Accelerating AI Applications on a RISC-V processor **LATAM-V**

Álvaro SCARRAMBERG
IMT Atlantique
Brest, France
alvaro.scarramberg
@imt-atlantique.net

Rogerio BOMBARDELLI
IMT Atlantique
Brest, France
rogerio.kaciava-bombardelli
@imt-atlantique.net

Joaquin OPAZO
IMT Atlantique
Brest, France
joaquin.opazo-yentzen
@imt-atlantique.net

***Abstract*—This paper presents the solution developed by the LATAM-V Team from IMT Atlantique, Brest campus, for the 4th National RISC-V Student Contest 2023-2024, sponsored by Thales, the GDR SOC2, and CNFM. The contest focused on accelerating AI applications on a RISC-V processor, challenging participants to optimize performance and efficiency. Our solution leverages the fact that an 8-bits representation is used for all data in the target AI application despite the fact that the registers in the RISC-V CV32A6 architecture are 32 bits. Taking this into account, we added and implemented an SIMD inspired custom instruction to the RISC-V ISA resulting in an acceleration of 8.4 times for the target convolutional neural network application. Through simulation, FPGA implementation, and validation, our solution is demonstrated to be generally applicable for the acceleration of all neural network-based AI applications. This paper outlines the design, implementation, and evaluation of our solution, demonstrating its potential for advancing AI acceleration on RISC-V processors.**

I. Introduction

The National RISC-V Student Contest [1], a prestigious competition organized by **Thales** Research & Technology France in collaboration with the **GDR SOC²** and the **CNFM**, presents a unique opportunity for students interested in the forefront of processor design and open-source hardware. **Thales**, a global leader in critical information systems, is committed to fostering innovation in this field. Additionally, the contest provides a cutting-edge platform for students to both contribute and learn from the growing interest in the RISC-V open Instruction Set Architecture (ISA).

Building upon the success of the previous three editions (2020 to 2023), the 2024 contest [1] focuses on accelerating an Artificial Intelligence (AI) application – specifically, **MNIST** digit recognition – on the *CV32A6* RISC-V soft-core processor. The ***CV32A6*** is a 32-bit variant of the open-source **ARIANE** core, designed by ETH Zürich and integrated into the **OpenHW** Group's CVA6 series. Students should compete to modify the *CV32A6* architecture to achieve the fastest possible **MNIST** digit recognition using the fewest processing cycles. The winning team will propose the most efficient solution for this AI application on open-source hardware.

A. Hardware

The FPGA[1] development board targeted for this context is the Digilent Zybo Z7-20 (Figure 1). In addition, the development kit includes two cables: a **JTAG-HS2** for bitstream programming and a **Pmod USBUART** for the output console. This reference hardware platform was provided by the Algorithm-Architecture Interaction (2AI) team of the Lab-STICC laboratory.

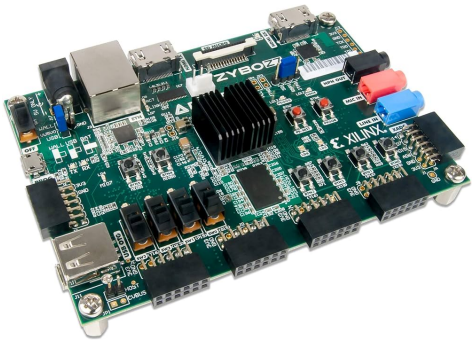

Fig. 1. Zybo Z7-20 Board

B. Software

For the software preparation, the following tools are used:

- **Questa Tool 10.7**: This simulator [2] is used to measure power consumption and the graph of all the control signals to verify that everything is in order.
- **Vitis/Vivado 2020.1**: This is the software used to program the FPGA with the bitstream using the Zynq 7000 from AMD-Xilinx [3].
- **Docker**: is used to ease the installation of RISC-V tools including the toolchain and OpenOCD

[1]Field-Programmable Gate Array

C. AI Context

AI is a transformative technology that enables machines to perform tasks that typically require human intelligence. At its core, AI seeks to replicate human cognitive functions such as learning, reasoning, problem-solving, perception, and decision-making. The field of AI encompasses a wide range of techniques and approaches, with the ultimate goal of creating intelligent systems that can perceive their environment, learn from data, and make informed decisions.

For the context of this challenge, we target the well-known **MNIST**[2] dataset as our database. It consists of 60,000 24x24 images of handwritten digits between 0 and 9. These images will be the input to a Convolutional Neural Network (CNN), which consists of 4 layers, 2 convolutional and 2 fully connected. The output is an array of 10 values, where the highest value represents the number that our application estimates as the handwritten number.

Fig. 2. MNIST dataset examples.

Figure 2 shows some samples of the MNIST dataset, which is a commonly used dataset of handwritten digits. Figure 3 shows the layers of the CNN that is being used to classify the digits in the MNIST dataset.

The first two layers of the neural network are convolutional layers. *Convolutional* layers are designed to extract features from images. The next layer in the architecture is a *fully connected* layer. Fully connected layers are used to combine the features that have been extracted by the convolutional layers into a single output vector. The output vector is then used to classify the image.

The final layer of the CNN is also fully connected but its output is not passed through a non-linearity such as a ReLU activation function, that is why this layer is also classified as linear. This final linear layer is used to map the output vector from the fully connected layer to the output classes. In the case of the MNIST dataset, the output classes are the 10 digits (0-9).

[2]*Modified National Institute of Standards and Technology*

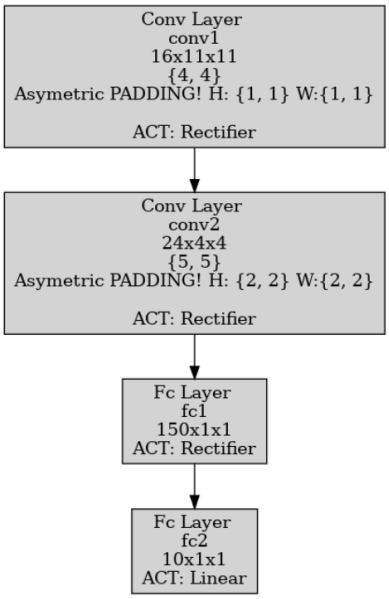

Fig. 3. Target CNN architecture description.

II. MNIST INFERENCE BASELINE

In order to find ways to optimize the inference of our target application, it's necessary to have a good understanding of the inner functionning of the baseline architecture.

Function	Number of calls
macsOnRange	10274
clamp	2480
sat	2480
saturate	2480
read_mnist_input	4
convcellPropagate1	2
envRead	1
fccellPropagateDATA$_T$	1
fccellPropagateUDATA$_T$	1
feof_mnist_input	1
maxPropagate1	1
processInput	1
propagate	1
readStimulus	1

Fig. 4. Number of calls for each function of the target application.

From Figure 4 we can see that the function that is used the most is related to a multiply and accumulate operation. This is expected since the operation of a neural network can be characterized as a series of matrix-to-vector multiplications which in turn are computed by the addition of the results of a series of multiplications.

In the baseline, the RISC-V assembly implementation of the *macsOnRange* function consists of an unrolled loop where there are **4 instructions** that are used and they are **two loads** of 8 bit values associated to both the input and weight and then the **multiplication** and **addition** required by the MAC operation. This is also illustrated in the listing (1).

```
lbu s2,0(s9) // We load 1 input
lb s3,0(a2) // We load 1 weight
mul s6,s2,s3 // Multiply both values
add a5,a5,s6 // Accumulate with previous results
```

Listing 1. Baseline assembly loop.

Our optimizations focus on accelerating this function by adding two custom instructions to the RISC-V ISA. This allows us to not only reduce the total number of loops used by means of parallelization, but also to reduce the number of instructions used inside each loop.

III. LATAM-V Optimizations

In this section, we present the main 3 ideas which we developed to optimize the inference of the convolutional neural network application.

A. Multiplication parallelization

As we previously mentioned, in the baseline architecture, the data are loaded using 8-bits format which evidently does not take full advantage of CV32A6 registers which are of size 32 bits.

Based on this observation, our first optimization consists in casting both the input data and weights into 32 bit variables to make sure that 4 are loaded at a time. Once we have two registers containing the packages of 8-bits numbers, we will perform the multiplications and their subsequent additions in parallel by means of a custom instruction which we call **CustomMAC**.

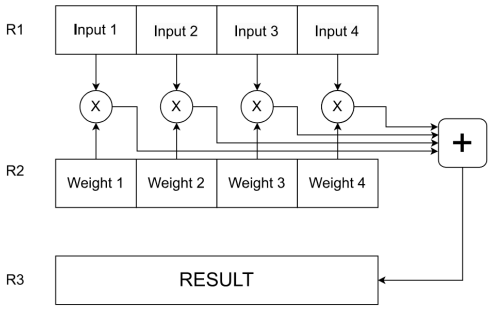

Fig. 5. **CustomMAC** representation.

Thus, given that we parallelize the number of loads, additions, and multiplications by a factor of 4, we should expect a similar level of acceleration as a result of this optimization.

This instruction is implemented in hardware by the use of the CV-X-IF interface available in the CV32A6 RISC-V softcore. We designed a Multiply-Accumulate Unit that acts as a tightly coupled co-processor to the CPU and allows it to offload our custom instructions. The use of this interface greatly simplified the development time of our optimizations.

B. Accumulate operation omission

Our above discussed version of the **CustomMAC** operation requires an instruction to add the new result to the accumulated result of previous MAC operations. Since we are targeting a custom instruction able to perform both the multiplications and the additions, we decided to buffer the result of the previous operations inside the MAC unit and add it to the new result of the **CustomMAC** operation when its called. This second optimisation idea is shown in Figure 6.

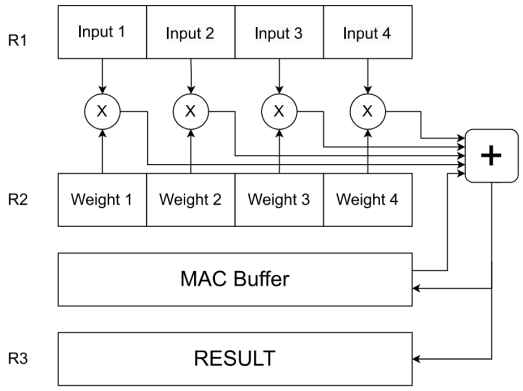

Fig. 6. **CustomMAC** with accumulate buffer.

When a new set of MAC operations needs to be performed, this buffer must be initialized to zero. To deal with this issue, we added a second custom instruction **RetrieveMAC** that returns the result of the current set of operations and initializes the buffer to zero so the next multiplications can be performed.

As described previously, the multiply accumulate loop contains 4 instructions, 2 for loading the weight and input, 1 for performing the multiplication, and lastly and addition to previous results, which we will omit since our 'CustomMac' instruction performs this last operation as well as the corresponding multiplications. This omission reduces the instruction count by 25%, as our loop now comprises only three instructions instead of four. Considering this reduction, we should anticipate an additional speed-up of 1.33 times.

Similarly to the baseline architecture, the assembly implementation of our custom MAC function consists of an unrolled loop where the main instructions that are used are represented in the following assembly code:

```
lw s2,0(s9) // We load 32 bits -> 4 inputs
lw s3,0(a2) // We load 32 bits -> 4 weights
customMAC s6,s2,s3 // 4 parallel Mac's
```

Listing 2. MAC function RISC-V Assembly implementation.

Note that this sequence of instructions is repeated multiple times as an unrolled loop and, although not shown, the end of this loop is immediately followed by a **RetrieveMAC** instruction.

C. Online input buffering

The operation of convolutional neural networks can be decomposed into a series of Matrix-Vector multiplications,

where the weights of the network play the role of these matrices and the inputs of different channels are concatenated to form the vectors.

This observation is important because this means that the inputs need to be used multiple times to compute the output. Consequently, the current implementation of the CNN inference requires loading a single input multiple times, which is inherently inefficient. Therefore, finding a way to avoid this should lead to greater performance.

With this goal in mind, we added the possibility of buffering the inputs inside our Multiply Accumulate Unit within a buffer consisting of 128 cells of 32 bits each. We designed this input buffer with the idea of letting the user decide if using it or not.

In order to do this, we added a functionality to the **RetrieveMAC** instruction. If no arguments are given to the instruction (Input registers initialized to zero) then the MAC Unit will behave as usual, but if a USE INPUT BUFFER flag is used as an argument then it will configure the co-processor to use the buffer for the next MAC operations. In this case, the first source register of the **CustomMAC** instruction will be ignored, and instead one of the buffered inputs will be used to complete the operation. This means that the assembly code loop presented in the listing (2) will only require one load instead of two:

```
lw a3,0(a4) // We load 32 bits -> 4 weights
customMAC a3,a5,a3 // a5 is not used
```

Listing 3. Input buffer use with CustomMAC instruction.

To obtain a better understanding of this idea, we can see in the following figure an illustration on the operation of a fully connected neural network layer:

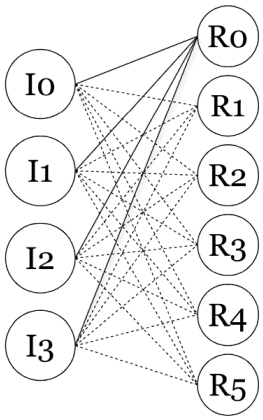

Fig. 7. Fully connected layer representation.

In this figure, we can see that in a fully connected layer each input is used multiple times to compute all the outputs of the layer, which also happens in convolutional neural networks.

Therefore, our proposal consists of storing the inputs in a buffer inside our MAC Unit which will then allow us to load them only the first time they are needed, which occurs when we compute the first output of the layer. Subsequently, when computing the next outputs of the layer, we will only need to load the weights, not the inputs, as shown in the assembly code in listing (3).

Eliminating another instruction from this loop means that on top of previous optimizations we get a **50% speedup**. We called this optimization '**Online input buffering**' since we buffer the inputs in parallel when they are used for their first MAC operation, effectively eliminating any overhead related to storing them in the buffer.

The intended operation of the buffer is the following:

1) When any given set of inputs are to be used for the first time, the MAC Unit will be configured in its default state, which can be manually set by calling the **RetrieveMAC** instruction with source registers set to zero. Doing this will reset the buffer address so that it points to the first cell in the buffer.
2) The moment the first MAC operation is performed, in parallel the buffer will store the first input and it will increase its buffer address to point to the next cell in this memory to get ready for the next input to be stored.
3) Once all inputs have been used for their first respective multiplications, the user will require for the current MAC operation result to be retrieved and therefore it will call **RetrieveMAC**, at which moment the co-processor will reset the buffer address to zero and it will also check if the USE INPUT BUFFER flag has been passed.
4) In this case, for the next MAC operations it will not modify the input buffer, rather the data passed through the register associated to the inputs will be replaced by that stored in the buffer which allows the user to avoid having to load the inputs again.
5) Until the end of all associated MAC operations of this set of inputs, whenever the user needs to retrieve a result it will need to pass the USE INPUT BUFFER flag, otherwise the normal operation of the MAC Unit will be resumed and thus the input buffer will not be used and it will repopulate its memory with the next set of inputs.

To further clarify its operation, Figure 8 illustrates the use of the input buffer. However, as mentioned previously, the existence of this buffer does not necessitate its use; normal MAC operations can be performed as long as the USE INPUT BUFFER flag is never set.

This concludes all our proposed optimizations. Other improvements could have been envisioned, which could have tackled input sparsity for example. Given that we are storing inputs in our MAC Unit, we could have found a way of detecting if they are zero or not and skip their respective multiplications since they would be redundant. However due to time constraints, and doubts about the legality of this optimization in the context of the competition we decided not to explore this approach any further.

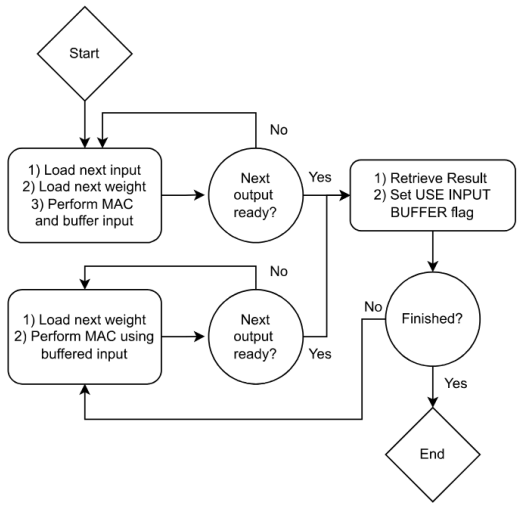

Fig. 8. Input buffer utilization flow.

IV. Results

In this section we present the results of our optimizations. Tables I and II show both the number of RISC-V instructions and clock cycles required by the execution of the inference of the convolutional neural network application applied on a single sample of the MNIST dataset. We measured performance by using the Questa simulator and by actually running the application on the FPGA.

We differentiate the performance of our optimizations with and without using the input buffer. If we analyze the results on the FPGA, we can see that in terms of number of clock cycles, using the SIMD CustomMAC instruction produces already a speedup of 5.25 times, and when considering the third optimization, "Online input buffering", an acceleration of 8.4 times is achieved.

These results are compatible with our previous analysis of the proposed optimizations. The full use of our modifications theoretically should leads us to a 8x performance boost given that we parallelized operations by a factor of 4 and we reduced the number of necessary instructions by 50% when it comes to the main loop of the application (related to the execution of the MACs).

Given that MACs are not the only operations involved in our application's execution, we expected that in practice there would be a slightly smaller performance boost. Surprisingly, however, the number of clock cycles is even smaller than our theoretical analysis had suggested. This discrepancy can be attributed to our strategic restructuring of the application code to leverage our custom instructions effectively. In response to the limited assembly instructions within the main loop and the constant pipeline stalls due to dependency on preceding instruction results, we reorganized our implementation of the **macsOnRange** function to optimize the utilization of the CV32A6 pipeline.

Lastly, there exists a difference between the Questa simulation and the FPGA performance measurements which becomes quite noticeable when we analyze the performance of our optimizations. The Questa simulation suggests an acceleration of 9.4 times compared to the actual 8.4 times measured on the FPGA.

We were not able to identify in detail the true origin of this difference, but since the net performance difference remains approximately the same for all measurements (Between 35k to 37k clock cycles) we do not believe that this inaccuracy in the simulation is an indication of an error or problem in the implementation of our optimizations.

In our GitHub repository you can find the report files of the FPGA implementation of our solution. The following are some of the parameters that must be reported in the context of the competition.

1) The total number of LUTs required for the hardware implementation of our solution is **26051** (From 'cva6_fpga.utilization.rpt').
2) The number of reported FFs is **18485** (From 'cva6_fpga.utilization.rpt').
3) The reported slack is of **0.424ns** (From 'cva6_fpga.timing.rpt') which indicates that there are no timing violations and consequently the critical path delay is compatible with the selected clock frequency (Maximum clock frequency of **51.08** MHz).
4) Our final measurements on the FPGA result in a total number of clock cycles of **280.145** and the application correctly predicts the input to be a '**4**' with a credence of '**82**'.

TABLE I
Results on Questa Simulation

MNIST Application	N° RISC-V Inst.	N° of Clock Cycles
Baseline	1.731.593	2.316.326
CustomMAC	268.393	410.997
CustomMAC + Input Buffer	169.116	245.487

TABLE II
Results on FPGA

MNIST Application	N° RISC-V Inst.	N° of Clock Cycles
Baseline	1.731.593	2.353.806
CustomMAC	268.393	448.409
CustomMAC + Input Buffer	169.116	280.145

V. Conclusion

In summary, our solution to accelerate AI applications on a RISC-V processor has shown promising results in improving performance and efficiency. By introducing custom instructions and parallelization techniques, we successfully enhanced the execution of the convolutional neural network application. This work demonstrates the importance of customized hardware optimizations in maximizing the potential of available resources and open-source hardware tools for AI tasks. Optimizations including multiplication parallelization, accumulate operation omission, and online input buffering led to an **8.4 times** acceleration in the execution of the convolutional neural network application with the MNIST dataset. This research contributes to the ongoing development of AI technologies and provides valuable insight for future efforts to accelerate AI on RISC-V processors.

Acknowledgments

Finally we would like to thank our professors at IMT Atlantique which not only guided us throughout the duration of this project but also were essential for our formation in the domains of FPGA development, RISC-V and embedded systems in general.

In particular, we extend our gratitude to Ali Al Ghouwayel, Amer Baghdadi, Stefan Weithoffer and Yehya Nasser, all professors at the Brest campus of IMT Atlantique and members of the Algorithm-Architecture Interaction (2AI) team of the Lab-STICC laboratory.

We would also like to thank the organizers behind this project: Thales, CNFM and GDR SOC2.

References

[1] Thales, "National RISC-V student contest CV32A6." https://github.com/ThalesGroup/cva6-softcore-contest, 2023.

[2] Siemens, "Questa advanced simulator." https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/.

[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.