RISC-V ISA extension for CNN acceleration on the CV32A6 core

John GNANGUESSIM
INSA Toulouse
Toulouse, France
gnanguessim@insa-toulouse.fr

Nell PARTY
INSA Toulouse
Toulouse, France
party@insa-toulouse.fr

Hugo MICHEL
INSA Toulouse
Toulouse, France
hmichel@insa-toulouse.fr

Achille CAUTE
INSA Toulouse
Toulouse, France
caute@insa-toulouse.fr

Eric ALATA
LAAS-CNRS
Toulouse, France
eric.alata@laas.fr

Daniela DRAGOMIRESCU
LAAS-CNRS
Toulouse, France
daniela.dragomirescu@laas.fr

Philippe LELEUX
LAAS-CNRS
Toulouse, France
philippe.leleux@laas.fr

Vincent MIGLIORE
LAAS-CNRS
Toulouse, France
vincent.migliore@laas.fr

Gael LOUBET
LAAS-CNRS
Toulouse, France
gael.loubet@laas.fr

***Abstract*—Since 2009, with the appearance of ImageNet challenges Convolutional Neural Network algorithms have become more powerful. They can handle larger images with better accuracy. To reach better performance, algorithms became more complex with more parameters. Therefore, hardware must speed up the execution to maintain acceptable execution time. The aim of this study is to decrease the number of execution cycles of a convolutional neural network program on an RISC-V processor. The acceleration was tested on the Mixed National Institute of Standards and Technology (MNIST) dataset with a 4-layer CNN. This program was executed on a CV32A6 processor running on a FPGA. Even though there are numerous studies on hardware acceleration for CNN, there are none for the CV32A6 processor. To start the study, we analyzed the program to obtain the number of cycles, the number of instructions, the percentage of cache miss, the execution trace...,etc. Based on this information we implemented a new instruction MAC8 that does four multiplications in parallel and accumulate the results. With this solution, we observed that the execution is 5.5 times faster. In the future, a study on the optimization of the loading instructions should be done since. Load instructions represent the major part of the instructions executed.**

***Index Terms*—RISC-V, CVA6, MAC, hardware acceleration, CNN**

I. Introduction

Convolutional neural networks (CNNs), a state-of-the-art method in computer vision and machine learning, capture significant features from visual data such as images and videos. Advances since LeNet's 1990 introduction include deeper architectures such as VGG, Inception, and ResNet. These architectures, used for image classification, object detection, and segmentation, remain complex with performance optimisation challenges. Representing INSA Toulouse, our team INSAi is composed of four fourth-year Automation & Electronics engineering students specializing in Embedded Systems at the Master level 1. Five tutors from the INSA Toulouse electronics and computer engineering department guided us: Daniela Dragomirescu, Eric Alata, Gaël Loubet, Philippe Leleux, and Vincent Migliore.

We aim to enhance our algorithm by observing its current functionality to identify potential areas of improvement areas. This method provides a deeper insight into the consequences of our actions. Furthermore, we aspire to develop a broader improvement applicable not only to this algorithm but also to Convolutional Neural Networks (CNNs) generally. Ultimately, our goal is to extend the acceleration advantages to more intricate algorithms.

II. Execution analysis

A. Code Structure

For this study, we focus on the MNIST dataset. This dataset contains more than 50,000 images of handwritten numbers from 0 to 9. To recognize these numbers, we use a 4-layer Convolutional Neural Network. Table I below gives an overview of the network.

TABLE I
Structure of the network

Layer	Type	Characteristics
1st Layer (Conv1)	Convolutional layer	Inputs: 24x24 Kernel: 4x4 Filters: 16 Stride: 2
2nd Layer (Conv2)	Convolutional laye	Inputs: 11x11 Kernel: 5x5 Filters: 24 Stride: 2
3rd Layer (fc1)	Fully Connected Layer	Number of perceptrons: 150
4th Layer (fc2)	Fully Connected Layer	Number of perceptrons: 10

There are ten outputs because the last layer contains ten perceptrons (see Table I). Each output represents the probability that the input is a number from 0 to 9.

This algorithm uses 8-bit weights and 8-bit inputs. In addition, all the data are integers thanks to the quantization process.

B. Performance analysis

To reduce the number of cycles required for execution, in-depth knowledge of the system is essential. To measure performance, we used the Control/Status Registers (CSR). Once properly configured, these registers can provide a great deal of information. We decided to focus not only on the number of instructions executed and the number of cycles required, but also on memory accesses with events such as L1 D-cache accesses, L1 D-cache misses, load accesses and store accesses.

Figure 1 below, provides an overview of the execution performance.

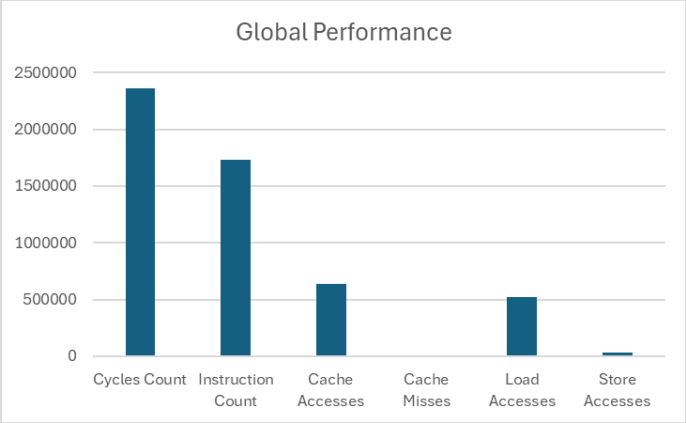

Fig. 1. Global performance of the algorithm execution

The execution of the algorithm requires 2358461 cycles and 1731593 instructions. In addition, 36.8% of the instructions are memory accesses and 81.8% of them are load accesses. Finally, only 1.8% of the load accesses are cache misses. This means that the memory optimisation is sufficient to limit cache misses. Cache misses need to be avoided because it takes a large number of cycles to deal with them.

As explained above, the algorithm is divided into four layers. To determine the layer that needs the larger number of cycles and instructions, we proceed as previously to obtain the number of instructions and the number of cycles per layer. A summary is shown in the Table II.

TABLE II
PERFORMANCE OF THE ALGORITHM PER LAYER

Layer	Number of cycles	Number of instructions
Conv1	324957	218584
Conv2	1467834	1097891
fc1	55725	408321
fc2	14757	10650

The second layer represents 63.3% of the instructions and 62% of the cycles. On the other hand, the second fully connected layer (fc2) time process is negligible with only 0.6% of the cycles. It indicates that the acceleration on convolutional processes has more impact on the final number of cycles.

C. Execution trace analysis

To complete the study of the system, we analyzed the execution trace. This trace represents the number of times each instruction is executed on the processor. To create this trace, the program must be run on a simulator. For this analysis we used Spike, an RISC-V ISA (Instruction Set Architecture) simulator. By analyzing the occurrence of each instruction we obtain the following graph.

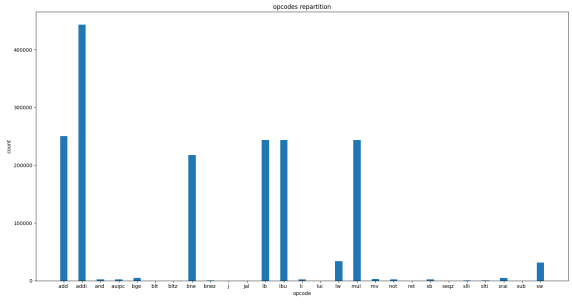

Fig. 2. Occurrence of the eight most executed instructions

Figure 2 shows that addition and load instructions are the most executed. In fact, 40% of the instructions are additions (addi, add) and 30% are loads (lb, lbu and lw). Finally, multiplication (mul) is another of the most executed instructions.

These three instructions : addition, multiplication and load, correspond to the computation needed for the perceptron computation in the fully connected layers but it is mostly present for the convolution in the convolutional layers. In fact, 184576 multiplications and 182256 additions are needed for the two convolutional layers. This represents 75% of all the multiplications performed in the program.

D. Conclusion

In conclusion, trying to reduce cache misses is ineffective because the proportion of cache misses is negligible compared to the number of memory accesses.

On the other hand, the program studied is composed of four layers, with two convolutional layers that require many cycles and instructions. These instructions are mainly additions and multiplications, since they are the main operations for the convolution computation. In addition, load instructions are also executed many times. This analysis shows that improving the convolution operation can greatly reduce the number of execution cycles.

III. MAC UNIT

By studying the current system, we better understand the workings of the convolutional networks that cause such a large number of register loads, additions and multiplications. One by one, these networks load one weight, in this case 1 byte, from the kernel and one weight from the processed image, then multiply them and store the result. After passing through them all, the stored results are added together. However, these successive multiplications use different weights from each other, so they are independent of each other. Based on this observation and inspired in part by the use of vectorized processors, we came up with the idea of mutualizing these loads by creating a new Multiply ACcumulate unit, MAC, which performs the addition and multiplication of 4 weights of images and kernels in a single load.

A. Hardware

Implementing this unit means redefining the behavior of the FPGA architecture. In order to do this, we have created a new unit called MAC8, 8 for the processed 8-bit weights, based on the model of the existing units.

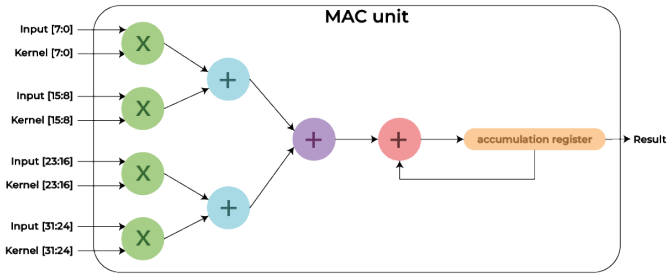

Fig. 3. Occurrence of the eight most executed instructions

As illustrated in figure 3, this unit receives two inputs, 4 bytes from the kernel and 4 bytes from the input matrix, i.e., the image being processed. In the first stage, it multiplies each kernel byte with a byte of the image, calculating 4 parallel products. In the second stage, these 4 results are summed two by two. These two results are then added together in a third stage.

The output of this third stage is summed with the previous results in a dedicated 32-bit register. The value of this register is stored in a general purpose register at each cycle.

By parallelizing loads and operations and as the data in each stage is independent of each other, the stage is full and there is no waiting time, so it might take 4 cycles for 8 weights of each input to be processed. In our case we choose to use it with à fix latency of one cycle by operating a small reduction in the operating frequency, in order to make it as fast as possible.

B. MAC instructions

Without instructions to use it, the MAC unit is ineffective. Therefore, we created two instructions to be executed by the MAC unit. The first one is the mac8_init, it computes the convolution of the inputs and initializes the accumulate register with the result. The second instruction, mac8_acc, also computes the convolution, but it accumulates the result with the previous value of the accumulation register.

There are several types of instructions that are supported by the RISC-V, as shown in Figure 4 below.

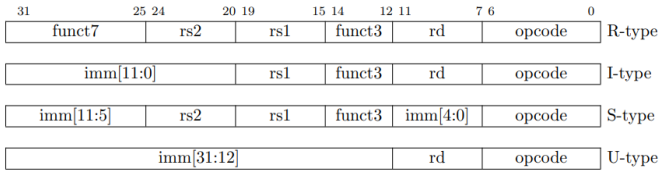

Fig. 4. Structure of the different type of instruction supported by the RISC-V

Both instructions created for the MAC unit are R-type, because, there are always two 32-bit operands and a return register. We need a return register because, each time one of the mac8 instructions is executed, the content of the accumulate register is copied to in a general purpose register. This behavior allows us not to create a third instruction to move the contents of the accumulate register to a general purpose register.

TABLE III
STRUCTURE OF THE MAC8 INSTRUCTIONS

Instruction R-Type	funct7 31..25	rs2 24..20	rs1 19..15	funct3 14..12	rd 11..7	opcode 6..0
MAC8_INIT	0x0	rs2	rs1	0x2	rd	custom_0 (0xB)
MAC8_ACC	0x0	rs2	rs1	0x0	rd	custom_0 (0xB)

As shown in Table IV, all the custom instructions have the same opcode to simplify implementation. The opcode used is $(0001011)_2$. The instructions are distinguished by the function 3 code. The code of the mac8_init instruction is $(010)_2$ while the code of the mac8_acc instruction is $(000)_2$. The opcode and the function 3 code are used by the decoding stage of the processor to identify the operands, the functional unit and the return register of the instruction being executed.

C. Software

Since the MAC unit is implemented in hardware and the new mac8 instructions are recognized by the RISC-V, this new unit must be introduced into the program. We have modified the macsOnRange function in order to perform the convolution using the MAC unit. This function is used in the

convcellPropagate1 function to perform the convolutions for the two convolution layers.

As explained previously, the MAC unit takes two 32-bit vectors as input. One corresponds to four kernel weights and the other to four elements of the input matrix. The mac_pack32 function packs the elements of the input matrix and the kernel into of 32-bit vectors. However, these vectors must be stored at addresses that are divisible by four, so mac_pack32 also properly aligns the vectors in the memory.

Then, the mac8_init instruction in macsOnRange is called with assembly code, the arguments being the vectors previously packed by the mac_pack32 function. This instruction initializes the accumulation register of the MAC unit. The mac8_acc instruction is then used to perform convolutions.

To limit the dependencies, four operations are started in sequence. By running successive mac8_acc instructions, we are able to control the weights used and then choose to use only independent operands. Furthermore, we run four consecutive mac8_acc instructions because it corresponds to a convolution between the kernel and the input matrix in the first layer.

Finally, the activation function is applied to the result. In the convolutional layers, the activation function is the Rectified Linear Unit (ReLU). This function preserves the value if it is positive and sets it to zero if it is negative.

IV. Getting MAC8_FU's operands

The misalignment of loaded registers is another issue contributing to the high number of loads on these convolutional networks. The 2-byte stride used in these networks often requires loading a 4-byte weights, overlapping two different words in memory, which are themselves aligned, illustrated on Figure 5.The loading of this misaligned words presents a recurring pattern that is inefficient. all the loading of weights and input matrices was done byte by byte, stored in registers, and then shifted and combined to create 4-byte vectors for the MAC8 unit. This process involved 10 instructions.

To address this, we implemented a solution that compares the address alignment of the four-byte vector for the MAC unit. If the vector is already 32 bits aligned, we simply use a load word. However, if it is not aligned, we load the two words that contain a part of the words desired and perform two shifts and an 'or' function to rearrange the data into a 32-bit register. These three instructions can be easily implemented in hardware.

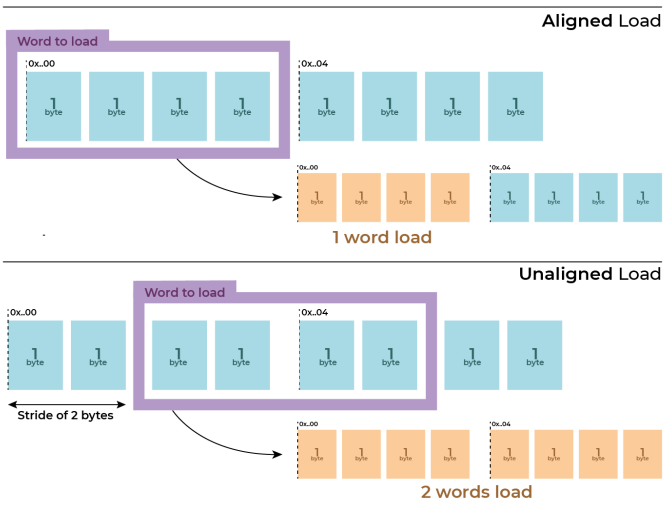

Fig. 5. Illustration of the non alignment problem in loads

A. MIX instruction

We implemented a new custom instruction called 'mix.' It takes two registers as operands and returns the result to a general-purpose register. It performs the following operations:

$$rd = (rs1 >> 16)|(rs2 << 16)$$

In other words, this instruction combine the 16 last bits of the operand A with the 16 first bits of the operand B in a 32-bit register.

TABLE IV
STRUCTURE OF THE MAC8 INSTRUCTIONS

Instruction R-Type	funct7 31..25	rs2 24..20	rs1 19..15	funct3 14..12	rd 11..7	opcode 6..0
MIX	0x0	rs2	rs1	0x1	rd	custom_0 (0xB)

Due to the data access method, we only need to shift the data by 16 bits. Therefore, we did not use any immediate field, and we implemented the instruction as an R-Type instruction. The structure of the 'mix' instruction is given in Table IV. As for the architecture itself, it is quite simple. All operations are performed combinatorially before being stored in a flip-flop. The result is then available for the next stage until the next positive edge of the clock.

With the 'mix' instruction the ten instructions done to load a misaligned words is then replace with three instructions, two 'load word' instructions and one 'mix' instruction.

The 'load word' instructions are more optimal since they take the same time as loading a single byte. Therefore, even though only half the data is used per 'load word,' we save time overall.

B. Software

We made some changes to the software to load an aligned 4-byte vector with just one 'load word' instruction whenever possible. To achieve this, we added a condition that checks

if the address is aligned or not. If it is aligned, we use a single 'load word' instruction. If not, we use two 'load word' instructions followed by one 'mix' instruction for unaligned loads. This approach remains highly efficient, as the compiler optimizes the code by checking the condition once before entering the MAC loop.

V. Methodology

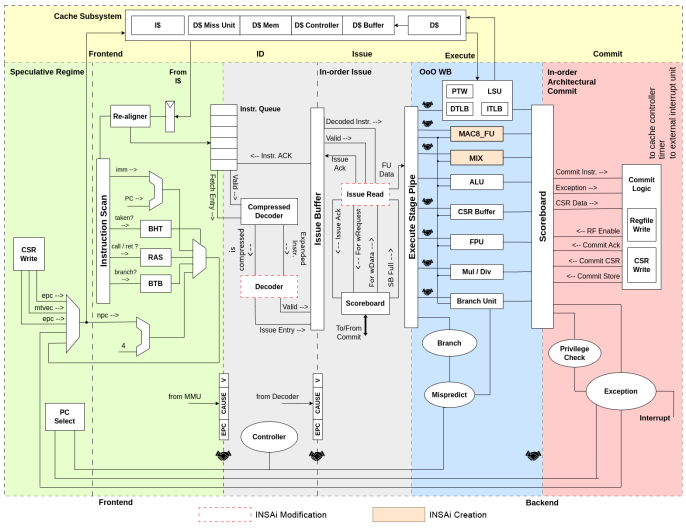

Fig. 6. Ariane architecture including our modifications, [2]

The whole architecture including the optimizations described above, sythesized on Figure 6, is tested on an image from the MNIST dataset. The CNN algorithm used is the one described in the Execution Analysis section. The test is performed directly on the FPGA. The FPGA used is the Zybo Z7-20. The components available on the Zybo Z7-20 are listed in the Table V.

TABLE V
List of components of the Zybo Z7-20

Componants	Zybo Z7-20
Look-up Tables (LUTs)	53200
Flip-Flop	106400
Block RAM	630 KB
DSP Slices	220

The efficiency of the optimizations is measured by the number of cycles needed to execute the algorithm. However, other parameters such as frequency, components used in the FPGA, power consumption, etc. should also be taken into account.

The validity of the optimization is given by the number of recognized digits and the credence associated. It should be 82. The credence corresponds to the confidence assigned to the result.

VI. Results

First, the digit is detected with a credence of 82. This means that the optimizations give the same result as the baseline. Figure 7 shows the number of cycles and the number of instructions needed to execute the CNN algorithm on the RISC-V. These values are compared to the baseline.

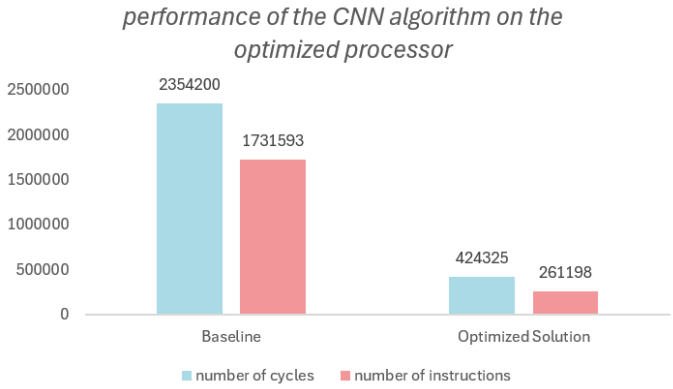

Fig. 7. Performance of the CNN algorithm on the optimized processor in simulation and in real

The test shows 81.9% reduction in the number of cycles and 84.9% reduction in the number of instructions (see Figure 7). The results show the effectiveness of the added solution.

In addition, the maximum frequency decreases slightly from 51.54 MHz to 51.19 MHz, a reduction of 0.68%. The baseline has a power consumption of 0.285 W, while the optimized solution is at 0.287 W. Therefore, there is only a 0.7% difference with the implementation of the MAC unit and the MIX unit.

Finally, we observe the utilization of the components of the FPGA. Figure 8 summarizes the area used by the solutions.

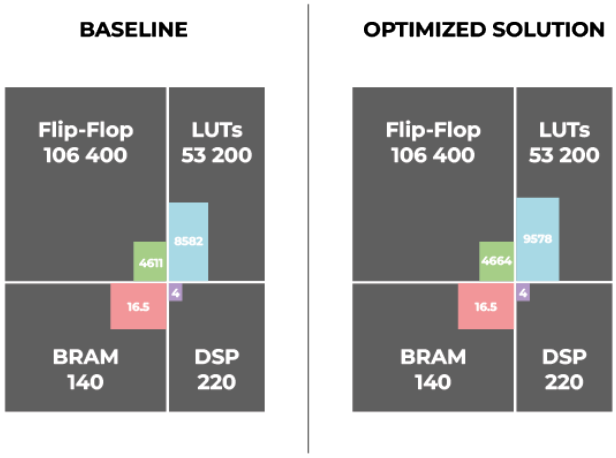

Fig. 8. Components utilization in the FPGA

Finally, Figure 8 shows that there is not much difference between the baseline and the optimized solution. This means that the optimized RISC-V can be extended to add new optimizations.

In summary, the number of cycles is reduced by 81.9% thanks to the optimized solution. Moreover, the other parameters such as the maximum frequency, the power consumption or the components used on the FPGA did not increase much.

VII. Conclusion

In conclusion, we have designed a processor that executes a specific CNN algorithm 5.5 times faster. To achieve this result we implemented two new units. The first one is a MAC unit that performs the convolutions faster by paralyzing the multiplications and the additions. The second one is the MIX unit, which performs combinatorial operations to reconstruct a word that is not aligned in the memory.

However, the MIX unit is limited by the need to check whether the word is aligned or not. It adds some jump instructions in the execution trace that reduce the expected performance.

There are many solutions to reduce the number of execution cycles that we have not explored in this study. As explained in the execution analysis part, the number of loads is significant. To reduce the cost of these load instructions, it is possible to create a preload unit. This unit will load data thanks to a specific loading pattern. It makes it possible to compute the CNN operations in sequence without having to load the data in between.

References

[1] OpenHW Group, "CVA6 user manual" [Online]. Available: https://docs.openhwgroup.org/projects/cva6-user-manual/.

[2] ThalesGroup, "Ariane_Overview" [Online]. Available:https://github.com/ThalesGroupcva6-softcore-contest/blob/cv32a6_contest_23_24/docs/03_cva6_design_staticariane_overview.drawio.png.

[3] INSAi, "INSAi_CVA6-softcore-contest" [Online]. Available: ...