

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

**Implémentation efficiente de produits de convolution sur des opérandes
représentés sur moins de 8 bits**

THÉO DUPUIS

Département de génie électrique

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie électrique

Août 2023

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

**Implémentation efficace de produits de convolution sur des opérandes
représentés sur moins de 8 bits**

présenté par **Théo DUPUIS**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Tarek OULD BACHIR, président

Jean Pierre DAVID, membre et directeur de recherche

Yvon SAVARIA, membre et codirecteur de recherche

Guy BOIS, membre

REMERCIEMENTS

Je tiens tout d'abord à remercier mon directeur et codirecteur de recherche Jean Pierre David et Yvon Savaria pour leur précieux soutien, leurs conseils ainsi que leurs encouragements tout au long de ma maîtrise.

Je remercie aussi toutes les personnes avec qui ce travail a été réalisé et plus particulièrement MohammadHossein AskariHemmat et Yoan Fournier qui ont permis d'aboutir à ce travail.

Je remercie plus généralement le personnel de Polytechnique Montréal, le département de génie électrique ainsi que les différentes personnes qui ont fait en sorte que mon double diplôme se déroule dans les meilleures conditions possibles.

RÉSUMÉ

Les Réseaux de Neurones Convolutifs (CNN) sont utilisés dans de nombreux domaines et offrent une grande précision au détriment de la portabilité. Les progrès récents dans les techniques de quantification ont démontré que les Réseaux de Neurones Quantifiés (QNN) sur moins d'un octet atteignent une précision comparable, voire supérieure à celle des réseaux calculés en virgule flottante, tout en réduisant considérablement les coûts de calcul et l'empreinte mémoire. Cependant, les opérations sur des opérandes de taille inférieure à l'octet sur du matériel moderne standard sont généralement effectuées de manière sous optimale en raison de l'absence de prise en charge de telles précisions. Pour remédier à cette limitation, nous présentons dans ce travail différentes modifications d'Ara, un processeur 64-bit conforme à la norme RISC-V "V" en sources libres. Nous introduisons deux processeurs vectoriels spécialisés nommés Sparq et SparqV2, conçus spécifiquement pour accélérer l'inférence des QNN en utilisant une technique appelée ULPPACK. De plus, nous comparons les performances de ces processeurs à celles de Quark, un autre processeur basé sur Ara qui utilise la technique d'opérations binaires séquentielles pour les opérations sur des opérandes inférieurs à l'octet. Sparq(V2) et Quark sont implémentés avec la technologie GLOBAL FOUNDRIES 22FDX FD-SOI et étendent le jeu d'instructions (ISA) en ajoutant des instructions spécialisées pour améliorer l'efficacité des calculs sur des opérandes de tailles inférieures à l'octet. L'unité de calcul en virgule flottante est également supprimée pour minimiser la surface et la consommation d'énergie. Dans notre implémentation physique, nous démontrons qu'une voie (lane dans la terminologie relative à Ara) sur Sparq présente une surface inférieure de 43,3% par rapport à une voie d'Ara. Pour illustrer les performances de Sparq et de Quark, nous mettons en œuvre une opération de convolution 2D vectorisée pour des opérandes de basse précision. Nous montrons que Sparq peut accélérer de manière significative les calculs sur des opérandes de tailles inférieurs à l'octet, avec des accélérations moyennes respectives de 2.95 fois et 2.40 fois par rapport aux convolutions 2D en virgule flottante sur 32 bits pour différentes couches du modèle ResNet-18 en exploitant une résolution sur 2 et 4 bits. De plus, nous montrons que pour des opérandes de faible précision supérieure à 2 bits, l'utilisation de la méthode ULPPACK sur Sparq permet d'obtenir de meilleures performances que la méthode d'opérations binaires séquentielles utilisée par Quark.

ABSTRACT

Convolutional Neural Networks (CNNs) have found extensive use in various applications, offering high accuracy but limited portability when implemented with full precision. Recent advances in quantization techniques have shown that sub-byte Quantized Neural Networks (QNNs) can achieve accuracy comparable or even superior to their floating point counterparts while significantly reducing computational costs and memory requirements. However, performing sub-byte computations on standard hardware is generally suboptimal due to the lack of support for such precisions. In this work, we present several modifications of Ara, an open-source 64-bit RISC-V “V” compliant processor, to address this limitation. We introduce two specialized sub-byte vector processors called Sparq and SparqV2, designed specifically to accelerate QNN inference using a technique known as ULPPACK. Additionally, we compare the performance of these processors to Quark, another custom processor based on Ara that employs the bit-serial technique. Both Sparq(V2) and Quark are implemented using the GLOBAL FOUNDRIES 22FDX FD-SOI technology and extend their Instruction Set Architecture (ISA) by incorporating custom instructions to enhance sub-byte computation efficiency. To minimize area and power usage, the floating-point unit is removed. In our physical implementation, we demonstrate that Sparq’s lanes reduce the cell area by 43.3% compared to Ara’s lanes. To showcase the performance of Sparq and Quark, we implement an ultra-low-precision vectorized conv2d operation, leveraging the dedicated hardware. Our results indicate that Sparq achieves significant acceleration in sub-byte computations, leading to speedups of 2.95 times and 2.40 times over floating-point 32-bit 2D convolutions for different layers of the ResNet-18 model processed in 2-bit and 4-bit resolution, respectively. Furthermore, we show that for low-precision operands above 2-bit, utilizing the ULPPACK method on Sparq yields better performance than the bit-serial method employed by Quark.

TABLE DES MATIÈRES

REMERCIEMENTS	iii
RÉSUMÉ	iv
ABSTRACT	v
TABLE DES MATIÈRES	vi
LISTE DES TABLEAUX	ix
LISTE DES FIGURES	x
LISTE DES SIGLES ET ABRÉVIATIONS	xiv
CHAPITRE 1 INTRODUCTION	1
1.1 Modèles de réseaux de neurones convolutifs	1
1.2 Réduire la taille et la complexité des modèles	2
1.3 Implémentation sur matérielle et limitations	3
1.4 Organisation du mémoire	3
CHAPITRE 2 REVUE DE LITTÉRATURE	5
2.1 Produit de convolution et corrélation croisée	5
2.1.1 Définitions	5
2.1.2 Implémentation	6
2.2 Quantification	10
2.2.1 Quantification uniforme	10
2.2.2 Consommation d'énergie et impact sur les performances	12
2.2.3 Impact sur la précision	13
2.2.4 Quantification inférieure à l'octet	14
2.3 Implémentation du produit de convolution 2D inférieur à 8 bits	15
2.3.1 Limitations	16
2.3.2 Accélération logicielle	16
2.4 Matériel cible	19
2.4.1 Processeurs vectoriels à usage général	20
2.4.2 Accélérateurs FPGA et ASIC	21

CHAPITRE 3	DÉMARCHE DE RECHERCHE ET ORGANISATION	24
CHAPITRE 4	ARTICLE 1 : SPARQ : A CUSTOM RISC-V VECTOR PROCESSOR FOR EFFICIENT SUB-BYTE QUANTIZED INFERENCE	26
4.1	Détails de l'article	26
4.2	Contexte	26
4.3	Contributions	27
4.4	Abstract	28
4.5	Introduction	28
4.6	Background and Related Work	29
4.6.1	Sub-byte Convolutional Neural Networks	29
4.6.2	Sub-byte Computation on Commodity Hardware	30
4.7	Software implementations	31
4.7.1	Optimized Vector Conv2d	31
4.7.2	ULPPACK	32
4.8	Proposed Architecture	32
4.8.1	vmacsr Custom Instruction	32
4.8.2	2D Convolution Algorithm	33
4.9	Results and Performance Analysis	35
4.9.1	Performance Analysis	35
4.9.2	Physical Implementation	37
4.10	Conclusion	37
4.11	Acknowledgments	38
CHAPITRE 5	AUTRES ARCHITECTURES ET ANALYSE DES PERFORMANCES	39
5.1	Méthodologie	39
5.1.1	Compteur de cycles	39
5.1.2	Calcul des performances	40
5.1.3	Génération des données et essai	41
5.2	SparqV2	41
5.2.1	Différences et motivations	41
5.2.2	Analyse des performances	44
5.2.3	Implémentation physique	47
5.3	Quark	48
5.3.1	Différences et motivations	48
5.3.2	Analyse des performances	50
5.3.3	Implémentation physique	52

CHAPITRE 6	COMPARAISON DES PERFORMANCES	54
6.1	Comparaison sur des architectures standards	54
6.2	Comparaison entre Quark et SparqV2	55
6.2.1	Performances	56
6.2.2	Caractéristiques	58
CHAPITRE 7	DISCUSSION GÉNÉRALE	60
CHAPITRE 8	CONCLUSION ET RECOMMANDATIONS	63
8.1	Synthèse des travaux	63
8.2	Limitations des solutions proposées	64
8.3	Améliorations futures	64
RÉFÉRENCES		65

LISTE DES TABLEAUX

Tableau 2.1	Taille de ResNet-18 [1] à différents niveaux de quantifications	12
Tableau 2.2	Consommation d'énergie en picojoules [pJ] pour une multiplication et une addition en 45nm / 0.9V, tiré de [2]	12
Tableau 2.3	Consommation d'énergie en picojoules [pJ] pour un accès en lecture mémoire sur 64 bits, tiré de [2]	12
Tableau 2.4	Précision de différents modèles ResNet [1] et EfficientNet [3] sur ImageNet [4] et impact relatif de la quantification sur des entiers 8 bits (int8) après l'entraînement (PTQ) et intégrée à l'entraînement (QAT), tiré de [5]	14
Tableau 2.5	Comparaison de la précision [%] entre différentes méthodes de quantification de ResNet [1] et EfficientNet [3] sur 2 à 4 bits sur l'ensemble ImageNet [4]. <i>WxAy</i> correspond à la précision des poids (W) et des activations (A). Les précisions de référence sont issues des résultats de LSQ [6] et LSQ+ [7]	15
Tableau 2.6	Accélération moyenne du débit d'images traitées sur ResNet-18 et ResNet-50 [1] à l'aide de QNNPACK [8] et ULPPACK [9] sur différents ensembles de tests par rapport au modèle virgule flottante 32 bits sur un Raspberry Pi 4, tiré de [9]	19
Tableau 4.1	Accuracy on Quantized Resnet18 Using LG-LSQ [10]	30
Tableau 4.2	Physical Implementation of Ara and Sparq Lanes	37
Tableau 6.1	Récapitulatif des caractéristiques des différentes solutions proposées pour une configuration sur quatre voies	59

LISTE DES FIGURES

Figure 2.1	Illustration de l'opération de convolution 2D dans les couches de convolution pour le calcul d'une valeur sur le tenseur de sortie. F est le tenseur de poids, I est le tenseur d'entrée (ou d'activation) et S , le tenseur de sortie	7
Figure 2.2	Illustration de l'opération de convolution 2D par multiplication matricielle (GEMM) après l'utilisation de la transformation image à colonne (Im2Col) sur le tenseur d'entrée (ou activation) I et le tenseur de filtres F	8
Figure 2.3	Exemple d'empaquetage de deux vecteurs $\{w_0, w_1\} = \{3, 1\}$ et $\{a_0, a_1\} = \{2, 0\}$ pour effectuer une opérations séquentielles binaires sur une précision de 2 bits. Les vecteurs empaquetés sont $\bar{w}_0 = \{3\}$, $\bar{w}_1 = \{2\}$, $\bar{a}_0 = \{0\}$ et $\bar{a}_1 = \{2\}$	17
Figure 2.4	Exemple du produit scalaire de deux vecteurs $w = \{w_0, w_1\}$ et $a = \{a_0, a_1\}$ pour effectuer une opération bit-sérielle sur une précision de 2 bits	18
Figure 2.5	Exemple d'exécution de diverses instructions sur une architecture parallèle (a) et vectoriel (b). N et M correspondent respectivement au nombre d'unités de calculs et à la longueur des vecteurs à traiter	20
Figure 2.6	Diagramme simplifié d'un accélérateur générique de Réseau de Neurones Profond (DNN) avec N unités de multiplication matricielle. Chaque unité est composée d'une grille d'unités de calculs (PE) chacune embarquant un réseau systolique de cellules de multiplication-accumulation (MAC). Les divers niveaux de tampons permettent d'augmenter la localité des données et sont contrôlées par l'unité de contrôle	22
Figure 4.1	Multiplication between 2 packed 8-bit vectors with 1-bit precision on weights and activation. The dot product of the two packed elements is computed using a single multiplication on the MSB. The result then needs to be shifted to retrieve the dot product.	31
Figure 4.2	Diagram of the multiply-shift-accumulate operation on 8-bit packed registers with 1-bit precision for activations and weights. A shifter is inserted between the multiplication and the accumulation	32
Figure 4.3	<code>vmacsr</code> encoding example	33

Figure 4.4	Performance comparison between several <i>conv2d</i> implementation using a 7×7 kernel size. Ultra-Low-Precision (ULP) and Low-Precision (LP) <i>conv2d</i> take advantage of the vmacsr instruction on Sparq while W1A1, W2A2, W3A3 <i>conv2d</i> run on native RISC-V “V” ISA	35
Figure 4.5	Relative speedup over the 16-bit <i>conv2d</i> implementation on the overflow-free precision region. The kernel size is 7×7 over a $32 \times 256 \times 256$ input. (a) for the native implementation benchmarked on Ara (b) for the accelerated version taking advantage of the vmacsr instruction benchmarked on Sparq	36
Figure 4.6	Ara and Sparq lanes placed and routed designs. ■ is the vector register file, ■ is the operands queue, ■ is the vector fixed point multiplication and division units with or without the FPU, ■ is the vector ALU	36
Figure 5.1	Schéma simplifié de l’instruction de multiplication-accumulation intégrant le décaleur en cascade piloté par le registre vxsh configurable avec l’instruction vsetsh dans SparqV2. vxsh.amount correspond à la valeur de décalage, vxsh.dir permet de sélectionner la direction (droite/gauche) et vxsh.sign configure le type de décalage (arithmétique/logique)	42
Figure 5.2	Illustration de l’instruction vsetsh permettant de configurer le registre vxsh à l’aide d’une valeur immédiate sur 8 bits. A/L signifie Arithmétique ou Logique, D/G signifie Droite ou Gauche. La valeur de N détermine le nombre de bit permettant d’encoder la valeur de décalage configurable disponible. ‘-’ correspond à des bits dont l’état n’est pas pris en compte	42
Figure 5.3	Comparaison des performances en opérations par cycle pour différente précision des opérandes sur un noyau de 3×3 . Les implémentations ULP et LP-conv2d tirent parti du matériel spécialisé sur SparqV2 tandis que W1A1, W2A2 et W3A3-conv2d sont implémentées sur Ara. $WxAy$ correspond à la précision des opérandes avec x la précision des poids et y la précision des activations en bits	45

Figure 5.4	Accélération relative par rapport à une implémentation en virgule flottante 32 bits dans la région sans débordement. L'activation est un tenseur de $64 \times 256 \times 256$ convolué avec un noyau 3×3 . (a) est l'implémentation d'ULPPACK sur Ara sans modification matérielle et (b) correspond à l'implémentation d'ULPPACK accélérée par le décaleur en cascade sur SparqV2. $WxAy$ correspond à la précision des opérandes avec x la précision des poids et y la précision des activations en bits . . .	46
Figure 5.5	Implémentation d'une voie de SparqV2 en utilisant la technologie GF22 nm. Le décaleur en cascade est implémenté avec $N = 6$ bits. VRF est le banc de registres vectoriels, OQ la file d'attente d'opérandes, VMUL l'unité de multiplication et VALU l'unité de calcul arithmétique et logique	47
Figure 5.6	Comparaison des performances en opérations par cycle pour différentes précisions des opérandes sur un noyau de 3×3 sur Quark. $WxAy$ correspond à la précision des opérandes avec x la précision des poids et y la précision des activations en bits	50
Figure 5.7	Accélération relative sur Quark par rapport à une implémentation en virgule flottante 32 bits. L'activation est un tenseur de $64 \times 32 \times 32$ avec un noyau de 3×3 . $WxAy$ correspond à la précision des opérandes avec x la précision des poids et y la précision des activations en bits . .	51
Figure 5.8	Implémentation après placement et routage de Quark en comparaison d'Ara en GF22 nm avec, VRF le banc de registres vectoriels, OQ la file d'attente d'opérandes, VFPU l'unité de calcul en virgule flottante et VALU l'unité de calcul arithmétique et logique. Image tirée de [11] avec autorisation de l'auteur	52
Figure 6.1	Comparaison des performances en opérations par cycle d'une multiplication matricielle $512 \times 512 \times 512$ sur un Raspberry Pi 4 et d'une opération de convolution 2D entre un tenseur d'entrée de $1 \times 56 \times 25 \times 25$ et un tenseur de 512 filtres de $56 \times 3 \times 3$ sur Ara en utilisant ULPPACK [9]. $WxAy$ correspond à la précision des poids (W) et des activations (A) en bits	54
Figure 6.2	Comparaison des performances en opérations par cycle pour des précisions d'opérande de 1 et 2 bits sur un noyau de 3×3 sur SparqV2 et Quark (4 voies)	56

- Figure 6.3 Accélération relative de couches de convolution de ResNet-18 [1] sur l'ensemble CIFAR-10 [12] pour différentes quantifications sur SparqV2 et Quark. L'abscisse représente le numéro de la couche avec en premier, les couches du modèle [a-b]. Ensuite sont donnés dans l'ordre, [nombre de canaux de sorties, nombre de canaux d'entrée, hauteur et largeur de l'entrée, hauteur et largeur du noyau] 57
- Figure 6.4 Accélération relative de couches de convolution de VGG16 [13] sur l'ensemble ImageNet [4] pour différentes quantifications sur SparqV2 et Quark. L'abscisse représente le numéro de la couche avec en premier, les couches du modèle [a-b]. Ensuite sont donnés dans l'ordre, [nombre de canaux de sorties, nombre de canaux d'entrée, hauteur et largeur de l'entrée, hauteur et largeur du noyau] 57

LISTE DES SIGLES ET ABRÉVIATIONS

ASIC	Application Specific Integrated Circuit Circuits Intégrés Spécifique à une Application
CISC	Complex Instruction Set Computer Ordinateur à Jeu d’Instruction Complexe
CNN	Convolutionnal Neural Network Réseau de Neurones Convolutif
CPU	Central Processing Unit Unité Centrale de Traitement
CSR	Control and Status Registers Registres de Contrôle et de Statut
DFT	Discrete Fourier Transform Transformée de Fourier Discète
DNN	Deep Neural Network Réseau de Neurones Profond
DRAM	Dynamic Random Access Memory Mémoire à Accès Aléatoire Dynamique
FFT	Fast Fourier Transform Transformée de Fourier Rapide
FLOP	Floating Point Operations Opérations en Virgule Flottante
FPGA	Field Programmable Gate Array Circuit Logique Programmable
FPU	Floating Point Unit Unité de Calcul en virgule Flottante
GEMM	GEneral Matrix Multiplication Multiplication Matricielle Générique
GF22	Global Foundries 22FDX FD-SOI
GPP	General Purpose Processor Processeur à Usage Général
GPU	Graphical Processing Unit Unité de Traitement Graphique
GPU	Graphical Processing Unit Unité de Traitement Graphique
GPVP	General Purpose Vector Processor Processeur Vectoriel à Usage Général
HDL	Hardware Description Langage Langage de Description Matériel
IDFT	Inverse Discrete Fourier Transform Transformée de Fourier Discrète Inverse
Im2Col	Transformation image à colonne Image to column transform
ISA	Instruction Set Architecture Jeu d’instruction
LSB	Less Significant Bit Bit le Moins Significatif
MAC	Multiply-Accumulate Multiplication-Accumulation
MB	MegaBytes MégaOctets
MMU	Matrix Multiplication Unit Unité de Multiplication Matricielle
MSB	Most Significant Bit Bit le Plus Significatif
MSE	Mean Square Error Erreur Quadratique Moyenne
NN	Neural Network Réseau de Neurones
PQT	Post-Training Quantization Quantification Post-Entraînement
QAT	Quantized Aware Training Quantification Intégrée à l’Entraînement

QCNN	Quantized Convolutional Neural Network Réseau de Neurones Convolutif Quantifié
QDNN	Quantized Deep Neural Network Réseau de Neurones Profond Quantifié
QNN	Quantized Neural Network Réseau de Neurones Quantifié
ReLU	Rectified Linear Unit Unité Linéaire Rectifiée
RISC	Reduced Instruction Set Computer Ordinateur à Jeu d'Instruction Réduit
RTL	Register Transfer Level Niveau de Transfert de Registre
RVV	RISC-V Vector Extension Extension Vectorielle RISC-V
SIMD	Single Instruction, Multiple Data Instruction Unique à Données Multiples
SVE	Scalable Vector Extension Extension Vectorielle Configurable
TPU	Tensor Processing Unit Unité de Traitement Tensoriel

CHAPITRE 1 INTRODUCTION

Inspiré par les travaux de David H. Hubel et Torsten Wiesel sur le cortex de la vision chez le chat dans les années 1960 [14], Kunihiko Fukushima propose Neocognitron en 1980 [15], considéré comme le premier Réseau de Neurones Convolutifs (CNN). Basés sur l'opération de convolution, les CNN sont aujourd'hui employés dans de nombreux domaines afin de réaliser des tâches complexes. Parmi ces applications, nous retrouvons notamment des tâches appliquées au suivi d'objets [16], à la reconnaissance vocale [17] ou à la classification d'images [18].

1.1 Modèles de réseaux de neurones convolutifs

Ces dix dernières années, une évolution rapide des capacités des Réseaux de Neurones Convolutifs a pu être observée. Grâce à de nouvelles méthodes et architectures, ces réseaux sont devenus plus précis et flexibles au détriment d'une augmentation de la taille des modèles, ainsi que de la complexité en calculs. En 2012, AlexNet [19] est proposé pour classifier l'ensemble de données ImageNet [4]. Avec une précision de 63.3% [20], il suscite un regain d'intérêt pour les CNN et l'apprentissage profond dans les applications de vision par ordinateur. Une dizaine d'années plus tard, CoAtNet-7, modèle environ $4\times$ plus large et $3500\times$ plus complexe qu'AlexNet est présenté avec une précision de 90.9% [21]. D'autres modèles plus simples et compacts sont proposés dans un objectif de compromis entre la précision, l'encombrement en mémoire et la complexité. Par exemple, EfficientNet-B0 (2019) est un modèle d'environ 21 millions d'octets (MB) pour 390 millions d'opérations (MFLOP) atteignant une précision de 77.1% [3]. Avec un encombrement en mémoire plus de $10\times$ moindre et une complexité environ $2\times$ inférieure à AlexNet, ce modèle présente tout de même une précision plus élevée. En revanche, le coût lié à l'entraînement et à l'inférence demeure particulièrement important et requiert l'emploi de matériels dédiés. Des appareils tels que des Unités de Traitement Graphique (GPU) et des systèmes haute performance consommant une grande quantité d'énergie sont nécessaires. L'utilisation de tels équipements pose un réel défi quant au déploiement de ces modèles sur des appareils dont les ressources sont limitées. Cette problématique a ouvert à la recherche de solutions permettant de porter ces modèles complexes sur des appareils contraints en réduisant leur taille et leur complexité.

1.2 Réduire la taille et la complexité des modèles

Dans l’objectif de réduire la charge de calculs et la taille des modèles tout en tentant de conserver une précision maximale, de nombreuses méthodes indépendantes mais complémentaires, telles que l’élagage de paramètres [22] ou la compression de données [23], ont été proposées.

Il existe deux méthodes principales d’élagage largement utilisées : l’élagage non structuré et l’élagage structuré. L’élagage non structuré consiste à éliminer les connexions les moins significatives d’un modèle en se basant sur un seuil prédéfini, souvent exprimé en proportion des poids à conserver. Cette approche permet de réduire la taille du modèle de manière proportionnelle à la quantité de paramètres supprimés. Cependant, elle entraîne des irrégularités dans l’ordre des données ainsi que des représentations éparpillées, et il peut être difficile d’exploiter pleinement cette méthode pour bénéficier d’une accélération de l’exécution lors du déploiement du modèle sur certaines cibles matérielles. Contrairement à l’élagage non structuré, qui élimine des paramètres individuels, l’élagage structuré supprime des canaux, des filtres, voire des couches entières d’un modèle. Cet élagage permet de conserver une structure régulière en mémoire, qui facilite le déploiement sur les architectures matérielles en plus d’offrir une réduction de la taille du modèle ainsi qu’une augmentation des performances. En revanche, l’augmentation de la granularité d’élagage de cette méthode par rapport à un élagage non structuré réduit sa flexibilité, ce qui peut conduire à des gains moins importants sur la taille du modèle ainsi qu’à une perte plus importante de sa précision.

Il existe deux catégories distinctes de processus de compression : la compression avec perte et la compression sans perte. La compression sans perte vise à réduire la taille du modèle en modifiant sa structure ou sa représentation en mémoire, par exemple en utilisant des précisions variables pour représenter les paramètres en fonction de leur fréquence d’apparition [23]. Toutefois, ce sont les compressions avec pertes qui offrent généralement les taux de compression les plus importants au détriment d’une perte d’information. Une méthode de compression avec perte couramment utilisée dans le contexte des modèles de Réseaux de Neurones est la quantification de ses paramètres. Au cours de ces dernières années, de nombreuses stratégies de compression par quantification [24] visant à réduire la précision des paramètres ont été développées. Permettant d’adresser simultanément la problématique de la taille du modèle et l’efficacité de ses calculs, la quantification repose sur la réduction du nombre de bits employés à la représentation des paramètres d’un modèle. Typiquement représenté en virgule flottante 32 bits (fp32), une quantification sur des entiers standards 8 bits (int8) permet de diviser par quatre la taille du modèle tout en permettant l’emploi d’instructions plus efficaces et économes en énergie. En outre, il est possible de quantifier les paramètres avec une précision inférieure à 8 bits, ce qui réduit davantage la taille du modèle tout en limitant la perte de

précision en fonction des modèles et des ensembles de données considérés [6, 7, 24–26]. Le sujet central de ce mémoire concerne précisément l’amélioration des performances des opérations des modèles de CNN dans le contexte d’une quantification inférieure à l’octet.

1.3 Implémentation sur matérielle et limitations

L’implémentation des modèles de Réseaux de Neurones Convolutifs sur des Unités de Traitement Graphique (GPU) a révolutionné l’apprentissage machine. Les GPUs présentent l’avantage principal d’offrir un traitement parallèle efficace des données, répondant à un besoin en calculs grandissant des modèles. Cependant, leur utilisation présente des inconvénients majeurs liés à la consommation élevée d’énergie et à leur encombrement spatial qui les rendent inutilisables dans des contextes de ressources contraintes ou limitées.

Pour palier ces inconvénients, de nouvelles solutions matérielles adaptées ont émergé. De nombreux circuits accélérateurs spécialement conçus pour les CNN ont été développés, en utilisant des circuits logiques programmables (FPGA) [27, 28], ou des circuits intégrés spécifiques à une application (ASIC) [29, 30]. Ces circuits présentent l’avantage d’être extrêmement performants et économes en énergie, mais ils manquent de flexibilité. Contrairement aux circuits spécifiques, le regain d’intérêt récent pour les jeux d’instructions vectoriels extensibles [31, 32] sur des architectures d’ordinateur à jeu d’instruction réduit (RISC), a permis d’augmenter les capacités de parallélisation des calculs tout en conservant une grande flexibilité. Les processeurs à usage généraux (GPP) embarquant ces jeux d’instructions vectoriels [33–35] sont capables de diminuer considérablement le temps d’exécution d’algorithmes basés sur des opérations vectorielles, comme les opérations de convolution. En revanche, ils présentent des performances inférieures à celle des circuits spécifiques et possèdent l’inconvénient de ne supporter que les représentations standards de nombres. De façon naïve, ils ne peuvent donc tirer avantage d’une quantification d’un modèle inférieur à l’octet.

1.4 Organisation du mémoire

Ce mémoire se concentre principalement sur l’intégration efficace de l’opération de convolution 2D sur une précision inférieure à l’octet sur des jeux d’instructions vectoriels. Dans cet objectif, nous proposons de modifier Ara [34], un processeur RISC-V vectoriel 64 bits en sources libres. Nous proposons des algorithmes tirant parti de l’ajout de nos instructions spécialisées pour accélérer des convolutions 2D sur des opérandes inférieurs à 8 bits. Le chapitre 4 présente Sparq. Sparq est un processeur basé sur Ara et conçu pour l’accélération d’opérations sur des opérandes de moins de 8 bits en utilisant la méthode ULPPACK [9].

L’ajout d’une instruction spécialisée de multiplication-décalage-accumulation (**vmacsr**) permet une accélération comprise entre $1.7\times$ à $3.2\times$ sur des opérandes de 2 à 4 bits par rapport à une convolution 2D sur 16 bits avec un noyau 7×7 . Les modifications permettent aussi de réduire la surface nécessaire pour l’implémentation d’une voie (lane dans la terminologie d’Ara) sur la technologie GLOBAL FOUNDRIES 22FDX FD-SOI (GF22) de 43.3%. Le chapitre 5 présente une amélioration de Sparq nommé SparqV2 qui généralise l’instruction **vmacsr** en introduisant un registre configurable permettant de manipuler un décaleur en cascade tout en conservant les performances de Sparq. Un autre processeur nommé Quark y est aussi présenté. Quark [11] utilise la méthode d’opérations binaires séquentielles pour accélérer l’exécution de la convolution 2D par un facteur compris entre 3.1 et 5.21 sur respectivement 1 et 2 bits par rapport à une convolution 2D sur virgule flottante 32 bits. Enfin, le chapitre 6 compare les performances des différentes implémentations dans des contextes réalistes. Les modifications ainsi que les algorithmes proposés permettent une accélération moyenne comprise entre 2.75 et 8.35 de l’exécution des couches de convolutions pour différents modèles de CNN sur différents ensembles de données par rapport à leur équivalent en virgule flottante 32 bits.

Ce travail s’inscrit dans le projet “RISC-V Vector Processor for High-throughput Multidimensional Sensor Data Processing and Machine Learning Acceleration at the Edge” effectué en collaboration entre Polytechnique Montréal, l’ETH de Zürich, OpenHardware Group et CMC Microsystems. Ce projet est financé par CMC Microsystems, Mitacs et OpenHardware Group.

CHAPITRE 2 REVUE DE LITTÉRATURE

L'objectif de ce travail est de diminuer le temps d'exécution de l'opération de convolution 2D, principalement dans le contexte des Réseaux de Neurones Convolutifs Quantifiés (QCNN). Dans ce cadre, la section 2.1 définit le produit de convolution 2D dans le contexte des Réseaux de Neurones Convolutifs (CNN), puis la section 2.2 introduit le processus de quantification de ces réseaux. La section 2.3 présente les solutions logicielles et matérielles permettant d'effectuer une convolution 2D efficace sur des opérandes dont la représentation est inférieure à 8 bits. Enfin, la section 2.4 présente brièvement les solutions matérielles permettant d'accélérer l'exécution d'algorithmes vectoriels tels que la convolution 2D.

2.1 Produit de convolution et corrélation croisée

2.1.1 Définitions

Les Réseaux de Neurones Convolutifs correspondent à un type de Réseaux de Neurones (NN) particulier destiné au traitement de structures représentées sur deux dimensions. Ces réseaux sont employés dans des domaines tels que la détection d'objet [36], la classification d'image [18], la reconnaissance vocale [17], la segmentation d'images [37], et d'autres applications. Les Réseaux de Neurones Convolutifs sont nommés d'après l'opération mathématique principale dont ils tirent parti, à savoir l'opération de convolution. Mathématiquement, le produit de convolution de deux fonctions continues $f(x)$ et $g(x)$, $x \in \mathbb{R}$ est définie par :

$$f(x) * g(x) = \int_{-\infty}^{+\infty} f(x-t) \times g(t) dt \quad (2.1)$$

Son équivalent sur deux fonctions discrètes $f[n]$ et $g[n]$, $n \in \mathbb{Z}$, est alors défini par :

$$f[n] * g[n] = \sum_{k=-\infty}^{+\infty} f[n-k] \times g[k] \quad (2.2)$$

Toutefois, ce n'est pas l'opération de convolution qui est employée dans les CNN, mais une opération légèrement différente nommée produit de corrélation croisée. Cette opération est identique à la convolution à ceci près qu'elle ne retourne pas la fonction jouant le rôle de fenêtre glissante représentée par la fonction f dans l'équation 2.2. Mathématiquement, l'opération de corrélation croisée sur deux fonctions discrètes $f[n]$ et $g[n]$, $n \in \mathbb{Z}$ est définie par :

$$f[n] \star g[n] = \sum_{k=-\infty}^{+\infty} f[n+k] \times g[k] \quad (2.3)$$

Dans le cas particulier où les fonctions sont symétriques, l'opération de convolution et de corrélation croisée sont alors confondues. En généralisant l'équation 2.3 à des structures discrètes sur deux dimensions, nous pouvons alors définir une opération de corrélation croisée 2D par :

$$S[m, n] = I \star F = \sum_i \sum_j I[m+i, n+j] \times F[i, j] \quad (2.4)$$

Dans l'équation 2.4, nous pouvons considérer que I représente une matrice d'entrée (ou d'activation) et F une matrice de filtre. Dans un CNN, l'opération de convolution croisée est généralement effectuée sur des tenseurs, intégrant une notion de canaux, ainsi que de nombre de filtres. L'opération de corrélation croisée multicanaux sur deux dimensions entre un tenseur d'entrée I et un tenseur de filtres F est alors définie par :

$$S[k, m, n] = \sum_c \sum_i \sum_j I[c, m+i, n+j] \times F[k, c, i, j] \quad (2.5)$$

L'équation 2.5 définit l'opération de convolution 2D généralement employée dans les couches de convolutions d'un CNN, bien qu'elle corresponde à une opération de corrélation croisée 2D sur des tenseurs. Nous présentons ici une définition de l'opération entre un unique tenseur d'entrée (ou d'activation) et de multiples tenseurs de filtres. Cette version est plus adaptée pour des appareils à ressources limitées, car elle réduit notamment la latence de traitement des données, paramètre important lors de la phase d'inférence d'un CNN. Dans la suite de ce mémoire, l'emploi du terme *convolution* est fait sans distinction pour faire référence à un produit de convolution 2D ou à une opération de corrélation croisée 2D multicanaux sur des tenseurs comme décrit à l'équation 2.5.

2.1.2 Implémentation

Implémentation naïve

Il existe de nombreuses façons de réaliser une opération de convolution et de nombreux algorithmes permettant d'accélérer son exécution ont été proposés. L'Algorithme 1 présente une implémentation naïve de l'opération de corrélation 2D présentée dans l'équation 2.5 à l'aide de boucles imbriquées. Cette fonction, nommée *convolution directe*, est une implémentation naïve stationnaire sur la sortie et nécessite un nombre de multiplication-accumulation (MAC)

Algorithme 1 Implémentation naïve d'une corrélation croisée 2D avec des boucles imbriquées

```

 $C$       : Nombre des canaux de l'entrée et des filtres
 $H_o, W_o$  : Hauteur et largeur de la sortie
 $K, R, S$   : Nombre, hauteur et largeur des filtres
1: pour  $k \leftarrow 1$  to  $K$  faire
2:   pour  $c \leftarrow 1$  to  $C$  faire
3:     pour  $m \leftarrow 1$  to  $H_o$  faire
4:       pour  $n \leftarrow 1$  to  $W_o$  faire
5:         pour  $i \leftarrow 1$  to  $R$  faire
6:           pour  $j \leftarrow 1$  to  $S$  faire
7:              $S[k, m, n] += F[k, c, i, j] \times I[c, m + i, n + j]$ 

```

correspondant à KCH_oW_oRS , c'est-à-dire le produit des dimensions du tenseur de sortie et de filtres. Les dimensions de la sortie sont définies en fonction des dimensions de l'entrée en suivant :

$$\begin{cases} H_o = H_i - R + 1 \\ W_o = W_i - S + 1 \end{cases} \quad (2.6)$$

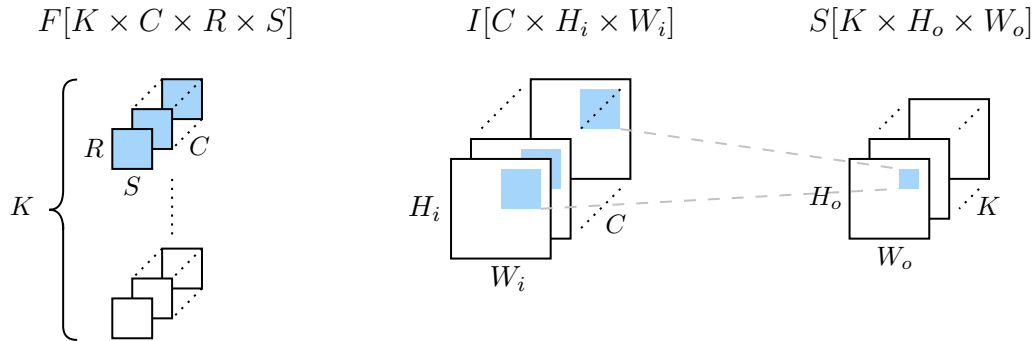


FIGURE 2.1 Illustration de l'opération de convolution 2D dans les couches de convolution pour le calcul d'une valeur sur le tenseur de sortie. F est le tenseur de poids, I est le tenseur d'entrée (ou d'activation) et S , le tenseur de sortie

Graphiquement, l'opération est illustrée par la Figure 2.1. La convolution se décompose alors en 4 étapes :

- (1) Chaque canal d'un tenseur de filtres F est superposé au canal de l'entrée I correspondant.
- (2) Chaque élément des filtres est multiplié un à un avec les éléments de l'entrée.
- (3) Tous les produits calculés à l'étape précédente sont sommés afin de construire un élément de sortie.
- (4) L'opération est répétée pour toutes les superpositions possibles des filtres sur l'entrée jusqu'à ce que tous les éléments de sorties soient calculés.

Contrairement aux autres couches des modèles de CNN, la convolution présente le temps d'exécution le plus important. Lors de la phase d'inférence, entre 50% et 90% des opérations sont dédiées à son calcul [38–41]. En outre, l'accroissement de la précision des modèles de CNN s'accompagnent généralement d'une augmentation du nombre de couches de convolution, augmentant la charge de calcul dédiée à cette opération. Par conséquent, il est pertinent de s'intéresser à augmenter sa performance.

Implémentations performantes

Des algorithmes plus performants sont généralement employés afin d'accélérer l'exécution de l'opération de convolution. Dans cette section, nous présentons brièvement les deux principaux algorithmes ainsi que leurs avantages et inconvénients.

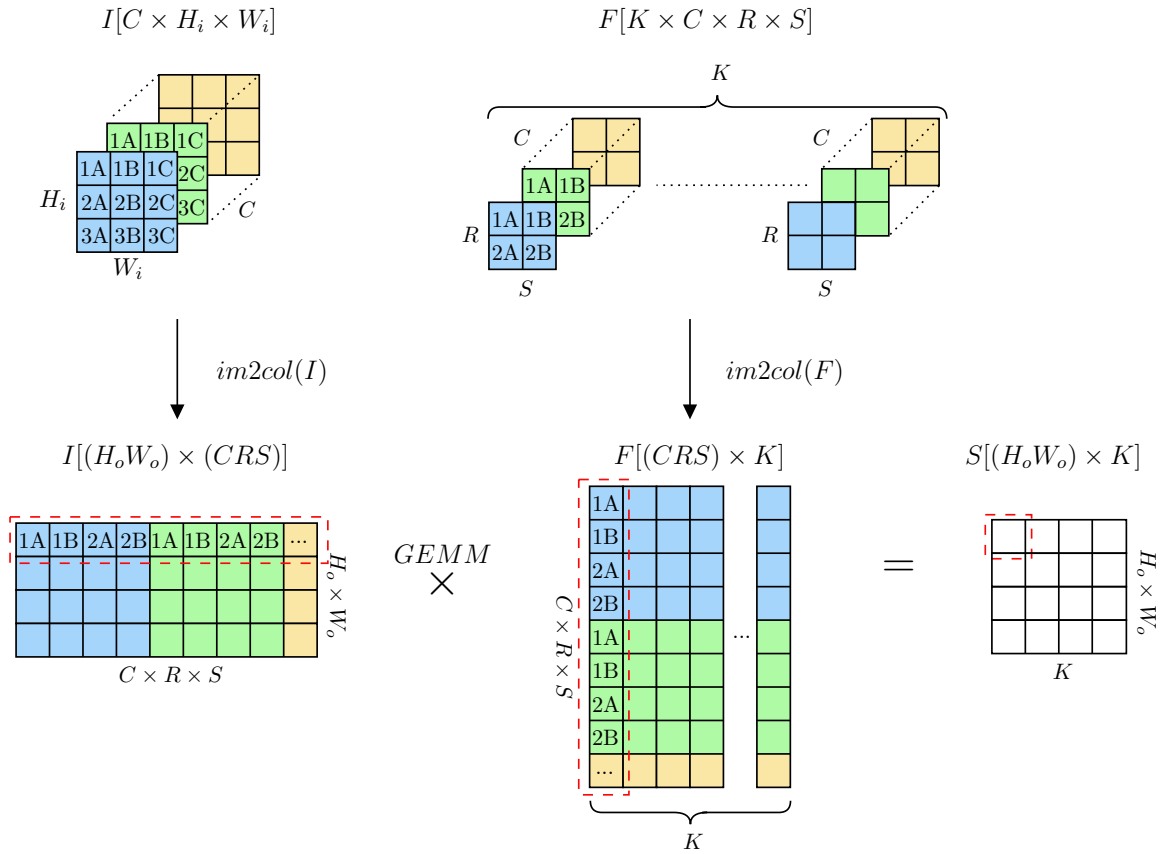


FIGURE 2.2 Illustration de l'opération de convolution 2D par multiplication matricielle (GEMM) après l'utilisation de la transformation image à colonne (Im2Col) sur le tenseur d'entrée (ou activation) I et le tenseur de filtres F

- **Im2Col/GEMM** [42] est un algorithme de convolution constitué de deux opérations. L'idée principale est de transformer une convolution sur des tenseurs en une multiplica-

tion matricielle. Comme illustré sur la Figure 2.2, chaque fenêtre de l'entrée, permettant le calcul d'une valeur de sortie, est aplatie horizontalement en suivant l'ordre des canaux et chaque filtre est aplati verticalement en suivant l'ordre des canaux et horizontalement en suivant l'ordre des filtres. Une multiplication matricielle sur deux dimensions peut ensuite être appliquée entre la matrice de filtres et la matrice d'entrée afin de calculer la matrice de sorties. Dans une application réaliste, la matrice de sortie est stockée sur une mémoire possédant seulement une dimension et il en est de même pour un tenseur de sortie. Cette matrice est donc identique en mémoire au résultat de l'opération de convolution entre tenseurs.

Cette méthode tire parti d'une réorganisation des valeurs d'entrées et de filtres de façon contiguë dans la mémoire, permettant une meilleure localité des données à chaque lecture en mémoire. En outre, les multiplications matricielles sont généralement implémentées dans des bibliothèques d'algèbre linéaire hautement optimisées, dont l'exécution est particulièrement rapide. Néanmoins, et bien que la transformation des filtres puisse être effectuée en amont du processus, chaque nouvelle entrée doit être transformée avant la multiplication matricielle, ce qui diminue la performance [43]. Enfin, comme illustré sur la Figure 2.2, cette transformation multiplie l'espace mémoire nécessaire au stockage à chaque tenseur d'entrée, par un facteur $R \times S$, avec respectivement R et S la hauteur et la largeur d'un filtre.

- **La transformée de Fourier rapide (FFT)** [44] est un algorithme permettant de calculer la transformée de Fourier discrète (DFT) d'une séquence. L'idée est d'utiliser cette transformée afin de réaliser l'opération de convolution temporelle de l'équation 2.5 dans le domaine fréquentiel.

Soit x et $y, \in \mathbb{C}^N$ et X et Y leur transformée de Fourier discrète respective, alors d'après le théorème de convolution :

$$\begin{aligned} x[n] * y[n] &\xrightarrow{DFT_k} X[k] \odot Y[k] \\ x[n] * y[n] &\xleftarrow{IDFT_n} X[k] \odot Y[k] \end{aligned}$$

avec $*$ l'opérateur de convolution et $\odot$ la multiplication élément à élément.

La transformée de Fourier discrète permet de remplacer l'opération de convolution dans le domaine temporel, par une opération de multiplication élément à élément dans le domaine fréquentiel. Il est alors possible d'utiliser la transformation réciproque, la transformée de Fourier discrète inverse (IDFT), pour recouvrer le résultat de la convolution dans l'espace temporel. En généralisant ce processus sur des objets de deux

dimensions, nous obtenons un moyen efficace d’effectuer une convolution [45, 46].

Toutefois, en raison de la nature cyclique de la transformée de Fourier, des effets de bords apparaissent et une partie des résultats doit être écartée. Aussi, bien que réduisant le calcul à une multiplication pour chaque valeur d’entrée, les multiplications doivent être effectuées sur des nombres complexes, ce qui réduit la performance de cette méthode. Néanmoins, l’opération de DFT et de IDFT sont aussi généralement implémentées de façon efficace dans des bibliothèques, ce qui rend cette méthode plus performante que l’Algorithme 1.

Généralement, le calcul des valeurs de sortie est subdivisé pour être effectuées parallèlement, afin d’augmenter la performance. En outre, de nombreux travaux proposent des variations ou des combinaisons de ces méthodes, principalement dans l’objectif de tirer avantage des architectures cibles [40, 47, 48]. Il est impossible de décrire tous les algorithmes de façon exhaustive, et beaucoup d’autres méthodes permettent d’accélérer l’opération de convolution tel que l’algorithme de Karatsuba [49], l’algorithme de Winograd [45, 50], l’algorithme de Strassen [51] et bien d’autres. Ils sont plus rarement employés, car ils ne permettent pas une accélération aussi importante ou leurs implémentations sont encore limitées ou contraignantes.

2.2 Quantification

Typiquement, les poids et activations d’un CNN sont représentés sur des formats en virgule flottante 32 bits. Le processus de quantification permet l’approximation du modèle original en virgule flottante 32 bits vers un format de plus basse précision (typiquement 1 à 8 bits) en quantifiant les paramètres correspondant aux poids, mais aussi les activations. Nous nous intéressons ici uniquement aux quantifications sur des représentations entières. Néanmoins, d’autres travaux prometteurs s’orientent vers des formats spécifiques, par exemple en virgule flottante sur 8 bits [52–55]. Toutefois, ces travaux correspondent à des formats non standardisés dont l’emploi reste encore marginal.

2.2.1 Quantification uniforme

Tout d’abord, il convient de définir une fonction capable de quantifier les poids et les activations d’un CNN sur une plage finie de valeurs. La fonction fait correspondre des valeurs réelles représentées en virgule flottante en entrée à leur représentation entière dans une précision plus basse. Un choix simple de fonction de quantification est :

$$x_q = Q(x) = \text{clip}\left(\left\lfloor \frac{x}{s} \right\rfloor - z\right) \quad (2.7)$$

Où x_q représente le vecteur de valeurs quantifiées du vecteur x et Q la fonction de quantification. L'échelle de quantification est représentée par le nombre réel s , le point de référence est représenté par le nombre z et $\lfloor \cdot \rfloor$ correspond à la fonction d'arrondi, qui permet de tronquer la partie décimale. La fonction $clip(\cdot)$ permet de saturer les valeurs dans une certaine plage $[U; L]$ dont le choix est détaillé dans l'équation 2.9. Cette méthode définit une façon d'effectuer une quantification, où toutes les valeurs quantifiées sont espacées uniformément.

Les valeurs réelles x peuvent être reconstruites en utilisant une opération réciproque à l'équation 2.7. Cette fonction est appelée fonction de déquantification, et permet de reconstruire les valeurs déquantifiées par l'opération suivante :

$$\bar{x} = s(Q(x) + z) \quad (2.8)$$

Le vecteur de valeurs réelles déquantifiées $\bar{x}$ est alors légèrement différent de x . Cette différence correspond à l'erreur de quantification due à l'arrondi effectué lors du processus de quantification. Une étape importante de la quantification uniforme est le choix du facteur d'échelle s utilisé par la fonction de quantification et de déquantification (équations 2.7 et 2.8). Dans le cas d'une quantification symétrique, un choix simple de facteur d'échelle est :

$$s = \frac{U - L}{2^n - 1} \quad (2.9)$$

Cette équation permet de répartir les valeurs quantifiées sur toute la plage disponible. $[U, L]$ représente la plage de coupure des valeurs réelles. Dans le cas d'une quantification symétrique, les valeurs réelles $|x|$ supérieures à $U - L$ débordent alors de la plage quantifiée $[-2^{n-1}, 2^{n-1} - 1]$ et sont saturées par la fonction $clip(\cdot)$ (équation 2.7), avec n le nombre de bits utilisés pour représenter les valeurs quantifiées. les valeurs U et L peuvent être déterminées de différentes façons, une méthode simple consiste à fixer la plage de coupure pour intégrer toutes les valeurs de x de façon symétrique, en utilisant $U - L = 2 \times \max\{|x|\}$. La quantification symétrique sur des valeurs signées est généralement utilisée sur les poids et présente l'avantage de pouvoir fixer la variable z de l'équation 2.7 et 2.8 à zéro ce qui réduit la quantité de calculs à effectuer [56]. Dans le cas de l'utilisation d'une fonction d'activation ReLU, les activations sont quantifiées sur des valeurs non signées en utilisant le facteur d'échelle défini à l'équation 2.9 avec $U - L = \max\{x\}$.

D'autres méthodes ont depuis été proposées. Elles offrent de meilleurs résultats, en tenant, par exemple, compte de la distribution statistique des paramètres, tel que proposé par SAWB [25] pour la quantification des poids. Pour les activations, PACT [25] propose une méthode permettant de minimiser l'erreur quadratique moyenne (MSE) due à la quantification en

entraînant un paramètre α tenant compte de l’erreur induite par la quantification ainsi que par la saturation des valeurs dans le cas de l’utilisation d’une fonction d’activation ReLU. Jusqu’ici, nous avons exclusivement traité de la quantification uniforme. Cependant, il est possible d’utiliser une quantification non-uniforme, dont le pas entre les valeurs quantifiées est variable [57–60]. Elle améliore la capture de l’information lors du processus de quantification, permettant aux modèles quantifiés de présenter une précision accrue par rapport à une quantification uniforme. Toutefois, ces stratégies sont difficiles à déployer efficacement sur le matériel existant. Elles nécessitent plus d’accès en mémoire et requièrent généralement l’usage de calcul en virgule flottante. Par conséquent, la quantification uniforme demeure pour l’instant la méthode communément employée.

2.2.2 Consommation d’énergie et impact sur les performances

La quantification d’un modèle offre de multiples avantages. En réduisant la précision des paramètres, l’espace mémoire alloué aux modèles est fortement réduit.

TABLEAU 2.1 Taille de ResNet-18 [1] à différents niveaux de quantifications

Précision du modèle	Taille du modèle [MByte]	Facteur de réduction
Virgule flottante 32 bits	42.80	-
Entier 8 bits	10.87	3.94
Entier 2 bits	2.89	14.81
Entier 1 bits	1.45	29.52

Le Tableau 2.1 présente la taille de ResNet-18 pour différents niveaux de quantification. La taille du modèle diminue de façon proportionnelle avec la précision employée pour la représentation des poids. En outre, l’utilisation d’opérandes entiers de plus basse précision plutôt qu’une représentation en virgule flottante est directement lié à une réduction de la consommation [2] ainsi qu’à une augmentation de l’efficacité des instructions.

TABLEAU 2.2 Consommation d’énergie en picojoules [pJ] pour une multiplication et une addition en 45nm / 0.9V, tiré de [2]

Représentation	MUL [pJ]	ADD [pJ]
Virgule flottante 32 bits	3.7	0.9
Virgule flottante 16 bits	1.1	0.4
Entier 32 bits	3.1	0.1
Entier 8 bits	0.2	0.03

TABLEAU 2.3 Consommation d’énergie en picojoules [pJ] pour un accès en lecture mémoire sur 64 bits, tiré de [2]

Mémoire (taille)	Consommation d’une lecture
Cache (8kB)	10 pJ
Cache (32kB)	20 pJ
Cache (1MB)	100 pJ
DRAM	1.3-2.6 nJ

Le Tableau 2.2 présente la consommation d’énergie sur une technologie 45nm. La consommation d’énergie pour une instruction de multiplication et d’addition augmente avec la précision des opérandes, avec par exemple, une consommation $15.5\times$ plus importante entre une multiplication sur un entier 8 bits et 32 bits. De plus, les instructions sur des opérandes entiers nécessitent une énergie moins importante que les instructions en virgule flottante pour un même nombre de bits. L’utilisation d’une représentation sur des nombres entiers est donc plus efficace d’un point de vue énergétique que la représentation en virgule flottante.

En outre, la réduction de la taille du modèle permet une meilleure localité des données, dont un nombre plus important peut être stocké dans une mémoire cache. Le Tableau 2.3 présente l’énergie requise pour une lecture 64 bits dans différentes tailles de mémoire cache ou de mémoire à accès aléatoire dynamique (DRAM). Un accès en lecture d’une donnée en DRAM y est présenté comme 13 à $260\times$ plus demandant en énergie qu’une lecture dans une mémoire cache. Le processus de quantification apporte donc deux bénéfices majeurs du point de vue de la consommation énergétique. Il permet de réduire l’énergie allouée aux opérations ainsi que l’énergie allouée aux accès en mémoire.

En plus de la réduction de consommation énergétique, la réduction de la précision des paramètres permet de diminuer les besoins en bande passante de la mémoire, ce qui réduit le goulot d’étranglement et améliore l’efficacité des opérations. La quantification permet donc aussi d’augmenter de façon significative les performances des algorithmes de CNN, pendant la phase d’entraînement, mais surtout lors de la phase d’inférence. Aussi, la réduction de la précision des instructions (par exemple, de 32 bits à 8 bits) permet d’augmenter les capacités de parallélisme de certains matériels par la subdivision des chemins de données. Sur Ara [33,34], un processeur vectoriel RISC-V 64-bit, l’emploi d’instructions sur une granularité de 8 bits, permet de multiplier le nombre d’instructions exécutées en un cycle par 4 en comparaison à des instructions 32 bits.

2.2.3 Impact sur la précision

Lors de la quantification, une partie de l’information est perdue au moment de l’approximation induite par la réduction de précision. Toutefois, il a été empiriquement montré qu’un grand nombre de CNN entraînés sur des entiers 8 bits présentent une dégradation négligeable sur la précision. Certains modèles présentent même une meilleure précision que leur équivalent en virgule flottante 32 bits [24]. Ce phénomène s’explique par le bruit induit par le processus de quantification sur les paramètres du modèle. Ce bruit est à l’origine d’un effet de généralisation sur le modèle qui résulte en une précision accrue [26,61].

Le Tableau 2.4 présente différentes précisions de modèles quantifiés associées à leur précision

TABLEAU 2.4 Précision de différents modèles ResNet [1] et EfficientNet [3] sur ImageNet [4] et impact relatif de la quantification sur des entiers 8 bits (int8) après l’entraînement (PTQ) et intégrée à l’entraînement (QAT), tiré de [5]

Modèle	Précision (fp32) [%]	Impact de la quantification (int8) sur la précision	
		Après l’entraînement	Intégrée à l’entraînement
ResNet-50	75.05	-0.09	+0.07
ResNet-50v2	75.36	+0.12	+0.29
ResNet-101	76.47	-0.15	-0.21
ResNet-101v2	76.89	+0.05	+0.26
EfficientNet-B0	76.97	-5.26	-1.15
EfficientNet-B3	81.36	-2.48	-1.88

sur une représentation en virgule flottante 32bits. En introduisant une quantification sur des entiers 8 bits après que l’entraînement ait été effectué sur 32 bits (PTQ), nous pouvons observer une précision relativement similaire pour les modèles ResNet. Cependant, lorsque le processus de quantification est directement intégré à l’entraînement (QAT), une amélioration de la précision est visible. La précision des modèles EfficientNet est négativement impactée par la quantification. Nous pouvons tout de même noter que l’intégration du processus pendant l’entraînement permet de compenser partiellement la perte de précision.

Toutefois, plus le niveau de quantification augmente, plus le bruit induit est important. Le seuil à partir duquel le phénomène de généralisation du modèle s’inverse, c’est-à-dire que la quantification dégrade la précision, est relatif à chaque modèle. S’il est aujourd’hui accepté que la quantification intégrée durant l’entraînement sur 8 bits résulte en une précision similaire, voire accrue pour certains modèles et ensemble de données [24], la précision à des niveaux de quantification plus élevés, typiquement en utilisant des représentations inférieures à 8 bits, présente des résultats plus contrastés [26, 61].

2.2.4 Quantification inférieure à l’octet

Jusqu’à maintenant, nous avons détaillé le processus de quantification de modèles représentés en virgule flottante 32 bits à une représentation sur des entiers de 8 bits. Ces représentations offrent l’avantage d’être standardisées, les virgules flottantes dans la norme IEEE-754 [62], et les entiers 8 bits par l’utilisation de la représentation en complément à 2. Les instructions sur ces représentations sont supportées sur quasiment toutes les architectures RISC et CISC [31, 63], offrant ainsi la possibilité d’implémenter l’Algorithme 1 et ses variations de façon simple et efficiente. Toutefois, il est tout à fait possible de quantifier un modèle de CNN sur une

précision inférieure à l’octet pour tirer avantage d’une réduction encore plus importante de la taille du modèle. Un pan entier de la recherche actuelle sur l’optimisation des Réseaux de Neurones est orienté vers ce sujet.

TABLEAU 2.5 Comparaison de la précision [%] entre différentes méthodes de quantification de ResNet [1] et EfficientNet [3] sur 2 à 4 bits sur l’ensemble ImageNet [4]. $WxAy$ correspond à la précision des poids (W) et des activations (A). Les précisions de référence sont issues des résultats de LSQ [6] et LSQ+ [7]

Modèle	Précision FP32	LSQ (2020)			LSQ+ (2020)		
		W2A2	W3A3	W4A4	W2A2	W3A3	W4A4
ResNet-18	70.5	66.7	69.4	70.7	66.8	69.4	70.8
ResNet-50	76.9	71.6	73.4	73.7	-	-	-
EfficientNet-B0	76.1	43.5	67.5	71.9	49.1	69.9	73.8

Le Tableau 2.5 présente les précisions de modèles entraînés sur des niveaux de quantifications de 2 à 4 bits. LSQ [6] implémente une quantification uniforme pour les poids et activations similaire au processus détaillé dans la section 2.2 et propose de modifier le gradient dans la phase de propagation en arrière durant l’entraînement. LSQ+ [7] étend les capacités de LSQ en implémentant une quantification plus adaptée aux activations permettant de rendre la méthode plus flexible. Lors d’une quantification sur 2 bits, la dégradation de la précision est importante, particulièrement sur EfficientNet-B0, ce qui coïncide avec les résultats du Tableau 2.4. Toutefois, sur les modèles ResNet, la dégradation de précision est relativement faible par rapport aux modèles sur virgule flottante en 32bits, en plus d’une diminution importante de la taille des modèles. Néanmoins, les jeux d’instructions modernes, vectoriels ou scalaires, ne supportent pas nativement les instructions entre des opérandes inférieurs à l’octet. L’implémentation de ces modèles est alors sous-optimale sur de telles architectures.

2.3 Implémentation du produit de convolution 2D inférieur à 8 bits

Dans la section précédente, nous avons introduit les avantages de la quantification, permettant de réduire la taille et d’accélérer l’exécution de la phase d’inférence d’un CNN. Nous avons soulevé le problème de l’implémentation des algorithmes lors d’une quantification sur une précision inférieure à 8 bits. Dans cette section, nous détaillons les solutions matérielles et logicielles existantes afin de palier l’absence de prise en charge efficace d’opérations sur des données représentées sur moins de 8 bits.

2.3.1 Limitations

Historiquement, l’emploi de l’octet comme unité de calcul et de stockage dans les systèmes numériques est un héritage de la représentation des caractères [64, 65]. Bien que de nouveaux matériels tentent d’intégrer des instructions avec une précision inférieure à l’octet, comme le Nvidia T4 [66] avec un mode sur 4 bits, les systèmes numériques à usage général reposent principalement sur l’usage de la représentation 8 bits comme unité, tant pour le stockage que pour les instructions. De plus, les formats standards, points flottants ou entiers aussi supportés par ces systèmes, sont représentés sur des multiples de 8 bits [62, 67]. Cette limitation se retrouve dans les extensions vectorielles des jeux d’instructions RISC disponibles sur les processeurs vectoriels [31, 32], ce qui rend l’implémentation d’une opération de convolution sur une précision inférieure à 8 bits sous optimale sur ce type de matériel. Une implémentation de l’Algorithme 1, est donc naïvement bornée par la performance de son implémentation sur 8 bits. Toutefois, diverses méthodes ont été proposées afin de palier cette contrainte.

2.3.2 Accélération logicielle

Les méthodes présentées dans cette section offrent l’avantage de ne pas nécessiter de modifications du matériel, permettant de bénéficier d’une grande flexibilité quant à la cible matérielle utilisée. Elles sont cependant bien moins performantes que des solutions matérielles dédiées. Les solutions présentées dans cette section reposent sur la parallélisation des calculs d’un produit scalaire représenté par l’opérateur $\cdot$ comme illustré à l’équation 2.10 avec un exemple sur les vecteurs $w = \{w_0, w_1\}$ et $a = \{a_0, a_1\}$. En utilisant cette opération élémentaire, il est alors possible de construire des opérations tensorielles plus complexes comme la convolution 2D présentée dans la section 2.1.2. L’idée principale de ces méthodes est d’empaqueter l’information de plusieurs opérandes basse précision (< 8 bits) dans un ou plusieurs registres de taille standard (> 8 bits) afin d’effectuer plusieurs opérations en utilisant une unique instruction, ce qui permet de paralléliser une partie des calculs.

$$w \cdot a = \begin{bmatrix} a_0 & a_1 \end{bmatrix} \times \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} = \begin{bmatrix} a_0 w_0 + a_1 w_1 \end{bmatrix} \quad (2.10)$$

Opérations binaires séquentielle

Les opérations binaires séquentielles [68, 69] offrent une solution permettant d’effectuer efficacement un produit scalaire entre des opérandes en parallélisant une partie des opérations. Lors du calcul du produit scalaire, chaque bit des opérandes est traité séquentiellement en utilisant un unique emplacement d’un registre pouvant contenir X bits. Le reste du registre

est alors rempli avec $X - 1$ autres bits de même poids appartenant à d'autres opérandes afin d'effectuer X multiplications binaires en parallèle avec une seule instruction logique ET. Les X bits du registre résultants sont ensuite sommés entre eux puis le résultat est décalé afin de correspondre au poids original pour compléter la somme partielle précédente.

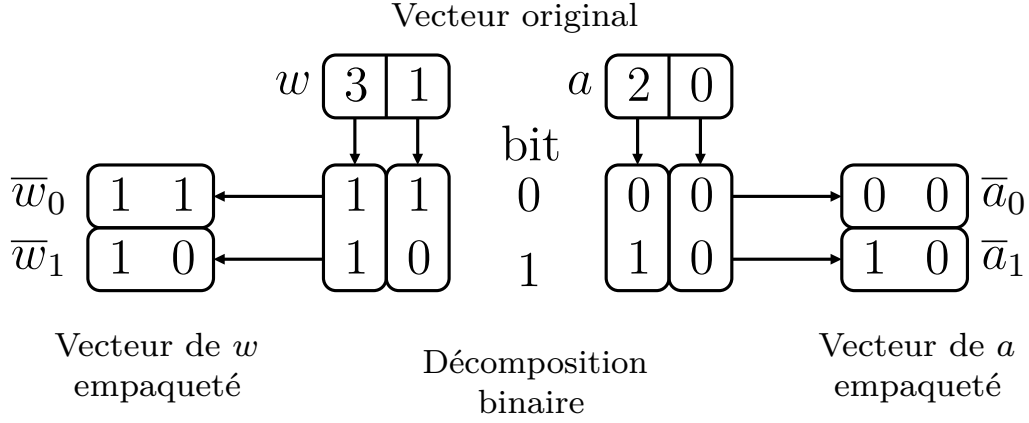


FIGURE 2.3 Exemple d'empaquetage de deux vecteurs $\{w_0, w_1\} = \{3, 1\}$ et $\{a_0, a_1\} = \{2, 0\}$ pour effectuer une opérations séquentielles binaires sur une précision de 2 bits. Les vecteurs empaquetés sont $\bar{w}_0 = \{3\}$, $\bar{w}_1 = \{2\}$, $\bar{a}_0 = \{0\}$ et $\bar{a}_1 = \{2\}$

Afin de réaliser l'opération présentée à l'équation 2.10, la première étape consiste à emballer les opérandes afin de réaliser les multiplications entre paires d'opérandes de manière binaire en parallèle avec une instruction logique. La Figure 2.3 illustre l'empaquetage de deux vecteurs contenant deux opérandes encodés sur 2 bits en adéquation avec l'équation 2.10. Les vecteurs $\bar{w}_i$ et $\bar{a}_i$ contiennent les bits de mêmes poids des valeurs respectives contenues dans w et a .

$$w \cdot a = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} 2^{i+j} \times \text{popcount}(\bar{w}_i \& \bar{a}_j) \quad (2.11)$$

L'équation 2.11 présente l'opération générale permettant de calculer le produit scalaire entre les vecteurs w et a avec M et N , leur précision respective. La fonction $\text{popcount}(\cdot)$ retourne la quantité de bits à l'état haut d'un élément avant que le résultat partiel soit décalé pour faire correspondre son poids.

Néanmoins, cette méthode présente plusieurs inconvénients, l'empaquetage des bits requiert une quantité importante d'opérations de décalage et masquage. Chaque bit doit être décalé pour être placé au bon index du registre contenant les bits de même poids. Aussi, bien que l'opération de multiplication binaire (réalisée par un ET logique dans l'équation 2.11) permet d'effectuer plusieurs produits binaires en parallèle, le débit reste fortement atténué par les manipulations nécessaires pour effectuer l'opération de popcount . Cette instruction n'étant

pas supportée nativement par les jeux d'instructions vectoriels, la modification du matériel permettant de l'ajouter afin d'obtenir de bonnes performances est nécessaire. Enfin, le nombre d'opérations à réaliser est proportionnel au produit des précisions $M \times N$ des opérandes, ce qui a pour conséquence de faire rapidement chuter les performances à mesure que la précision des opérandes augmente. Ces inconvénients permettent à la méthode par opérations binaires séquentielles, de n'être plus compétitive en performance qu'une implémentation équivalente sur 8 bits sur une plage de précision particulièrement réduite, typiquement sur des opérandes représentés entre 1 et 2 bits [9, 68].

Accélération de calcul très basse précision via un empaquetage efficient

Plus récemment, une méthode logicielle orientée vers l'accélération de calcul très basse précision via un empaquetage efficient (ULPPACK) [9] a été proposée. Cette méthode introduit un schéma d'empaquetage bien moins complexe que pour les opérations binaires séquentielles détaillées dans la section 2.3.2. Étendant ce travail, FULLPACK [70] introduit une stratégie de compression permettant un stockage optimal des données, réduisant le stress sur la bande passante de la mémoire et augmentant les performances de l'opération de convolution en utilisant ULPPACK. Bien qu'elle soit moins performante que les opérations binaires séquentielles dans le cas d'une précision binaire (W1A1), elle offre un gain substantiel de la performance des opérations sur des opérandes dont la précision est comprise entre 1 et 4 bits, permettant une plus grande flexibilité.

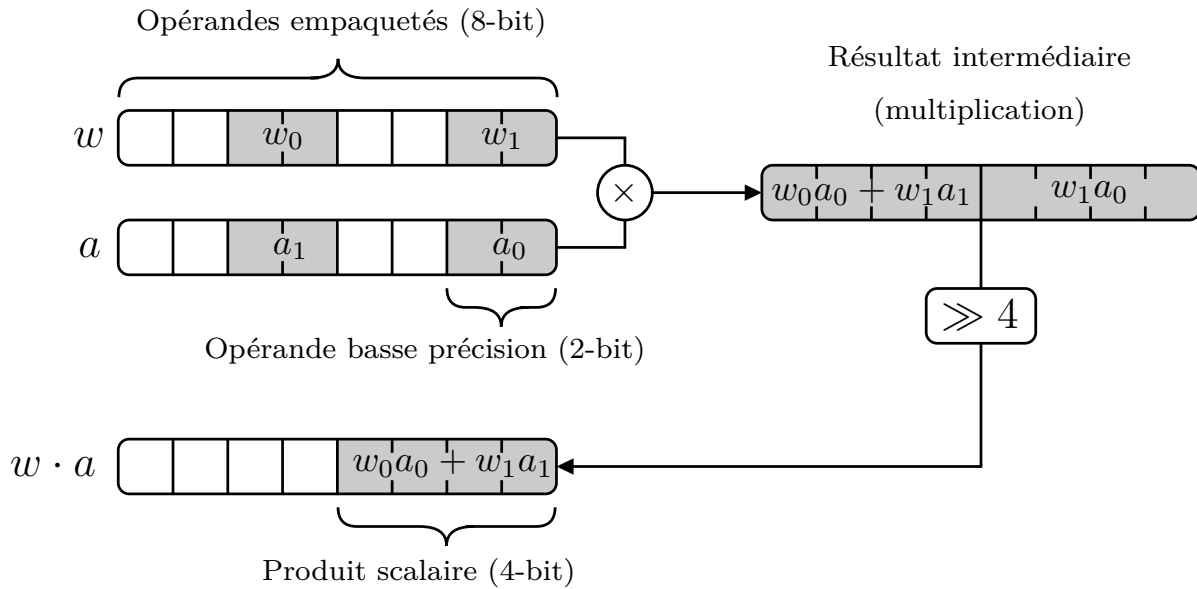


FIGURE 2.4 Exemple du produit scalaire de deux vecteurs $w = \{w_0, w_1\}$ et $a = \{a_0, a_1\}$ pour effectuer une opération bit-sérielle sur une précision de 2 bits

L'idée de cette méthode est de calculer un produit scalaire en empaquetant plusieurs opérandes inférieurs à 8 bits dans un registre de plus grande dimension (typiquement 8 ou 16 bits). En utilisant l'exemple de produit scalaire de l'équation 2.10, la Figure 2.4 montre les étapes à suivre pour réaliser un produit scalaire entre deux vecteurs w et a dont les valeurs sont représentées sur 2 bits. Elles sont d'abord empaquetées dans deux registres 8 bits par index opposés, qui sont ensuite multipliés. Le résultat est ensuite décalé afin de retrouver le résultat du produit scalaire contenu sur les 4 bits de poids fort. Mathématiquement, l'opération de multiplication entre les opérandes empaquetés sur 8 bits est décrite par :

$$(a_0 + 2^4 a_1) \times (w_1 + 2^4 w_0) = \underbrace{2^8 w_0 a_1}_{\text{débordement}} + 2^4 (\underbrace{w_0 a_0 + w_1 a_1}_{\text{produit scalaire}}) + \underbrace{w_1 a_0}_{\text{résultat secondaire}} \quad (2.12)$$

TABLEAU 2.6 Accélération moyenne du débit d'images traitées sur ResNet-18 et ResNet-50 [1] à l'aide de QNNPACK [8] et ULPPACK [9] sur différents ensembles de tests par rapport au modèle virgule flottante 32 bits sur un Raspberry Pi 4, tiré de [9]

Modèle*	QNNPACK	ULPPACK		
	W8A8	W4A4	W3A3	W2A2
ResNet-18	1.5×	2.0×	2.3×	2.3×
ResNet-50	1.7×	2.3×	2.6×	2.7×

*La première et dernière couches ne sont pas quantifiées en suivant [25].

Le Tableau 2.6 présente l'accélération de la performance sur des modèles quantifiés ResNet [1] par rapport aux équivalents en virgule flottante 32 bits sur une architecture RISC ARM vectorielle avec un unique thread. QNNPACK [8] est une bibliothèque de fonctions décrite en assembleur, intégrées dans PyTorch et développée par Facebook pour les architectures ARM et x86 vectoriels. Elle est utilisée comme référence pour la précision de 8 bits. Les résultats indiquent que l'utilisation d'ULPPACK sur une plage de précision variant de 2 à 4 bits résulte en une accélération comprise entre 2× et 2.7× du débit des modèles en comparaison du modèle en virgule flottante 32 bits. En outre, elle offre une accélération significative par rapport à l'utilisation de QNNPACK sur les mêmes modèles pour une quantification des poids et des activations sur des entiers 8 bits.

2.4 Matériel cible

Dans les sections précédentes, nous avons défini l'opération de convolution ainsi que les algorithmes généralement employés. Nous avons aussi démontré l'intérêt de quantifier des

modèles de Réseaux de Neurones Convolutifs, et par conséquent l'opération de convolution, sur des précisions inférieures à l'octet. Dans ce contexte, il existe une multitude de matériels à usage général et spécialisé permettant d'implémenter des CNN, en insistant sur l'exécution de l'opération de convolution.

2.4.1 Processeurs vectoriels à usage général

Les processeurs vectoriels à usage général (GPVP) sont un sous type de processeurs parallèles embarquant une architecture conçu pour effectuer des instructions vectorielles efficaces et efficaces. Les premières architectures vectorielles proviennent des années 1970 avec le superordinateur Cray-1 [71] mais la diminution du coût et l'augmentation des performances des micro-processeurs ont poussé à leur déclin dans les années 1990, notamment grâce à l'émergence de processeurs super-scalaires et de processeurs à mots d'instruction très longs (VLIW). Néanmoins, la réduction de la taille des transistors suivant la loi de Moore [72] ainsi que la mise à l'échelle de Dennard [73] étant de plus en plus difficile à suivre pour les fabricants de semi-conducteurs, les dix dernières années ont vu le retour d'architectures vectorielles sur des processeurs RISC. Ces processeurs offrent des performances accrues pour une consommation d'énergie optimale grâce à des micro-architectures parallèles efficaces.

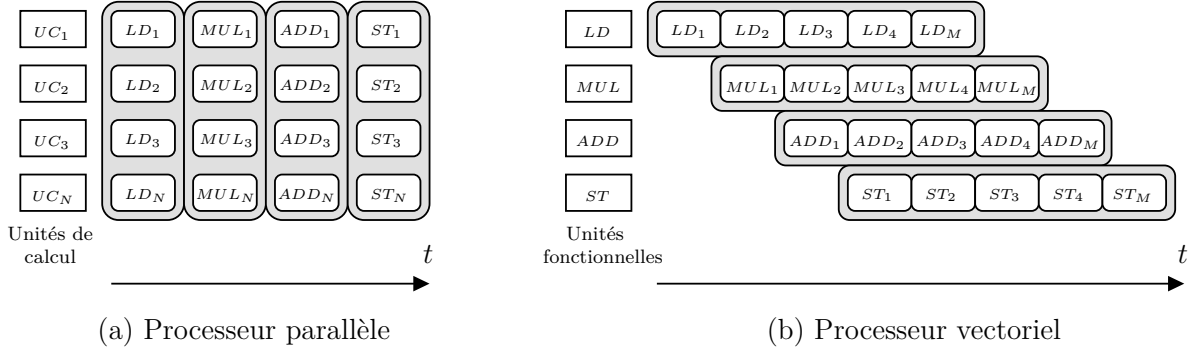


FIGURE 2.5 Exemple d'exécution de diverses instructions sur une architecture parallèle (a) et vectoriel (b). N et M correspondent respectivement au nombre d'unités de calculs et à la longueur des vecteurs à traiter

ARM, grandement inspiré par Cray-1, a lancé sur le marché son extension de jeu d'instruction vectoriel configurable (SVE) [31] en 2016. Le premier système adoptant cette extension est le A64FX de Fujitsu (2019) avec une performance maximale de 2.7 TFLOP/s sur une technologie 7nm pour une surface d'environ 400mm^2 [74]. À titre d'exemple, la GPU Nvidia K80 (2014) en 28nm revendique 2.8 TFLOP/s pour une surface de 561mm^2 [75], ce qui correspond à une performance et une surface relativement équivalente, malgré un écart de technologie

important. Le jeu d'instruction libre RISC-V a, lui aussi, proposé une extension vectorielle [32] qui continue d'être développée et améliorée. Étant en sources libres, de nombreux travaux de recherche proposent l'introduction de nouvelles micro-architectures permettant d'utiliser l'extension RISC-V "V" [33–35]. Ces architectures se placent à mi-chemin entre les CPU et les GPU en proposant des performances plus importantes que les GPP ainsi qu'une consommation d'énergie réduite par rapport à des GPU.

En utilisant des instructions simples sur plusieurs données (SIMD), une unique instruction spécifie à plusieurs unités de calcul l'instruction devant être opérée. Cette stratégie permet de diminuer le nombre d'instructions à décoder pour contrôler chaque unité au détriment de la flexibilité, car toutes les unités effectuent dorénavant la même instruction sur des données différentes. Ce processus est particulièrement efficace dans le contexte d'opérations vectorielles où un grand nombre d'opérations identiques et indépendantes peuvent être parallélisées.

Les processeurs vectoriels cascaden les différentes instructions en suivant l'exemple de la Figure 2.5. Plusieurs unités fonctionnelles spécialisées (unités de lecture/écriture, d'opération de multiplication, d'opération arithmétiques et logiques et d'autres), peuvent exécuter des micro-opérations séquentiellement, jusqu'à avoir traité toutes les données spécifiées. Contrairement à une architecture parallèle où chaque unité de calcul réalise la même instruction au même instant, parallélisant les calculs, l'architecture vectorielle parallélise l'utilisation des unités spécialisées. Elle permet donc d'effectuer un grand nombre de calculs en parallèle sur une quantité arbitraire de micro-opérations, contrairement à un processeur parallèle dont la quantité de micro-opérations simultanée est limitée par le nombre d'unités de calcul. Ce processus permet aux architectures vectorielles d'être agnostiques quant à la taille des vecteurs à traiter, ce qui les rend plus flexibles et adaptables.

2.4.2 Accélérateurs FPGA et ASIC

En utilisant des jeux d'instructions sur des GPVP, il est simple de se concentrer sur les techniques d'accélération logicielles indépendantes du matériel, offrant une grande flexibilité pour les utilisateurs. Néanmoins, les méthodes d'accélération les plus efficaces tirent parti d'une connaissance poussée du matériel permettant une optimisation de la partie logicielle afin qu'elles prennent les contraintes et les avantages du matériel en considération. Dans ce contexte, l'emploi de Circuits Intégrés pour Application Spécifique (ASIC) et de Circuits Logiques Programmables (FPGA) permettent d'intégrer des circuits spécifiques à l'accélération de certaines applications. Ils sont notamment utilisés dans le cadre de l'accélération des calculs des CNN. Ces circuits ont deux objectifs principaux :

L'accélération du débit de traitement des modèles par l'implantation de matériel dédié à certaines opérations et une capacité de parallélisation importante.

La réduction de la consommation d'énergie par le développement de matériel conçu spécifiquement pour cette tâche.

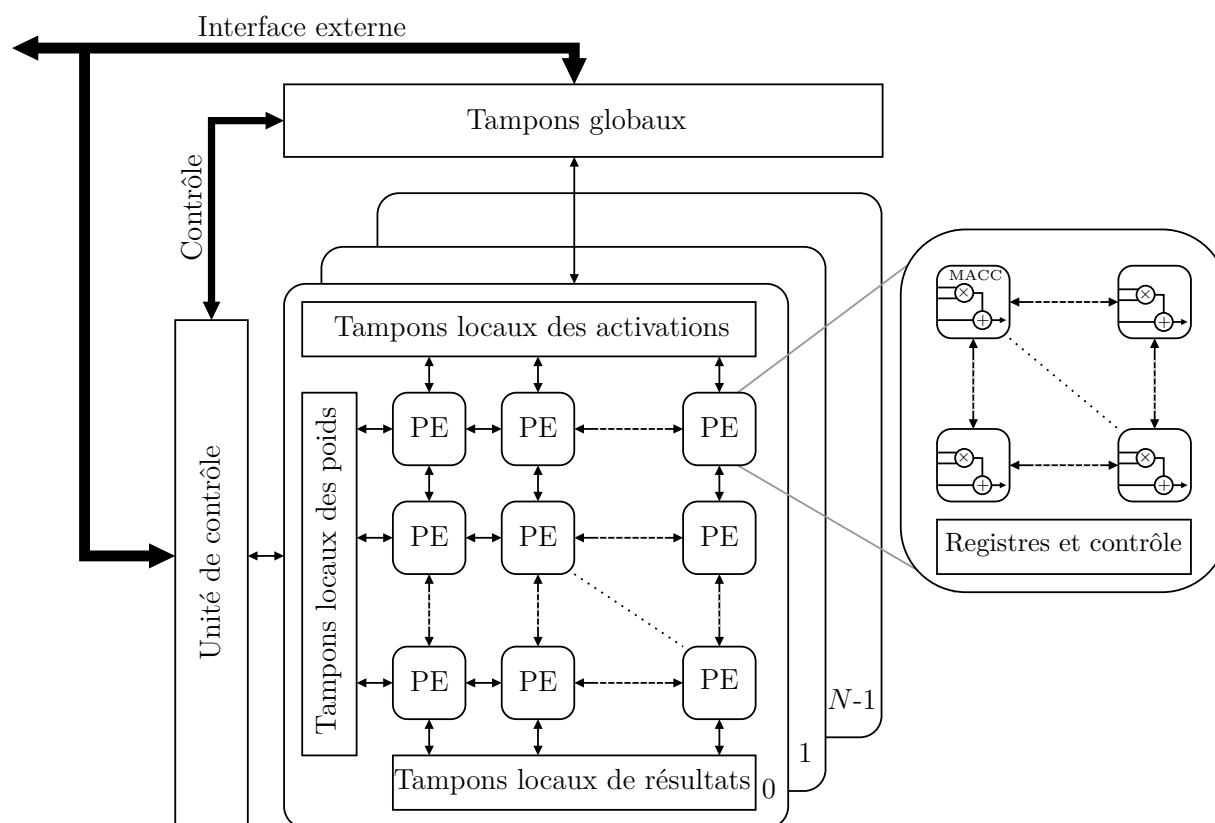


FIGURE 2.6 Diagramme simplifié d'un accélérateur générique de Réseau de Neurons Profond (DNN) avec N unités de multiplication matricielle. Chaque unité est composée d'une grille d'unités de calculs (PE) chacune embarquant un réseau systolique de cellules de multiplication-accumulation (MAC). Les divers niveaux de tampons permettent d'augmenter la localité des données et sont contrôlés par l'unité de contrôle

Les premiers accélérateurs sur ASIC sont apparus dans les années 1990, mais ont été rapidement mis de côté avec la chute en popularité des Réseaux de Neurons. Avec l'introduction d'AlexNet en 2012, l'intérêt pour le domaine des NN fut relancé. Néanmoins, ces nouveaux modèles nécessitent une quantité importante d'opérations élémentaires et répétitives à effectuer. Cette problématique a rapidement relancé les travaux de recherches vers le développement de circuits d'accélération afin de réduire les contraintes temporelles de l'exécution de NN et CNN. Aujourd'hui, toutes les entreprises de technologies et de semi-conducteurs, telles que Google,

Apple, Meta, Intel, Nvidia ou encore ARM, ont massivement investi dans la conception et le développement de circuits de traitement spécifiques dédiés à l'accélération d'opérations de Réseaux de Neurones.

Ces accélérateurs embarquent généralement des Unités de Multiplication Matricielle (MMU) composée d'une grille d'Unités de Calcul (PE) utilisées pour les produits entre les poids et activations. Chaque unité est composée d'un réseau systolique de cellules de multiplication-accumulation (MAC). La Figure 2.6 donne à voir un exemple simplifié d'accélérateur générique pour Réseau de Neurones Profonds (DNN). À chaque cycle, chaque PE multiplie ses entrées entre elles puis les additionne avec la somme partielle précédente. Cette somme est ensuite propagée à la prochaine PE afin d'y effectuer les produits et accumulations suivantes. Ceci permet d'effectuer une multiplication matricielle en une seule instruction en faisant appel à l'accélérateur. En comparaison avec des GPU, possédant un grand nombre d'unités de calcul dont chacune est constituée d'un processeur compact et spécialisé avec une mémoire partagée, l'utilisation d'unités composées seulement d'un circuit de MAC permet d'augmenter la densité et les performances des circuits. La surface du circuit accélérateur est approximativement proportionnelle au nombre d'unités de MAC et le coût de chaque MAC peut être minimisé en réduisant la précision en bits des opérandes, ce qui permet de réduire la consommation ainsi que la surface nécessaire. En se basant sur cette architecture, l'Unité de Traitement Tensoriel (TPU) 28nm de Google [76] cadencée à 700MHz présentée en 2016 revendique un débit de 92 TOP/s sur des entiers 8 bits pour une enveloppe thermique de 75W et une surface de 331mm² [75]. La GPU Nvidia K80 [77], le concurrent cadencé à 560MHz au même moment en 28nm, revendique 2.8 TFLOP/s sur virgule flottante 32 bits pour une enveloppe thermique de 150W et une surface de 561mm² [75]. Malgré la différence des types de données, le TPU de Google embarquant le réseau systolique offre une meilleure performance par watt que son concurrent pour un encombrement inférieur. Aujourd'hui, cet écart tend à se réduire avec l'introduction de GPU orientés vers des applications d'apprentissage machine comme la Nvidia T4 [66].

Si les implémentations utilisant un flux de conception ASIC permettent d'obtenir les meilleures performances par watt, leur coût ainsi que l'absence de reprogrammation possible les limitent à des applications à grande échelle. En comparaison, les FPGA constituent une alternative proposant une solution bien moins performante, mais reprogrammable et plus accessible financièrement. Ils permettent de réduire la consommation d'énergie par rapport à des GPU et peuvent être facilement adaptés à des tailles de données non standards, par exemple inférieures à 8 bits. En outre, divers outils comme fpgaConvNet [78] ou FINN [79] sont disponibles pour la génération automatisée de circuits permettant de réaliser l'inférence de modèles de CNN haut niveau optimisés à la cible FPGA.

CHAPITRE 3 DÉMARCHE DE RECHERCHE ET ORGANISATION

Dans les sections précédentes, nous avons défini l’opération de convolution dans le contexte des Réseaux de Neurones Convolutifs ainsi que divers algorithmes performants permettant de la réaliser. Nous avons aussi présenté les bénéfices liés à la quantification des modèles de CNN et la problématique d’implémentation sur le matériel moderne lors d’un processus de quantification inférieur à 8 bits. L’avènement d’architectures embarquant des extensions vectorielles permet de profiter d’une performance accrue par rapport à un CPU tout en maintenant un encombrement moindre que des GPU. Dans ce contexte, de nouvelles micro-architectures supportant ces extensions vectorielles sont développées, notamment sur des processeurs RISC. Un grand nombre de travaux s’orientent sur l’architecture ARM embarquant l’extension SVE dont la version fonctionnelle est disponible sur de multiples processeurs et sur certains systèmes populaires comme les Raspberry Pi. Toutefois, l’extension vectorielle RISC-V, bien que moins mature que ARM, est en sources libres, ce qui permet d’effectuer des modifications de manière simple sur la micro-architecture d’un processeur RISC-V. Par exemple, Ara [33, 34] est un processeur RISC-V supportant l’extension vectorielle basée sur le processeur scalaire CVA6 [80]. En plus d’afficher des performances élevées, sa description matérielle en sources libres lui permet d’être pleinement modifiable et c’est sur cette plateforme que ce travail s’oriente.

Dans un premier temps, nous utilisons Ara pour implémenter des algorithmes efficaces de convolution 2D sur des opérandes standards (8, 16, 32 et 64 bits). Puis, nous implémentons des algorithmes efficaces de convolution 2D sur des opérandes inférieurs à l’octet en utilisant ULPPACK [9], afin d’accélérer l’exécution des couches de convolution dans un contexte d’inférence de modèle de Réseaux de Neurones Convolutifs. Les programmes sont alors développés et vérifiés sur Ara. Ces algorithmes implémentés en langage assembleur intégrés directement dans un code C sont basés sur une implémentation en virgule flottante 64 bits en sources libres fournis avec Ara qui bénéficient d’une utilisation des voies particulièrement élevée (> 90%) [34] correspondant à une optimisation importante de l’exécution du programme.

Dans un second temps, nous cherchons à modifier Ara afin de produire des processeurs conçus pour l’inférence de Réseaux de Neurones Convolutifs Quantifiés. L’objectif est de réduire la surface et la consommation énergétique, tout en augmentant les performances des couches de convolution. Dans ce contexte, le chapitre 4 présente les performances de ULPPACK [9] introduit dans la section 2.3.2. Sparq, un processeur RISC-V vectoriel basé sur une version modifiée d’Ara, est présenté sous la forme d’un article de conférence publié à NEWCAS 2023.

Dans le chapitre 5, nous proposons une seconde version de Sparq. Présentant des performances identiques à Sparq mais plus flexible grâce à l'utilisation d'un circuit de décalage configurable, SparqV2 est aussi orienté vers l'exécution efficiente d'opérations sur des opérandes inférieurs à 8 bits en employant ULPPACK. Dans ce même chapitre, nous présentons Quark [11], un autre processeur basé sur Ara dans le cadre du même projet et présenté sous la forme d'un article de conférence publié à ISCAS 2023. Contrairement à Sparq(V2), Quark [11] utilise la méthode d'opérations binaires séquentielles pour accélérer les calculs convolutifs, proposant des performances particulièrement élevées pour des opérandes dont la précision est inférieure à 2 bits. Nous analysons les performances de SparqV2 et de Quark ainsi que l'impact des modifications de leur micro-architecture sur leur implémentation physique dans la technologie GLOBALFOUNDRIES 22FDX FD-SOI.

Enfin, dans le chapitre 6, nous comparons les performances de ULPPACK implémentée sur Ara avec l'implémentation de cette méthode sur une architecture ARM, sur laquelle cette méthode est originellement implémentée. Nous comparons ensuite les performances sur différentes couches de convolutions de CNN sur Ara, Sparq(V2) et Quark. L'objectif de ce mémoire est d'évaluer les performances ainsi que les avantages et inconvénients de tous les processeurs proposés dans un contexte de quantifications de modèles en deçà de 8 bit.

CHAPITRE 4 ARTICLE 1 : SPARQ : A CUSTOM RISC-V VECTOR PROCESSOR FOR EFFICIENT SUB-BYTE QUANTIZED INFERENCE

4.1 Détails de l'article

Sparq : A Custom RISC-V Vector Processor for Efficient Sub-Byte Quantized Inference

Par **Théo Dupuis**, Yoan Fournier, MohammadHossein AskariHemmat, Nizar El Zarif, François Leduc-Primeau, Jean Pierre David, Yvon Savaria.

Accepté à la 21^{ème} Conférence IEEE Interrégionale NEWCAS, le 11 avril 2023.

4.2 Contexte

Cet article présente un processeur vectoriel conçu pour l'exécution efficace d'opérations présentes dans les Réseaux de Neurones Convolutifs, nommé Sparq pour “**S**ub-byte vector **P**rocessor designed for the **A**cceler**R**ation of **Q**NN inference”. Ce travail s'inscrit dans une collaboration entre OpenHardware Group, CMC Microsystems, ETH Zürich et Polytechnique Montréal dans le cadre du projet “RISC-V Vector Processor for High-throughput Multidimensional Sensor Data Processing and Machine Learning”. Dans ce contexte, le groupe de travail de Polytechnique Montréal propose aussi des modifications matérielles sur Ara, le processeur vectoriel RISC-V 64-bit développé par l'ETH de Zürich.

La première étape est le développement en assembleur d'une bibliothèque d'opérations de convolution 2D permettant d'offrir des fonctions de convolution tirant parti de l'architecture vectorielle d'Ara sur tous les types de données (entier 8, 16, 32 et 64 bits et virgule flottante 32 et 64 bits). Une fois ces fonctions rendues disponibles, nous nous sommes intéressés à l'accélération logicielle d'opérations basse précision (<8 bits) en utilisant diverses méthodes, notamment ULPPACK [9] et les opérations binaires séquentielles [68]. Nous proposons ensuite plusieurs modifications avec pour objectif d'accélérer matériellement l'exécution de ces opérations, en plus de vérifier leur fonctionnement sur Ara. Dans ce contexte, une première version modifiée d'Ara nommé Quark [11] a été proposée, testée et publiée. Ce processeur

embarque des instructions spécialisées dédiées aux opérations binaires séquentielles et permet d’accélérer significativement les opérations de convolution sur des opérandes d’une précision comprise entre 1 et 2 bits. À la suite de cette publication, nous avons décidé de proposer une implémentation de la méthode ULPPACK sur Ara, avec et sans ajout d’instructions spécialisées. L’objectif est d’augmenter les performances des opérations de convolution 2D sur des opérandes inférieurs à 8 bits et de comparer les performances de ce processeur modifié nommé Sparq avec Quark.

Sparq propose l’ajout d’une instruction spécialisée très peu coûteuse du point de vue matérielle et énergétique afin de tirer parti d’une stratégie de calcul de très basse précision sur des matériels standards proposée récemment : ULPPACK [9]. L’ajout de cette instruction permet une accélération de $1.7\times$ à $3.2\times$ selon la précision des opérandes. Nous proposons aussi de retirer l’unité de calcul en virgule flottante, ce qui permet de réduire la surface d’une voie de 43.3% et la consommation énergétique de 58.8%.

4.3 Contributions

En se basant sur l’article décrivant l’implémentation d’ULPPACK [9], Théo Dupuis propose de modifier Ara [33, 34], un processeur RISC-V vectoriel 64-bit afin d’accélérer l’opération de convolution 2D. Pour ce faire, il propose l’ajout d’une instruction spécialisée de multiplication-décalage-accumulation (vmacsr) afin de réduire les contraintes sur les accumulations locales de la méthode ULPPACK. Il propose une implémentation logicielle avec et sans l’instruction spécialisée permettant de comparer les performances sur différentes précisions d’opérandes. Yoan Fournier effectue l’implémentation du processeur modifié sur la technologie GLOBALFOUNDRIES 22FDX FD-SOI. MohammadHossein AskariHemmat a fourni de précieux conseils concernant les modifications matérielles à apporter tout au long de ce travail. La supervision de François Leduc-Primeau, Jean Pierre David et Yvon Savaria en plus de leurs conseils et retours ont grandement contribué à la version définitive de cet article.

Sparq : A Custom RISC-V Vector Processor for Efficient Sub-Byte Quantized Inference

4.4 Abstract

Convolutional Neural Networks (CNNs) are used in a wide range of applications, with full-precision CNNs achieving high accuracy at the expense of portability. Recent progress in quantization techniques has demonstrated that sub-byte Quantized Neural Networks (QNNs) achieve comparable or superior accuracy while significantly reducing the computational cost and memory footprint. However, sub-byte computation on commodity hardware is sub-optimal due to the lack of support for such precision. In this paper, we introduce Sparq, a Sub-byte vector Processor designed for the Acceleration of QNN inference. This processor is based on a modified version of Ara, an open-source 64-bit RISC-V “V” compliant processor. Sparq is implemented in GLOBALFOUNDRIES 22FDX FD-SOI technology and extends the Instruction Set Architecture (ISA) by adding a new multiply-shift-accumulate instruction to improve sub-byte computation efficiency. The floating-point unit is also removed to minimize area and power usage. To demonstrate Sparq performance, we implement an ultra-low-precision (1-bit to 4-bit) vectorized conv2d operation taking advantage of the dedicated hardware. We show that Sparq can significantly accelerate sub-byte computations with respectively 3.2 times, and 1.7 times acceleration over an optimized 16-bit 2D convolution for 2-bit and 4-bit quantization.

Index Terms - RISC-V, Vector ISA, Sub-byte, Convolution

4.5 Introduction

Convolutional Neural Networks (CNNs) are used in a broad range of applications that include image processing, speech recognition, and natural language processing. As new CNN architectures and optimization methods allow for higher accuracy, the required amount of computations and on/off-chip memory accesses tends to increase. The high complexity and energy consumption pose a challenge for their deployment on resource-constrained devices such as low-power systems.

In the last few years, research has focused on minimizing computational cost and memory footprint while trying to preserve accuracy. This was achieved through the emergence of multiple new optimization techniques. Among them, reducing the bit-precision of weights and activations allows for a decrease in both the computational complexity and the memory

footprint. In extreme cases, where weights and activations are quantized down to 1-bit or 2-bit, the resulting models typically experience an accuracy drop of less than 10% compared to their full-precision counterparts [25, 81–84].

At the same time, new hardware solutions are proposed to address the issue of complexity through the emergence of vectorized Instruction Set Architectures (ISAs). Supporting these new instructions, vector processors can achieve high efficiency through parallelization of tensor computations while maintaining the flexibility of being fully programmable [31, 33, 34]. However, they are limited by the minimum granularity of their vector registers. Generally fixed at 8-bit, this limitation makes these architectures sub-optimal for sub-byte computing.

In this paper, we propose to modify Ara [34], a 64-bit RISC-V RVV1.0 compliant vector processor to add a new vectorized multiply-shift-accumulate instruction, `vmacsr`. In order to minimize area and power usage, we remove the vector floating-point unit (FPU). The main idea is to increase the performance of ultra-low-precision computations with a cost-effective instruction. To demonstrate the performance of Sparq, we implement several low-precision conv2d algorithms over a wide range of precisions, using the recently proposed ULPPACK [9] technique. We demonstrate that the new `vmacsr` instruction allows us to mitigate the constraints related to the ULPPACK technique and significantly improve performance. Using up to 2-bit quantization, we achieve a speedup of $3.2\times$ over an optimized 16-bit conv2d implementation and $1.7\times$ using up to 4-bit quantization. Lastly, we implement the processor in GLOBAL FOUNDRIES 22FDX FD-SOI technology and compare power and area usage between Sparq and Ara.

4.6 Background and Related Work

4.6.1 Sub-byte Convolutional Neural Networks

Allowing to address both computation complexity and memory footprint, quantization has aroused a real interest. In fact, quantizing properly can achieve considerable memory savings with limited accuracy drop. On Deep Neural Networks such as ResNeXt [85] and EfficientNet [3], only a low accuracy degradation was observed when quantizing the full-precision 32-bit weights and activations down to 8-bit [24]. In recent work, ultra-low-precision quantization (≤ 8 -bit), which we refer to as sub-byte quantization, has demonstrated that minimal accuracy degradation was achievable with new network structures and training strategies. In some cases sub-byte Quantized Neural Networks (QNNs) perform better than their full-precision counterparts [6, 10] as shown in Table 4.1.

This was made possible by the introduction of architectures and methods focused on ultra-

TABLEAU 4.1 Accuracy on Quantized Resnet18 Using LG-LSQ [10]

Dataset	Model	Precision (W/A)	Top-1	Top-5
ImageNet	Resnet18	LG-LSQ(3/3)	70.31	89.55
		LG-LSQ(4/4)	70.78	89.77
		FP32	69.76	89.08

low-precision. LSQ [6], SSG [10] and DSQ [86] propose to learn the quantization parameters for both weights and activations by minimizing the quantization loss of the network during training. SAWB [25] focuses on the weight quantization parameter using the weight distribution to estimate the optimal quantization scale. To address the issue of unbounded activation range after ReLU, PACT [25] proposes to train a clipping parameter to find the balance point between clipping and quantization error. Focusing on binary or ternary quantization, CNN architectures tailored for 1-bit or 2-bit such as XNORNet [82] or BinaryNet [83] have been introduced. While achieving very high efficiency and low memory footprint, their accuracy suffers a significant drop when compared to full-precision models. All these methods try to minimize the accuracy drop of sub-byte QNN by minimizing the error due to quantization on weights and activations.

4.6.2 Sub-byte Computation on Commodity Hardware

CNNs rely heavily on the use of the *conv2d* operation, which can make up to 90% of the computation [40, 41]. By using ultra-low-precision quantization, or specific binary/ternary architectures, the computational cost can be drastically reduced. Typically, general-purpose processors cannot take advantage of sub-byte operands since their ISA is suited for byte instructions at best. The same applies to their corresponding vector extension ISA whose granularity is typically limited to vectorized 8-bit instructions. Thus, naive implementations of sub-byte algorithms are bounded by the performance of the 8-bit implementation.

Several acceleration algorithms have been introduced to counterbalance this issue. For example, bit-serial computation allows each operand bit to be processed in a serial manner. As a result, the operation between any arbitrary N -bit and M -bit precision operand can be computed. However, the computation complexity is defined by $\mathcal{O}(N \times M)$, meaning that this method is only suitable for ultra-low-precision, typically less than 3-bit. For this precision range, the performance over fp32 can be multiplied by a factor of 2 to 6 depending on the precision [69]. This paved the way for the development of specialized bit-serial hardware [11, 87].

Other techniques, such as ULPPACK, offer a less invasive alternative while achieving higher performance than standard precision implementation. Even though the expected speedup is lower than with bit-serial, typically $2\times$ to $4\times$ acceleration over fp32 [9], this method covers a wider precision range, typically 1-bit to 4-bit, by utilizing densely packed operands with commodity Single Instruction, Multiple Data (SIMD) architectures. Although no specialized hardware is required, this algorithm can benefit from specialized instruction such as a multiply-shift-accumulate.

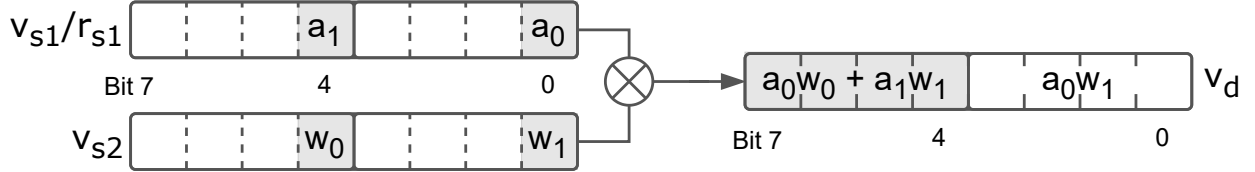


FIGURE 4.1 Multiplication between 2 packed 8-bit vectors with 1-bit precision on weights and activation. The dot product of the two packed elements is computed using a single multiplication on the MSB. The result then needs to be shifted to retrieve the dot product.

4.7 Software implementations

In this section, we provide an overview of different precision *conv2d* algorithms benchmarked on Sparq. Every implementation, except for the provided benchmarks, is handwritten using inline assembly, unrolled, and stored using a channel-first memory layout for the input, kernel, and output tensors.

4.7.1 Optimized Vector Conv2d

A wide range of benchmarks is provided with Ara¹, including a double-precision (DP) *conv2d* function. The DP implementation achieves high utilization of Ara’s lanes by using the available optimized vector slides to maximize efficiency through high data reuse. We implemented int16 and fp32 *conv2d* based on the structure of the convolution example to serve as the baseline for our comparisons. The choice of a dedicated convolution algorithm over an image to column (*im2col*) operation followed by a GEneral Matrix Multiplication (GEMM) technique is motivated by the reduction of the memory footprint induced by the *im2col* operation. Thus, fewer on/off-chip memory accesses are required to compute the output. We benchmarked the int16 *conv2d* on Sparq and the fp32 version on Ara. Our implementations of int16 and fp32 achieve a lane utilization of respectively 93.8% and 93.6% at $1 \times 32 \times 512 \times 512$ input size.

1. See <https://github.com/pulp-platform/ara>

4.7.2 ULPPACK

ULPPACK [9] is a software-only technique aiming towards acceleration of ultra-low-precision computation via an effective packing operand scheme. As illustrated in Figure 4.1 with an 8-bit register example, by packing multiple low-precision operands in wider registers, a single multiplication instruction completes the dot product between multiple operands and returns the result on the 4 most significant bits. Formally, the result can be expressed as follows :

$$(a_0 + 2^4 a_1) \times (w_1 + 2^4 w_0) = 2^8 a_1 w_0 + 2^4 (a_0 w_0 + a_1 w_1) + a_0 w_1$$

The dot product of packed vectors has to be extracted through a right logical shift (LSR) before accumulation. Local accumulations of the non-shifted result can be done to alleviate the number of shift operations, but are limited by the width of the low-precision result. Moreover, increasing the precision of weights and activations results in a reduction in the number of possible local accumulations due to overflow, implying a lower speedup. In the Figure 4.1 example, using 8-bit elements to compute the product of 2 packed operands limits the result on a 4-bit width. By using 1-bit precision on weights and activation, 8 local accumulations are possible without risking overflow. This method is an efficient way to compute vector-based algorithms since it computes the dot product between multiple low-precision operands with a unique higher precision multiplication, which significantly reduces the computational cost and resource usage.

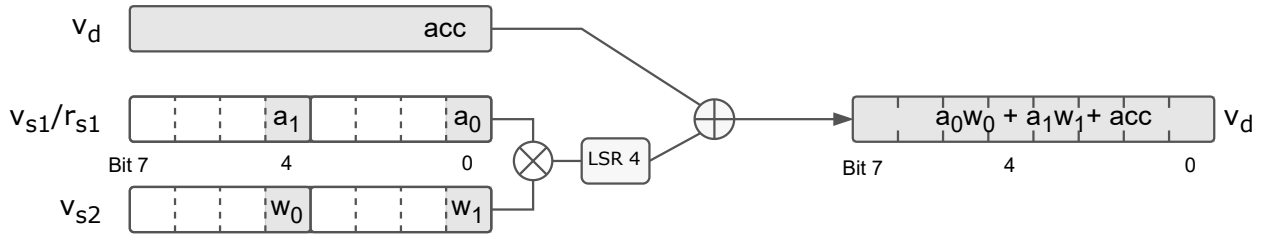


FIGURE 4.2 Diagram of the multiply-shift-accumulate operation on 8-bit packed registers with 1-bit precision for activations and weights. A shifter is inserted between the multiplication and the accumulation

4.8 Proposed Architecture

4.8.1 vmacsr Custom Instruction

To address the issue of local accumulation overflow described in Section 4.7.2, we propose to implement a multiply-shift-accumulate operation as a custom vector instruction. As depicted

in Figure 4.2, using the multiply-shift-accumulate operation, the low-precision multiplication is shifted and accumulated. This avoids the need for an intermediate register, a logical right shift, and an addition.

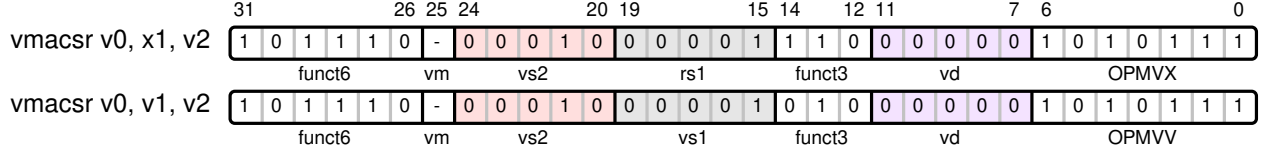


FIGURE 4.3 `vmacsr` encoding example

In this work, we mainly focus on a specific precision range following the condition $N + M \leq 7$ -bit, where N and M are the operand’s precision in bits. In this range, quantized models offer sufficient precision to achieve comparable or superior accuracy compared to full-precision models. We implement a *conv2d* with 16-bit packed register granularity to achieve high performance on this range. However, the use of multiply-shift-accumulate is applicable on higher precision operands and elements width, addressing the same overflow issue. To implement the new instruction, we modified Ara’s dispatcher to specify the `vmacsr` `funct6` encoding. As our instruction behaves very closely to `vmacc`, we use the free `funct6` encoding following `vmacc`’s `funct6` [32]. We implement `vmacsr` in both `OPMVV` and `OPMVX` format allowing vector-vector and vector-scalar instructions to be used. The encoding of `vmacsr` is described in Figure 4.3. It should be noted that for this work, only the vector-scalar `vmacsr` instructions is used. We then modified Ara’s SIMD multiplier to describe the expected behavior of the instruction, denoted as follows :

$$V_d \leftarrow V_d + ((V_{s1} \times V_{s2}) >> M)$$

where M is the shifted value on the resulting product before accumulation. In this work, we only pack two operands per register to reduce the overhead related to the packing operation carried out at runtime. Thus, M is hard-wired at half the granularity of the vector registers.

4.8.2 2D Convolution Algorithm

To our knowledge, the ULPPACK method has only been implemented on an ARM CPU using the ARM Neon intrinsic ISA extension [9]. In addition, by rearranging the input and filters with an *im2col* operation, the convolution is performed with a GEMM. However, Ara’s dedicated *conv2d* algorithm takes advantage of the `vslidedown` instruction provided by the RISC-V “V” ISA extension, which does not have any ARM equivalent. Thereby, the output

Algorithm 2 Proposed vector conv2d using `vmacsr`

```

 $H, C$       : Input height, channels
 $F_h, F_w$    : Kernel height and width
 $M$          : Packed operand per element
 $R, V$       : Scalar and vector registers (initialized to zero)
1: for  $h \leftarrow 1$  to  $H$  do
2:    $V_{F_h} \leftarrow 0$ 
3:   for  $c \leftarrow 1$  to  $\frac{C}{M}$  do
4:      $V_0 \leftarrow$  load one packed input row
5:     for  $i \leftarrow 1$  to  $F_w$  do
6:        $R_{[1:F_h]} \leftarrow$  load the  $i^{th}$  packed kernel column
7:       for  $j \leftarrow 1$  to  $F_h$  do
8:          $V_j \leftarrow \text{vmacsr}(R_j, V_0, V_j)$ 
9:          $V_0 \leftarrow \text{vslidedown}(V_0, 1)$ 
10:    if  $h \geq F_h$  then
11:       $O[h] \leftarrow V_1$  ▷ store one output row
12:      for  $j \leftarrow 1$  to  $F_h - 1$  do
13:         $V_j \leftarrow V_{j+1}$ 

```

stationary ULPPACK *conv2d* algorithm presented in Algorithm 2 is based on the *conv2d* described in Section 4.7.1. Although the operand packing operation is not detailed in the algorithm, it is carried out at runtime. We use the ULPPACK P1 packing [9] scheme to compute the contribution of M channels at each iteration, where M is the number of operands packed per register. Following the example in Figure 4.1, a_i and w_j respectively represent activation and weight values from the channel i and j .

Algorithm 2 presents a simplified ultra-low-precision *conv2d* implementation using the `vmacsr` instruction available on Sparq. For each loaded packed input row (line 4), the algorithm performs a vector multiply-shift-accumulate operation (line 8) to compute the partial result with the first packed kernel column. The input is then slided by one element to the left (line 9) to accommodate the next kernel column, and this operation is repeated until all the kernel columns and input channels have been processed. In order to complete and store the first output row (line 11), F_h packed input rows must be processed (line 10). The partial results contained in the following vector registers (line 13) are moved down to be completed one by one with each new packed input row.

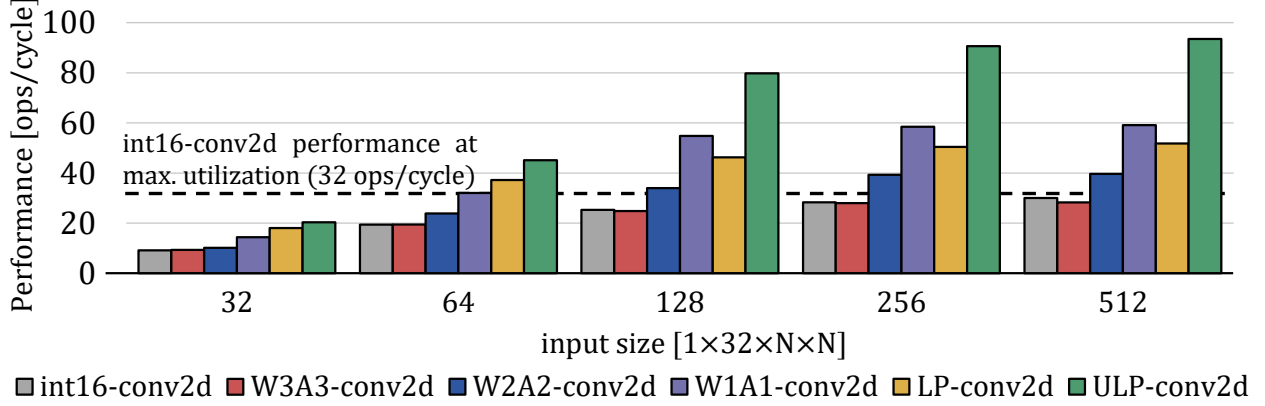


FIGURE 4.4 Performance comparison between several *conv2d* implementation using a 7×7 kernel size. Ultra-Low-Precision (ULP) and Low-Precision (LP) *conv2d* take advantage of the *vmacsr* instruction on Sparq while W1A1, W2A2, W3A3 *conv2d* run on native RISC-V “V” ISA

4.9 Results and Performance Analysis

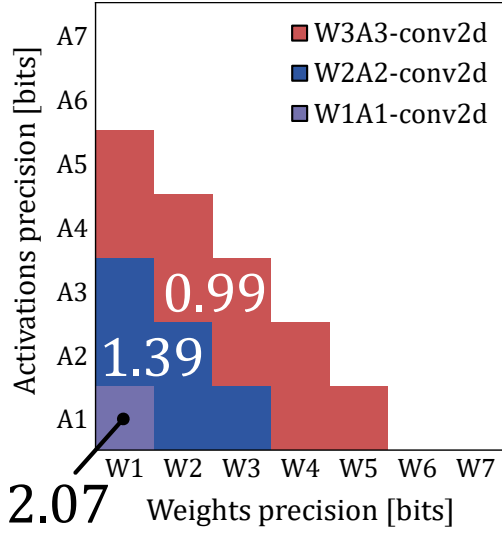
4.9.1 Performance Analysis

We compare the performance of the different *conv2d* implementations over several precisions for activations and weights using RTL simulations with a 4-lane configuration for both Ara and Sparq. Our measured execution time includes both activations and weights packing done at runtime. However, the overhead induced by weights packing could be avoided by offline preprocessing. Figure 4.4 presents the performance in operations per cycle over the different *conv2d* implementations.

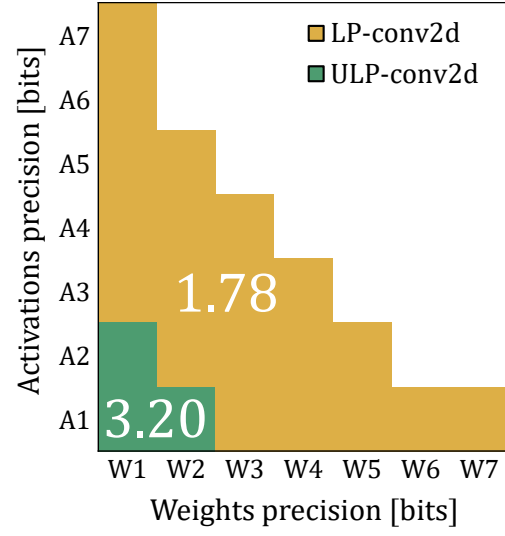
As expected, the performance of ULPPACK running on Ara shows an improvement over its 16-bit counterpart. Using 7×7 favors high data reuse, hence the small gap with the theoretical throughput. Moreover, using Sparq’s *vmacsr* instruction removes the constraint of local accumulation, offering two major benefits :

Performance increase resulting from the expected overall reduction in the number of instructions to compute the output matrix as presented in Figure 4.4.

Higher precision range, as shown in Figure 4.5, is obtained without modifying the algorithm. The range is only limited by the 8-bit wide dot product result when using 16-bit packed registers for Low-Precision (LP) and the 4-bit result for the Ultra-Low-Precision (ULP).

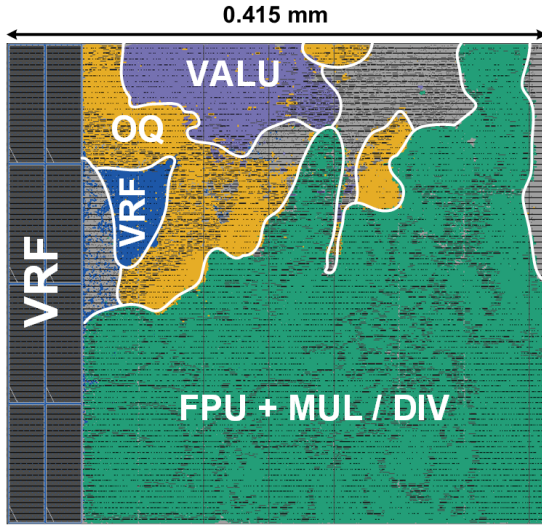


(a) Native implementation

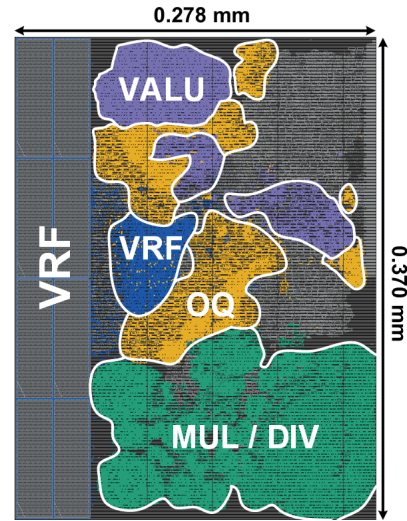


(b) vmacsr implementation

FIGURE 4.5 Relative speedup over the 16-bit *conv2d* implementation on the overflow-free precision region. The kernel size is 7×7 over a $32 \times 256 \times 256$ input. (a) for the native implementation benchmarked on Ara (b) for the accelerated version taking advantage of the *vmacsr* instruction benchmarked on Sparq



(a) Ara lane



(b) Sparq lane

FIGURE 4.6 Ara and Sparq lanes placed and routed designs. ■ is the vector register file, ■ is the operands queue, ■ is the vector fixed point multiplication and division units with or without the FPU, ■ is the vector ALU

TABLEAU 4.2 Physical Implementation of Ara and Sparq Lanes

	Ara Lane	Sparq Lane
Number of Lanes	4	4
VRF Size [KiB]	16	16
Lane Cell Area [mm ²]	0.120	0.068
Lane Core Frequency*[GHz]	1.346	1.464
Lane Power* [mW]	159.2	65.6

*At typical corner (TT/0.8V/25°C)

4.9.2 Physical Implementation

Since the `vmacsr` circuit is located in the vector engine, we only implemented one lane of Ara and Sparq using GLOBALFOUNDRIES 22FDX FD-SOI technology. For synthesis and back-end, we used the SYNOPSIS DESIGN COMPILER S-2021.06-SP5 and CADENCE INNOVUS v21.15, respectively. The physical implementation results of these lanes are summarized in Table 4.2. As anticipated, the Sparq lane exhibits significant improvements in terms of area (-43.3%) and power consumption (-58.8%) when compared to a standard Ara lane, primarily due to the FPU removal. Furthermore, the inclusion of the `vmacsr` instruction did not impact the typical corner frequency, indicating that it did not affect the critical path of the design. In fact, the removal of the FPU actually yielded an increase (+8.7%) in the maximum lane clock speed. It is important to note that in Ara, the critical path is primarily located in the VLSU and SLDU, both of which are part of the interconnection between lanes [34]. These interconnections were not implemented as part of this work since the addition of `vmacsr` would have had a negligible impact on those modules. Figure 4.6 shows the physical layout of both designs.

4.10 Conclusion

This paper introduced the extension of the RISC-V “V” ISA with a multiply-shift-accumulate custom instruction, allowing better performance of sub-byte computations. We presented Sparq, a FPU-free RISC-V vector processor supporting the multiply-shift-accumulate instruction. To demonstrate its improved performance, we implemented a *conv2d* algorithm for 1 to 4-bit precision operands. In this range, quantized models offer sufficient precision to achieve comparable or superior levels of accuracy compared to full-precision models. We showed that, with `vmacsr`, we achieved up to $1.7\times$ and $3.2\times$ speedup over an optimized 16-bit *conv2d*

depending on the precision. Lastly, the implementation reports of the modified version in GF22 nm showed that removing the FPU significantly reduced the area and power usage, while the newly added instruction had no impact on the critical path.

In future work, we plan to improve the flexibility of `vmacsr` using a runtime configurable shifter, as well as testing the proposed algorithm on an FPGA emulation of Sparq.

4.11 Acknowledgments

The authors thank CMC Microsystems and Global Foundries for access to design tools and technologies. This research was funded by CMC Microsystems, Mitacs and the OpenHW Group.

CHAPITRE 5 AUTRES ARCHITECTURES ET ANALYSE DES PERFORMANCES

5.1 Méthodologie

5.1.1 Compteur de cycles

Afin de mesurer les performances des différents algorithmes de convolution 2D sur les architectures proposées, nous utilisons un compteur de cycles défini dans les registres de statut et de contrôle (CSR) de la norme RISC-V [63]. Ce registre de 64 bits contient le nombre de cycles exécuté sur un cœur du processeur (ou *hart* dans la terminologie du jeu d'instruction RISC-V). Une instruction barrière (ou *fence* dans la terminologie du jeu d'instruction RISC-V) est utilisée avant de lire la valeur du CSR du compteur de cycle. Cette instruction impose une contrainte sur l'ordre d'exécution des instructions afin de s'assurer que toutes les écritures mémoires soient effectuées avant que la valeur du registre du compteur de cycle ne soit lue.

Les fonctions *start_timer()*, *stop_timer* et *int64_t get_timer()* sont définies dans un fichier d'en-têtes de l'environnement de compilation d'Ara nommé *runtime.h*. Ces fonctions permettent respectivement de copier l'opposé de la valeur courante du registre compteur de cycle dans une variable *timer*, d'ajouter la valeur courante de ce même registre à la variable *timer* et de retourner la valeur de la variable *timer*. La variable *timer* ainsi que son adresse sont définies directement lors de la compilation par le script de liens dans la mémoire cache d'Ara.

Ainsi, les appels aux fonctions permettant de contrôler la variable *timer* sont effectués avant et après l'appel à la fonction de convolution 2D de la manière suivante :

```
#include "runtime.h"

start_timer();    // timer -= compteur_cycle
conv2d(...);     // appel de fonction conv2d
stop_timer();     // timer = compteur_cycle

int64_t Nombre_de_cycles = get_timer(); // retourne timer
```

5.1.2 Calcul des performances

Afin de rendre compte du niveau d'optimisation d'une fonction, nous utilisons une métrique appelée utilisation dont l'unité est le pourcentage (%). La valeur d'utilisation correspond à la proportion de cycles utiles sur le nombre de cycles total. Un cycle est considéré comme utile s'il est employé à exécuter une opération nécessaire au calcul du résultat. Comme détaillé dans la section 2.1.2, une convolution 2D directe nécessite KCH_oW_oRS instructions MAC, soit $2KCH_oW_oRS$ opérations de multiplication et d'addition. Afin de déterminer le taux d'utilisation d'une implémentation, il est nécessaire d'obtenir sa performance en opérations par cycle. À l'aide du compteur de cycle défini précédemment, nous pouvons déterminer la performance d'une implémentation de convolution 2D en utilisant l'équation suivante :

$$\text{Performance} = \frac{\text{Nombre d'opérations}}{\text{Nombre de cycle}} = \frac{2KCH_oW_oRS}{\text{Nombre de cycles}} \quad [\text{ops/cycle}] \quad (5.1)$$

Afin de déterminer le taux d'utilisation d'une implémentation sur Ara, nous devons tenir compte du nombre de voies, ainsi que de la granularité des instructions. L'utilisation d'une implémentation est alors définie par :

$$\begin{aligned} \text{Utilisation} &= 100 \times \frac{\text{Performance} \times \text{Granularité des instructions}}{2 \times \text{Nombre de voies} \times \text{Largeur des chemins de données}} \\ &= 100 \times \frac{2KCH_oW_oRS \times \text{Granularité des instructions}}{2 \times \text{Nombre de voies} \times \text{Nombre de cycles} \times 64} \quad [\%] \end{aligned} \quad (5.2)$$

Le facteur 2 au dénominateur provient de la disponibilité d'une instruction MAC sur le jeu d'instruction RISC-V qui s'exécute en un seul cycle sur Ara. Il permet de compenser la prise en compte de deux opérations (addition et multiplication) dans le cas général. Par exemple, dans le cadre d'une implémentation en virgule flottante 32 bits sur une configuration avec quatre voies, l'utilisation est définie par :

$$\text{Utilisation}_{fp32} = 100 \times \frac{2KCH_oW_oRS \times 32}{2 \times 4 \times \text{Nombre de cycles} \times 64} \quad [\%] \quad (5.3)$$

Idéalement, la valeur de l'utilisation est de 100%. Néanmoins, dans un contexte plus réaliste, une partie des cycles est dédiée à d'autres tâches, telles que les lectures/écritures en mémoire ou la résolution de conflits dans la pipeline. Plus la valeur d'utilisation est élevée, plus l'optimisation de l'implémentation est importante.

5.1.3 Génération des données et essai

Lors de la compilation de notre code d’essai, un script Python génère un fichier de mémoire nommé `data.S` décrivant, en assembleur, les différents tenseurs d’entrée (ou activation), de filtres (ou poids) ainsi qu’un tenseur de sortie dont toutes les valeurs sont initialisées à zéro dans la mémoire cache L2 d’Ara. Les tenseurs d’entrée et de filtres contiennent des valeurs générées aléatoirement dans la plage disponible correspondant à la précision souhaitée en entier (8, 16, 32 ou 64 bits) ou en virgule flottante (32 ou 64 bits). Ce fichier contient aussi les dimensions de chaque tenseur qui sont passés en paramètres lors de l’appel à la fonction de convolution 2D. En outre, un tenseur contenant le résultat attendu de la convolution entre le tenseur d’entrée et de filtres contenant les valeurs aléatoires est généré à l’aide des bibliothèques NumPy [88] et SciPy [89]. Il permet de comparer le résultat de la convolution réalisé à l’aide des instructions vectorielles RISC-V sur Ara avec le résultat d’une même opération réalisée avec des fonctions de plus haut niveau et de valider le résultat. Tous les fichiers sources sont compilés en utilisant la chaîne d’outils LLVM 16.0.6 [90] compatible avec l’extension RVV 1.0 du jeu d’instruction RISC-V. Les résultats des opérations de convolution sur Ara proviennent de simulations au niveau transfert de registre (RTL) sur QUESTASIM FPGA STARTER EDITION 2021.2.

5.2 SparqV2

Le processeur Sparq, introduit dans le chapitre 4, utilise la méthode ULPPACK présentée dans la section 2.3.2 afin d’implémenter l’opération de convolution 2D sur des opérandes représentés sur moins de 8 bits. L’exécution de cette opération est accélérée par l’ajout d’une instruction spécialisée de multiplication-décalage-accumulation (`vmacsr`) dont le décalage est fixé en fonction de la granularité du registre vectoriel utilisé. Ceci permet d’augmenter la performance de la convolution en comparaison d’une implémentation purement logicielle comme illustré sur la Figure 4.4. Toutefois, cette instruction demeure limitée quant à son emploi pour d’autres applications que celle présentée, notamment par l’implémentation d’un décalage fixé matériellement.

5.2.1 Différences et motivations

Pour palier ce problème, nous proposons le remplacement de l’instruction `vmacsr` par un registre de contrôle configurable pendant l’exécution, nommé `vxsh`. La valeur de `vxsh` permet de spécifier un type (arithmétique ou logique), une direction (droite ou gauche) et une valeur de décalage positive ou nulle du résultat de la multiplication via une instruction spécialisée `vsetsh`.

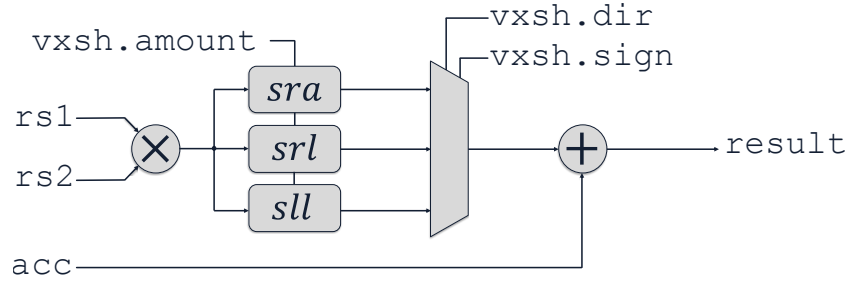


FIGURE 5.1 Schéma simplifié de l'instruction de multiplication-accumulation intégrant le décaleur en cascade piloté par le registre `vxsh` configurable avec l'instruction `vsetsh` dans SparqV2. `vxsh.amount` correspond à la valeur de décalage, `vxsh.dir` permet de sélectionner la direction (droite/gauche) et `vxsh.sign` configure le type de décalage (arithmétique/logique)

Cette modification, implémentée sur SparqV2 permet de maintenir les performances de Sparq sur la convolution tout en offrant plus de flexibilité. Par exemple, l'ajout du décaleur en cascade contrôlé par `vxsh` permet d'effectuer des opérations de multiplication sur des représentations en virgule fixe arbitraires plus aisément qu'en utilisant le jeu d'instruction initial. La Figure 5.1 présente un schéma simplifié de l'instruction de multiplication-accumulation intégrant le décaleur en cascade configuré par le registre `vxsh`.

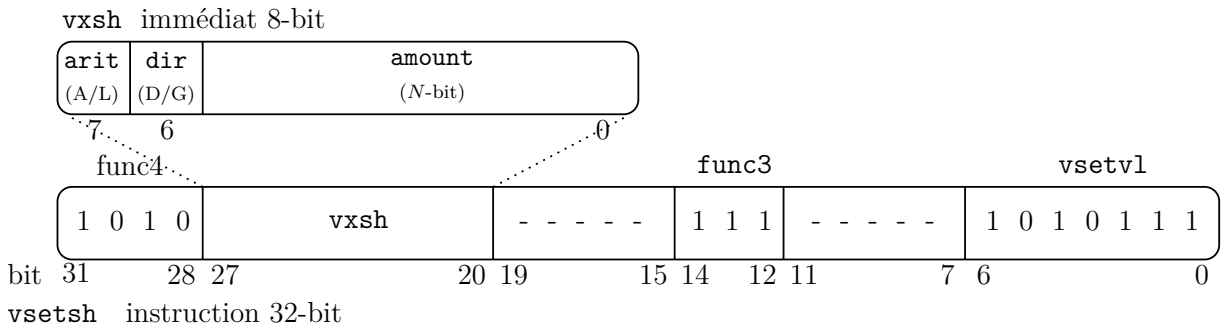


FIGURE 5.2 Illustration de l'instruction `vsetsh` permettant de configurer le registre `vxsh` à l'aide d'une valeur immédiate sur 8 bits. A/L signifie Arithmétique ou Logique, D/G signifie Droite ou Gauche. La valeur de N détermine le nombre de bit permettant d'encoder la valeur de décalage configurable disponible. '-' correspond à des bits dont l'état n'est pas pris en compte

La Figure 5.2 présente l'instruction `vsetsh` permettant de configurer le registre `vxsh`. La valeur maximale de `vxsh.amount` est déterminée par N , le nombre de bits. Nous fixons N à 6 bits, ce qui permet d'effectuer un décalage maximal de $2^6 - 1 = 63$. La granularité

maximale d'un registre vectoriel sur SparqV2 étant de 64-bit, un décaleur en cascade de cette taille permet de couvrir tous les décalages possibles, notamment ceux utilisés dans nos implémentations de convolution 2D utilisant ULPPACK.

Toutes les implémentations dont les performances et les plages de précisions sont présentées par la suite en utilisant la méthode ULPPACK. Elles emploient la stratégie d'empaquetage P1 [9] sur des registres vectoriels entiers de 16 bits. L'emploi d'une granularité de 16 bits constitue un compromis entre la flexibilité et le débit d'exécution. En utilisant ULPPACK sur Ara, le nombre d'instructions par cycle est proportionnel à la précision des instructions et à la granularité des registres. En augmentant la granularité des registres vectoriels, un nombre plus important d'opérandes de basse précision peut être empaqueté. Ceci améliore le nombre d'opérations basse précision (< 8 bits) traitées en une instruction, mais réduit le débit d'instructions par cycle en ne profitant pas de la capacité de subdivision des chemins de données sur Ara. Employer des registres de 8 bits permettrait d'obtenir le meilleur débit d'instruction par cycle. Toutefois, l'emploi d'une granularité de 8 bits limite la précision des opérandes à un maximum de 2 bits (voir équation 2.12). Pour ces raisons, nous avons fait le choix d'utiliser des registres 16 bits, tout comme les auteurs de ULPPACK [9]. Enfin, le choix d'utiliser la stratégie P1 permet de se limiter à des instructions de même précision, ici 16 bits vers 16 bits, ce qui accroît les performances, mais limite la précision des opérandes.

Algorithme 3 Proposition d'une fonction conv2d vectorisée utilisant ULPPACK sur SparqV2

H, C : Hauteur et canaux de l'entrée
 F_h, F_w : Hauteur et largeur du noyau
 M : Nombre d'opérandes empaquetés dans un élément
 R, V : Registres scalaires et vectoriels (initialisés à 0)

```

1:  $vxsh \leftarrow$  configuration du décalage
2: pour  $h \leftarrow 1$  to  $H$  faire
3:    $V_{F_h} \leftarrow 0$ 
4:   pour  $c \leftarrow 1$  to  $\frac{C}{M}$  faire
5:      $V_0 \leftarrow$  charge la  $h^{ème}$  rangée empaquetée du tenseur d'activation
6:     pour  $i \leftarrow 1$  to  $F_w$  faire
7:        $R_{[1:F_h]} \leftarrow$  charge la  $i^{ème}$  colonne empaquetée du noyau
8:       pour  $j \leftarrow 1$  to  $F_h$  faire
9:          $V_j \leftarrow vmacc(R_j, V_0, V_j)$ 
10:       $V_0 \leftarrow vslidedown(V_0, 1)$ 
11:   si  $h \geq F_h$  alors
12:      $O[h] \leftarrow V_1$  ▷ écrit une rangée de sortie complétée en mémoire
13:     pour  $j \leftarrow 1$  to  $F_h - 1$  faire
14:        $V_j \leftarrow V_{j+1}$ 
15:  $vxsh \leftarrow 0$  ▷ remise à zéro du décalage
  
```

L’Algorithme 3 présente une implémentation simplifiée similaire à celle présentée dans le chapitre 4 avec le matériel dédié de SparqV2. L’instruction `vmacs` est remplacée par l’instruction `vmacc` (ligne 9) qui embarque le décaleur en cascade entre la multiplication et l’accumulation. Le registre `vxsh` permettant de contrôler le décaleur en cascade est configuré afin que les résultats de multiplications soient décalés en suivant le nombre d’opérandes par élément et la taille d’un élément en bit (ligne 1). Pour chaque rangée empaquetée en suivant la méthode ULP-PACK (ligne 5) l’algorithme réalise une opération de multiplication-décalage-accumulation (ligne 9) comme décrite à la Figure 5.1, afin de calculer la contribution au résultat de la première colonne empaquetée du noyau. Le registre contenant la rangée empaquetée d’entrée est ensuite glissée d’un élément vers la gauche (ligne 10) pour que les éléments correspondent au calcul de la colonne empaquetée du noyau suivante. Ce processus est répété jusqu’à ce que les contributions de toutes les colonnes empaquetées du noyau sur tous les canaux de l’entrée soient traitées. Pour compléter et écrire une rangée en mémoire (ligne 12), F_h rangées empaquetées d’entrée doivent avoir été traitées (ligne 11). Les résultats partiels contenus dans les registres suivants (ligne 14) sont ensuite déplacés dans le registre d’index précédent afin que chaque somme partielle d’une nouvelle rangée d’entrée empaquetée permette de les compléter. Lorsque tous les calculs sont complétés, `vxsh` est remis à zéro, ce qui a pour effet de supprimer tout décalage sur les résultats des instructions multiplications (ligne 15).

5.2.2 Analyse des performances

Dans cette section, nous analysons les performances de l’opération de convolution 2D sur SparqV2 et Ara en utilisant ULPPACK [9]. Nous détaillons aussi la problématique de dépendance entre performance et précision des opérandes. Tout comme Sparq, les opérandes des tenseurs d’activation, filtres et sorties sont stockées en mémoire de la même façon que détaillé à la section 4.7. Tous les résultats proviennent de simulations RTL sur QUESTASIM FPGA STARTER EDITION 2021.2 avec les configurations $VLEN = 4096$ le nombre de bit par registre vectoriel et $NR_LANES = 4$ le nombre de voies sur Ara et SparqV2.

Performances

Les performances sur SparqV2 sont identiques à Sparq, car la structure des algorithmes et des instructions spécialisées reste identique. Toutefois, dans le chapitre 4, seules les performances sur un noyau de dimension 7×7 sont présentées. Ce type de noyau étant pratiquement absent des Réseaux de Neurones Convolutifs modernes, nous présentons les performances de SparqV2 dans le cadre de convolution avec des noyaux de filtres de taille 3×3 . Malgré les capacités de

réutilisation de données moins importantes que les convolutions sur des noyaux 7×7 , SparqV2 montre des performances significativement accrues en comparaison des implémentations sur virgule flottante 32 bits.

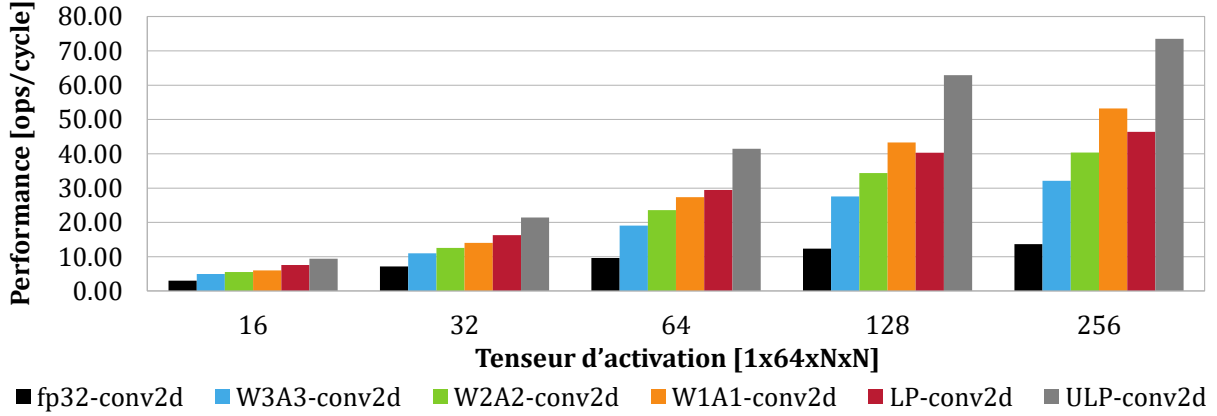


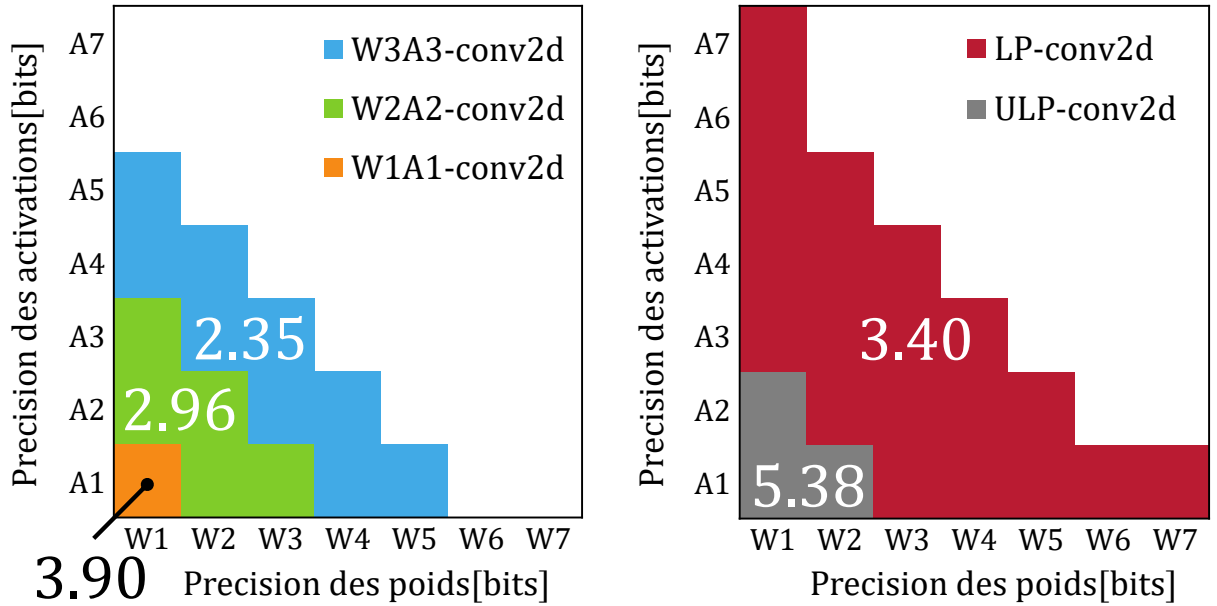
FIGURE 5.3 Comparaison des performances en opérations par cycle pour différente précision des opérandes sur un noyau de 3×3 . Les implémentations ULP et LP-conv2d tirent parti du matériel spécialisé sur SparqV2 tandis que W1A1, W2A2 et W3A3-conv2d sont implémentées sur Ara. $WxAy$ correspond à la précision des opérandes avec x la précision des poids et y la précision des activations en bits

La Figure 5.3 montre la performance d’ULPPACK sur Ara et sur SparqV2 pour différentes précisions. Pour un tenseur d’entrée de $64 \times 256 \times 256$, l’utilisation d’ULPPACK sans modification sur Ara permet d’augmenter la performance de l’opération de convolution 2D significativement par rapport à l’implémentation en virgule flottante 32 bits, avec respectivement 3.90 à 2.35 pour une précision sur les poids et les activations de 1 à 3 bits. Sur SparqV2, l’utilisation de l’instruction spécialisée et du décaleur en cascade configurée par `vxsh`, permet d’éliminer les accumulations locales nécessaires sur Ara. Éliminer la contrainte des accumulations locales permet de réduire considérablement le nombre d’instructions à utiliser, et accélère l’exécution de la convolution par un facteur compris entre 3.40 et 5.38 sur un tenseur d’entrée $64 \times 256 \times 256$ pour une précision variable des opérandes détaillée dans la section suivante par rapport à une exécution sur 32 bits. Notons que pour cette dimension de tenseur, l’implémentation en virgule flottante 32 bits offre une utilisation de 85.3%.

Plage de précision

Contrairement aux implémentations d’ULPPACK sur Ara, les implémentations sur SparqV2 peuvent être utilisées sur une plage plus importante de précision. Cette différence est due au niveau d’optimisation qui est relâché sur SparqV2 grâce à l’ajout du matériel spécialisé. Afin

d'obtenir des performances optimales sur Ara, des accumulations locales sont effectuées jusqu'à la limite de débordement théorique, dépendamment de la précision des opérandes. Or, comme détaillé dans la section précédente, l'ajout du décaleur en cascade entre la multiplication et l'accumulation de l'instruction (macc) permet de supprimer la contrainte sur les accumulations locales. Par conséquent, les fonctions de convolution peuvent opérer de façon efficiente sur une plage plus importante de précision.



(a) Implémentations sur Ara

(b) Implémentations sur SparqV2

FIGURE 5.4 Accélération relative par rapport à une implémentation en virgule flottante 32 bits dans la région sans débordement. L'activation est un tenseur de $64 \times 256 \times 256$ convolué avec un noyau 3×3 . (a) est l'implémentation d'ULPPACK sur Ara sans modification matérielle et (b) correspond à l'implémentation d'ULPPACK accélérée par le décaleur en cascade sur SparqV2. $WxAy$ correspond à la précision des opérandes avec x la précision des poids et y la précision des activations en bits

La Figure 5.4 montre les plages de précision disponibles évitant le débordement pour chaque fonction de convolution sur des noyaux 3×3 présentée sur la Figure 5.3. Le matériel spécialisé sur SparqV2 permet bien une plus grande flexibilité sur la précision, en plus de présenter une augmentation des performances.

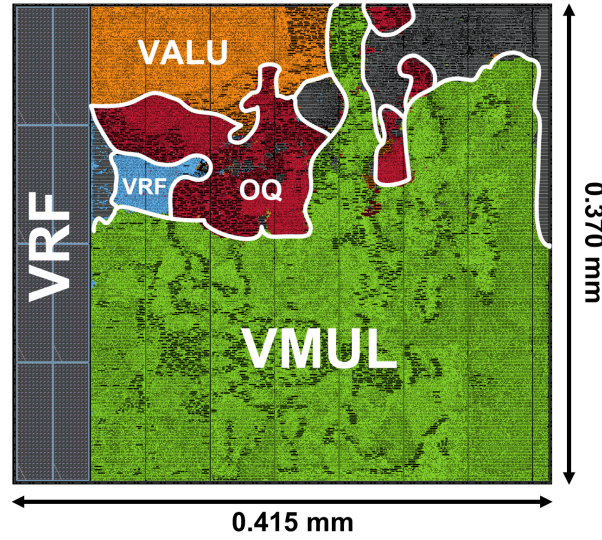


FIGURE 5.5 Implémentation d’une voie de SparqV2 en utilisant la technologie GF22 nm. Le décaleur en cascade est implémenté avec $N = 6$ bits. VRF est le banc de registres vectoriels, OQ la file d’attente d’opérandes, VMUL l’unité de multiplication et VALU l’unité de calcul arithmétique et logique

5.2.3 Implémentation physique

Comme pour Sparq, une voie de SparqV2 a été implémentée sur la technologie GLOBAL FOUNDRIES 22FDX FD-SOI (GF22) à l’aide de SYNOPSIS DESIGN COMPILER S-2021.06-SP5 pour la synthèse et CADENCE INNOVUS v21.15 pour la conception de l’arrière-plan. Les résultats de cette implémentation sont présentés en Figure 5.5. Nous pouvons observer que l’intégration d’un décaleur en cascade nécessite beaucoup plus de surface que l’instruction effectuant un décalage fixe dans Sparq. La surface utilisée est pour l’implémentation physique d’une voie de SparqV2 est aussi importante que pour Ara malgré le retrait de l’unité de calcul en virgule flottante. L’implémentation physique de la voie contenant le décaleur en cascade, n’a montré aucune dégradation du chemin critique. Ceci démontre que l’ajout du matériel spécialisé ne dégrade pas la fréquence d’horloge utilisable en comparaison d’Ara. Théoriquement, le nombre de multiplexeurs nécessaire à l’intégration du décaleur en cascade est défini par $N \log_2(N)$, avec N le nombre de bit spécifiant le décalage. L’implémentation de la Figure 5.5 correspond à une implémentation avec un décaleur en cascade de $N = 6$ bits permettant un décalage maximale de $2^6 - 1 = 63$. Toutefois, cette valeur peut être réduite en fonction de l’application afin de minimiser la surface nécessaire.

5.3 Quark

Quark [11] repose, tout comme Sparq, sur le processeur vectoriel RISC-V 64-bit Ara. Toutefois, plutôt que d'embarquer du matériel spécialisé pour la méthode ULPPACK, il a été conçu afin d'optimiser les opérations binaires séquentielles présentées dans la section 2.3.2. Quark a été développé, conçu et testé dans le cadre du même projet que Sparq ainsi que par la même équipe. L'objectif est là aussi d'augmenter la performance de l'opération de convolution sur des représentations basse précision d'opérandes afin d'améliorer le débit de traitement de Réseaux de Neurones Convolutifs Quantifiés (QCNN).

5.3.1 Différences et motivations

L'emploi de la méthode d'opérations binaires séquentielles permet d'augmenter considérablement les performances dans le cas de précision très basse des opérandes, typiquement 1 à 2 bits. Cette méthode est théoriquement plus performante que ULPPACK dans cette plage de précision. Pour rappel, l'opération mathématique principale à réaliser dans la méthode d'opérations binaires séquentielles est la suivante :

$$w \cdot a = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} 2^{i+j} \times \text{popcount}(\bar{w}_i \& \bar{a}_j)$$

avec respectivement $\bar{w}$ et $\bar{a}$ les vecteurs empaquetés des vecteurs quelconques w et a et N et M leur précision respective en bits.

Un grand nombre de modifications sur l'unité vectorielle ont été apportées à Quark afin d'augmenter les performances de la convolution 2D. Parmi ces modifications, nous retrouvons :

Une instruction vpopcount, ayant pour objectif de sommer le nombre de bit à l'état haut d'un élément. Son comportement sur chaque élément d'un vecteur peut être décrit par l'opération suivante :

$$\text{popcount}(s_1) = \sum_{i=0}^{N_{s_1}-1} s_1 \& 2^i$$

avec N_{s_1} le nombre de bits contenu dans l'élément s_1 .

Une instruction vshac permettant d'effectuer une opération de décalage-accumulation. Elle est notamment utilisée afin de décaler les résultats intermédiaires pour qu'ils

correspondent à leur poids respectif dans la somme. Elle est définie par l'opération suivante sur chaque élément d'un vecteur :

$$\text{vshac}(d, s_1, s_2) = 2^{s_1} \times s_2 + d$$

Une instruction **vbitpack** qui réalise tout le processus d'empaquetage des opérandes basse précision dans les registres décrit dans la section 2.3.2. Cette instruction est primordiale, car c'est le processus d'empaquetage qui constitue le goulot d'étranglement dans la méthode d'opérations séquentielles binaires. Elle est décrite plus précisément dans l'article proposant Quark [11].

Sans ces modifications, les charges supplémentaires de l'opération d'empaquetage et de comptage de population (*popcount*) ne permettent pas d'atteindre un gain en performance justifiant l'emploi de représentations à très basse précision. En outre, l'implémentation de ces instructions ne constitue pas une contrainte importante comme nous le détaillons dans la section 5.3.3.

Algorithme 4 Proposition d'une fonction conv2d vectorisée utilisant des opérations binaires séquentielles

H, C : Hauteur et canaux de l'entrée
 F_h, F_w : Hauteur et largeur du noyau
 P : Taille d'un élément des registres vectoriels (en bits)
 R, V : Registres scalaires et vectoriels (initialisés à 0)
 M, N : Précision des poids et des activations (en bits)

```

1: pour  $h \leftarrow 1$  à  $H$  faire
2:    $V_{F_h} \leftarrow 0$ 
3:   pour  $c \leftarrow 1$  à  $\frac{C}{P}$  faire
4:      $V_0 \leftarrow$  charge la  $h^{\text{ème}}$  rangée empaquetée du tenseur d'activation
5:     pour  $i \leftarrow 1$  à  $F_w$  faire
6:        $R_{[1:F_h]} \leftarrow$  charge la  $i^{\text{ème}}$  colonne empaquetée du noyau
7:       pour  $j \leftarrow 1$  à  $F_h$  faire
8:         pour  $m \leftarrow 0$  à  $M - 1$  faire
9:           pour  $n \leftarrow 0$  à  $N - 1$  faire
10:             $V_{F_h+j+1} \leftarrow \text{vpopcount}(V_0 \& R_j)$ 
11:             $V_j \leftarrow \text{vshac}(V_{F_h+j+1}, (n + m), V_j) \quad \triangleright V_j += 2^{n+m} \times V_{F_h+j+1}$ 
12:           $V_0 \leftarrow \text{vslidedown}(V_0, 1)$ 
13:       si  $h \geq F_h$  alors
14:          $O[h] \leftarrow V_1 \quad \triangleright$  écrit une rangée de sortie complétée en mémoire
15:       pour  $j \leftarrow 1$  à  $F_h - 1$  faire
16:          $V_j \leftarrow V_{j+1}$ 
  
```

L'Algorithme 4 présente une implémentation simplifiée de la convolution 2D basse précision à l'aide de méthode d'opérations binaires séquentielles et des instructions spécifiques embarquées sur Quark. Pour chaque rangée empaquetée à l'aide de **vbitpack**, l'algorithme réalise $N \times M$ opérations de comptage de population et de décalage-accumulation (lignes 10 et 11) afin de calculer les résultats partiels liés à la première colonne empaquetée du noyau. Les éléments de la rangée d'entrée sont ensuite glissés vers la gauche d'un élément (ligne 12) afin de pouvoir calculer la contribution de la colonne empaquetée suivante du noyau. Cette opération est répétée jusqu'à ce que toutes les colonnes du noyau aient été prises en compte. Pour compléter et écrire une rangée en mémoire (ligne 14), F_h rangées empaquetées d'entrée doivent avoir été traitées (ligne 13). Les résultats partiels contenus dans les registres suivants (ligne 16) sont ensuite déplacés dans le registre d'index précédent afin que chaque somme partielle d'une nouvelle rangée d'entrée empaquetée permette de les compléter.

5.3.2 Analyse des performances

Dans cette section, nous analysons les performances de Quark utilisant la méthode d'opérations binaires séquentielles décrite dans l'Algorithme 4 et les comparons à celle d'une convolution 2D typique sur Ara en virgule flottante 32 bits. Les opérandes des tenseurs d'entrée, filtres et sorties sont stockées en mémoire de la même façon que détaillé à la section 4.7. Néanmoins, les résultats de la convolution 2D sont stockés sur des objets entiers de 32 bits, contre 16 bits sur Sparq et SparqV2. Tous les résultats proviennent de simulations RTL sur QUESTASIM FPGA STARTER EDITION 2021.2 avec les configurations $VLEN = 4096$ le nombre de bit par registre vectoriel et $NR_LANES = 4$ le nombre de voies.

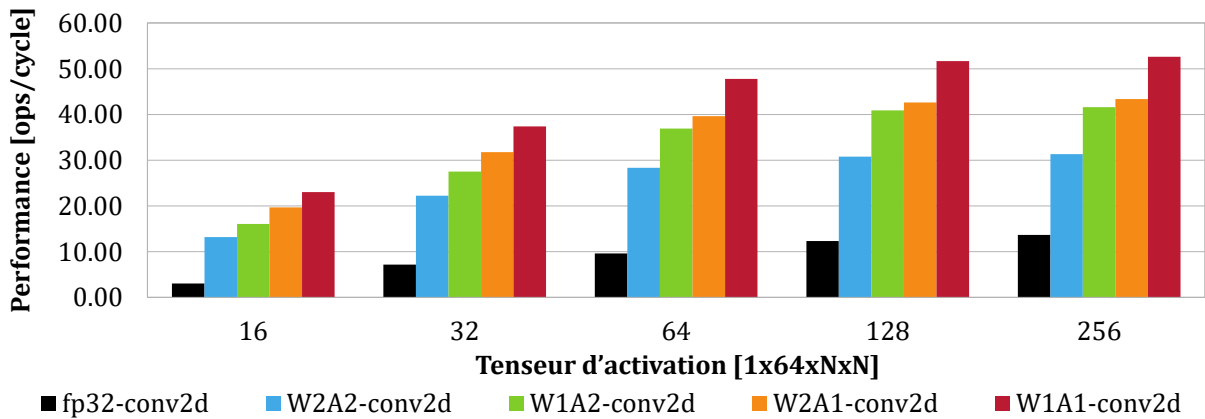


FIGURE 5.6 Comparaison des performances en opérations par cycle pour différentes précisions des opérandes sur un noyau de 3×3 sur Quark. $WxAy$ correspond à la précision des opérandes avec x la précision des poids et y la précision des activations en bits

Performances

Quark emploie la méthode de traitement basée sur des opérations binaires séquentielles présentée à la section 2.3.2. Nous limitons son utilisation à des précisions d'opérandes très basses, de 1 à 2 bits.

La Figure 5.6 présente les performances de l'opération de convolution sur Quark en utilisant un noyau 3×3 . Comme attendu, la performance chute rapidement avec l'augmentation de la précision. Avec une précision de 2 bits sur les poids et les activations, la performance est en moyenne $1.65 \times$ moins élevée que celle sur 1 bit. Malgré cette différence, Quark montre une performance accrue par un facteur de 5.21 et 3.10 pour une précision respective de 1 et 2 bits par rapport à la performance en virgule flottante 32 bits, dans le cas d'une convolution 3×3 sur un tenseur d'activation de $64 \times 32 \times 32$.

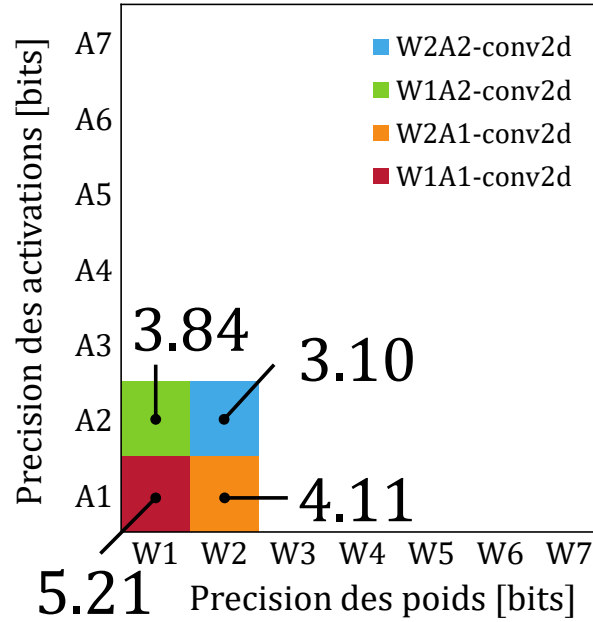


FIGURE 5.7 Accélération relative sur Quark par rapport à une implémentation en virgule flottante 32 bits. L'activation est un tenseur de $64 \times 32 \times 32$ avec un noyau de 3×3 . $WxAy$ correspond à la précision des opérandes avec x la précision des poids et y la précision des activations en bits

Plage de précision

La méthode d'opérations séquentielles binaires n'offre pas autant de flexibilité que la méthode ULPPACK sur Sparq. Par son empaquetage spécifique sur les poids et les activations, cette

méthode nécessite une implémentation logicielle spécifique à couple de précision. En outre, l'instruction spécialisée `vbitpack` permettant le processus d'empaquetage des données dans les registres [11] a été conçu pour une précision de 2 bits sur les poids et les activations. S'il est possible de l'employer de façon relativement efficace pour des précisions sur 1 bit, il est plus difficile de l'utiliser efficacement pour des précisions supérieures à 2 bits. Ce phénomène limite fortement la plage de précision disponible sur Quark.

La Figure 5.7 illustre l'accélération relative des performances de Quark par rapport à une représentation en virgule flottante 32 bits sur Ara, pour une convolution entre un tenseur d'activation de $64 \times 32 \times 32$ et un noyau 3×3 . La plage de précision des implémentations de convolutions 2D présentée dans la section performance de Quark y est présentée. Malgré une augmentation des performances, les plages de précisions disponibles sur Quark sont significativement limitées par la méthode d'opérations binaires séquentielles. Cette différence est particulièrement importante en comparaison des plages de précision d'ULPPACK présentées en Figure 5.4. La différence d'accélération entre les couples de précision W1A2 et W2A1 s'explique par la complexité du processus d'empaquetage qui augmente avec la précision des opérandes ainsi que par la quantité plus importante de valeurs d'activation que de filtre.

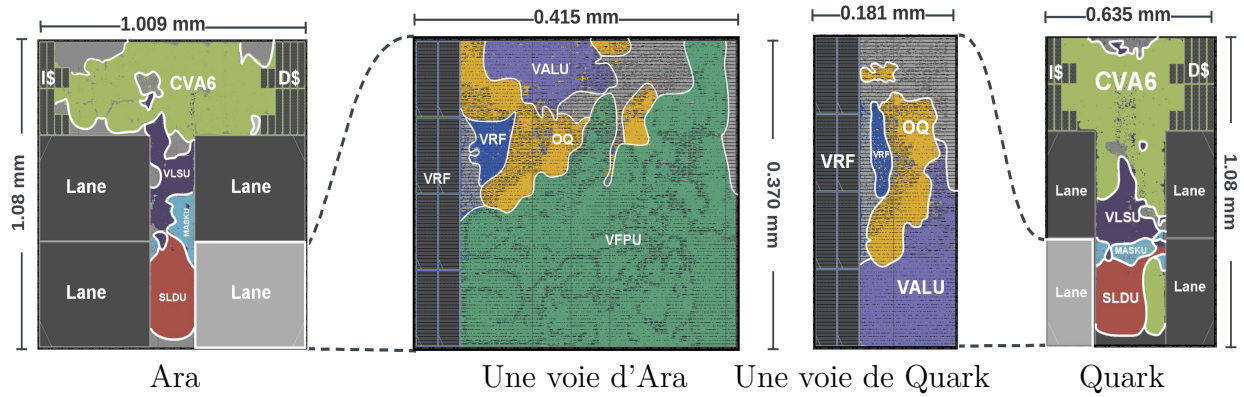


FIGURE 5.8 Implémentation après placement et routage de Quark en comparaison d'Ara en GF22 nm avec, VRF le banc de registres vectoriels, OQ la file d'attente d'opérandes, VFPU l'unité de calcul en virgule flottante et VALU l'unité de calcul arithmétique et logique. Image tirée de [11] avec autorisation de l'auteur

5.3.3 Implémentation physique

Contrairement à ULPPACK sur Ara, Sparq et SparqV2, cette méthode ne requiert aucune multiplication et est effectuée avec seulement des instructions logiques (décalages et opérations

bit à bit) et d'additions (*cf.* équation 2.11). Ceci nous permet de retirer l'unité de calcul en virgule flottant comme sur Sparq et SparqV2 mais plus généralement, de retirer l'unité de multiplication complète. Son retrait minimise la surface nécessaire à l'implémentation physique de Quark par rapport à Ara en plus de réduire fortement la consommation énergétique.

Deux possibilités s'offrent alors :

Avec une configuration 4 voies, comme présenté sur la Figure 5.8 la surface de Quark est divisée par 2.3 et permet de réduire significativement la consommation énergétique tout en maintenant des performances entre $3\times$ et $6\times$ plus importantes que sur Ara en virgule flottante 32 bits selon la précision des opérandes. Cette configuration met l'accent sur la minimisation de la surface et de la consommation de Quark.

Avec une configuration 8 voies, la surface de Quark est identique à celle d'Ara et la consommation d'énergie estimée est équivalente. Néanmoins, en doublant le nombre de voies, les performances de l'opération de convolution 2D sont approximativement multipliée par deux par un processus de parallélisation plus important. Cette configuration met alors l'accent sur la performance au détriment d'une surface équivalente à Ara.

La Figure 5.8 illustre l'implémentation complète de Quark en GF22 nm et compare la surface du processeur et d'une voie à celle d'Ara. Le retrait de l'unité de calcul en virgule flottante, contenant aussi l'unité de multiplication, permet de diviser la surface nécessaire à l'implémentation d'une voie par respectivement 2.3 et 2.61 pour 4 et 8 voies. Par rapport à Ara, cette réduction permet de diviser par 1.58 la surface du processeur complet pour une configuration sur 4 voies et de présenter une surface similaire pour la configuration sur 8 voies.

CHAPITRE 6 COMPARAISON DES PERFORMANCES

6.1 Comparaison sur des architectures standards

Dans cette section, nous comparons les performances d'ULPPACK sur un Raspberry Pi 4 (module embarquant un processeur d'architecture ARMv8) présenté dans l'article introduisant ULPPACK [9] et Ara (architecture RISC-V GCV) basée sur un algorithme de convolution direct.

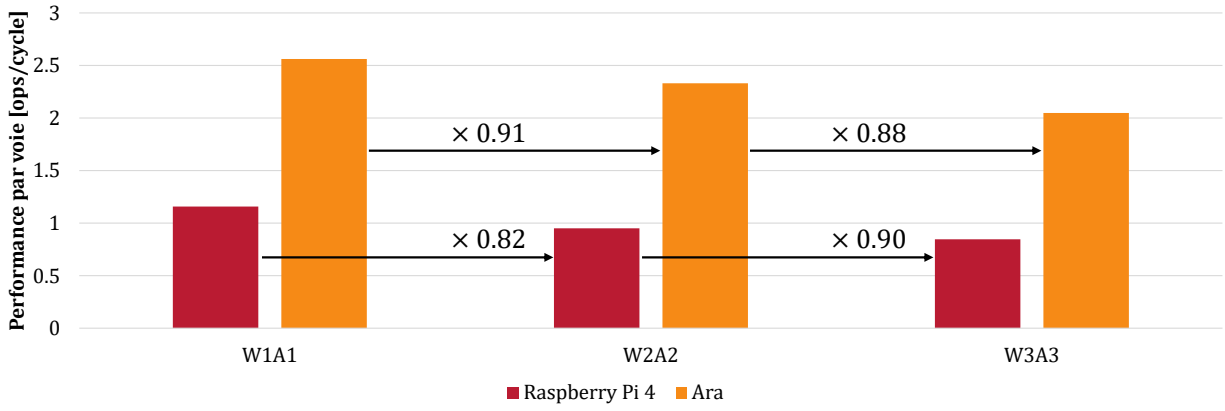


FIGURE 6.1 Comparaison des performances en opérations par cycle d'une multiplication matricielle $512 \times 512 \times 512$ sur un Raspberry Pi 4 et d'une opération de convolution 2D entre un tenseur d'entrée de $1 \times 56 \times 25 \times 25$ et un tenseur de 512 filtres de $56 \times 3 \times 3$ sur Ara en utilisant ULPPACK [9]. $W \times A_y$ correspond à la précision des poids (W) et des activations (A) en bits

Sur le processeur ARM, l'opération implémentée par la méthode ULPPACK est une multiplication matricielle, alors que nous utilisons un algorithme dédié à une convolution 2D sur Ara. Comme décrit dans la section 2.1.2, en modifiant la disposition des valeurs présentes dans les tenseurs, il est possible d'effectuer une convolution en utilisant un produit matriciel. Les performances sur le Raspberry Pi 4 sont issues d'une multiplication matricielle entre deux matrices de 512×512 éléments. Afin de comparer le produit matriciel sur ARM avec une convolution sur Ara, nous proposons de fixer un noyau $R \times S$ de 3×3 et un tenseur d'activation dont la hauteur et la largeur sont égales. En effectuant l'opération réciproque de `im2col` comme présentée sur la Figure 2.2, nous déterminons les dimensions des tenseurs d'entrée et de filtres de la convolution 2D par :

$$\left\{ \begin{array}{l} R = S = 3 \\ C \times R \times S = 512 \leftrightarrow C = \frac{512}{9} \approx 56 \\ H_o \times W_o = 512 \leftrightarrow H_o = W_o = \sqrt{512} \approx 23 \\ K = 512 \end{array} \right. \quad (6.1)$$

Si les performances sur le processeur ARM sont issues d’une multiplication matricielle, elles ne tiennent pas compte de cette modification de disposition (im2col) qui doit être effectuée avant la multiplication pour correspondre à une convolution et qui réduit la performance. En outre, dans un souci de comparaison juste des performances de ces deux architectures, nous transposons les performances en opérations par cycle et par voie afin de supprimer la différence de fréquence d’horloge et de micro-architecture.

La Figure 6.1 compare les performances de la méthode ULPPACK sur une Raspberry Pi 4 et sur Ara. Nous observons qu’Ara présente une performance en opérations par cycle par voie plus importante que sur l’architecture ARM qui s’explique par la quantité moins importante d’accès mémoire à effectuer dans notre implémentation. En outre, comme illustré sur la Figure 5.3, la performance de notre algorithme est fortement dépendante de la largeur du tenseur d’activation, car la vectorisation est effectuée sur cette dimension. Au contraire, l’implémentation sur ARM propose de vectoriser les opérations sur les canaux, ce qui rend ses performances dépendantes de cette dimension. Notons que, par la différence d’architecture, de micro-architecture, d’opération et de contexte d’application entre les deux systèmes, il est difficile de donner une conclusion claire sur cette comparaison. En revanche, malgré les différences énoncées, l’évolution relative des performances démontre que le comportement de notre implémentation sur Ara est similaire à l’implémentation sur Raspberry Pi 4 en suivant la précision des opérandes.

6.2 Comparaison entre Quark et SparqV2

Dans le chapitre précédent, nous avons introduit deux processeurs vectoriels modifiés basés sur Ara, à savoir SparqV2 et Quark. Nous avons décrit les modifications apportées et analysé leur performance dans le contexte d’opérations de convolution 2D sur des tailles arbitraires. Dans cette section, nous comparons les performances des deux processeurs modifiés lors d’une opération de convolution 2D sur les couches de convolution de modèles de Réseaux de Neurones Convolutifs. Enfin, nous comparons dans un tableau récapitulatif, les différences entre Ara, SparqV2 et Quark et leur domaine d’intérêt. Tous les résultats proviennent de simulations RTL sur QUESTASIM FPGA STARTER EDITION 2021.2 avec les configurations $VLEN = 4096$ le nombre de bit par registre vectoriel et $NR_LANES = 4$ le nombre de voies.

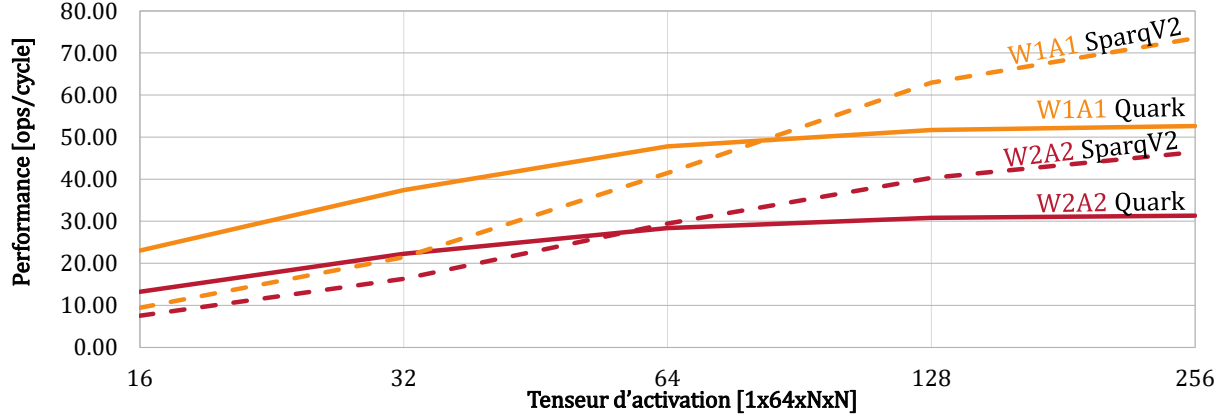


FIGURE 6.2 Comparaison des performances en opérations par cycle pour des précisions d'opérande de 1 et 2 bits sur un noyau de 3×3 sur SparqV2 et Quark (4 voies)

6.2.1 Performances

En comparant les performances entre SparqV2 et Quark, nous pouvons observer empiriquement plusieurs différences :

La dimension des entrées joue un rôle important sur la performance. Quark présente des performances plus importantes que SparqV2 pour des entrées inférieures à une hauteur et une largeur de 32×32 indépendamment du nombre de canaux. Cette différence s'explique par le relâchement de la charge sur les opérations d'empaquetage sur Quark, comme détaillé dans l'article éponyme [11]. Notons, que cette contrainte semble s'estomper lorsque la taille du tenseur d'activation augmente.

La limite de performance semble, au contraire, profiter à SparqV2. En extrapolant les courbes de la Figure 6.2, nous observons que Quark est en limite de performances quand celles de SparqV2 continue de croître de façon plus importante, indiquant une limite maximale de performance plus importante. En outre, les performances de SparqV2 sur des opérands de précisions W1A1 et W2A2 correspondent aux précisions minimales dans les plages disponibles en suivant la Figure 5.4. Ceci indique que les performances de SparqV2 sont identiques pour des précisions d'opérands plus importantes, là où nous observerions une réduction de performance sur Quark.

L'utilisation de la méthode ULPPACK sur SparqV2 semble plus indiquée pour le traitement d'objet de grande dimension, alors que Quark présente de meilleures performances sur des objets d'entrée avec une taille réduite et une précision très basse des opérands, entre 1 et 2 bits. Afin de vérifier cette hypothèse dans un cadre plus réaliste, nous comparons l'accélération de performance entre SparqV2 et Quark sur les couches de Réseaux de Neurones Convolutifs.

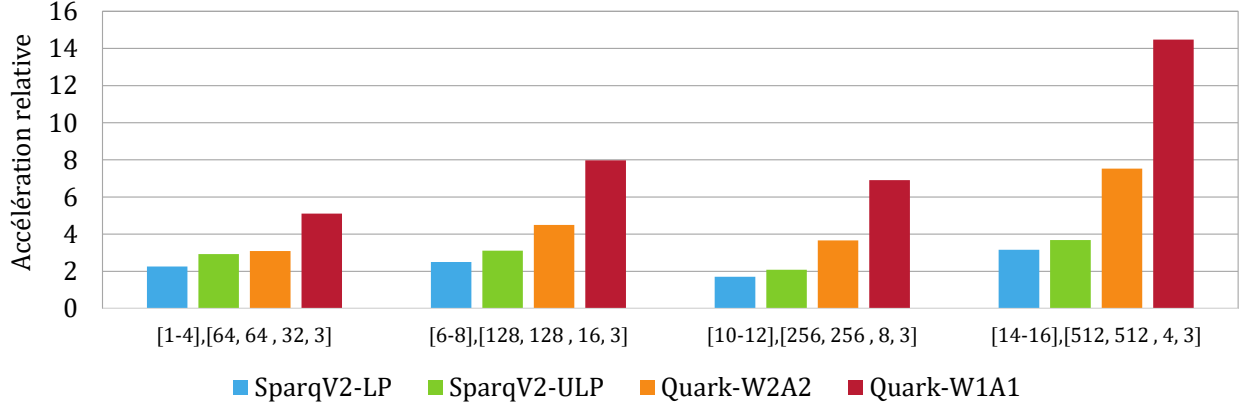


FIGURE 6.3 Accélération relative de couches de convolution de ResNet-18 [1] sur l'ensemble CIFAR-10 [12] pour différentes quantifications sur SparqV2 et Quark. L'abscisse représente le numéro de la couche avec en premier, les couches du modèle [a-b]. Ensuite sont donnés dans l'ordre, [nombre de canaux de sorties, nombre de canaux d'entrée, hauteur et largeur de l'entrée, hauteur et largeur du noyau]

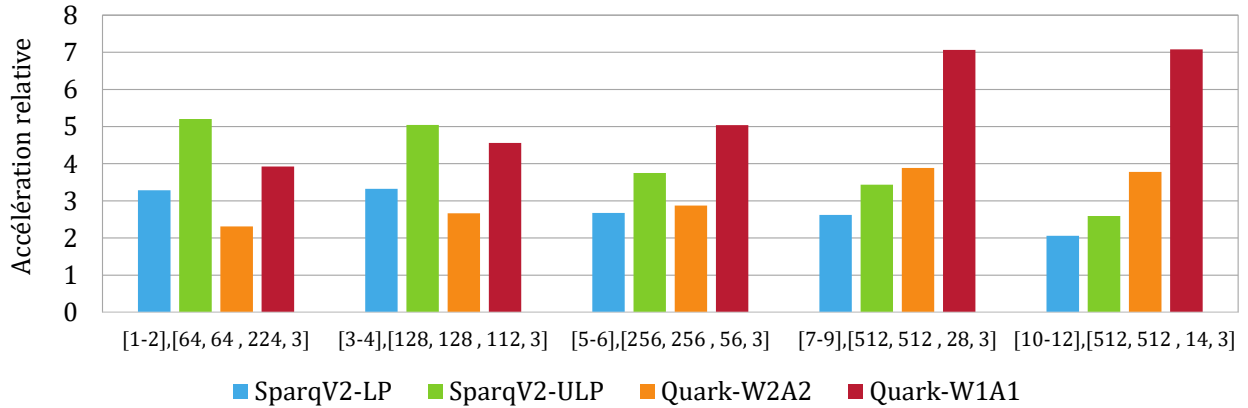


FIGURE 6.4 Accélération relative de couches de convolution de VGG16 [13] sur l'ensemble ImageNet [4] pour différentes quantifications sur SparqV2 et Quark. L'abscisse représente le numéro de la couche avec en premier, les couches du modèle [a-b]. Ensuite sont donnés dans l'ordre, [nombre de canaux de sorties, nombre de canaux d'entrée, hauteur et largeur de l'entrée, hauteur et largeur du noyau]

La Figure 6.3 montre l'accélération relative de certaines couches de convolutions de ResNet-18 [1] par rapport à une convolution en virgule flottante 32 bits sur l'ensemble de données CIFAR-10 [12]. Nous pouvons observer que sur toutes les couches présentées, Quark permet d'obtenir l'accélération la plus importante avec en moyenne une accélération de 8.35 et 4.58 pour respectivement une quantification 1 bit et 2 bits sur les poids et activations. Sur les mêmes quantifications, SparqV2 accélère par 2.95 et 2.40 le temps d'exécution des couches de

convolutions. Sur cet ensemble de données, les tenseurs d’activations ont une taille réduite, il est donc cohérent que Quark soit plus performant.

La Figure 6.4 présente l’accélération relative de certaines de ses couches de convolutions sur le modèle VGG16 [13] par rapport à sa représentation en virgule flottante 32 bits sur l’ensemble de données ImageNet [4]. Ici encore, Quark présente la meilleure accélération moyenne avec 5.79 et 3.22 pour une quantification sur 1 bit et 2 bits contre 3.84 et 2.72 pour SparqV2. Néanmoins, l’écart est moins important que sur la Figure 6.3. Ceci s’explique par les dimensions plus importantes des tenseurs d’activations en employant ImageNet sur les premières couches. La première couche des deux modèles n’est pas représentée, en effet, elle possède un nombre restreint de canaux d’entrée et est généralement non quantifié, ce qui ne permet pas de tirer parti de nos méthodes.

Toutefois, ces comparaisons sont à nuancer, car si les performances sur Quark correspondent bien à des quantifications sur 1 et 2 bits. Les implémentations sur SparqV2 englobe une plage de précision bien plus grande sur laquelle les performances sont identiques. Par exemple, en utilisant la zone de précision LP sur SparqV2 (*cf.* Figure 5.4), nous obtenons des performances identiques pour des couples de quantification similaires sur les activations et poids sur 2 ou 3 bits. Il en est de même dans le cadre de convolution à précision mixte.

Quark est donc particulièrement performant dans le cas de quantification de très basse précision. SparqV2, plus qu’un concurrent, est complémentaire à Quark et permet d’implémenter des couches de convolutions efficaces sur une plage plus importante de précision grâce à une méthode différente.

6.2.2 Caractéristiques

Chaque modification présentée précédemment présente des avantages et des inconvénients. Nous proposons ici de récapituler les caractéristiques de chaque solution matérielle. Nous nous limitons à quatre voies pour tous les processeurs, car c’est la configuration par défaut d’Ara. De plus, cette configuration permet de minimiser la consommation énergétique en plus de la surface par rapport à une configuration avec plus de voies.

Tout d’abord, il est important de préciser qu’une accélération de la performance est possible sans modification du matériel sur Ara. Il est donc possible, en utilisant la méthode ULPPACK [9], de réduire le temps d’exécution de la convolution 2D basse précision en utilisant le jeu d’instruction vectoriel initial. Toutefois, comme nous l’avons détaillée durant ce mémoire, l’ajout d’instructions spécialisées, pour les deux méthodes présentées, permet de réduire de façon significative le temps d’exécution de la convolution.

TABLEAU 6.1 Récapitulatif des caractéristiques des différentes solutions proposées pour une configuration sur quatre voies

	Ara [34]	Sparq	SparqV2	Quark [11]
Méthode <i>conv2d</i>	ULPPACK	ULPPACK	ULPPACK	Opérations séquentielles
Précision opérandes	1 à 3 bits	1 à 4 bits	1 à 4 bits	1 à 2 bits
Accélération approximative	1.5 à 3×	3 à 6×	3 à 6×	4 à 8×
Modifications	-	Pas de FPU	Pas de FPU	Pas de multiplications
Surface relative (voie)	1	0.57	1	0.39

Finalement, chaque architecture proposée s’oriente vers l’optimisation d’un paramètre clef. Le Tableau 6.1 résume les caractéristiques des différentes architectures. SparqV2 s’oriente vers un objectif de flexibilité en permettant d’accélérer de façon importante la performance de la convolution sur une plage de précision des opérandes importante. De plus, les modifications présentes dans SparqV2 peuvent être utilisées dans le contexte d’autres applications que la convolution 2D, ce qui s’oriente dans la direction de la flexibilité quand Quark s’oriente vers un objectif de minimisation de surface et de la consommation d’énergie. En revanche, ce dernier présente une plage de précision particulièrement limitée et le retrait de toute l’unité de multiplication (virgule flottante et entière) pose une contrainte importante. En outre, les instructions spécifiques disponibles sur Quark sont exclusivement destinées à la convolution par opérations binaires séquentielles, et il est difficile d’en tirer avantage dans d’autres contextes.

CHAPITRE 7 DISCUSSION GÉNÉRALE

Afin de déployer des Réseaux de Neurones Convolutifs complexes sur des appareils dont les ressources sont contraintes, il est nécessaire d’effectuer certains compromis. Ces derniers tentent de réduire la complexité ainsi que la taille des modèles, permettant d’obtenir des performances de traitement raisonnables au prix d’une perte de la précision. Ces réseaux trouvent leur intérêt sur des systèmes électroniques embarqués, des téléphones intelligents ou encore l’internet des objets, où le traitement des données est effectué sur l’appareil basse consommation plutôt que de s’appuyer sur un traitement centralisé infonuagique. De nombreux travaux de recherches, portant sur les processus d’élagage et de quantification, tentent de réduire leur impact sur la perte de précision. Si la quantification en dessous de l’octet présentée dans ce mémoire semble être une solution prometteuse, les capacités des architectures de processeurs standards posent un frein pour son emploi. Malgré cette limite, le retour d’architectures de processeurs vectorielles constitue une opportunité intéressante proposant un compromis entre la flexibilité de leurs architectures standardisées ainsi que leur gain en performance sur la parallélisation des opérations vectorielles tout en présentant une consommation réduite par rapport à des GPU. Ils constituent des cibles idéales pour les phases d’inférence des Réseaux de Neurones, principalement constitués d’opérations vectorielles, matricielles ou tensorielles. Dans l’objectif de palier ces limitations, nous présentons plusieurs méthodes logicielles permettant d’effectuer des opérations sur des opérandes d’une précision inférieures à l’octet. Aussi, nous implémentons ces méthodes sur une nouvelle architecture vectorielle en sources libres. Ceci nous permet, en plus de travailler sur du matériel moderne adapté aux tâches sur les NN, d’y effectuer des modifications aisément. Nous vérifions les performances liées à l’apport des méthodes logicielles ainsi que des modifications matérielles au travers d’opérations de convolution 2D multicanaux, s’inscrivant directement dans un contexte de Réseaux de Neurones Convolutifs Quantifiés (QCNN) sur une précision inférieure à l’octet.

Le travail de ce mémoire est effectué en simulation RTL sur le processeur vectoriel RISC-V Ara [33, 34] 64 bits cadencé à 1GHz (jeu d’instruction RISC-V GCV) sans système d’exploitation. Nous proposons d’utiliser la méthode ULPPACK [9] qui offre une solution algorithmique afin d’accélérer l’exécution d’opérations sur des opérandes de précision arbitraires inférieures à 8 bits. Cet article propose, initialement, une implémentation purement logicielle de ULPPACK sur une Raspberry Pi 4, un module embarquant un processeur quatre cœurs cortex-A7 cadencé à 1.5GHz (jeu d’instruction ARM v8) sur un système d’exploitation UBUNTU 64 BITS 20.04 LTS. La comparaison entre les performances des deux processeurs est alors difficile, car ils ne s’orientent pas vers les mêmes applications. Par conséquent, nous avons choisi

d'appuyer principalement nos comparaisons sur les performances d'une convolution 2D en virgule flottante 32 bits sur Ara. Ces implémentations de référence présentent une utilisation des voies comparables, voire supérieures à 90% selon les dimensions de filtres et d'activation, ce qui correspond à une valeur élevée démontrant un haut niveau d'optimisation [33, 34]. Elles jouent donc le rôle de référence pour les comparaisons dans ce travail. Celui-ci s'oriente principalement sur l'accélération de l'exécution des CNN pour des quantifications entre 1 et 4 bits des activations et des poids, bien que des méthodes comme la stratégie P2 d'ULPPACK [9] proposent des algorithmes pour augmenter les performances sur des précisions plus élevées. Le choix de se limiter à une telle plage de précision des opérandes est motivée par les différents travaux présentant des précisions identiques aux modèles en précision virgule flottante 32 bits dont il est fait référence dans la section 2.2.

Dans ce contexte, nous avons démontré que ULPPACK [9], sans aucune modification matérielle, permet d'augmenter considérablement la performance d'une convolution dans des configurations d'opérandes représentés sur moins de 8 bits. Ceci permet de démontrer qu'elle constitue une méthode pertinente pour l'accélération de calcul sur des opérandes basse précision sur une architecture RISC-V "V", hypothèse qui n'avait pas été vérifiée auparavant. En partant de ce constat, nous avons ensuite proposé des modifications de l'unité vectorielle d'Ara afin de créer divers processeurs spécifiques orientés vers l'exécution de la phase d'inférence de QCNN. Nous avons alors utilisé des structures d'algorithmes identiques aux versions sur Ara afin que les comparaisons soient pertinentes. Finalement, nous montrons que les modifications proposées permettent d'augmenter les performances de façon encore plus importante au détriment de modifications matérielles. Néanmoins, le processus de modification permet de retirer les modules jugés superflus dans le contexte d'inférence de QCNN, ce qui permet de réduire la surface de l'implémentation ainsi que leur consommation énergétique. Par ailleurs, nous avons réalisé le même travail en employant la méthode d'opérations séquentielles binaires sur une autre architecture basée sur Ara et nommée Quark [11]. Les modifications apportées ainsi que l'emploi de cette méthode permettent à Quark de proposer des performances encore supérieures aux autres architectures présentées tout en offrant une réduction plus importante de la surface nécessaire. En revanche, la supériorité de cette architecture se limite à une précision maximale des opérandes de 2 bits, significativement moins qu'avec ULPPACK.

En proposant ces modifications d'Ara, nous démontrons qu'il est possible d'obtenir des architectures conçues pour l'inférence de Réseaux de Neurones Convolutifs Quantifiés à un coût énergétique et surfacique très faible tout en proposant des performances élevées. Ces caractéristiques permettent à Ara et, plus particulièrement aux variations présentées dans ce mémoire, de s'établir comme des architectures en sources libres de choix pour un traitement d'informations au plus près de l'utilisateur dans un contexte d'environnement à ressources

contraintes. Néanmoins, l'utilisation d'une architecture de processeurs RISC-V "V" tel que disponible sur Ara et ses variations présente des inconvénients importants, notamment liés à son manque de maturité qui se traduit par un support partiel du jeu d'instruction induit par l'évolution récente et constante du standard. En revanche, l'accessibilité au code source de l'implémentation d'Ara permet de modifier sa micro-architecture et de l'adapter au besoin. Cette tâche est impossible à effectuer sur un processeur ARM, à cause des micro-architectures propriétaires. Les modifications proposées dans les diverses architectures présentées permettent, en plus de répondre à un besoin d'augmentation des performances dans un contexte de QCNN (1 à 4 bits), sont facilement réalisables.

CHAPITRE 8 CONCLUSION ET RECOMMANDATIONS

8.1 Synthèse des travaux

Les chapitres 4 et 5 introduisent différentes modifications du matériel au travers de divers processeurs. Tous ces processeurs sont basés sur Ara, processeur RISC-V vectoriel 64 bits. Les propositions de modifications reposent sur l’objectif d’accélération d’algorithmes existant de convolution 2D basse précision (< 8 bits), à savoir les opérations binaires séquentielles [68] et ULPPACK [9].

Introduit dans le chapitre 4, Sparq utilise la méthode ULPPACK [9] en plus d’une instruction spécialisée de multiplication-décalage-accumulation afin d’accélérer l’exécution des produits de convolution 2D. Cette instruction a pour objectif de décaler le résultat d’une multiplication par la moitié de la granularité des registres vectoriels. Elle permet de supprimer des contraintes de débordement liées à ULPPACK [9] menant à une augmentation de la performance par rapport à une implémentation similaire sur Ara. Elle permet aussi d’augmenter les performances (entre $1.7\times$ et $3.2\times$) sur l’exemple d’un noyau 7×7 sur une précision de 1 à 4 bits. Nous proposons ensuite l’implémentation en GF22 nm en retirant l’unité de calcul à virgule flottante d’une voie, ce qui résulte en une diminution de 43.3% de la surface et de 58.8% de la consommation énergétique d’une voie de Sparq par rapport à SparqV2.

Par la suite, dans la première partie du chapitre 5, nous proposons d’améliorer l’instruction de multiplication-décalage-accumulation par l’introduction d’un registre configurable permettant de sélectionner une valeur et un type arbitraire de décalage au détriment d’une surface accrue. Nous illustrons que le gain en performance sur cette nouvelle version SparqV2 est toujours significatif en montrant les performances sur un produit de convolution 2D sur un noyau 3×3 , très présent dans les CNN modernes. Nous obtenons une augmentation des performances comprise entre $5.38\times$ et $3.40\times$ selon la précision.

Quark [11] est introduit dans la deuxième partie du chapitre 5 et implémente la méthode d’opérations binaires séquentielles. Pour ce faire, il embarque des instructions spécialisées dédiées à des opérations spécifiques et à l’empaquetage des données. Les modifications permettent d’obtenir une accélération comprise entre $3.5\times$ et $5.7\times$ supérieure à une implémentation en virgule flottante 32 bits sur un noyau 3×3 sur une quantification respective de 1 et 2 bits. Son implémentation sur la technologie GF22nm permet de réduire la surface des voies de 57.5% ainsi que sa consommation d’énergie par 48% ce qui est légèrement meilleur que Sparq.

8.2 Limitations des solutions proposées

Les solutions proposées dans ce travail présentent plusieurs limitations. Du point de vue des performances présentées, tous les résultats proviennent d’une simulation au Niveau Transfert de Registres (RTL) où toutes les données sont placées dans une mémoire cache idéale de taille infinie. Or dans un contexte réaliste, les paramètres d’un modèle ne pourraient pas physiquement être contenus dans la mémoire cache. Cela signifie qu’un grand nombre de transferts entre une mémoire externe et la cache devraient être effectués afin qu’au moment où le programme ait besoin de l’information, elle soit disponible en un nombre de cycles minimal. Dans notre cas, aucun défaut de mémoire cache, où l’information nécessaire n’est pas présente dans la cache et une lecture mémoire plus lente est nécessaire, n’est pris en compte.

Un second point est le choix d’avoir conservé les représentations basse précision (< 8 bits) sur des objets mémoire de la taille d’un octet standard. Cela signifie qu’une grande proportion de l’information à chaque lecture en mémoire est inutile. En compactant les données en mémoire comme proposée dans FULLPACK [70] pour la méthode ULPPACK, les performances pourraient être améliorées significativement par la réduction de la bande passante nécessaire.

Enfin, toutes les implémentations logicielles ont été réalisées en vectorisant les calculs sur la largeur des tenseurs. Ce choix a dû être fait à cause du manque de prise en charge de certaines instructions au moment du développement des premières fonctions. Cette vectorisation permet d’obtenir de très bonnes performances sur des activations de grandes tailles. Au contraire, des activations de plus petites tailles avec un grand nombre de canaux ne tirent pas un avantage aussi important de cette méthode de vectorisation. Or les modèles présentent généralement des tenseurs avec un grand nombre de canaux, mais des hauteurs et des largeurs limitées, ce qui ne correspond pas à des cas optimaux pour nos fonctions.

8.3 Améliorations futures

Des travaux futurs pourraient s’orienter sur l’intégration de ces fonctions dans un framework, permettant l’implémentation simple de modèles complets et la vérification des résultats de précision de ces mêmes modèles. Une partie inexplorée dans ce travail détaillée dans la section 8.2 se trouve dans le choix de vectorisation des algorithmes de convolution 2D. Il serait intéressant de développer d’autres fonctions permettant de tester les performances d’Ara, Quark et Sparq en vectorisant les calculs sur les canaux plutôt que sur la largeur. Enfin, il serait particulièrement intéressant de s’orienter sur une véritable plateforme matérielle, FPGA ou ASIC afin de vérifier les performances des implémentations dans un cadre réaliste en utilisant des mémoires externes.

RÉFÉRENCES

- [1] K. He *et al.*, “Deep residual learning for image recognition,” dans *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, p. 770–778.
- [2] M. Horowitz, “1.1 computing’s energy problem (and what we can do about it),” dans *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, p. 10–14.
- [3] M. Tan et Q. Le, “EfficientNet : Rethinking model scaling for convolutional neural networks,” dans *Proceedings of the 36th International Conference on Machine Learning*, ser. *Proceedings of Machine Learning Research*, K. Chaudhuri et R. Salakhutdinov, édit., vol. 97. PMLR, 09–15 Jun 2019, p. 6105–6114. [En ligne]. Disponible : <https://proceedings.mlr.press/v97/tan19a.html>
- [4] J. Deng *et al.*, “Imagenet : A large-scale hierarchical image database,” dans *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, p. 248–255.
- [5] G. C. Sergio *et al.* (2022) Accelerating quantized networks with qat toolkit and tensorrt. [En ligne]. Disponible : <https://developer.nvidia.com/blog/accelerating-quantized-networks-with-qat-toolkit-and-tensorrt/>
- [6] S. K. Esser *et al.*, “Learned step size quantization,” dans *International Conference on Learning Representations*, 2020. [En ligne]. Disponible : <https://openreview.net/forum?id=rkgO66VKDS>
- [7] Y. Bhalgat *et al.*, “Lsq+ : Improving low-bit quantization through learnable offsets and better initialization,” 06 2020, p. 2978–2985.
- [8] M. Dukhan, Y. Wu et H. Lu, “Qnnpack : Open source library for optimized mobile deep learning,” 2018.
- [9] J. Won *et al.*, “Ulp-pack : Fast sub-8-bit matrix multiply on commodity simd hardware,” dans *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi et C. Wu, édit., vol. 4, 2022, p. 52–63. [En ligne]. Disponible : <https://proceedings.mlsys.org/paper/2022/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf>
- [10] S.-T. Lin *et al.*, “Lg-lsq : Learned gradient linear symmetric quantization,” 2022.
- [11] M. AskariHemmat *et al.*, “Quark : An integer risc-v vector processor for sub-byte quantized dnn inference,” 2023. [En ligne]. Disponible : <https://arxiv.org/abs/2302.05996>
- [12] A. Krizhevsky, V. Nair et G. Hinton, “Cifar-10 (canadian institute for advanced research).” [En ligne]. Disponible : <http://www.cs.toronto.edu/~kriz/cifar.html>

- [13] K. Simonyan et A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv 1409.1556*, 09 2014.
- [14] D. Hubel et T. Wiesel, “Receptive fields, binocular interaction, and functional architecture in the cat’s visual cortex,” *Journal of Physiology*, vol. 160, p. 106–154, 1962.
- [15] K. Fukushima, “Neocognitron : A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” *Biological Cybernetics*, vol. 36, p. 193–202, 1980.
- [16] Q. Chu *et al.*, “Online multi-object tracking using cnn-based single object tracker with spatial-temporal attention mechanism,” 10 2017, p. 4846–4855.
- [17] O. Abdel-Hamid *et al.*, “Convolutional neural networks for speech recognition,” *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 22, n°. 10, p. 1533–1545, 2014.
- [18] M. Cimpoi, S. Maji et A. Vedaldi, “Deep filter banks for texture recognition and segmentation,” dans *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, p. 3828–3836.
- [19] A. Krizhevsky, I. Sutskever et G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” dans *Advances in Neural Information Processing Systems*, F. Pereira *et al.*, édit., vol. 25. Curran Associates, Inc., 2012. [En ligne]. Disponible : https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf
- [20] W. Li, X. Zhu et S. Gong, “Harmonious attention network for person re-identification,” 02 2018.
- [21] Z. Dai *et al.*, “Coatnet : Marrying convolution and attention for all data sizes,” dans *Advances in Neural Information Processing Systems*, A. Beygelzimer *et al.*, édit., 2021. [En ligne]. Disponible : <https://openreview.net/forum?id=dUk5Foj5CLf>
- [22] Y. Lecun, J. Denker et S. Solla, “Optimal brain damage,” vol. 2, 01 1989, p. 598–605.
- [23] R. Gajjala *et al.*, “Huffman coding based encoding techniques for fast distributed deep learning,” 12 2020.
- [24] M. Nagel *et al.*, “A white paper on neural network quantization,” *ArXiv*, vol. abs/2106.08295, 2021.
- [25] J. Choi *et al.*, “Accurate and efficient 2-bit quantized neural networks,” dans *Conference on Machine Learning and Systems*, 2019.
- [26] M. AskariHemmat *et al.*, “Qreg : On regularization effects of quantization,” 2022.

- [27] M. Alwani *et al.*, “Fused-layer cnn accelerators,” dans *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, p. 1–12.
- [28] Y. Ma *et al.*, “Optimizing the convolution operation to accelerate deep neural networks on fpga,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, n^o. 7, p. 1354–1367, 2018.
- [29] Y.-H. Chen *et al.*, “Eyeriss : An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE Journal of Solid-State Circuits*, vol. 52, n^o. 1, p. 127–138, 2017.
- [30] Y.-H. Chen *et al.*, “Eyeriss v2 : A flexible accelerator for emerging deep neural networks on mobile devices,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, n^o. 2, p. 292–308, 2019.
- [31] N. Stephens *et al.*, “The ARM scalable vector extension,” *IEEE Micro*, vol. 37, n^o. 2, p. 26–39, mar 2017. [En ligne]. Disponible : <https://doi.org/10.1109%2Fmm.2017.35>
- [32] K. Asanović et A. Waterman, “The risc-v vector extension, v1.0,” 2021. [En ligne]. Disponible : <https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf>
- [33] M. Perotti *et al.*, “A new ara for vector computing : An open source highly efficient RISC-v v 1.0 vector processor design,” dans *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, jul 2022. [En ligne]. Disponible : <https://doi.org/10.1109%2Fasap54787.2022.00017>
- [34] M. Cavalcante *et al.*, “Ara : A 1-GHz+ scalable and energy-efficient RISC-v vector processor with multiprecision floating-point support in 22-nm FD-SOI,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, n^o. 2, p. 530–543, feb 2020. [En ligne]. Disponible : <https://doi.org/10.1109%2Ftvlsi.2019.2950087>
- [35] K. Patsidis *et al.*, “Risc-v 2 : A scalable risc-v vector processor,” 10 2020, p. 1–5.
- [36] Z.-Q. Zhao *et al.*, “Object detection with deep learning : A review,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, p. 1–21, 01 2019.
- [37] F. Sultana, A. Sufian et P. Dutta, “Evolution of image segmentation using deep convolutional neural network : A survey,” *Knowledge-Based Systems*, vol. 201-202, p. 106062, 2020. [En ligne]. Disponible : <https://www.sciencedirect.com/science/article/pii/S0950705120303464>
- [38] S. Di Mascio *et al.*, “On-board decision making in space with deep neural networks and risc-v vector processors,” *Journal of Aerospace Information Systems*, vol. 18, p. 1–17, 06 2021.

- [39] P.-H. Huang *et al.*, “Pops : An off-peak precomputing scheme for privacy-preserving computing,” *J. Supercomput.*, vol. 78, n°. 15, p. 16841–16860, oct 2022. [En ligne]. Disponible : <https://doi.org/10.1007/s11227-022-04552-x>
- [40] T. Abtahi, A. Kulkarni et T. Mohsenin, “Accelerating convolutional neural network with fft on tiny cores,” dans *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, p. 1–4.
- [41] J. Cong et B. Xiao, “Minimizing computation in convolutional neural networks,” dans *Artificial Neural Networks and Machine Learning – ICANN 2014*, S. Wermter *et al.*, édit. Cham : Springer International Publishing, 2014, p. 281–290.
- [42] K. Chellapilla, S. Puri et P. Simard, “High Performance Convolutional Neural Networks for Document Processing,” dans *Tenth International Workshop on Frontiers in Handwriting Recognition*, G. Lorette, édit., Université de Rennes 1. La Baule (France) : Suvisoft, oct. 2006, <http://www.suvisoft.com>. [En ligne]. Disponible : <https://inria.hal.science/inria-00112631>
- [43] A. Vasudevan, A. Anderson et D. Gregg, “Parallel multi channel convolution using general matrix multiplication,” 04 2017.
- [44] J. W. Cooley et J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” *Mathematics of computation*, vol. 19, n°. 90, p. 297–301, 1965.
- [45] A. Lavin et S. Gray, “Fast algorithms for convolutional neural networks,” dans *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, p. 4013–4021.
- [46] L. Chi, B. Jiang et Y. Mu, “Fast fourier convolution,” dans *Advances in Neural Information Processing Systems*, H. Larochelle *et al.*, édit., vol. 33. Curran Associates, Inc., 2020, p. 4479–4488. [En ligne]. Disponible : https://proceedings.neurips.cc/paper_files/paper/2020/file/2fd5d41ec6cfab47e32164d5624269b1-Paper.pdf
- [47] C. Eleftheriadis et G. Karakostas, “Energy-efficient fast fourier transform for real-valued applications,” *IEEE Transactions on Circuits and Systems II : Express Briefs*, vol. 69, n°. 5, p. 2458–2462, 2022.
- [48] A. Vasudevan, A. Anderson et D. Gregg, “Parallel multi channel convolution using general matrix multiplication,” dans *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2017, p. 19–24.
- [49] A. Karatsuba, “Multiplication of multidigit numbers on automata,” dans *Soviet physics doklady*, vol. 7, 1963, p. 595–596.
- [50] S. Winograd, *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics, 1980. [En ligne]. Disponible : <https://epubs.siam.org/doi/abs/10.1137/1.9781611970364>

- [51] V. Strassen, “Gaussian Elimination is Not Optimal (1969),” dans *Ideas That Created the Future : Classic Papers of Computer Science*. The MIT Press, 02 2021. [En ligne]. Disponible : <https://doi.org/10.7551/mitpress/12274.003.0032>
- [52] M. Baalen *et al.*, “Fp8 versus int8 for efficient deep learning inference,” 03 2023.
- [53] S. K. Venkataramanaiah *et al.*, “A 28nm 8-bit floating-point tensor core based cnn training processor with dynamic activation/weight sparsification,” dans *ESSCIRC 2022-IEEE 48th European Solid State Circuits Conference (ESSCIRC)*, 2022, p. 89–92.
- [54] X. Sun *et al.*, “Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks,” dans *Advances in Neural Information Processing Systems*, H. Wallach *et al.*, édit., vol. 32. Curran Associates, Inc., 2019. [En ligne]. Disponible : https://proceedings.neurips.cc/paper_files/paper/2019/file/65fc9fb4897a89789352e211ca2d398f-Paper.pdf
- [55] P. Micikevicius *et al.*, “Fp8 formats for deep learning,” 09 2022.
- [56] H. Wu *et al.*, “Integer quantization for deep learning inference : Principles and empirical evaluation,” 2020.
- [57] E. Park, J. Ahn et S. Yoo, “Weighted-entropy-based quantization for deep neural networks,” dans *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, p. 7197–7205.
- [58] S. Jung *et al.*, “Learning to quantize deep networks by optimizing quantization intervals with task loss,” 2018.
- [59] G. Tej Pratap, R. Kumar et N. Pradeep, “Hybrid and non-uniform quantization methods using retro synthesis data for efficient inference,” dans *2021 International Joint Conference on Neural Networks (IJCNN)*, 2021, p. 1–8.
- [60] S. Seo et J. Kim, “Efficient weights quantization of convolutional neural networks using kernel density estimation based non-uniform quantizer,” *Applied Sciences*, vol. 9, n°. 12, 2019. [En ligne]. Disponible : <https://www.mdpi.com/2076-3417/9/12/2559>
- [61] A. Abdolrashidi *et al.*, “Pareto-optimal quantized resnet is mostly 4-bit,” *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Jun 2021. [En ligne]. Disponible : <http://dx.doi.org/10.1109/CVPRW53098.2021.00345>
- [62] “Ieee standard for floating-point arithmetic,” *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, p. 1–84, 2019.
- [63] D. A. Patterson et J. L. Hennessy, *Computer Organization and Design RISC-V Edition : The Hardware Software Interface*, 1^{er} éd. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2017.

- [64] R. W. Bemer, “A proposal for a generalized card code for 256 characters,” *Commun. ACM*, vol. 2, n^o. 9, p. 19–23, sep 1959. [En ligne]. Disponible : <https://doi.org/10.1145/368424.368435>
- [65] W. Buchholz, *Planning a Computer System : Project Stretch*. USA : McGraw-Hill, Inc., 1962.
- [66] Z. Jia *et al.*, “Dissecting the nvidia turing t4 gpu via microbenchmarking,” *ArXiv*, vol. abs/1903.07486, 2019.
- [67] “Ieee standard for binary floating-point arithmetic,” *ANSI/IEEE Std 754-1985*, p. 1–20, 1985.
- [68] S. Lawson, “Vlsi signal processing : a bit-serial approach,” *Electronics and Power*, vol. 32, n^o. 2, p. 169–, 1986.
- [69] M. Cowan *et al.*, “Automatic generation of high-performance quantized machine learning kernels,” ser. CGO 2020. New York, NY, USA : Association for Computing Machinery, 2020, p. 305–316. [En ligne]. Disponible : <https://doi.org/10.1145/3368826.3377912>
- [70] H. Katebi, N. Asadi et M. Goudarzi, “Fullpack : Full vector utilization for sub-byte quantized inference on general purpose cpus,” 2022.
- [71] R. M. Russell, “The cray-1 computer system,” *Commun. ACM*, vol. 21, p. 63–72, 1978.
- [72] G. E. Moore, “Cramming more components onto integrated circuits,” *Electronics*, vol. 38, n^o. 8, April 1965.
- [73] R. Dennard *et al.*, “Design of ion-implanted mosfet’s with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, n^o. 5, p. 256–268, 1974.
- [74] M. Sato *et al.*, “Co-design for a64fx manycore processor and ”fugaku”,” dans *SC20 : International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, p. 1–15.
- [75] N. Jouppi *et al.*, “A domain-specific architecture for deep neural networks,” *Communications of the ACM*, vol. 61, p. 50–59, 08 2018.
- [76] N. P. Jouppi *et al.*, “In-datacenter performance analysis of a tensor processing unit,” 2017.
- [77] NVIDIA, “Tesla k80 gpu accelerator,” 2015. [En ligne]. Disponible : <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/Tesla-K80-BoardSpec-07317-001-v05.pdf>
- [78] S. I. Venieris et C.-S. Bouganis, “fpgaconvnet : A framework for mapping convolutional neural networks on fpgas,” dans *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, p. 40–47.

- [79] Y. Umuroglu *et al.*, “Finn : A framework for fast, scalable binarized neural network inference,” dans *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA : Association for Computing Machinery, 2017, p. 65–74. [En ligne]. Disponible : <https://doi.org/10.1145/3020078.3021744>
- [80] F. Zaruba et L. Benini, “The cost of application-class processing : Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, n°. 11, p. 2629–2640, Nov 2019.
- [81] Z. Liu *et al.*, “Bi-real net : Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm,” dans *Computer Vision – ECCV 2018*, V. Ferrari *et al.*, édit. Cham : Springer International Publishing, 2018, p. 747–763.
- [82] M. Rastegari *et al.*, “Xnor-net : Imagenet classification using binary convolutional neural networks,” vol. 9908, 10 2016, p. 525–542.
- [83] M. Courbariaux *et al.*, “Binarized neural networks : Training deep neural networks with weights and activations constrained to +1 or -1,” 2016.
- [84] M. Courbariaux, Y. Bengio et J.-P. David, “Binaryconnect : Training deep neural networks with binary weights during propagations,” dans *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA : MIT Press, 2015, p. 3123–3131.
- [85] S. Xie *et al.*, “Aggregated residual transformations for deep neural networks,” 07 2017, p. 5987–5995.
- [86] R. Gong *et al.*, “Differentiable soft quantization : Bridging full-precision and low-bit neural networks,” 10 2019, p. 4851–4860.
- [87] P. Judd *et al.*, “Stripes : Bit-serial deep neural network computing,” dans *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, p. 1–12.
- [88] C. R. Harris *et al.*, “Array programming with NumPy,” *Nature*, vol. 585, n°. 7825, p. 357–362, sept. 2020. [En ligne]. Disponible : <https://doi.org/10.1038/s41586-020-2649-2>
- [89] P. Virtanen *et al.*, “SciPy 1.0 : Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods*, vol. 17, p. 261–272, 2020.
- [90] C. Lattner et V. Adve, “Llvm : a compilation framework for lifelong program analysis transformation,” dans *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, p. 75–86.