Accélérer les applications d'intelligence artificielle sur un processeur RISC-V

Paul BOUCHARD
Polytech Sorbonne
Paris, France
paul.bouchard@etu.sorbonne-universite.fr

Lucas GAUVAIN
Polytech Sorbonne
Paris, France
lucas.gauvain@etu.sorbonne-universite.fr

Christopher AL KHAWAND
Polytech Sorbonne
Paris, France
christopher.al_khawand@etu.sorbonne-universite.fr

Paris, 13 mai 2024

Le concours national des étudiants RISC-V, organisé par Thales, le GDR SOC² et le CNFM, visait à accélérer l'algorithme de reconnaissance de chiffres MNIST en modifiant le cœur processeur RISC-V CV32A6. Notre équipe Polytech Sorbonne a optimisé le processeur avec des instructions assembleur personnalisées, combinant SystemVerilog et langage C, pour réduire le nombre de cycles nécessaires à la reconnaissance tout en maintenant une haute précision. L'accent a été mis sur la reconnaissance du chiffre 4 et la compatibilité avec la carte FPGA Zybo-Z7. Ce concours illustre l'importance de l'optimisation architecturale pour les applications d'intelligence artificielle embarquée sur des processeurs de type RISC-V.
Mots clés : RISC-V, FPGA, Vitis, MNIST, Processeur

1 Introduction

Thales, en collaboration avec le GDR SOC² et le CNFM, organise la $4^{ème}$ édition du concours national des étudiants RISC-V. Thales est une entreprise de premier plan dans les systèmes d'information critiques pour les domaines de la sécurité, de la défense, de l'aérospatiale et du transport terrestre. Cette compétition s'adresse aux étudiants intéressés par l'électronique, les architectures informatiques et l'intelligence artificielle embarquée, et qui souhaitent relever le défi de la modification architecturale du cœur de processeur RISC-V CV32A6 dans le but d'accélérer une application d'intelligence artificielle, en l'occurrence la reconnaissance de chiffres de la base de données MNIST. Le problème posé aux étudiants est le suivant : accélérer l'algorithme de reconnaissance de chiffres de la base de données MNIST en apportant des évolutions architecturales au cœur du processeur CV32A6. Les équipes participantes doivent proposer des solutions qui minimisent le nombre de cycles nécessaires pour reconnaître un ensemble donné de chiffres. Les contraintes imposées incluent notamment la taille de la conception matérielle devant s'adapter à la carte FPGA Zybo-Z7, le maintien des probabilités de reconnaissance des chiffres à 82% et la limitation des modifications apportées au code source de l'application MNIST tout en gardant la fréquence maximale du FPGA à ±20% de la fréquence maximale de base. La plus grande partie du processeur CV32A6 a été écrite en SystemVerilog. Le code de l'algorithme d'intelligence artificielle à quant a lui été écrit en langage C. L'objectif est donc d'utiliser ces deux langages pour réaliser des instructions assembleur personnalisées.

2 Methodologie du projet

Dans tout ce qui suit, la formule suivante est utilisée pour calculer le pourcentage relatif d'optimisation :
$\%_{reduction} = 100 \times (cycles_{initiaux} - cycles_{optimises})/cycles_{initiaux}$

2.1 Lancement du code de base

En faisant tourner le code source fourni sur la carte FPGA, nous obtenons les résultats suivants dans un terminal minicom :

- Chiffre reconnu : 4
- Crédence : 82
- En simulation :
  - 1 731 593 instructions
  - 2 316 644 cycles machine
- Sur FPGA :
  - 1 731 593 instructions
  - 2 354 247 cycles machine
- $f_{max} = 50$ MHz
- 8582 LUTs
- 4611 FFs
- 16 RAMB36

L'objectif du projet étant de minimiser le nombre de cycles machine, nous devons donc profiler le code C de l'algorithme afin de déterminer les sections du code les plus onéreuses.

2.2 Profilage du code

Afin de profiler le code de l'algorithme fourni, nous avons développé deux fonctions, "measure()" et "endMeasure()", qui permettent de calculer le nombre de cycles entre leurs appels respectifs.

```
void measure(const char name[100]) {
    unsigned long startIgnore = read_csr(mcycle);
    callStackIndex++;
    strncat(callPath, name, 99);
    callStack[callStackIndex].startCycle = read_csr(
        mcycle);
    unsigned long endIgnore = read_csr(mcycle);
    callStack[callStackIndex].wastedCycles =
        endIgnore - startIgnore;
}

void endMeasure() {
    unsigned long startIgnore = read_csr(mcycle);
    unsigned long delta = read_csr(mcycle) -
        callStack[callStackIndex].startCycle;
    if (delta >= callStack[callStackIndex].
        wastedCycles) {
        delta -= callStack[callStackIndex].
            wastedCycles;
    }
    printf("%s %lu\n", callPath, delta);
    callStackIndex--;
    if (callStackIndex >= 0) {
        for (int i = strlen(callPath) - 1; i >= 0; i
            --) {
            if (callPath[i] == '/') {
                callPath[i] = '\0';
                break;
            }
        }
        unsigned long endIgnore = read_csr(mcycle);
        callStack[callStackIndex].wastedCycles +=
            endIgnore - startIgnore + callStack[
            callStackIndex + 1].wastedCycles;
    }
}
```

Nous les avons ensuite utilisés autour de chaque boucle et appel de fonction dans le code C fourni. A l'aide d'un script python que nous avons développé pour interpréter ces données, nous obtenons le rapport de profilage suivant :

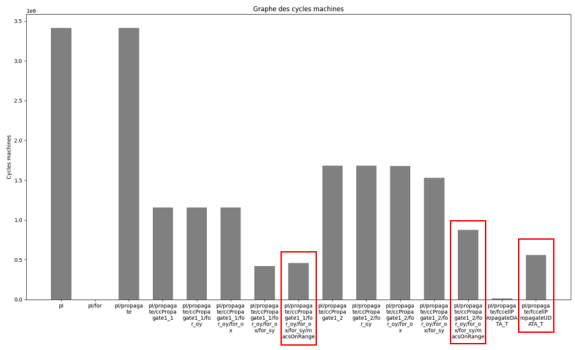

FIGURE 1 – Profilage du code de base

Ce premier profilage nous indique que la fonction macsOnRange consomme beaucoup de cycles machine (1 800 000 cycles). Nous allons ainsi l'améliorer.
N.B. : le nombre de cycles machine indiqué par notre profileur peut différer du nombre de cycles machine réel. Ce que l'on cherche surtout ici est de trouver la fonction la plus onéreuse relativement aux autres fonctions.

2.3 Création d'une instruction personnalisée

En analysant le code, nous avons découvert que les inputs et les weights sont des entiers codés sur 8 bits. Cependant, les registres du processeur CV32A6 sont eux codés sur 32 bits. Ainsi, à chaque itération dans la fonction macsOnRange, nous n'utilisons pas les 24 bits restant de chaque registre. Notre idée d'amélioration consiste ainsi à paralléliser la multiplication et la sommation, en créant notre propre instruction que l'on appelle MAD pour 'Multiply and Add'.

L'appel de cette custom instruction se fera de la manière suivante : MADUS(Rd, R1, R2). En théorie et dans un monde parfait, cette instruction permettra de passer d'environ 1 800 000 cycles à 450 000 cycles (4 fois moins de cycles machine).

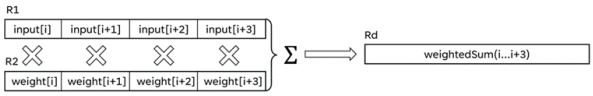

FIGURE 2 – Représentation de l'instruction MAD

En pratique, ce modèle ne fonctionne pas toujours et certains problèmes apparaissent, notamment au niveau des offsets des tableaux passés. En effet, dans notre version de la fonction macsOnRange, on convertit les tableaux d'octets passés en paramètre en tableaux de mots (4 octets). Cette approche ne marche que si l'adresse initiale des tableaux passés est un multiple de 4, sinon on tombe sur un exception handler vu que le processeur ne peut pas faire des loads non-alignés (donc chevauchants 2 cases mémoire), comme illustré sur les figures ci-dessous.

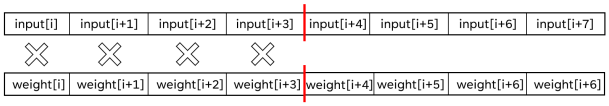

FIGURE 3 – Alignement mémoire correct

Le problème d'alignement mémoire se représente ainsi :

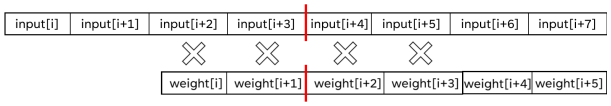

FIGURE 4 – Problème d'alignement mémoire

Pour pallier à ce problème d'alignement mémoire, nous vérifions l'alignement avant d'exécuter l'instruction MAD. Cette vérification entraîne cependant un coût assez conséquent de cycles machine. Nous modifions la fonction macsOnRange de la sorte :

```
#ifndef MADUS
#define MADUS(Rd, R1, R2) asm volatile(".insn␣r␣0
    x33,␣0x0,␣0x03,␣%0,␣%1,␣%2\n":"=r"(Rd):"r"(R1
    ),"r"(R2):);
#endif

static void macsOnRange(const UDATA_T* __restrict
    inputs,
                        const WDATA_T* __restrict
                            weights,
                        SUM_T* __restrict
                            weightedSum,
                        int nb_iterations)
{
    int32_t weightedSumRes;
    int nb_iter4 = ((uintptr_t)inputs%4 == 0 && (
        uintptr_t)weights%4 == 0) ? nb_iterations
        /4 : 0;
    uint32_t* inputs32 = (uint32_t*)inputs;
    int32_t* weights32 = (int32_t*)weights;
    for (int iter = 0; iter < nb_iter4; ++iter) {
        MADUS(weightedSumRes, inputs32[iter],
            weights32[iter]);
        *weightedSum += weightedSumRes;
    }

    int nb_iter2 = nb_iterations/2;
    uint16_t* inputs16 = (uint16_t*)inputs;
    int16_t* weights16 = (int16_t*)weights;
    for (int iter = nb_iter4*2; iter < nb_iter2;
        ++iter) {
        MADUS(weightedSumRes, inputs16[iter],
            weights16[iter]);
        *weightedSum += weightedSumRes;
    }
}
```

L'intégration de cette instruction personnalisée dans le programme C a conduit aux résultats suivants lors de l'exécution du programme sur la carte :

- Chiffre reconnu : 4
- Crédence : 82
- 481 730 instructions (-72% par rapport au programme de base)
- 803 345 cycles machine (-66% par rapport au programme de base)

Cette fonction peut cependant encore être améliorée, notamment en traitant le cas du nombre d'itérations égal à 4 à part puisqu'une boucle for génère beaucoup d'instructions inutiles pour ce nombre d'itérations.

```
#ifndef MADUS
#define MADUS(Rd, R1, R2) asm volatile(".insn␣r␣0
    x33,␣0x0,␣0x03,␣%0,␣%1,␣%2\n":"=r"(Rd):"r"(R1
    ),"r"(R2):);
#endif

static void macsOnRange(const UDATA_T* __restrict
    inputs,
                        const WDATA_T* __restrict
                            weights,
                        SUM_T* __restrict
                            weightedSum,
                        int nb_iterations)
```

```
{
    int32_t weightedSumRes;
    if (nb_iterations == 4) {
        if ((uintptr_t)inputs%4 == 0 && (uintptr_t
             )weights%4 == 0) {
            MADUS(weightedSumRes, ((uint32_t*)
                 inputs)[0], ((int32_t*)weights)
                 [0]);
            *weightedSum += weightedSumRes;
        } else {
            MADUS(weightedSumRes, ((uint16_t*)
                 inputs)[0], ((int16_t*)weights)
                 [0]);
            *weightedSum += weightedSumRes;
            MADUS(weightedSumRes, ((uint16_t*)
                 inputs)[1], ((int16_t*)weights)
                 [1]);
            *weightedSum += weightedSumRes;
        }
    } else {
        int nb_iter4 = ((uintptr_t)inputs%4 == 0
             && (uintptr_t)weights%4 == 0) ?
             nb_iterations/4 : 0;
        uint32_t* inputs32 = (uint32_t*)inputs;
        int32_t* weights32 = (int32_t*)weights;
        for (int iter = 0; iter < nb_iter4; ++iter
             ) {
            MADUS(weightedSumRes, inputs32[iter],
                 weights32[iter]);
            *weightedSum += weightedSumRes;
        }

        int nb_iter2 = nb_iterations/2;
        uint16_t* inputs16 = (uint16_t*)inputs;
        int16_t* weights16 = (int16_t*)weights;
        for (int iter = nb_iter4*2; iter <
             nb_iter2; ++iter) {
            MADUS(weightedSumRes, inputs16[iter],
                 weights16[iter]);
            *weightedSum += weightedSumRes;
        }
    }
}
```

L'intégration de cette amélioration a conduit aux résultats suivants lors de l'exécution du programme sur la carte :

- Chiffre reconnu : 4
- Crédence : 82
- En simulation :
  - 473 876 instructions (-72% par rapport au programme de base)
  - 747 449 cycles machine (-66% par rapport au programme de base)
- Sur FPGA :
  - 473 876 instructions (-72% par rapport au programme de base)
  - 784 094 cycles machine (-66% par rapport au programme de base)
- $f_{max} = \frac{1}{20ns+0.377ns} = 49.07$ MHz
- 9541 total LUTs
- 9315 logic LUTs
- 226 LUTRAMs
- 4848 FFs
- 16 RAMB36
- 1 RAMB18

2.4 Unité de calculs vectoriels

Nous avons remarqué que la grande majorité des appels à macsOnRange se faisaient avec un nombre d'itérations égal à 4, pour lesquelles la custom instruction MAD est suffisante et rapide. Par contre, il reste quand même plusieurs appels à cette fonction avec 80, 96 et 150 itérations, pour lesquels une unité de calculs vectoriels pourrait être très bénéfique.
La prochaine étape que nous avons donc envisagé est la création d'une unité de calculs vectoriels qui, au lieu de paralléliser 4 calculs, parallélise 128 calculs.
Nous avons donc créé une unité de calculs vectoriels qui offre 3 instructions assembleur :

- MADUIV(R1, Imm) : MAD unsigned input vectorial, qui permet d'envoyer un nombre 'Imm' d'inputs (chacun étant un uint8) vers l'unité de calculs vectoriels
- MADSWV(R1, Imm) : MAD signed weight vectorial, qui permet d'envoyer un nombre 'Imm' de poids (chacun étant un uint8) vers l'unité de calculs vectoriels
- MADEV(Rd) : MAD execute vectorial, qui permet d'effectuer les 128 multiplications-additions et de retourner le résultat

Nous avons fait le choix intentionnel de ne pas utiliser de boucles for pour transférer les inputs et les poids vers l'unité de calculs vectoriels pour éviter les instructions assembleurs non-nécessaires de contrôle de boucle. On charge également les données par paquets de 2 pour éviter les problèmes d'alignement mémoire évoqués précédemment (et vu que nos tests ont révélés que pour la configuration actuelle de l'algorithme donné, les données sont toujours alignées sur 2 octets).
Nos tests ont également révélés que le nombre d'itérations le plus souvent passés en paramètre sont 4, 80 et 96. Nous allons donc les traiter au cas-par-cas dans le if, et traiter tous les autres nombres d'itérations dans le else à l'aide de l'instruction MAD, comme illustré dans le code suivant :

```
#ifndef MADUS
#define MADUS(Rd, R1, R2) asm volatile(".insn␣r␣0
     x33,␣0x0,␣0x03,␣%0,␣%1,␣%2\n":"=r"(Rd):"r"(R1
     ),"r"(R2):);
#endif

#ifndef MADUIV
#define MADUIV(R1, Imm) asm volatile(".insn␣s␣0x23
     ,␣0x4,␣%1,␣%2(a0)\n":"=r"(R0):"r"(R1),"i"(Imm
     ):);
#endif

#ifndef MADSWV
#define MADSWV(R1, Imm) asm volatile(".insn␣s␣0x23
     ,␣0x5,␣%1,␣%2(a0)\n":"=r"(R0):"r"(R1),"i"(Imm
     ):);
#endif

#ifndef MADEV
#define MADEV(Rd) asm volatile(".insn␣r␣0x33,␣0x1,
     ␣0x03,␣%0,␣a0,␣a0\n":"=r"(Rd):);
#endif

static void macsOnRange(const UDATA_T* __restrict
     inputs,
                        const WDATA_T* __restrict
                             weights,
                        SUM_T* __restrict
                             weightedSum,
                        int nb_iterations)
{
    int32_t weightedSumRes;
    if (nb_iterations == 4) {
        MADUS(weightedSumRes, ((uint16_t*)inputs)
             [0], ((int16_t*)weights)[0]);
        *weightedSum += weightedSumRes;
        MADUS(weightedSumRes, ((uint16_t*)inputs)
             [1], ((int16_t*)weights)[1]);
        *weightedSum += weightedSumRes;
    } else if (nb_iterations == 80) {
        MADUIV(((uint16_t*)inputs)[0], 2);
        MADUIV(((uint16_t*)inputs)[1], 2);
        ...
        MADUIV(((uint16_t*)inputs)[38], 2);
        MADUIV(((uint16_t*)inputs)[39], 2);

        MADSWV(((uint16_t*)weights)[0], 2);
        MADSWV(((uint16_t*)weights)[1], 2);
        ...
        MADSWV(((uint16_t*)weights)[38], 2);
        MADSWV(((uint16_t*)weights)[39], 2);

        MADEV(weightedSumRes);

        *weightedSum += weightedSumRes;
    } else if (nb_iterations == 96) {
        MADUIV(((uint16_t*)inputs)[0], 2);
        MADUIV(((uint16_t*)inputs)[1], 2);
        ...
        MADUIV(((uint16_t*)inputs)[46], 2);
        MADUIV(((uint16_t*)inputs)[47], 2);

        MADSWV(((uint16_t*)weights)[0], 2);
        MADSWV(((uint16_t*)weights)[1], 2);
        ...
        MADSWV(((uint16_t*)weights)[46], 2);
        MADSWV(((uint16_t*)weights)[47], 2);
        MADEV(weightedSumRes);

        *weightedSum += weightedSumRes;
    } else {
        int nb_iterations2 = nb_iterations/2;
        for (int iter = 0; iter < nb_iterations2;
             ++iter) {
            MADUS(weightedSumRes, ((uint16_t*)
                 inputs)[iter], ((int16_t*)weights
                 )[iter]);
            *weightedSum += weightedSumRes;
        }
    }
}
```

L'intégration de cette unité de calculs vectoriels a conduit aux résultats suivants lors de l'exécution du programme sur la carte :

- Chiffre reconnu : 4
- Crédence : 82
- 572 826 instructions (-67% par rapport au programme de base)
- 1 156 589 cycles machine (-51% par rapport au programme de base)

3 Discussion des résultats

En analysant les résultats des différentes approches décrites dans cet article, nous pouvons remarquer que l'approche qui fait usage de l'instruction MAD optimisée est celle qui génère les meilleurs résultats, avec 784 094 cycles machine, soit une réduction de 66% par rapport au nombre de cycles original.
Avec l'instruction MAD toute seule, on aurait pu obtenir de meilleurs résultats si le gestionnaire de la mémoire du processeur CV32A6 offrait la possibilité de faire des chargements mémoire non-alignés, au dépit d'un chargement plus lent, vu que l'on n'aurait plus à vérifier le bon alignement mémoire dans le code C, qui s'avère comme étant très gourmand en cycles machine.

Il est aussi à noter que l'unité de calculs vectoriels aurait pu surpasser largement la performance de l'instruction MAD si l'interface CV-X-IF offrait la possibilité au co-processeur de faire des chargements de données depuis la mémoire et directement vers le co-processeur, au lieu de passer par le processeur pour faire ces chargements, ce qui aurait pu diviser par 2 le nombre de cycles machine dédiés au chargement des données dans l'unité de calculs vectoriels.

Nous avons également exploré la piste du chargement des inputs et des poids en une seule fois dans l'unité de calculs vectoriels et de ne passer que l'offset à l'appel de la fonction macsOnRange vu qu'ils sont constants, au lieu de les charger à chaque appel de macsOnRange, mais la taille d'au moins un des

tableaux de poids nous a vite fait faire demi-tour ($150 \times 24 \times 4 \times 4 = 57600$ éléments).

On va donc retenir le résultat final suivant, correspondant à la custom instruction MAD optimisée et sans unité de calculs vectoriels :

- Chiffre reconnu : 4
- Crédence : 82
- En simulation :
  - 473 876 instructions (-72% par rapport au programme de base)
  - 747 449 cycles machine (-66% par rapport au programme de base)
- Sur FPGA :
  - 473 876 instructions (-72% par rapport au programme de base)
  - 784 094 cycles machine (-66% par rapport au programme de base)
- $f_{max} = \frac{1}{20ns+0.377ns} = 49.07$ MHz
- 9541 total LUTs
- 9315 logic LUTs
- 226 LUTRAMs
- 4848 FFs
- 16 RAMB36
- 1 RAMB18

4 Conclusions

En conclusion, cette aventure a finalement été fructueuse, même si remplie de challenges. En effet, comprendre l'architecture du processeur CV32A6 et de son co-processeur, apprendre le SystemVerilog, comprendre l'algorithme d'IA mis en place, créer un profileur sur-mesure, apprendre à intégrer des instructions assembleur personnalisées en C et en SystemVerilog, et finalement penser aux architectures de l'instruction MAD et celle du MADV se sont avérées essentielles pour la réussite de ce projet.
Au final, et même si l'on espérait améliorer encore plus nos résultats, nous sommes très satisfaits des résultats obtenus actuellement, considérant notre niveau de M1 et toutes les nouvelles notions que nous avons dû assimiler avant de se lancer dans le développement.

Remerciements

Nous tenons d'abord à remercier Yann DOUZE, professeur agrégé à Sorbonne Université, pour nous avoir proposé de participer au concours et pour sa confiance en notre équipe représentant Sorbonne Université et Polytech Sorbonne.
Nous remercions également Bertrand GRANADO, Professeur des université à Sorbonne Université, pour avoir toujours répondu à nos questions et pour nous avoir éclairés dans nos idées de recherche.
Nous remercions Thibault HILAIRE, Cathy MOROT, Sylvain VIATEUR, Alexandre GUERRE et Hugo PAUGET BALLESTEROS, professeurs et encadrants et qui nous ont toujours suivi le long de ce projet.
Enfin, nous exprimons notre gratitude envers toute l'équipe de Thales, du CNFM et du GDR SOC² pour leur présence continue sur le forum mis à notre disposition, pour leur contribution précieuse concernant le fonctionnement du projet, et pour l'accès aux logiciels essentiels à la réalisation du projet