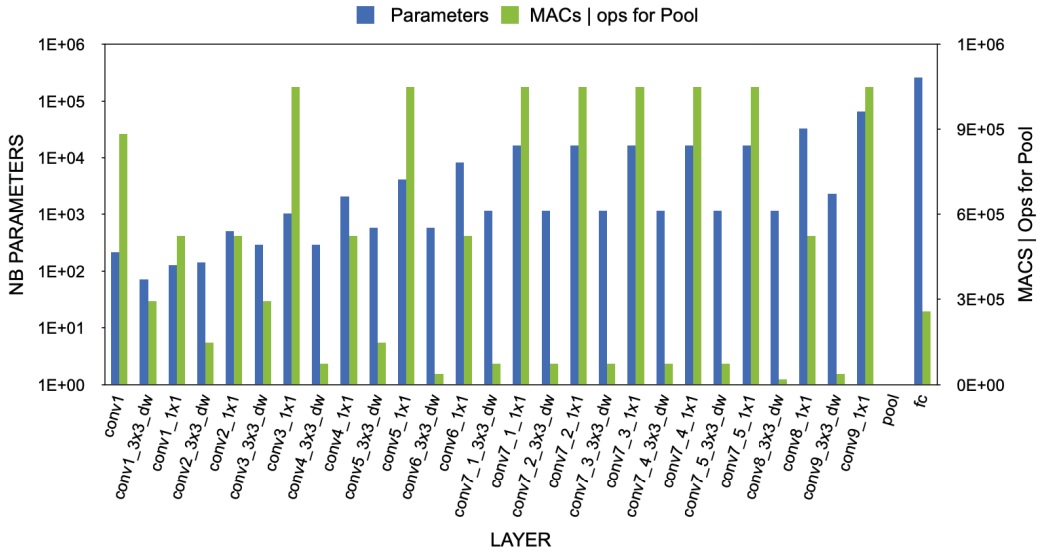


Figure 6.5 – Number of parameters and required computation of each layer of MobileNet-V1 in terms of number of MAC for convolutional layers, and number of Operations for pooling layers.

Optimizer module results

The Optimizer module performs per layer optimizations by using an intermediate representation of the network, including layers dimensions and types. The order of loops is an essential information for the GA, since the DNA material is based on it. It is set based on the dimensions of the layers and to the reuse opportunities.

Once all the required information regarding the loops order and tiling possibilities are acquired by the GA, it then runs until a set of pertinent solutions is found. This found set comprises various solutions having different combinations of surface and latency, enabling the search algorithm to pick one satisfactory solution for each layer based on their computing and memory needs as well as their position in the CNN. Figure 6.6 shows various configuration points of a few layers of the MobileNet-V1 network, convolutional layers in Figure 6.6(a) - 6.6(g) and pooling layer in Figure 6.6(h). As it can be seen, area and latency vary in opposite ways, i.e., optimizing the area by decreasing resource usage comes at a performance cost, and reducing the latency (optimizing the performance) comes at an area cost. Since the search algorithm selects a solution in a set of Pareto-optimal solutions for each layer, it is worth noting that only Pareto-optimal solutions will

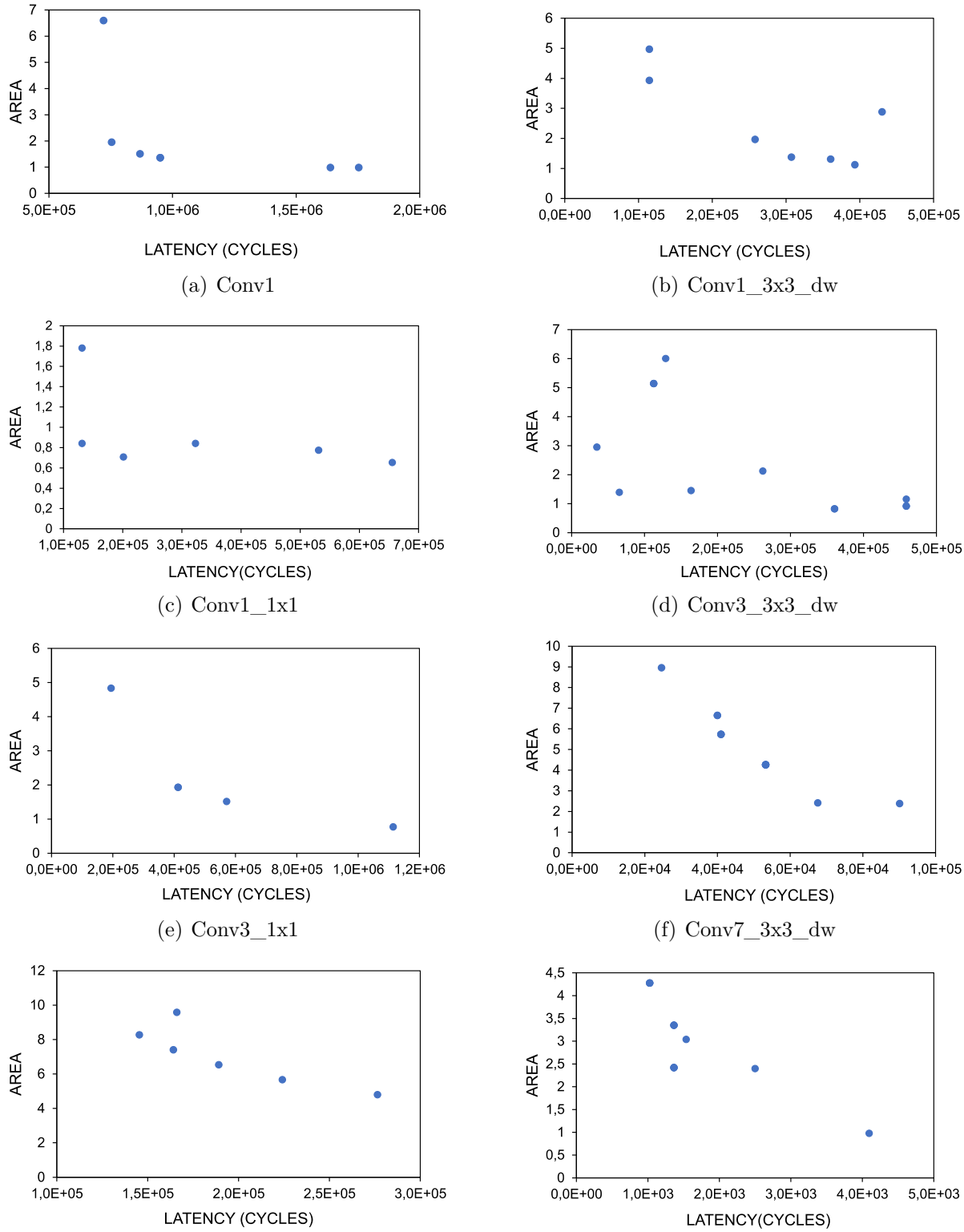
Layers	Loops Order	Optimizations	LUT	FF	DSP	BRAM	Latency (Cycles)
conv1	M, R, C, N, KY, KX	U-1; N; N; P-5; Op-*conv3_sepfunc4*; N	511	257	9	4	529920
conv1_3x3_dw	M, R, C, N, KY, KX	N; N; N; P-1; Op-*conv3_sepop3*; U-1	401	293	9	1	258048
conv1_1x1	M, R, C, N, KY, KX	U-1; N; N; U-1; U-1; P-1	75	100	1	1	1310720
conv2_3x3_dw	M, R, C, N, KY, KX	P-5; N; U-1; N; Op-*conv3_sepop3*, N	419	200	9	1	122564
conv2_1x1	M, R, C, N, KY, KX	N; N; P-6; U-2; N; N	291	233	2	3	258144
conv3_3x3_dw	M, R, C, N, KY, KX	N; N; N; U-1; P-3, U-2	670	423	2	1	262144
conv3_1x1	M, R, C, N, KY, KX	P-1; U-2; U-4; N; N, N	392	150	3	1	2530130
conv4_3x3_dw	R, C, M, N, KY, KX	U-1; N; P-8; N; Op-*conv3_sepop3*, N	952	392	9	1	68096
conv4_1x1	M, R, C, N, KY, KX	U-8; N; N; N; N; P-2; N	147	129	1	2	1157882
conv5_3x3_dw	R, C, M, N, KY, KX	N; N; N; P-5, N; N	844	421	1	2	131072
conv5_1x1	M, R, C, N, KY, KX	N; N; N; P-2; N; N	133	261	1	2	2217924
conv6_3x3_dw	R, C, M, N, KY, KX	U-4; U-4; P-6; N; Op-*conv3_add4*; N	519	295	9	2	27751
conv6_1x1	M, R, C, N, KY, KX	N; N; U-4; N; N; N;	83	291	12	6	546133
conv7_1_3x3_dw	R, C, M, N, KY, KX	N; N; N; P-5; Op-*conv3_add4*; N	331	301	9	7	47268
conv7_1_1x1	M, R, C, N, KY, KX	N; N; N; P-2; N; N;	100	156	1	10	2119480
conv8_3x3_dw	R, C, M, N, KY, KX	N; N; P-6; N; Op-*conv3_sepfunc4*; N	469	172	9	7	10256
conv8_1x1	M, R, C, N, KY, KX	N; N; N; N; N; N	80	226	1	17	1122304
conv9_3x3_dw	R, C, M, N, KY, KX	N; N; N; N; Op-*conv3_sepop3*; N	472	220	9	12	43008
conv9_1x1	M, R, C, N, KY, KX	N; N; N; N; N; N	85	206	1	35	1654464
pool	M, R, C, N, KY, KX	N; U-1; N; N; N; U-4	94	80	0	3	4096
fc	M, N, KY, KX	N; N; N; N;	80	120	1	129	512000
Total	-	-	**7148**	**4926**	**103**	**336**	**23600396**

Table 6.1 – Configuration of each layer of MobileNet-V1 and resource and latency estimations.

be used to optimize the entire CNN.

Only one configuration is selected by the search algorithm for each layer. The selected configuration allows to fit the entire CNN accelerator on a single FPGA. Table 6.1 shows the configuration of each layer, generated by the GA, along with the estimated resource usage and latency. In most cases, the algorithm searches for the solution with the best area-latency trade-off. For example, for computationally intensive layers, the algorithm selects a solution with lower latency, without sacrificing the area. As for less computationally intensive layers, the algorithm selects a configuration point with low resource utilization without sacrificing the latency. For instance, *conv1* has a large number of MAC operations (see Figure 6.4), this means that the algorithm will seek a solution between these two latencies $5.0E+05$ and $1.0E+06$ (Figure 6.6(a)). This is compatible with the estimated latency, $5.3E+06$ cycles, in Table 6.1. For *conv1_3×3_dw*, the number of MACs is less than the half of *conv1*, thus the search algorithm picks a configuration with low resource usage having a latency value between $2.0E+05$ and $3.0E+05$ which is compatible with the latency $2.5E+05$ in Table 6.1.

As it can be seen in Table 6.1, the Optimizer module found no suitable solutions for some layers, especially the deeper ones where the depth of the feature map is greater than its height, such as layer *conv8_1×1*, *conv9_1×1* and *fc* where the optimiza-



(a) Conv1 (b) Conv1_3x3_dw

(c) Conv1_1x1 (d) Conv3_3x3_dw

(e) Conv3_1x1 (f) Conv7_3x3_dw

(g) Conv7_1x1 (h) Pool

Figure 6.6 – GA-based configurations of various MobileNet-V1 layers represented as area-latency points in a 2d space.

tions are set to N. The large depth led to higher BRAM usage, which means the Optimizer module was unable to find an appropriate solution without sacrificing the latency at the cost of the resource usage, and thus no optimizations were applied. It is worth noting that the optimizer module sought for solutions only for the first two *conv7* layers, i.e., *conv7_1_3×3_dw* and *conv7_3_1×1*, since the remaining *conv7* layers (*conv7×7_2_3×3_dw* to *conv7_5_1×1*) have exactly the same number of parameters and the same input and output pixels as the first ones. The Optimizer module relied on the connection graph of the MobileNet-V1 DNN (detailed in Figure 3.11 in chapter 2) to get the information on layers similarities. Hence, the Allocation *pragma* is set to use the same RTL resources for those layers, and thus reduce the overall resource utilization of the DNN accelerator.

Hardware Implementation Results

After identifying a suitable configuration for each layer by the Optimizer module, the Hardware Generation module includes, in the C-HLS source code, the right compiler directives and tool-specific commands (Vivado-HLS pragmas). Then, the configured C-HLS code is transferred to the Vivado-HLS 2020.1 tool, which generates the RTL representation of the entire CNN.

For comparison purposes, two hardware implementations of MobileNet-V1 were devised: an implementation with no specific optimizations and an optimized one using the configurations set by the Optimizer module. The Xilinx Zynq7000 xc7z030 FPGA was set as a target for both implementations. Figure 6.7 represents a per layer resource utilization, in percentage, of the non-optimized implementation. As for the optimized implementation, Figure 6.8 shows the resource utilization per layer (in percentage) after the generation of the RTL. As it can be seen, the classical implementation (non-optimized) requires less resources, especially in terms of DSP, LUT and FF. The number of BRAM shows a slight change from the classical implementation to the optimized one.

Regarding the latency, it can be seen from Figure 6.9 that the optimized layers, especially early layers, show a clear difference between the two implementations. Optimized layers, using operators and other combination of optimizations such as unrolling and pipelining, are faster than the non-optimized ones but use more resources.

For instance, *conv1* was optimized on multiple levels: the first one is the loop order, which is set in a way to encourage data locality and reduce unnecessary data transfer by moving the width and height loops (*l0* and *l1* in Listing 4.1) closer to the computing loops

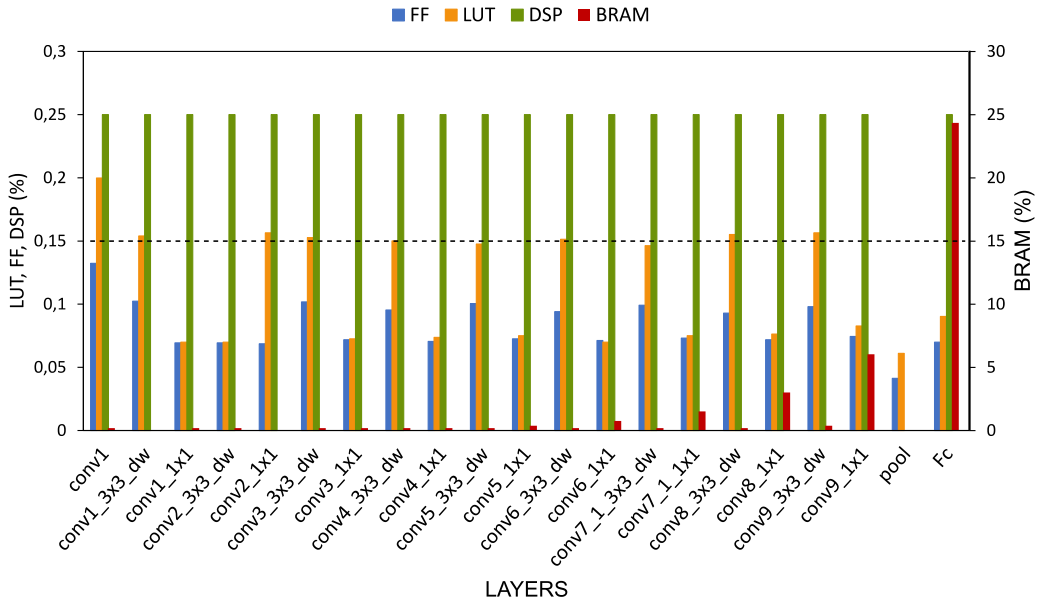


Figure 6.7 – Resource utilization, expressed in percentage, of the non-optimized implementation of MobileNet-V1 in terms of DSP, LUT, FF and BRAM.

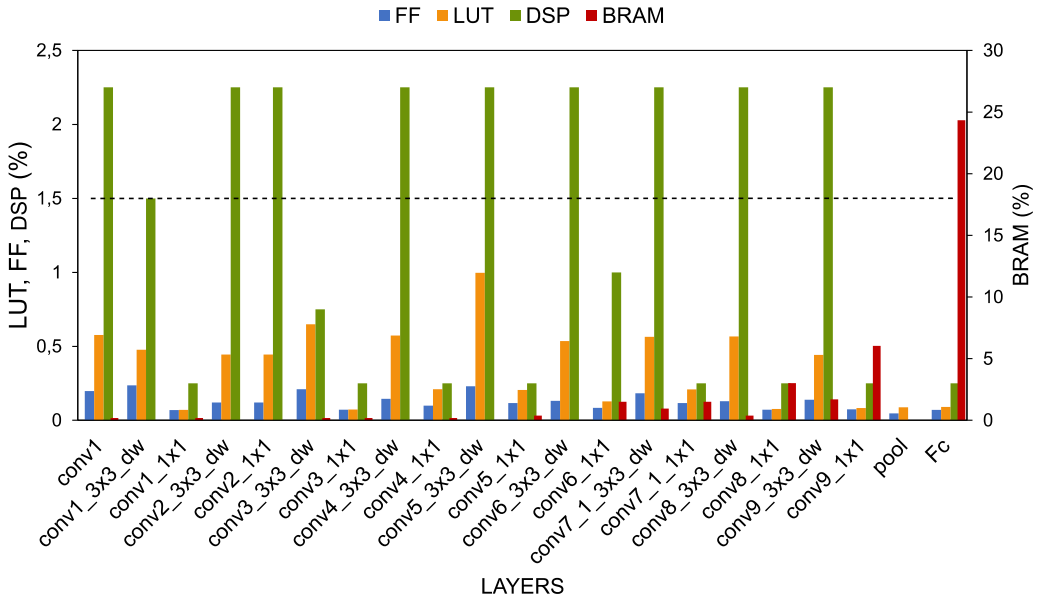


Figure 6.8 – Resource utilization, expressed in percentage, of an optimized implementation of MobileNet-V1 in terms of DSP, LUT, FF and BRAM.

(*l4* and *l5*). Moreover, an operator was used as a substitute of *l4* and *l5* to improve performance, where the kernel loop of the convolutional operation (*3×3* kernel) is separated into two functions. The first function consists of an unrolled multiplication loop, and the second one is an unrolled accumulation loop. This increased the resource utilization by 3× for LUT and 9× for DSP. However, the use of this operator is intended to accelerate the computation due to its ability of computing one *3×3* kernel in one clock cycle, at a frequency of *100*MHz (excluding data transfer), instead of executing one MAC operation per cycle. And finally, setting the pipeline directive on the channels loop *l3*, with an Initiation Interval of 5, enhanced the latency by implying a concurrent implementation of the operations in the loop. As a result, the whole latency is improved by *81.25*% compared to a classical implementation. Performance improvements of a few layers is presented in Figure 6.9.

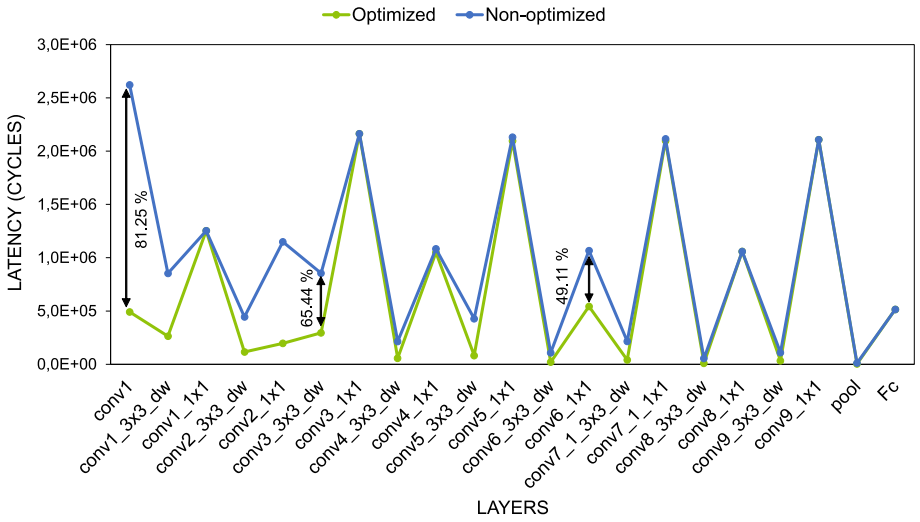


Figure 6.9 – Latency comparison, in cycles, between a non-optimized and an optimized implementation of MobileNet-V1.

Table 6.2 summarizes the computational and resource needs of a classical (non-optimized) implementation of MobileNet-V1, which only includes tool-specific optimizations that are usually applied automatically, as well as the optimized one. As it can be seen, the implementation of MobiletNet-V1 is optimized performance-wise. Most of the resources have slightly increased compared to a classical implementation. For instance, optimized MobileNet-V1 uses *2.75*× more LUTs, *1.46*× more FFs and *1.11*× more BRAMs. On the other hand, *5.25*× more DSPs are used, which is due to the employed optimizations like the unrolling and the chosen operator in certain layers. As for the latency, opti-

Resources (%)	Non-Optimized	Optimized	Gain (%)
LUT	2,8	7,8	-
FF	1,8	2,7	-
DSP	5,0	26,3	-
BRAM	87,4	78,5	-
Latency (cycles) $\times 1000$	29 972	23 227	22,5

Table 6.2 – Resource Usage (%) and Latency (cycles) of Optimized and Non-Optimized MobileNet-V1 implementations.

mized MobileNet-V1 is 22.5% faster compared to the non-optimized version, which can be explained by the applied optimizations that often include concurrent computation and parallel access to data, especially in the chosen operators. It is worth noting that only one synthesis was required in this case, since the estimated values of latency and resource usage are close to real values, with an error of 1.6% for latency and average 15.63% for all resources.

6.2.3 SHEFTENN Evaluation using SqueezeNet-V1.1

The same steps are applied to implement the SqueezeNet-V1.1 DNN with an input image of $3 \times 227 \times 227$. Regarding GA parameters, the mutation rate is set to 0.1 and the population size is set 300, same as MobileNet-V1. Besides, changing the FPGA target was mandatory, since the SqueezeNet-V1.1 is a larger network and the search algorithm *optimizeNetwork* was not able to find a viable solution to fit the whole network on the targeted FPGA. Hence, the Xilinx Virtex xc7vx980t is used instead. Currently, the proposed methodology generates a full accelerator for the whole CNN, which makes it difficult to use for large networks due to limited FPGA size. However, one of the perspectives is to automatically partition large DNNs as illustrated in Section "Discussion and Perspectives".

The resource usage and the latency of optimized and non-optimized versions of SqueezeNet-V1.1 are illustrated in Figure 6.10. From this figure, one can see that the optimized version of SqueezeNet-V1.1 has a lower latency but higher resource usage compared to the non-optimized one. Precisely, LUTs, FFs and DSPs have increased by 18.20%, 0.62% and 3.2% respectively compared to the non-optimized implementation. In contrast, BRAMs have slightly decreased by 0.05%. As for the latency, the optimized version of SqueezeNet-V1.1 is 39.24% faster than the classical implementation. One can see that the tool optimized the SqueezeNet-V1.1' implementation performance-wise without sacrificing re-

source usage, since most used resources do not exceed *50* % of the ones available on the targeted FPGA.

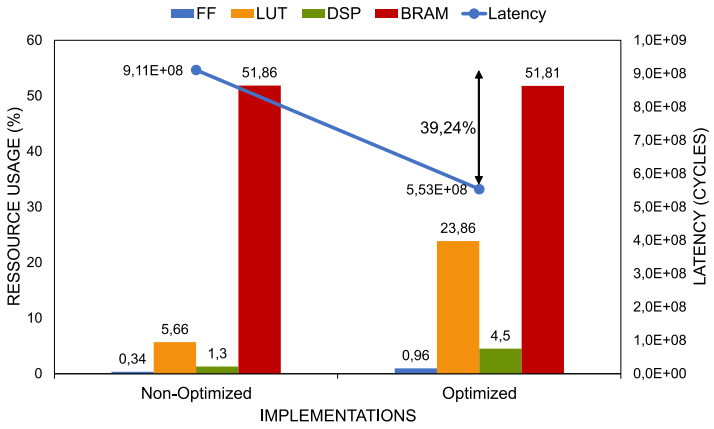


Figure 6.10 – Resource Usage (%) and Latency (cycles) of non-optimized and optimized SqueezeNet-V1.1 implementations.

To sum up, experimental results showed that the SHEFTENN framework was able to output viable solutions with an interesting area-performance trade-off, and to generate optimized implementations of accelerators for state-of-the-art neural networks, by using a custom-made GA encoding the loops order and their optimizations.

6.3 Conclusion

This chapter presented the implementation of the SHEFTENN methodology as an automated end-to-end hardware generation framework targeting embedded DNNs. The proposed design flow combines the advantages of the Characterization and the Optimizer modules to guide and optimize the Hardware Generation step. It has a new layer of optimization that diversifies the design space thanks to the library of HLS-based operators having distinct areas and latencies. The dataset is leveraged by the Optimizer module for optimization purposes. SHEFTENN employs a two-step optimization process that optimizes each layer separately based on area and latency models, and therefore lowers design time. It then uses a search algorithm to optimize the entire CNN hardware implementation. The results confirmed the efficiency of the proposed approach, which allowed to design accelerator implementations of MobileNet-V1 and SqueezeNet-V1.1. These hard-

ware designs are respectively 22.5% and 39.24% faster compared to HLS implementation. Thus, the proposed approach allows devising CNN accelerators to be embedded into next-generation embedded systems.

CONCLUSION

Summary

This dissertation first presented the state of the art of DNNs and showed the difference between training and inference. The light was put, in particular, on CNN algorithms that are efficient and suited for acceleration due to their intrinsic parallel structures, but are computationally intensive and have large memory needs. From here came the hardware acceleration problem of the CNNs' inference phase, especially in embedded systems, which was faced with various solutions. These solutions primary relied on ASIC hardware accelerators which offer dedicated and highly optimized solutions. Due to the lack of flexibility in dedicated ASIC architectures, the reconfiguration aspect of FPGAs is leveraged. FPGAs offer the possibility to tailor accelerators in terms of bitwidth and memory to the exact needs of the algorithms. Unfortunately, designing ASIC or FPGA-based DNN accelerators for inference requires time and expertise if done manually or even using high-level design methods, such as high-level synthesis (HLS). Therefore, another set of solutions, proposing hardware generation frameworks which exploit high-level design techniques (e.g. HLS), was presented that aims at reducing the gap between the application and the hardware architecture and thus reducing the complexity and the time of the design process.

After that, existing hardware generation approaches were compared with respect to different criteria. These criteria allowed to qualitatively evaluate each proposed approach in terms of the used level of abstraction, the methods used for the design space exploration, the hardware generation method and the overall optimization process of the hardware architecture to be generated. This comparison showed that the HLS is one of the most widely used for its ease of use and its flexibility. In addition, it showed that HLS is not fully leveraged at the algorithmic level and most approaches are based on existing templates or architectures. Regarding the optimization process, it widely relies on exhaustive search and it is also specific to the target architecture. Furthermore, the produced RTL, in most cases, cannot be optimized once synthesized. Based on this comparison, possible improvements to the automated design process were identified.

Hence, the presented approach in this thesis is an end-to-end automated methodology to design hardware accelerators for embedded DNNs. This methodology helps reducing the gap between abstract descriptions and hardware architectures, since hardware designers find it hard to master the wide and continuously evolving landscape of DNNs. Therefore, a first step consists in automatically and thoroughly characterizing the DNN to extract relevant metrics that ease the design process. Moreover, this methodology suggests exploiting high-level design techniques to reduce the design time and increase productivity, while enabling enough degrees of liberty for design space exploration. Unfortunately, the vast design space makes the exploration of all potential solutions very time-consuming. For this reason, the proposed methodology automates this exploration to ease the overall design process. This methodology was implemented as a framework, called SHEFTENN. The tool targets FPGAs and has no dependency on existing architectures or templates. The proposed framework consists in three interdependent modules working together to ensure an optimized RTL implementation. Each module has a specific role in the design process. The first module, Characterization, reduces the design space through metrics computation and analysis and generates augmented specifications which provides a first set of pre-optimizations. The second module, Hardware Generation, exploits the produced metrics and high-level languages, by using HLS, to generate an semi-optimized C-HLS source code. Finally, the Optimizer module, automatically explores the design space through a hybrid optimization process of the C-HLS code to optimize the resulting RTL. It is a two-step optimization process, which uses two separate algorithms, respectively targeting each layer and the whole DNN. The optimizer module proposes a trade-off between speed and quality of exploration using layer-based latency and area models as well as real syntheses.

The framework was evaluated on two state-of-the-art DNNs. The evaluation showed that the proposed approach is reliable and allows to devise hardware accelerators for DNNs with no user intervention in the design process. Framework-based implementations showed better performance and a better area-performance trade-off compared to classical implementations. This work should be considered as a first step of the implementation of the proposed methodology, and many perspectives were identified to improve it. The following section discusses the results and presents these perspectives.

Discussion and Perspectives

HLS offered a fast way to design AI chips, without a need for an advanced hardware expertise, while exploiting FPGAs. At the same time, many new challenges and interesting perspectives have emerged. One of the main challenges is the DSE due the variety of the parameters to modify. These parameters stand in the form of algorithmic coding style and high-level optimizations, which require many user interventions to achieve the desired design goal which lengthen design time. Therefore, HLS tools must evolve to automate this exploration, especially for AI accelerators, to reduce time-to-market of new accelerators. This automation must consider the rapidly changing landscape of DNNs as well as the embedded systems constraints.

This thesis presented a high-level design approach that exploits HLS. The high-level design process is tackled on three different levels. In the current implementation of the proposed framework, the design space is firstly reduced considering several optimization options deduced from the characterization phase, and then increased due to the combination of pragmas and HLS operators. The combination of those three levels seems essential to ensure an efficient and automated design flow and to reduce design time.

Although this work exploited the advantages of HLS and FPGAs offering respectively the speed of hardware design and the reconfiguration aspect, various aspects should be further enhanced to make the framework more generic and efficient. Some of these exciting perspectives are discussed below:

— In order to cope with algorithmic evolutions, the framework should be able to take into consideration newly-available DNN optimization techniques, such as pruning and quantization-aware training employed in hls4ml [43]. Considering such techniques in hardware implementation improves performance and reduce resource usage and energy consumption.

— Power consumption must be taken into account in the optimization process in order to optimize all aspects of the design. Therefore, a power model should be created and integrated in the Optimizer module. An example of modeling power is by following the same approach presented in section 5.3 which consists in creating a dataset of microbenchmarks of different layers. These benchmarks will be written in C, synthesized and simulated then fed to an RTL power estimator tool. The gathered data will be fed to the NCSS software to devise the appropriate model. Other solutions also exists and can be employed to estimate power consumption

as presented in [92].

— Adopted models in the optimizer module are related to layer configurations. Therefore, those models should be adapted for each newly available configuration to get an acceptable estimation with a small MSE. In addition, the collected data after simulating microbenchmarks depend on the employed FPGAs, since only two FPGAs were used to synthesize all the layers.

— Explore more parameters of the framework, especially the exploration of the trade-off between speed of finding a solution and precision of the employed estimation models.

— Future works will also pursue DNN partitioning to enable implementing large DNNs on smaller FPGAs. The partitioning algorithm will rely on the algorithmic representation of the network architecture (C-HLS source code) to find the right partitioning, i.e., a partitioning that reduces resource usage without sacrificing performance. This algorithm will be part of the search algorithm, its main role will be to find layers with similar memory requirements and computing needs to allocate them into the same RTL instance. Partitioning can also be on the layer level, i.e., intra-layer partitioning, to increase parallelism at the layer level. Furthermore, fpgaConvNet-like approaches that uses SDF can also be employed to find a suitable partitioning for the DNN.

— Thorough study of the GA results by studying its optimality results.

— Another interesting perspective is to seek other optimization algorithms capable of rapidly exploring the design space and providing accurate solutions, such as deep learning algorithms.

— Explore the effects of new pragmas to further extend the design space. Additionally, targeting sparse DNNs, from a hardware perspective, by introducing HLS-based compression techniques will enrich the proposed framework.

— Different possible options can be used to abstract the area in terms of FPGA resources, such as weighted solutions, which can be considered to favor designs requiring less LUTs or less BRAMs depending on the overall system to be implemented, or as added values of the resources on a form of linear combination of their usage.

— Compare SHEFTENN quantitatively with existing state-of-the-art frameworks by implementing the same networks used in the literature or by experimenting with existing open-source frameworks.

Contributions

All contributions during my doctoral studies are listed below. Publications are listed in reverse chronological order and classified among journals, conferences and patents:

Submitted in journals:

1. Ali, N.; Philippe, J. M.; Tain, B. & Coussy, P., "SHEFTENN : Software and Hardware Exploration Framework for Efficient Implementation on Embedded Systems". ACM Transactions on Design Automation of Embedded Systems (TODAES), 2022
2. Ali, N.; Philippe, J. M.; Tain, B. & Coussy, P., "Exploration and Generation of Efficient FPGA-based Deep Neural Network Accelerators". Journal of Signal Processing Systems (JSPS), 2022

To be Submitted in conferences:

1. Ali, N.; Philippe, J. M.; Tain, B. & Coussy, P. "An Integrated Design Space Exploration and Hardware Generation Tool Flow for Next-Generation Deep Neural Networks Accelerators", 2022.

Published:

1. Ali, N.; Philippe, J.-M.; Tain, B. & Coussy, P., "Exploration and Generation of Efficient FPGA-based Deep Neural Network Accelerators", 2021 IEEE Workshop on Signal Processing Systems (SiPS), 2021
2. Ali, N.; Philippe, J. M.; Tain, B.; Peyret, T. & Coussy, P.,"Deep Neural Networks Characterization Framework for Efficient Implementation on Embedded Systems", 2020 IEEE Workshop on Signal Processing Systems (SiPS), 2020, 1-6

Submitted patents:

1. High-level design flow for artificial intelligence accelerator (2021)

BIBLIOGRAPHY

[1] Martin Abadi et al., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015, URL: https://www.tensorflow.org/.

[2] I. Ahmad, M.K. Dhodhi, and F.H. Hielscher, « Design-Space Exploration for High-Level Synthesis », *in*: *Proceeding of 13th IEEE Annual International Phoenix Conference on Computers and Communications*, 1994, pp. 491–, DOI: 10.1109/PCCC.1994.504159.

[3] A. Aimar et al., « NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps », *in*: *IEEE Transactions on Neural Networks and Learning Systems* 30.*3* (Mar. 2019), pp. 644–656, ISSN: 2162-2388, DOI: 10.1109/TNNLS.2018.2852335.

[4] Giovanni Alcantara, « Empirical analysis of non-linear activation functions for Deep Neural Networks in classification tasks », *in*: *CoRR* abs/1710.11272 (2017), arXiv: 1710.11272, URL: http://arxiv.org/abs/1710.11272.

[5] N. Ali et al., « Deep Neural Networks Characterization Framework for Efficient Implementation on Embedded Systems », *in*: *2020 IEEE Workshop on Signal Processing Systems (SiPS)*, 2020, pp. 1–6, DOI: 10.1109/SiPS50750.2020.9195227.

[6] Nermine Ali et al., « Exploration and Generation of Efficient FPGA-based Deep Neural Network Accelerators », *in*: *2021 IEEE Workshop on Signal Processing Systems (SiPS)*, 2021, pp. 123–128, DOI: 10.1109/SiPS52927.2021.00030.

[7] Jonathan Bachrach et al., « Chisel: Constructing hardware in a Scala embedded language », *in*: *DAC Design Automation Conference 2012*, 2012, pp. 1212–1221, DOI: 10.1145/2228360.2228584.

[8] Soheil Bahrampour et al., « Comparative Study of Caffe, Neon, Theano, and Torch for Deep Learning », *in*: *CoRR* abs/1511.06435 (2015), arXiv: 1511.06435, URL: http://arxiv.org/abs/1511.06435.

[9] Junjie Bai, Fang Lu, Ke Zhang, et al., *ONNX: Open Neural Network Exchange*, 2019.

[10] D. Baptista, F. Morgado-Dias, and L. Sousa, « A Platform based on HLS to Implement a Generic CNN on an FPGA », *in*: *2019 International Conference in Engineering Applications (ICEA)*, 2019, pp. 1–7, DOI: 10.1109/CEAP.2019.8883473.

[11] Andrew Canis et al., « LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems », *in*: *ACM Trans. Embed. Comput. Syst.* 13.*2* (Sept. 2013), ISSN: 1539-9087, DOI: 10.1145/2514740, URL: https://doi.org/10.1145/2514740.

[12] Maurizio Capra et al., « An Updated Survey of Efficient Hardware Architectures for Accelerating Deep Convolutional Neural Networks », *in*: *Future Internet* 12.*7* (2020), ISSN: 1999-5903, DOI: 10.3390/fi12070113, URL: https://www.mdpi.com/1999-5903/12/7/113.

[13] A. Carbon et al., « PNeuro: A scalable energy-efficient programmable hardware accelerator for neural networks », *in*: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, Mar. 2018, pp. 1039–1044, DOI: 10.23919/DATE.2018.8342165.

[14] Benjamin Carrion Schafer, « Probabilistic Multiknob High-Level Synthesis Design Space Exploration Acceleration », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.*3* (2016), pp. 394–406, DOI: 10.1109/TCAD.2015.2472007.

[15] Catapult, 2021.

[16] Srimat Chakradhar et al., « A Dynamically Configurable Coprocessor for Convolutional Neural Networks », *in*: *SIGARCH Comput. Archit. News* 38.*3* (June 2010), pp. 247–257, ISSN: 0163-5964, DOI: 10.1145/1816038.1815993, URL: https://doi.org/10.1145/1816038.1815993.

[17] Chixiao Chen et al., « iFPNA: A Flexible and Efficient Deep Learning Processor in 28nm CMOS Using a Domain-Specific Instruction Set and Reconfigurable Fabric », *in*: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* PP (May 2019), pp. 1–1, DOI: 10.1109/JETCAS.2019.2914355.

[18] Yu-Hsin Chen, Joel Emer, and Vivienne Sze, « Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks », *in*: 2018.

[19] Yu-Hsin Chen, Joel Emer, and Vivienne Sze, « Eyeriss: A and Spatial Architecture and for Energy-Efficient and Dataflow and for Convolutional and Neural Networks », *in*: IEEE, 2016, pp. 367–379, DOI: 10.1109/ISCA.2016.40.

[20] Yu-Hsin Chen, Joel Emer, and Vivienne Sze, « Using Dataflow to Optimize Energy Efficiency of Deep Neural Network Accelerators », *in*: *IEEE Micro* 37.*3* (2017), pp. 12–21, DOI: 10.1109/MM.2017.54.

[21] Yu-Hsin Chen et al., « Eyeriss: An and Energy-Efficient Reconfigurable and Accelerator for Deep and Convolutional », *in*: *IEEE JOURNAL OF SOLID-STATE CIRCUITS* VOL. 52.*NO. 1* (Jan. 2017).

[22] Tianshi Chen, Zidong Du, and Ninghui Sun, « DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning », *in*: 2014, DOI: .org/10.1145/http://dx.doi.org/10.1145/2541940.2541967.

[23] Y. Chen et al., « Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices », *in*: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.*2* (2019), pp. 292–308, DOI: 10.1109/JETCAS.2019.2910232.

[24] Yuze Chi et al., « SODA: Stencil with Optimized Dataflow Architecture », *in*: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8, DOI: 10.1145/3240765.3240850.

[25] Young-kyu Choi and Jason Cong, « HLS-Based Optimization and Design Space Exploration for Applications with Variable Loop Bounds », *in*: *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, San Diego, California: Association for Computing Machinery, 2018, ISBN: 9781450359504, DOI: 10.1145/3240765.3240815, URL: https://doi.org/10.1145/3240765.3240815.

[26] Young-Kyu Choi et al., « FLASH: Fast, Parallel, and Accurate Simulator for HLS », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.*12* (2020), pp. 4828–4841, DOI: 10.1109/TCAD.2020.2970597.

[27] John Clow et al., « A pythonic approach for rapid hardware prototyping and instrumentation », *in*: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–7, DOI: 10.23919/FPL.2017.8056860.

[28] Jason Cong and Jie Wang, « PolySA: Polyhedral-Based Systolic Array Auto-Compilation », *in*: *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, San Diego, California: Association for Computing Machinery, 2018, ISBN: 9781450359504, DOI: 10.1145/3240765.3240838, URL: https://doi.org/10.1145/3240765.3240838.

[29] Jason Cong et al., « A Study on the Impact of Compiler Optimizations on High-Level Synthesis », *in*: *Languages and Compilers for Parallel Computing*, ed. by Hironori Kasahara and Keiji Kimura, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 143–157, ISBN: 978-3-642-37658-0.

[30] Jason Cong et al., « Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture », *in*: *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, San Francisco, California: Association for Computing Machinery, 2018, ISBN: 9781450357005, DOI: 10.1145/3195970.3195999, URL: https://doi.org/10.1145/3195970.3195999.

[31] Jason Cong et al., « High-Level Synthesis for FPGAs: From Prototyping to Deployment », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.*4* (2011), pp. 473–491, DOI: 10.1109/TCAD.2011.2110592.

[32] Ricardo C. Corrêa, Afonso Ferreira, and Pascal Rebreyend, « Scheduling Multiprocessor Tasks with Genetic Algorithms », *in*: *IEEE Trans. Parallel Distrib. Syst.* 10.*8* (Aug. 1999), pp. 825–837, ISSN: 1045-9219, DOI: 10.1109/71.790600, URL: https://doi.org/10.1109/71.790600.

[33] Philippe Coussy and Adam Morawiec, *High-Level Synthesis: From Algorithm to Digital Circuit*, 1st, Springer Publishing Company, Incorporated, 2008, ISBN: 1402085877.

[34] Philippe Coussy et al., « An Introduction to High-Level Synthesis », *in*: *IEEE Design Test of Computers* 26.*4* (2009), pp. 8–17, DOI: 10.1109/MDT.2009.69.

[35] Philippe Coussy et al., « GAUT: A High-Level Synthesis Tool for DSP applications », *in*: (June 2008).

[36] Hubel D.H. and Wiesel T.N., « Receptive fields and functional architecture of monkey striate cortex », *in*: *The Journal of Physiology* 195.*1* (1968), pp. 215–243, DOI: 10.1113/jphysiol.1968.sp008455.

[37] Steve Dai et al., « Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning », *in*: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 129–132, DOI: 10.1109/FCCM.2018.00029.

[38] G. Desoli et al., « 14.1 A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems », *in*: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, Feb. 2017, pp. 238–239, DOI: 10.1109/ISSCC.2017.7870349.

[39] Bin Ding, Huimin Qian, and Jun Zhou, « Activation functions and their characteristics in deep neural networks », *in*: *2018 Chinese Control And Decision Conference (CCDC)*, 2018, pp. 1836–1841, DOI: 10.1109/CCDC.2018.8407425.

[40] Janardhan Rao Doppa, Justinian Rosca, and Paul Bogdan, « Autonomous Design Space Exploration of Computing Systems for Sustainability: Opportunities and Challenges », *in*: *IEEE Design Test* 36.*5* (2019), pp. 35–43, DOI: 10.1109/MDAT.2019.2932894.

[41] Zidong Du et al., « ShiDianNao: Shifting Vision Processing Closer to the Sensor », *in*: 2015, DOI: .org/10.1145/2749469.2750389.

[42] ACE-Associated Compiler Experts, *CoSy*, 2016.

[43] Farah Fahim et al., « hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices », *in*: *CoRR* abs/2103.05579 (2021), arXiv: 2103.05579, URL: https://arxiv.org/abs/2103.05579.

[44] Lorenzo Ferretti, « Design space exploration in high-level synthesis », PhD thesis, Università della Svizzera italiana, 2020.

[45] *FROM RESEARCH TO PRODUCTION*, 2021, URL: https://pytorch.org/.

[46] Kunihiko Fukushima, « Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position », *in*: *Biological Cybernetics*, vol. 36, 4, 1980, pp. 193–202, DOI: 10.1007/BF00344251.

[47] Daniel D. Gajski et al., *SPECC: Specification Language and Methodology*, 2012.

[48] Vinayak Gokhale et al., « A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks », *in*: *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 696–701, DOI: 10.1109/CVPRW.2014.106.

[49] Ashish Gondimalla et al., « SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks », *in*: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 151–165, ISBN: 9781450369381, DOI: 10.1145/3352460.3358291, URL: https://doi.org/10.1145/3352460.3358291.

[50] Y. Guan et al., « FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates », *in*: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 152–159, DOI: 10.1109/FCCM.2017.25.

[51] Suyog Gupta et al., « Deep Learning with Limited Numerical Precision », *in*: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, Lille, France: JMLR.org, 2015, pp. 1737–1746.

[52] Song Han et al., « EIE: Efficient Inference Engine on Compressed Deep Neural Network », *in*: *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, Seoul, Republic of Korea: IEEE Press, 2016, pp. 243–254, ISBN: 9781467389471, DOI: 10.1109/ISCA.2016.30, URL: https://doi.org/10.1109/ISCA.2016.30.

[53] Kaiming He et al., « Deep Residual Learning for Image Recognition », *in*: (2015).

[54] Kartik Hegde et al., « UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition », *in*: *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, Los Angeles, California: IEEE Press, 2018, pp. 674–687, ISBN: 9781538659847, DOI: 10.1109/ISCA.2018.00062, URL: https://doi.org/10.1109/ISCA.2018.00062.

[55] John H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*, Cambridge, MA, USA: MIT Press, 1992, ISBN: 0262082136.

[56] M. Holzer, B. Knerr, and M. Rupp, « Design Space Exploration with Evolutionary Multi-Objective Optimisation », *in*: *2007 International Symposium on Industrial Embedded Systems*, 2007, pp. 126–133, DOI: 10.1109/SIES.2007.4297326.

[57] Andrew G. Howard et al., « MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications », *in*: *CoRR* abs/1704.04861 (2017), arXiv: 1704.04861, URL: http://arxiv.org/abs/1704.04861.

[58] Forrest N. Iandola et al., « SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size », *in*: *CoRR* abs/1602.07360 (2016), arXiv: 1602.07360, URL: http://arxiv.org/abs/1602.07360.

[59] « IEEE Standard for Standard SystemC Language Reference Manual - Redline », *in*: *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) - Redline* (2012), pp. 1–1163.

[60] Bluespec Inc., 2021.

[61] Intel, 2021.

[62] Intel, *Intel Distribution of OpenVINO toolkit*, 2021, URL: https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html.

[63] Yangqing Jia et al., « Caffe: Convolutional Architecture for Fast Feature Embedding », *in*: *arXiv preprint arXiv:1408.5093* (2014).

[64] Norman P. Jouppi et al., « In-Datacenter Performance Analysis of a Tensor Processing Unit », *in*: *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 1–12, ISBN: 9781450348928, DOI: 10.1145/3079856.3080246, URL: https://doi.org/10.1145/3079856.3080246.

[65] Sheng-Chun Kao and Tushar Krishna, « GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm », *in*: *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, Virtual Event, USA: Association for Computing Machinery, 2020, ISBN: 9781450380263, DOI: 10.1145/3400302.3415639, URL: https://doi.org/10.1145/3400302.3415639.

[66] *Keras: Deep Learning for humans*, 2020.

[67] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, « ImageNet Classification with Deep Convolutional Neural Networks », *in*: 2012.

[68] David C. Ku and Giovanni De Micheli, *Hardware C - A Language for Hardware Design*, tech. rep., STANFORD UNIV CA COMPUTER SYSTEMS LAB, Aug. 1, 1988.

[69] Kiseok Kwon et al., « Co-Design of Deep Neural Nets and Neural Net Accelerators for Embedded Vision Applications », *in*: *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, San Francisco, California: Association for Computing Machinery, 2018, ISBN: 9781450357005, DOI: 10.1145/3195970.3199849, URL: https://doi.org/10.1145/3195970.3199849.

[70] Sakari Lahti et al., « Are We There Yet? A Study on the State of High-Level Synthesis », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.*5* (2019), pp. 898–911, DOI: 10.1109/TCAD.2018.2834439.

[71] C. Lattner and V. Adve, « LLVM: a compilation framework for lifelong program analysis amp; transformation », *in*: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86, DOI: 10.1109/CGO.2004.1281665.

[72] *Gradient-Based Learning Applied to Document Recognition*, IEEE, 1998.

[73] Chao Li et al., « High-level synthesis for FPGAs: code optimization strategies for real-time image processing », *in*: *Journal of Real-Time Image Processing* 14 (2018), DOI: 10.1007/s11554-017-0722-3, URL: https://doi.org/10.1007/s11554-017-0722-3.

[74] Huimin Li et al., « A High Performance FPGA-based Accelerator for Large-Scale Convolutional Neural Networks », *in*: 2016.

[75] Jiajun Li et al., « SqueezeFlow: A Sparse CNN Accelerator Exploiting Concise Convolution Rules », *in*: *IEEE Transactions on Computers* 68.*11* (2019), pp. 1663–1677, DOI: 10.1109/TC.2019.2924215.

[76] Gai Liu et al., « Architecture and Synthesis for Area-Efficient Pipelining of Irregular Loop Nests », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.*11* (2017), pp. 1817–1830, DOI: 10.1109/TCAD.2017.2664067.

[77] Hung-Yi Liu and Luca P. Carloni, « On learning-based methods for design-space exploration with High-Level Synthesis », *in*: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–7.

[78] Hung-Yi Liu, Michele Petracca, and Luca P. Carloni, « Compositional system-level design exploration with planning of high-level synthesis », *in*: *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 641–646, DOI: 10.1109/DATE.2012.6176550.

[79] Shuangnan Liu, Francis CM Lau, and Benjamin Carrion Schafer, « Accelerating FPGA Prototyping through Predictive Model-Based HLS Design Space Exploration », *in*: *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[80] Derek Lockhart, Gary Zibrat, and Christopher Batten, « PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research », *in*: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 280–292, DOI: 10.1109/MICRO.2014.50.

[81] Liqiang Lu et al., « An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs », *in*: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 17–25, DOI: 10.1109/FCCM.2019.00013.

[82] Yufei Ma et al., « Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks », *in*: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, Monterey, California, USA: Association for Computing Machinery, 2017, pp. 45–54, ISBN: 9781450343541, DOI: 10.1145/3020078.3021736, URL: https://doi.org/10.1145/3020078.3021736.

[83] Giovanni Mariani et al., « OSCAR: An Optimization Methodology Exploiting Spatial Correlation in Multicore Design Spaces », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.5 (2012), pp. 740–753, DOI: 10.1109/TCAD.2011.2177457.

[84] Grant Martin and Gary Smith, « High-Level Synthesis: Past, Present, and Future », *in*: *IEEE Design Test of Computers* 26.4 (2009), pp. 18–25, DOI: 10.1109/MDT.2009.83.

[85] M. Motamedi et al., « Design space exploration of FPGA-based Deep Convolutional Neural Networks », *in*: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2016, pp. 575–580, DOI: 10.1109/ASPDAC.2016.7428073.

[86] *MyHDL: From Python to Silicon*, Manual available on Github, 2018, URL: https://github.com/myhdl/myhdl.

[87] *N2D2 - Neural Network Design & Deployment*, Manual available on Github, 2019, URL: https://github.com/CEA-LIST/N2D2/.

[88] *N2D2 - Neural Network Design & Deployment*, 2019, URL: https://n2d2.readthedocs.io/en/latest/quant/post.html#post-training-quantization.

[89] Vinod Nair and Geoffrey E. Hinton, « Rectified Linear Units Improve Restricted Boltzmann Machines », *in*: *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, Haifa, Israel: Omnipress, 2010, pp. 807–814, ISBN: 9781605589077.

[90] Razvan Nane et al., « A Survey and Evaluation of FPGA High-Level Synthesis Tools », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.*10* (2016), pp. 1591–1604, DOI: 10.1109/TCAD.2015.2513673.

[91] Razvan Nane et al., « DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler », *in*: *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 619–622, DOI: 10.1109/FPL.2012.6339221.

[92] Yehya Nasser et al., « RTL to Transistor Level Power Modeling and Estimation Techniques for FPGA and ASIC: A Survey », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.*3* (2021), pp. 479–493, DOI: 10.1109/TCAD.2020.3003276.

[93] NCSS, *NCSS 2021 Statistical Software*, 2021.

[94] *NVIDIA cuDNN*, 2021.

[95] Berkin Ozisikyilmaz, Gokhan Memik, and Alok Choudhary, « Efficient system design space exploration using machine learning techniques », *in*: *2008 45th ACM/IEEE Design Automation Conference*, 2008, pp. 966–969, DOI: 10.1145/1391469.1391712.

[96] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria, « ReSPIR: A Response Surface-Based Pareto Iterative Refinement for Application-Specific Design Space Exploration », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.*12* (2009), pp. 1816–1829, DOI: 10.1109/TCAD.2009.2028681.

[97] M. Palesi and T. Givargis, « Multi-objective design space exploration using genetic algorithms », *in*: *Proceedings of the Tenth International Symposium on Hardware/-Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, 2002, pp. 67–72, DOI: 10.1145/774789.774804.

[98] Angshuman Parashar et al., « SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks », *in*: *CoRR* abs/1708.04485 (2017), arXiv: 1708.04485, URL: http://arxiv.org/abs/1708.04485.

[99] Maurice Peemen et al., « Memory-centric accelerator design for Convolutional Neural Networks », *in*: *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 13–19, DOI: 10.1109/ICCD.2013.6657019.

[100] Phi-Hung Pham et al., « NeuFlow: Dataflow Vision Processing System-on-a-Chip », *in*: IEEE, 2012.

[101] Luca Piccolboni et al., « COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators », *in*: *CoRR* abs/1912.10823 (2019), arXiv: 1912.10823, URL: http://arxiv.org/abs/1912.10823.

[102] Christian Pilato and Fabrizio Ferrandi, « Bambu: A modular framework for the high level synthesis of memory-intensive applications », *in*: Sept. 2013, pp. 1–4, DOI: 10.1109/FPL.2013.6645550.

[103] Christian Pilato, Fabrizio Ferrandi, and Donatella Sciuto, « A design methodology to implement memory accesses in High-Level Synthesis », *in*: *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2011, pp. 49–58, DOI: 10.1145/2039370.2039381.

[104] Atul Rahman et al., « Design Space Exploration of FPGA Accelerators for Convolutional Neural Networks », *in*: IEEE, 2017.

[105] D.S. Harish Ram, M.C. Bhuvaneswari, and S.M. Logesh, « A Novel Evolutionary Technique for Multi-objective Power, Area and Delay Optimization in High Level Synthesis of Datapaths », *in*: *2011 IEEE Computer Society Annual Symposium on VLSI*, 2011, pp. 290–295, DOI: 10.1109/ISVLSI.2011.55.

[106] Darian Reyes Fernandez de Bulnes, Yazmin Maldonado, and Leonardo Trujillo, « Development of Multiobjective High-Level Synthesis for FPGAs », *in*: *Scientific Programming* 2020.*10* (2020), pp. 2628–2639, DOI: 10.1155/2020/7095048.

[107] Miguel Rivera-Acosta, Susana Ortega-Cisneros, and Jorge Rivera, « Automatic Tool for Fast Generation of Custom Convolutional Neural Networks Accelerators for FPGA », *in*: *Electronics* 8.*6* (2019), ISSN: 2079-9292, DOI: 10.3390/electronics8060641, URL: https://www.mdpi.com/2079-9292/8/6/641.

[108] F. Rosenblatt, « The perceptron: A probabilistic model for information storage and organization in the brain. », *in*: *Psychological Review* 65.*6* (1958), pp. 386–408, DOI: 10.1037/h0042519.

[109] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, « Learning Internal Representations by Error Propagation », *in*: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*, Cambridge, MA, USA: MIT Press, 1986, pp. 318–362, ISBN: 026268053X.

[110] Olga Russakovsky et al., « ImageNet Large Scale Visual Recognition Challenge », *in*: *International Journal of Computer Vision (IJCV)* 115.*3* (2015), pp. 211–252, DOI: 10.1007/s11263-015-0816-y.

[111] Benjamin Carrion Schafer, Takashi Takenaka, and Kazutoshi Wakabayashi, « Adaptive Simulated Annealer for high level synthesis design space exploration », *in*: *2009 International Symposium on VLSI Design, Automation and Test*, 2009, pp. 106–109, DOI: 10.1109/VDAT.2009.5158106.

[112] Benjamin Carrion Schafer and Kazutoshi Wakabayashi, « Divide and Conquer High-Level Synthesis Design Space Exploration », *in*: *ACM Trans. Des. Autom. Electron. Syst.* 17.*3* (July 2012), ISSN: 1084-4309, DOI: 10.1145/2209291.2209302, URL: https://doi.org/10.1145/2209291.2209302.

[113] Benjamin Carrion Schafer and Zi Wang, « High-Level Synthesis Design Space Exploration: Past, Present, and Future », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.*10* (2020), pp. 2628–2639, DOI: 10.1109/TCAD.2019.2943570.

[114] Dominik Scherer, Andreas Müller, and Sven Behnke, « Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition », *in*: *Artificial Neural Networks – ICANN 2010*, ed. by Konstantinos Diamantaras, Wlodek Duch, and Lazaros S. Iliadis, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 92–101, ISBN: 978-3-642-15825-4.

[115] Yakun Shao et al., « Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures », *in*: June 2014, pp. 97–108, ISBN: 978-1-4799-4394-4, DOI: 10.1109/ISCA.2014.6853196.

[116] R. Shathanaa and N. Ramasubramanian, « Design Space Exploration for Architectural Synthesis—A Survey », *in*: *Recent Findings in Intelligent Computing Techniques*, ed. by Pankaj Kumar Sa et al., Singapore: Springer Singapore, 2018, pp. 519–527, ISBN: 978-981-10-8636-6.

[117] Yongming Shen, Michael Ferdman, and Peter Milder, « Maximizing CNN Accelerator Efficiency Through Resource Partitioning », *in*: *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 535–547, ISBN: 9781450348
928, DOI: 10.1145/3079856.3080221.

[118] Cristina Silvano et al., « MULTICUBE: Multi-objective Design Space Exploration of Multi-core Architectures », *in*: *2010 IEEE Computer Society Annual Symposium on VLSI*, 2010, pp. 488–493, DOI: 10.1109/ISVLSI.2010.67.

[119] Karen Simonyan and Andrew Zisserman, « VERY DEEP and CONVOLUTIONAL NETWORKS and FOR LARGE-SCALE and IMAGE RECOGNITION », *in*: 2015.

[120] Andrea Solazzo et al., « Hardware Design and Automation of Convolutional », *in*: *2016 IEEE Computer Society Annual Symposium on VLSI*, IEEE, 2016, DOI: 10.1109/ISVLSI.2016.101.

[121] *SpinalHDL*, Manual available on Github, 2021, URL: https://github.com/SpinalHDL/SpinalHDL.

[122] C. Szegedy et al., « Going deeper with convolutions », *in*: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9, DOI: 10.1109/CVPR.2015.7298594.

[123] Mingxing Tan and Quoc V. Le, « EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks », *in*: *CoRR* abs/1905.11946 (2019), arXiv: 1905.11946, URL: http://arxiv.org/abs/1905.11946.

[124] Stephen M. Steve Trimberger, « Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore's Law Has Driven the Design of FPGAs Through Three Epochs: the Age of Invention, the Age of Expansion, and the Age of Accumulation », *in*: *IEEE Solid-State Circuits Magazine* 10.*2* (2018), pp. 16–29, DOI: 10.1109/MSSC.2018.2822862.

[125] Yaman Umuroglu et al., « FINN: A Framework for Fast, Scalable Binarized Neural Network Inference », *in*: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, Monterey, California, USA: Association for Computing Machinery, 2017, pp. 65–74, ISBN: 9781450343541, DOI: 10.1145/3020078.3021744, URL: https://doi.org/10.1145/3020078.3021744.

[126] Stylianos I. Venieris and Christos-Savvas Bouganis, « fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs », *in*: *IEEE Transactions on Neural Networks and Learning Systems* 30.*2* (2019), pp. 326–342, DOI: 10.1109/TNNLS.2018.2844093.

[127] Rangharajan Venkatesan et al., « MAGNet: A Modular Accelerator Generator for Neural Networks », *in*: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[128] Xilinx Vivado, 2020.

[129] Kazutoshi Wakabayashi, « C-Based Behavioral Synthesis and Verification Analysis on Industrial Design Examples », *in*: *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, Yokohama, Japan: IEEE Press, 2004, pp. 344–348, ISBN: 0780381750.

[130] Chao Wang et al., « DLAU: A Scalable Deep Learning Accelerator Unit on FPGA », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.*3* (2017), pp. 513–517, DOI: 10.1109/TCAD.2016.2587683.

[131] Zi Wang, Jianqi Chen, and Benjamin Carrion Schafer, « Efficient and Robust High-Level Synthesis Design Space Exploration through offline Micro-kernels Pre-characterization », *in*: *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 145–150, DOI: 10.23919/DATE48585.2020.9116309.

[132] P. N. Whatmough et al., « FixyNN: Efficient Hardware for Mobile Computer Vision via Transfer Learning », *in*: *The 2nd Conference on Systems and Machine Learning (SysML)*, 2019, URL: https://arxiv.org/pdf/1902.11128.

[133] Xilinx, *Vitis AI*, 2021, URL: https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html.

[134] Sotirios Xydis et al., « SPIRIT: Spectral-Aware Pareto Iterative Refinement Optimization for Supervised High-Level Synthesis », *in*: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.*1* (2015), pp. 155–159, DOI: 10.1109/TCAD.2014.2363392.

[135] Georgios Zacharopoulos et al., « Machine Learning Approach for Loop Unrolling Factor Prediction in High Level Synthesis », *in*: *2018 International Conference on High Performance Computing Simulation (HPCS)*, 2018, pp. 91–97, DOI: 10.1109/HPCS.2018.00030.

[136] Chen Zhang et al., « Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks », *in*: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170, ISBN: 9781450333153, DOI: 10.1145/2684746.2689060, URL: https://doi.org/10.1145/2684746.2689060.

[137] J. Zhao et al., « COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications », *in*: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 430–437, DOI: 10.1109/ICCAD.2017.8203809.

[138] Marcela Zuluaga et al., « "Smart" Design Space Sampling to Predict Pareto-Optimal Solutions », *in*: *SIGPLAN Not.* 47.*5* (June 2012), pp. 119–128, ISSN: 0362-1340, DOI: 10.1145/2345141.2248436, URL: https://doi.org/10.1145/2345141.2248436.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Université Bretagne Sud

Titre : Approche haut niveau pour la génération automatique d'accélérateurs matériels optimisés pour les réseaux de neurones profonds

Mot clés : Synthèse haut-niveau, Réseaux de neurones convolutionnels, Systèmes embarqués, Méthodologies pour l'EDA, Accélerateurs matériels

Résumé : Étant l'une des solutions de pointe dans le domaine de la vision par ordinateur, les réseaux de neurones convolutifs (CNN) évoluent rapidement. En effet, les chercheurs déploient de nombreux efforts pour améliorer la précision de ces algorithmes afin de répondre à des tâches de détection et de reconnaissance d'objet de plus en plus complexes. L'amélioration des performances applicatives, ou encore la diminution des besoins de calcul et de mémoire des CNNs sont principalement apportées par de nouvelles topologies ou par l'introduction de nouveaux types de couches. Les CNN se caractérisent par leur parallélisme intrinsèque et sont bien adaptés à l'accélération matérielle. Malgré ces innovations, les besoins importants en mémoire et la complexité de calcul des CNN les rendent difficiles à intégrer dans les systèmes embarqués. De plus, l'évolution rapide des algorithmes rend difficile leur maîtrise par les architectes matériels, ce qui rend l'écart logiciel-matériel existant de plus en plus important. En outre, la conception d'accélérateurs matériels prend un temps important. Dans cette thèse, les défis du processus de conception des accélérateurs de réseaux de neurones sont abordés en étudiant divers aspects du flot de conception, de l'application, de la génération de matériel et de l'exploration de l'espace de conception. Ces aspects sont abordés via SHEFTENN, une approche d'exploration logicielle et matérielle de bout en bout pour la génération d'accélérateurs CNN. SHEFTENN vise à réduire l'écart entre le matériel et le logiciel en introduisant une phase de caractérisation visant à étudier la description algorithmique d'un point de vue matériel et en utilisant ensuite des techniques modernes de conception de haut-niveau pour les systèmes électroniques (ESL) telles que la synthèse de haut niveau pour générer du matériel à partir d'une description logicielle. Tirant parti des métriques de caractérisation, l'exploration automatique de l'espace de conception est effectuée par un module d'optimisation hybride qui utilise une combinaison d'évaluations de l'architecture candidate basées sur des modèles et des synthèses réelles. Les résultats expérimentaux montrent l'efficacité de l'approche proposée et débouchent sur des perspectives prometteuses.

Title: High-level approach for the automatic generation of optimized hardware accelerators for deep neural networks

Keywords: High-Level Synthesis, Convolutional Neural Networks, Embedded Systems, Methodologies for EDA, Hardware Accelerators

Abstract: Being one of the cutting edge solutions in the computer vision field, Convolutional neural networks (CNNs) are rapidly evolving. Indeed, researchers put many efforts to enhance the accuracy of these algorithms to meet increasingly complex object detection and recognition tasks. The improvement in application performance, or even the reduction in the computation and memory requirements of CNNs, are mainly brought about by new topologies or by the introduction of new types of layers. CNN are characterized by their spatial parallelism and are well adapted for hardware acceleration. Despite these innovations, the large memory requirements and computational complexity of CNNs make them difficult to embed in embedded systems. Additionally, the fast algorithmic evolutions are difficult to follow and master for hardware architects, which makes an existing software-hardware gap growing wider. Besides, designing hardware accelerators is a time consuming. In this dissertation, the challenges of the design process of neural networks accelerators are tackled by investigating various aspects of the design flow, the application, the hardware generation, and the exploration of the design space. These aspects are addressed through SHEFTENN, an end-to-end software and hardware exploration approach for CNN accelerator generation. SHEFTENN aims at reducing the gap between hardware and software by introducing a characterization phase to study the algorithmic description with a hardware point of view and by using modern Electronic System Level (ESL) design techniques such as high-level synthesis to generate hardware with a software point of view. Leveraging characterization metrics, automatic design space exploration is performed by an optimization module which uses a combination of model-based evaluations and real syntheses. Experimental results show the effectiveness of the proposed approach and lead to promising perspectives.