Accélération d'Application d'IA sur un Processeur RISC-V — Breizh Team

Adrien Pellé
IMT Atlantique
Brest, France
adrien.pelle@imt-atlantique.net

Le Nam Hieu Nguyen
IMT Atlantique
Brest, France
nam.nguyen-le@imt-atlantique.net

Rémi Vassal
IMT Atlantique
Brest, France
remi-raymond.vassal@imt-atlantique.net

***Abstract*—Dans cet article, nous présentons une approche pour accélérer une application de réseau de neurones convolutif permettant la reconnaissance de chiffres manuscrits de la base de donnée MNIST [1], en modifiant la microarchitecture d'un processeur RISC-V, CVA6. Ces travaux s'inscrivent dans le cadre d'un concours étudiant organisé par Thales, le GDR SOC²[1] et le CNFM[2]. La première optimisation présentée s'appuie sur l'implémentation d'instructions de l'extension "P" (Packed SIMD) du jeu d'instruction RV32, plus particulièrement de l'instruction "SMAQA", permettant d'effectuer une opération de multiplication-accumulation en SIMD (Single Instruction, Multiple Data). La seconde optimisation porte sur l'extension du nombre de ports de lecture du register file jusqu'à 5 ports, permettant de lire 5 opérandes en parallèle. Enfin, l'utilisation d'un buffer pour stocker les valeurs des entrées a permis d'améliorer encore les performances. L'ensemble de ces optimisations a entraîné un facteur d'accélération d'environ ×6.74.**

***Index Terms*—RISC-V, CVA6, Convolutionnal Neural Network, SIMD, FPGA**

I. Introduction

L'accélération des réseaux de neurones convolutifs est devenue cruciale pour répondre aux exigences croissantes en termes de performances et d'efficacité énergétique dans le domaine de l'intelligence artificielle. Les réseaux de neurones convolutifs sont largement utilisés dans diverses applications telles que la vision par ordinateur, la reconnaissance vocale et la classification d'images. Toutefois, leur déploiement sur des architectures conventionnelles telles que les processeurs embarqués est limité en termes de vitesse et de consommation d'énergie.

Dans ce contexte, nous présentons dans ce papier notre méthode afin de réduire le temps d'exécution des réseaux de neurones convolutifs sur un processeur RISC-V. L'architecture d'instruction ouverte RISC-V, au cœur de nos travaux, suscite un intérêt croissant tant dans le milieu académique qu'industriel. Son modèle open-source et sa flexibilité permettent une personnalisation poussée, favorisant l'innovation dans la conception de systèmes embarqués.

Ces travaux s'inscrivent dans le cadre d'un concours étudiant dont l'objectif est l'optimisation de la microarchitecture du processeur CVA6, afin d'accélérer une application de réseau de neurones reconnaissant des chiffres de la base de données MNIST. Ce concours, organisé par Thales Research & Technology France, en collaboration avec le GDR SOC² et le CNFM, constitue une opportunité unique permettant aux étudiants de contribuer et d'apprendre de l'intérêt croissant pour l'architecture ouverte RISC-V.

[1]Groupement de recherche System On Chip - Systèmes embarqués et Objets Connectés

[2]Groupement d'intérêt public pour la coordination nationale de la formation en microélectronique et en nanotechnologies

II. Contexte

A. CVA6

CVA6 est un processeur open-source qui implémente l'architecture RISC-V. CVA6 peut être configuré en 32 ou 64 bits : CV32A6 et CV64A6.

Ce processeur peut être configuré selon les besoins des utilisateurs et des applications grâce à plusieurs paramètres et fonctionnalités optionnelles (MMU, PMP, FPU, organisation et taille du cache...). Il vise les technologies FPGA (Field Programmable Gate Array) et ASIC (Application Specific Integrated Circuit).

En tant que processeur, CVA6 peut supporter plusieurs systèmes d'exploitation Linux (construites avec BuildRoot et Yocto), comme FreeRTOS et Zephyr. CVA6 dispose de l'interface de coprocesseur CV-X-IF pour étendre l'ensemble des instructions qu'il peut exécuter.

En plus de supporter les instructions spécifiées par l'architecture RISC-V, CVA6 permet l'ajout d'instructions cutsomisées. Cette flexibilité permet de créer des instructions dédiées à une seule application. CV32A6 et CV64A6 partagent le même code source SystemVerilog, disponible dans ce dépôt GitHub [2].

CV64A6 est une évolution industrielle d'ARIANE créée par l'ETH Zürich et l'Université de Bologne. CV32A6 est un ajout ultérieur de Thales. CVA6 est désormais géré par l'OpenHW Group par le biais de ses membres.

Dans le cadre de ce projet, le processeur CVA6 est porté sur une cible FPGA. La synthèse logique, le placement et le routage du circuit du processeur CVA6 est effectuée à l'aide de l'outil Vivado [3].

B. Application MNIST

L'application implémentée dans le processeur CVA6 est un réseau de neurones convolutif permettant la reconnaissance de chiffres manuscrits. Les images de ces chiffres sont tirées de la base de donnée MNIST. Chacune des images a une taille de 28 pixels de coté.

Ce réseau de neurones présente 4 couches au total : 2 couches de convolution et 2 couches entièrement connectées. Les pixels des images d'entrée, sont codés sous forme d'entiers non signés sur 8 bits. Les poids sont codés sous forme d'entiers signés sur 8 bits.

La fonction au coeur de cette application est une opération de Multiplication-ACcumulation (MAC), qui multiplie une valeur d'entrée (8 bits non signé) avec une valeur de poids (8 bits signés), et ajoute le résultat à une variable d'accumulation (32 bits signé).

C. Packed SIMD "P" Extension

L'extension "P" (Packed SIMD) [4], est une extension du jeu d'instruction RISC-V, permettant d'effectuer des opérations SIMD (Single Instruction, Multiple Data) à partir de GPR (General Purpose Register). Les GPRs sont de taille 32 bits pour RV32 et 64 bits pour RV64. Pour RV32, cette spécification définit des vecteurs de 4 éléments de 8 bits chacun, ou bien de 2 éléments de 16 bits chacun. Cette extension est encore au stade de la proposition et n'est pas officiellement ratifiée dans la norme RISC-V, contrairement à l'extension Vectorielle "V".

L'une des instructions spécifiée est l'instruction SMAQA, qui prend deux vecteurs de 32 bits (4 éléments de 8 bits chacun), multiplie chaque élément des vecteurs un à un, et renvoie la somme des 4 résultats de multiplications avec la valeur lue dans le registre de destination. Cette instruction a pour particularité de prendre 3 opérandes, et non 2 opérandes. En effet, la dernière opérande est lu dans le registre destination, adressée dans le champ "rd" de l'instruction. Nous avons remarqué que l'utilisation de cette instruction permettrait d'exploiter le parallélisme de notre application, en effectuant 4 opérations MAC en une seule instruction SMAQA.

Une implémentatation de l'extension "P" a été proposée dans la thèse de Davy Koene, de TU Delft [5]. L'architecture proposée dans cette thèse supporte une grande partie des instructions de l'extension "P" pour RV64. De plus, des implémentations de la toolchain supportant l'extension P sont disponibles sur le Github de plctlab.

Notre approche a été dans un premier temps de reprendre l'architecture proposée dans la thèse de Davy Koene, en nous concentrant sur les instructions qui permettraient d'accélérer l'application. Afin d'utiliser les instructions supportées par notre nouvelle architecture, il faut modifier la toolchain afin qu'elle puisse supporter l'extension "P", nous utiliserons alors un fork de la toolchain développée par plctlab (Binutils et GCC). Par ailleurs, avec cette approche, on peut ajouter des instructions cusotmisées, en plus des instructions de l'extension P.

III. Architecture

La structure du CVA6 est composée de 6 étages : le générateur de PC (Program Counter),l'instruction fetch (Ces deux étages constituent le frontend), le décodeur, l'étage d'issue, l'étage d'exécution et enfin l'étage de commit. Pour accélérer l'application MNIST, l'étage d'éxecution a été modifié. Pour décoder les nouvelles instructions, le décodeur a été modifié. Enfin, pour pouvoir lire plus de deux opérandes à la fois, l'étage d'issue a lui aussi été modifié. Notre première approche a été de changer l'architecture de CVA6 pour qu'elle puisse supporter un certain nombre d'instructions de l'extension P. Pour y parvenir, il nous a fallu dans un premier temps modifier l'ALU (Unité arithmétique et logique) de l'étage d'éxecution, pour qu'elle puisse supporter les additions, soustractions, décalages et comparaisons. Par la suite, le multiplieur a été modifié pour supporter les opérations de multiplication de l'extension P. Enfin, le multiplicateur a été modifié pour supporter des instructions customisées, spécialement dédiées à notre application.

Dans le cadre de notre travail, nous utilisons principalement l'instruction SMAQA (Fig.1), qui nous permet d'effectuer plusieurs opérations d'addition, de multiplication et d'accumulation en même temps, ce qui entraîne une augmentation du parallélisme des données, et la réduction des horloges requises. Afin d'implémenter SMAQA, nous devons modifier 3 étages du CVA6 : décodeur d'instructions, issue et exécution. Alors que le décodeur d'instructions n'est que marginalement ajusté pour reconnaître les nouvelles instructions dans le paquet SIMD et que l'étage d'issue est discutée dans la section III.C, l'étage d'exécution est clarifiée dans différentes sections car elle subit le plus de modifications. En particulier, l'exécution contient des unités spécialisées qui sont responsables de leurs propres fonctionnalités telles que l'addition ou la multiplication. Ces unités sont directement affectées à l'exécution des calculs, ce qui signifie que ces pièces doivent être prises en compte en premier dans le cas d'une accélération des performances. Concrètement, les modifications architecturales de l'étape d'exécution se sont concentrées principalement sur l'Unité Arithmétique et Logique, spécialisée dans les additions, et sur le multiplicateur, spécialisé dans les multiplications et l'accumulation.

A. ALU

1) Additions SIMD: Dans un premier temps, l'additionneur à été modifié pour supporter les instructions d'addition et soustraction SIMD de l'extension P. Ainsi, le nouvel additionneur supporte les additions SIMD sur 4 éléments de 8 bits, ou sur 2 éléments de 16 bits. Les additions signées et non signées sont supportées. Le nouvel additioneur permet également d'effectuer des "Halving additions". Ces instructions permettent d'effectuer une addition classique suivie d'un décalage vers la droite, ce qui à pour effet de diviser par deux le résultat de l'addition. Enfin, l'additioneur permet d'effectuer des additions saturées. Ces instructions permettent de saturer le résultat en cas d'overflow. Par exemple, si la somme non signée de deux éléments de 8 bits est supérieure à $2^8 - 1 = 255$, alors l'instruction renverra 255. De même pour les additions/soustractions non signées. En cas d'overflow positif, le résultat est saturé à $2^7 - 1 = 127$. En cas d'overflow négatif, le résulat est saturé à $-2^7 - 1 = -128$.

2) Opérations de décalage SIMD: L'ALU supporte les instructions SIMD de décalage à droite et à gauche, sur des éléments de 8 et 16 bits. Cependant, les instructions de décalages avec arrondi tels que spécifiés par l'extension P ne sont pas supportées.

3) Comparaisons SIMD: L'ALU supporte les instructions de comparaison SIMD tels que spécifiées par l'extension "P". Les comparaisons SIMD signées et non signées sont supportées, sur des éléments de 8 et 16 bits. De plus les opérations de minimum et maximum SIMD ont été ajoutées. Ces instructions renvoient le maximum ou minimum, signé ou non signé, entre chaque éléments des deux opérandes.

4) Instructions de paquetage/dépaquetage: Afin de faciliter le mouvement de données, l'extension P spécifie des instructions de paquetage et de dépaquetage. Les instructions de paquetage permettent de rassembler deux éléments de 16 bits de deux registres source, pour les placer dans un registre destination. Par exemple, l'instruction PKBB16 rd rs1 rs2, prends le premier élément de chaque registre source rsi[15:0], et place l'élement de rs1 dans la partie basse du registre destination rd[15:0], et l'élement de rs2 dans la partie haute rd[31:16]. Cette instruction sera utilisée dans l'application MNIST pour la première couche de convolution. Les instruction de dépaquetage permettent, au contraire, d'extraire deux élements de 8 bits d'un registre source, pour les placer dans la partie basse et la partie haute d'un registre destination. Les instructions de paquetage et de dépaquetage spécifiées par l'extension P sont supportées par notre processeur.

B. Multiplieur

Dans l'application MNIST, les réseaux de neurones sont essentiellement les poids. Chaque poids est multiplié par son entrée correspondante (un pixel de l'image ou une sortie d'une couche précédente), puis les produits sont accumulés pour former une sortie. Par conséquent, le SMAQA dans SIMD qui permet d'effectuer simultanément une opération d'addition-multiplication avec de nombreuses entrées et poids est idéal pour améliorer les calculs. Selon le package SIMD et l'architecture de CVA6, cette opération doit être implémentée dans l'unité multiplicatrice.

Concernant le type de données, les entrées sont des entiers non signés représentés sur 8 bits, les poids sont des entiers signés 8 bits. Le SMAQA effectue donc les calculs sous forme de signe étendu pour les poids (au lieu de zéro-étendu).

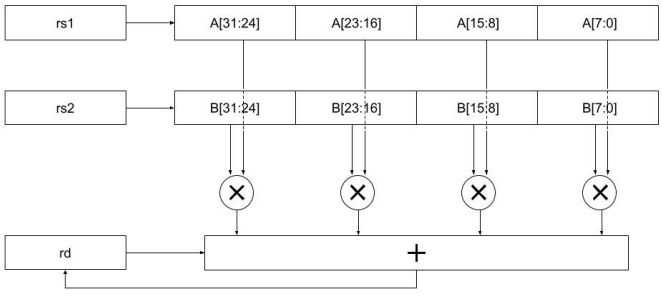

Fig. 1. Unité MAC

1) Instruction SMAQA: Dans un premier temps, l'instruction SMAQA (Figure 1) à 32 bits est implémentée, ce qui signifie que le calcul a l'opérande de 32 bits. Chaque poids et entrée sont de 8 bits, donc 4 paires d'entrées et de poids sont utilisées en même temps. L'opération SMAQA est visible sur la Fig. 1. 4 éléments de 8 bits sont enregistrés dans le GPR (Registre à usage général de longueur 32 bits), et l'instruction SMAQA prend trois opérandes en entrées : opérande A (4×8 bits), opérande B (4×8 bits) et opérande C (32 bits). Tout d'abord, chaque opérande est stockée dans un registre, puis une multiplication non signée est effectuée entre chaque élément de 8 bits. Ensuite, les résultats de chaque multiplication sont additionnés. Enfin, ils sont ajoutés à l'opérande C, et le résultat final est stocké dans le propre registre de l'opérande C, appelé registre de destination ou accumulateur. Initialement, il n'y a que 2 ports de lecture sur le fichier de registre, alors que l'instruction SMAQA utilise 3 entrées, ce qui nécessitent un port de lecture supplémentaire pour pouvoir lire l'opérande C depuis le registre de destination. Il faut donc ajouter un port de lecture au fichier de registre. L'adresse de l'opérande C est envoyée par l'étape d'émission à l'étape d'exécution, c'est la valeur lue dans le champ "rd" de l'instruction SMAQA.

Dans un deuxième temps, basant l'idée de SMAQA avec 32 bits, SMAQA avec 64 bits sont implémentés pour améliorer le parallélisme. Toutes les modifications sont effectuées dans la même conception, ce qui entraîne le fonctionnement de 5 opérandes avec 5 ports de lecture.

Pour utiliser un troisième opérande pour l'accumulation dans l'instruction SMAQA. En reproduisant la même architecture plusieurs fois, on peut augmenter les performances du processeur. En effet, l'opération SMAQA effectue une multiplication-accumulation SIMD avec des entrées sur des vecteurs de 32 bits, et des éléments de 8 bits. Pour augmenter les performances, une solution peut être d'augmenter la taille des vecteurs. Pour y parvenir, notre approche à été d'ajouter des opérandes en entrée du multiplieur SIMD.

2) Instruction SMAQA64: L'instruction SMAQA64 execute une opération MAC SIMD sur 64 bits , on utilisant deux unités MAC, la première prend en entrée l'opérande A, B et C, et la seconde prends les opérande D et E. Le résultat des deux unités est alors sommé et stocké dans le registre rd, comme pour l'instruction SMAQA. Pour permettre à l'étage d'exécution d'être alimenté avec 5 opérandes, nous avons utilisé 5 ports de lecture sur le register file. Cela demande une modification de l'issue stage, en particulier de l'issue read operands et du scoreboard. Les instructions RV32 étant limitées à 3 champs d'adressage des GPR (rs1, rs2, rd), il n'est pas possible d'adresser deux autres registre avec ce type d'instruction. Pour contourner le problème, nous utilisons des paires de registre. En effet, l'instruction SMAQA64 a10, a1, a3 prends comme opérandes : opérande A : valeur dans a1 (rs1) opérande B : valeur dans a3 (rs2) opérande C : valeur dans a10 (rd) opérande D : valeur dans a2 (rs1 + 1) opérande E : valeur dans a4 (rs2 + 1)

Dans notre cas, les registres rs1 et rs1+1 prennent des valeurs d'entrées, alors que rs2 et rs2+2 prennent des valeurs de poids.

On peut alors étendre cette logique pour adresser encore plus de registres. Nous avons essayé d'augmenter le nombre de ports de lecture du register file jusqu'à 9 ports, afin d'effectuer une opération MAC sur 8 éléments de 8 bits à la fois. Cependant, cela entraîne une augmentation du chemin critique et une violation des contraintes temporelles du processeur.

3) Instruction SMAQA128: Afin d'augmenter le parallélisme, on peut utiliser 4 unités MAC, prenant chacune deux opérandes : une entrée et un poids. En comptant l'opérande d'accumulation, il faut 9 opérandes au total. Cependant, comme nous ne pouvons utiliser que 9

opérande, nous utilisons un buffer pour enregistrer les entrées lors de l'itération de la première sortie. En effet, comme lorsqu'à chaque itération de la boucle calculant un élément de sortie, on réutilise les même entrées, on peut enregistrer les valeurs des entrées dans le buffer lors de la première itération. Il suffit, lors de prochaines itérations, de lire les valeurs d'entrées dans le buffer, sans avoir besoin de charger à nouveau les entrées dans des GPR en utilisant des instructions "load". Ainsi, l'instruction SMAQA128 (Figure 2 prends 4 opérandes. Ces opérandes correspondent à une sucession de 4 * 4 = 16 poids. Ces poids sont multipliés avec les entrées enregistrées dans le buffer d'entrées lors de la première itération. A chaque appel de l'instruction SMAQA128, 16 entrées sont multipliées avec 16 poids, et les résultats sont additionnés. Le résultat est alors accumulé dans un registre du multiplieur. A la fin de la boucle, la valeur de l'accumulateur est extraite, aditionnée au biais puis saturée.

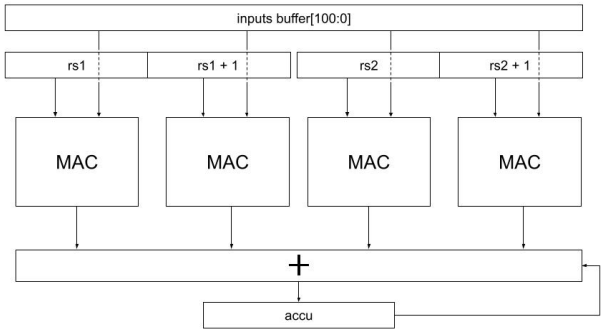

Fig. 2. Support pour l'instruction SMAQA128

IV. Software

A. Toolchain

Afin de supporter la nouvelle architecture, la toolchain doit être modifiée pour pouvoir générer les instructions P. Nous avons utilisé un fork du dépôt github de plctlab. Ce dépôt est une branche de gcc et binutils, supportant l'extension P. Ainsi, avec cette nouvelle toolchain, le compilateur génère automatiquement les instruction P, sans que nous ayons à les appeler avec de l'inline assembly. De plus, cette toolchain nous permet d'utiliser les fonctions intrinsèques de l'extension P. Ces instrinsèques nous permettent d'appeler les instructions P sans avoir à utiliser la directive "asm volatile".

Cependant, cette toolchain génère toutes les instructions définies par l'extension P. Cependant, nous n'avons pas implémenté le support matériel pour toutes ces instructions. En conséquence, les instructions générées par le compilateur et qui ne sont pas supportées par l'architecture du processeur lèvent des exceptions d'instructions illégales à l'éxecution. Pour contourner ce problèmre, nous avons modifié le compilateur afin qu'il ne génère pas certaines instructions. Nous avons alors désactivé les patterns des instructions non désirées, en modifiant les fichiers du compilateur gcc.

B. P extension instructions

L'extension P définie un total de 332 instructions. Notre processeur supporte 80 instructions telles que spécifiées par l'extension, et 81 si l'on compte l'instruction SMAQA qui a été modifiée pour être adaptée à notre application. Au total, 24.4% des instructions de l'extension P est supportée par notre processeur (Listing 1).

Additions/soustraction (40 total):
ADD16, SUB16, ADD8, SUB8, RADD8,
RSUB8, RADD16, RSUB16, URADD8, URSUB8,
URADD16, URSUB16, KADD16, UKADD16, KSUB16,
UKSUB16, KADD8, UKADD8, KSUB8, UKSUB8,
CRAS16, RCRAS16, URCRAS16, KCRAS16, UKCRAS16,
CRSA16, RCRSA16, URCRSA16, KCRSA16, UKCRSA16,
STAS16, RSTAS16, URSTAS16, KSTAS16, UKSTAS16,
STSA16, RSTSA16, URSTSA16, KSTSA16, UKSTSA16

Décalage (6 total):
SRA16, SRL16, SLL16, SRA8, SRL8, SLL8

Comparaison (10 total):
CMPEQ16, SCMPLT16, SCMPLE16, UCMPLT16,
UCMPLE16, CMPEQ8, SCMPLT8, SCMPLE8,
UCMPLT8, UCMPLE8

Maximum/minimum (8 total):
SMIN8, UMIN8, SMAX8, UMAX8,
SMIN16, UMIN16, SMAX16, UMAX16

Dépaquetage (10 total):
SUNPKD810, SUNPKD820, SUNPKD830, SUNPKD831,
SUNPKD832, ZUNPKD810, ZUNPKD820, ZUNPKD830,
ZUNPKD831, ZUNPKD832

Paquetage (4 total):
PKBB16, PKBT16, PKTB16, PKTT16

Saturation (2 total):
SCLIP32, UCLIP32

Multiplication/accumulation (modifiée, 1 total):
SMAQA

Listing 1. Liste des 81 instructions supportées de l'extension P

Notre première approche était d'implémenter une partie des instructions de l'extension P pour profiter de la génération automatique des instructions par le compilateur, et d'éventuelles optimisations. Cependant, la majeure partie de nos gains en performance est liée.

C. Custom instructions

Pour optimiser l'application MNIST spécifiquement, plusieurs instructions customisées sont supportées par le processeur :

- SMAQA64 :
  Cette instruction reprend la même architecture que l'instruction SMAQA, mais une unité MAC en plus, alimentée par deux opérandes supplémentaires. Ainsi, cette instruction permet d'effectuer une opération de multiplication-accumulation sur 8 entrées et 8 poids à la fois, avec un opérande pour l'accumulation. De plus, lorsque cette instruction est exécutée, les valeurs des entrées sont enregistrées dans un buffer. Les entrées enregistrées seront consommées lors de l'exécution de l'instruction SMAQA128.
- SMAQA128 :
  Cette instruction reprend la même architecture que

TABLE I
COMMPARAISON DE PERFORMANCES ENTRE DIFFÉRENTES ARCHITECTURES DE CVA6 POUR L'APPLICATION MNIST, SUR L'EVAL BOARD ZYBO-Z7

Instance		MNIST(Golden)	MNIST(SMAQA32)	MNIST(SMAQA64)	**MNIST(SMAQA128)**
Nombre de ports de lecture du Register File		2	3	5	5
CONV1	Nombre d'instructions	207 537	70 683	70 507	70 507
	Réduction (instructions)	0%	-65.9%	-66.0%	-66.0
	Nombre de cycles d'horloge	296 874	98 385	98 170	98 137
	Réduction (cycles)	0%	-66.9%	-66.9%	-66.9
CONV2	Nombre d'instructions	1 098 263	253 463	166 692	69 179
	Réduction (instructions)	0%	-76.9%	-84.8%	-93.7%
	Nombre de cycles d'horloge	1 468 114	435 107	294821	158 586
	Réduction (cycles)	0%	-70.4%	-79.9%	-89.2%
FC1	Nombre d'instructions	408 170	91 518	53118	24 057
	Réduction (instructions)	0%	-77.6%	-87.0%	-94.1%
	Nombre de cycles d'horloge	559 462	174 671	111 199	76 403
	Réduction (cycles)	0%	-68.8%	-80.1%	-86.3%
FC2	Nombre d'instructions	10 656	10 656	10 656	10 656
	Réduction (instructions)	0%	0%	0%	0%
	Nombre de cycles d'horloge	14 804	14 865	14 826	14 867
	Réduction (cycles)	0%	0%	0%	0%
Autre	Nombre d'instructions	432	65	816	50
	Réduction (instructions)	0%	-85.0%	+88.9%	-86.1
	Nombre de cycles d'horloge	67	207	8 897	1 545
	Réduction (cycles)	0 %	+223.4%	+13 801.5%	+2 314.1%
Total	Nombre d'instructions	1 731 593	426 385	301 789	174 449
	Réduction (instructions)	0%	-75.4%	-82.6%	-89.9%
	Nombre de cycles d'horloge	**2 354 247**	723 235	527 913	**349 538**
	Réduction (cycles)	**0%**	-69.3%	-77.6%	**-85.2%**

l'instruction SMAQA64, mais avec deux unités MAC en plus, pour un total de 4 unités MAC. Pour alimenter ces unités, il faut 4 vecteurs d'entrées et 4 vecteurs de poids (ces vecteurs sont constitués de 4 éléments de 8 bits chacun). Ainsi, cette instruction permet d'effectuer une opération de multiplication-accumulation sur 16 valeurs d'entrées et 16 valeurs de poids à la fois. L'instruction SMAQA128 prends 4 opérandes, qui sont les valeurs de 4 vecteurs de poids consécutifs, soit 16 valeurs de poids consécutives. Les vecteurs d'entrées sont consommées par les unités MAC sont fournis par le buffer d'entrées, précédemment chargé lors des exécutions consécutives de l'instruction SMAQA64. Contrairement aux instructions SMAQA précédentes, la valeur de la somme n'est pas propagée à l'aide d'un opérande supplémentaire. Le résultat de l'instruction SMAQA128 est ajoutée dans un accumulateur, qui sera lu et remis à zéro lors de l'exécution de l'instruction "SMAQA320".

V. RÉSULTATS

Tout les résultats présentés ont été obtenus avec l'outil Vivado v2021.1 (64-bit) [3].

A. Performances

Nous présentons les résulats pour différentes implémentations. Tout d'abord, les résultats de l'application MNIST de référence seront présentées. Puis, les résultats avec l'instruction SMAQA (sur des éléments de 32 bits, avec 3 ports de lecture au registerfile), les résultats avec l'instruction SMAQA64 (5 ports de lectures sur le register file). Enfin, les résultats avec l'instruction SMAQA128 sont présentés.

Nous comparons les performances de chaque fonction du réseau de neurone :

- CONV 1 : Première couche de convolution
- CONV 2 : Seconde couche de convolution
- FC1 : Première couche fully connected
- FC2 : Seconde couche fully connected

Les résultats de performance sont présentés dans le Tableau I. Pour chaque cas, le chiffre **4** est reconnu, avec une probabilité de **82**.

Notre première approche pour accélérer l'application MNIST a été de modifier la microarchitecture de CVA6 afin qu'elle puisse supporter l'extension P. Finalement, ces modifications ont d'ajouter jusqu'à 81 instructions de l'extension P. Cependant, la seule instruction ayant eu un impact significatif sur les performances est l'instruction SMAQA. En effet, cette instruction à permis de diminuer de 69.3% le nombre de cycles d'horloge nécéssaires à l'éxécution l'application MNIST. Dans un second temps, nous avons ajouté des instructions customisées à notre processeur. La première instruction customisée ajoutée est SMAQA64, qui utilise deux unités MAC (au lieu d'une seule pour SMAQA), et est alimentée par 5 opérandes. L'utilisation de 5 opérandes pour cette instruction à été rendue possible en ajoutant 2 ports de lecture au register file. SMAQA64 à permis de diminuer de 77.6% le nombre de cycles d'horloges. Enfin, le multiplieur à été modifié afin qu'il enregistre les opérandes d'inputs de l'instruction SMAQA64 dans un buffer d'inputs. La dernière instruc-

tion ajoutée, SMAQA128, s'appuie sur 4 unités MAC en parallèles. Les ports de "poids" de ces unités sont alimentés par les 4 opérandes de l'instruction SMAQA128 (rs1, rs2, rs1+1, rs2+2), tandis que les ports "d'entrées", sont alimentés par le buffer d'entrées. Finalement, cette dernière optimisation nous permet de réduire le nombre cycles d'horloges de 85.2%, passant de 2 354 247 cycles pour l'application originale, à 349 538 cycles avec nos optimisations. Cela correspond à une facteur d'accélération d'environ ×6.74.

B. Utilisation des ressources

Les optimisations apportées au processeur ont eu pour conséquence de pratiquement doubler le nombre de LUT (Look-up tables), et de doubler le nombre de Flips-flops. Les unités dont l'augmentation des ressources est la plus significative sont le multiplieur et l'étage d'issue. L'augmentation des ressources de l'issue stage est liée à l'utilisation de deux opérandes supplémentaires.

TABLE II
NOMBRE DE LUTS ET FLIP-FLOPS DE CVA6, AVANT ET APRÈS OPTIMISATION

	Instance	Valeur
Total LUTs	CVA6(référence)	8518
	CVA6(optimisé)	16 049
	Augmentation	+88.4%
Logic LUTs	CVA6(référence)	8302
	CVA6(optimisé)	15 877
	Augmentation	+91.2%
LUTRAMs	CVA6(référence)	216
	CVA6(optimisé)	172
	Augmentation	-20.4%
FFs	CVA6(référence)	4603
	CVA6(optimisé)	10 003
	Augmentation	+117.3%

C. Analyse temporelle

Les modifications apportées à la microarchitecture ont eu un impact sur le chemin critique, et par conséquent, ont entrainé une diminution de la fréquence maximale du processeur. L'ajouts de ports de lecture au register file et les modificactions apportées au multiplieur ont rallongé le chemin critique. La fréquence maximale est diminuée de 3.78%. Nous avons essayé d'étendre le nombre de ports de lecture du register file à 9 ports au total. Cependant, la fréquence maximale était alors diminué de plus de 20%, ce qui viole les contraintes temporelles imposées par le concours.

TABLE III
MESURES DU CHEMIN CRITIQUE ET DE LA FRÉQUENCE MAXIMALE, AVANT ET APRÈS OPTIMISATION

	Instance	Valeur
Slack (MET)	CVA6(référence)	0.907ns
	CVA6(optimisé)	0.157ns
	Changement	-82.7%
f_{max}	CVA6(référence)	52.38 MHz
	CVA6(optimisé)	50.40 MHz
	Changement	-3.78%

VI. CONCLUSION

Les améliorations apportées à la microarchitecture de CV32A6 ont permis une accélération significative de l'application MNIST, environ ×6.74. L'intégration de l'extension SIMD "P" et la conception de nouvelles instructions personnalisées ont réduit les cycles d'horloge de 85.2%, prouvant l'efficacité des approches de parallélisation des données dans les architectures RISC-V. Ce travail souligne le potentiel des processeurs RISC-V personnalisés pour les applications d'IA, offrant une voie prometteuse vers des systèmes plus efficaces. Pour aller plus loin, le nombre de ports de lectures du register file peut être augmenter pour pouvoir utiliser davantage d'opérandes et utiliser plus d'unités MAC en parallèle. Cependant, cela entrainerait une augmentation significative du chemin critique, menant à une violation de contraintes temporelles du processeur.

VII. REMERCIEMENTS

Nous tenons à remercier l'équipe d'enseignants de l'école IMT Atlantique et du laboratoire Lab-STICC[3] qui nous a accompagnés tout au long de ce concours : Yehya Nasser, Amer Baghdadi, Stefan Weithoffer et Ali Al Ghouwayel. Leur soutien et leurs conseils ont été déterminants pour mener à bien ce projet. Nous tenons également à remercier chaleureusement Amer Baghdadi et Ali Al Ghouwayel pour leur précieuse contribution à la rédaction de cet article.

Nous souhaitons également adresser nos remerciements aux organisateurs de la compétition pour nous avoir donné l'opportunité de participer à ce projet.

REFERENCES

[1] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.
[2] Thales, "National RISC-V student contest CV32A6." https://github.com/ThalesGroup/cva6-softcore-contest, 2023.
[3] AMD-Xilinx, "Vivado Design Suite 2020.1." https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html.
[4] C. Chang, "RISC-V P extension specification." https://github.com/riscv/riscv-p-spec.
[5] D. Koene, "Implementation and evaluation of Packed-SIMD instructions for a RISC-v processor," 2021.

[3]équipe 2AI (Algorithm-Architecture Interaction)