ADAC Team: Accelerating AI applications on a RISC-V processor

Felipe Paiva Alencar and Raffael Rocha Daltoe
Supervised by David Novo
LIRMM, Univ. Montpellier, CNRS, Montpellier, France

***Abstract*—This report investigates the optimization of the CV32A6 processor by integrating a dedicated co-processor named XADAC, an integer vectorized co-processor that implements custom vector instructions. The primary focus of our study is to accelerate the performance of a neural network trained with MNIST dataset to recognize handwritten digits. We have integrated custom instructions into the RISC-V toolchain and adapted the software to enable the use of our accelerator. As a result, we achieve a 5.47X speedup factor, corresponding to a 81.744% reduction in cycles.**

I. Introduction

In 2024, we were met with the opportunity to participate in a RISC-V national student contest. The Thales Group organized it, as well as two research groups: GDR SOC² and CNFM. The contest focused on accelerating AI applications on a RISC-V processor, specifically targeting architectural modifications of the CV32A6 RISC-V soft-core to enhance the performance of a neural network trained with MNIST dataset to recognize handwritten digits. The challenge is to modify the CV32A6 architecture to accelerate the application while adhering to specific constraints, such as not increasing memory sizes or decreasing the operational frequency significantly. To address the challenge, we make the following contributions:

- To couple a vector co-processor into CV32A6.
- Creation of new vector instructions to accelerate the MNIST CNN application.

II. Methodology

To optimize the execution speed of the MNIST CNN application on the CV32A6 processor, we employed several strategies including a thorough analysis of the existing code to understand its functionality and the implementation of performance counters to identify bottlenecks. We deactivated Translation Look-aside Buffers (TLBs) to simplify memory accesses, enabling us to design a co-processor that accesses the cache subsystem in a simpler way. Additionally, the toolchain was updated to support the new instruction opcodes introduced in the co-processor. To streamline it's implementation, we developed a robust CV-X-IF-like interface that allows for a modular and flexible co-processor design, thereby improving overall system performance. To verify the correctness of the hardware modifications we used some testing tools, they are described below.

- **Spike Simulation:** Spike is a RISC-V ISA simulator that provides a reference model for testing and debugging RISC-V binaries. Spike was integrated into the contest's Docker image to enhance testing efficiency and consistency. This integration allowed us to quickly verify the functional correctness of our software modifications.
- **Software Model of Accelerator:** We implemented the full functionality of our accelerator in C to validate the base idea of the instructions which we introduced. This software model was later used as a reference to verify the hardware implementation's correctness.
- **Verilator Simulation:** We created a test-bench that included only our accelerator to verify its functionality before integrating it into the core. Verilator was used to explore different simulators and as a learning exercise. Since Verilator's test benches are commonly written in C++, we were easily able to employ a C++ RISC-V behavioral model (mini-rv32ima [1]) to offload instructions to the RTL simulation of our accelerator
- **Questa Simulation:** Once the development phase of XADAC was completed and integrated into CV32A6, we used the provided Questa setup to verify the overall system's functionality and correctness.
  After validating all modifications in Questa, we moved to FPGA implementation. We used the FPGA described in the project repository and employed Vivado for synthesis and implementation. This final step ensured that our accelerator was ready for real-world application.

III. Performance counters

We've implemented a set of performance counters to provide a clearer picture of the processor's internal dynamics. The objective behind introducing these specific counters is to pinpoint performance bottlenecks with precision. This approach was

guided by the methodologies described in [2], which provided foundational insights for constructing our performance counters. It was necessary to add in the Software Layer the instructions to access these counters and describe in the hardware layer the new events to count.

As a result, we could identify the main bottleneck in the original application stemming from the large number of instructions, which achieve a reasonable 0.73 Instructions Per Cycle (IPC). Based on this insight, we chose to construct a co-processor that vectorizes execution, which reduces the number of instructions and boosts performance.

IV. Component Removal

In our optimization strategy for the CV32A6 processor, we removed non-essential components. This included reducing the size of the Instruction Cache and eliminating Translation Look-aside Buffers (TLBs).

We intentionally overcompensated in reducing the Instruction Cache size by 1kB, more than what was necessary to accommodate the 0.5kB Vector Register File. This strategy created a margin that ensured compliance with the contest's regulations, maintaining the total memory well below the absolute maximum size allowed.

Since the application runs bare-metal—that is, without an operating system—TLBs, which are typically used for virtual memory management, become redundant. Removing these buffers did not directly improve performance but was crucial for enabling a simpler co-processor design, since it performs accesses to the cache.

V. MNIST CNN Application

The MNIST CNN base code provided by the organizers was difficult to understand, so it was necessary to carry out a refactoring process to understand how it works. Then, it was possible understand that the application mainly relies on standard convolutional layers, illustrated by the Figure 1, where each kernel traverses the input (multidimensional matrix), performing multiplication and accumulation (MACC) operations between the kernel elements and the corresponding region in the input.

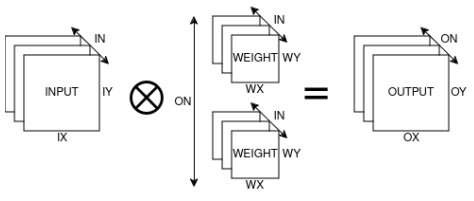


Figure 1. Convolutional Layer.

Algorithm 1 Convolutional Layer Algorithm

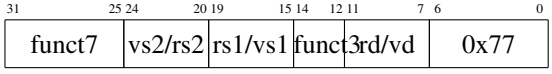
for $on = 0$ **to** $ON - 1$ **do**
 for $in = 0$ **to** $IN - 1$ **do**
  for $oy = 0$ **to** $OY - 1$ **do**
   for $ox = 0$ **to** $OX - 1$ **do**
    for $wy = 0$ **to** $WY - 1$ **do**
     for $wx = 0$ **to** $WX - 1$ **do**
      $O[on][oy][ox] \mathrel{+}=$
      $I[in][iy][ix] \times W[on][in][wy][wx]$

Additionally, the application included fully connected layers. However, we transformed these into convolutional layers, leveraging the fact that fully connected layers can be represented as a specific case of a convolutional layer.

VI. Creation of New Instruction Opcodes

After understanding the application, it became possible to devise new instruction opcodes to express parallelism. The devised instructions are: xadac.load, xadac.vbias, xadac.vmacc, and xadac.vactv. Below, we provide a general description of our instruction encoding, followed by the respective instruction opcode functional description.

XADAC Instruction



Instruction Fields:

- **funct7:** Immediate value (imm).
- **vs2 / rs2:** Represents the second vector source or second register source.
- **vs1 / rs1:** Represents the first vector source or first register source.
- **funct3:** XADAC opcode.
- **vd / vs3 / rd:** Represents the destination vector, third vector source or the destination register.
- **opcode:** RISC-V opcode.

A. xadac.load

The vload instruction loads data from a memory address (rs1) into a vector register (vd). It copies elements from the source into the vector register's elements, repeating the content if necessary according to the specified immediate (funct7).

Algorithm 2 Vector Load

procedure VLOAD(vd, $rs1$, imm)
 $ptr \leftarrow$ CAST $rs1$ TO uint8_t
 for $i = 0$ **to** $V8LEN - 1$ **do**
  $vrf\_u8[vd * V8LEN + i] \leftarrow ptr[i\%imm]$

B. xadac.vbias

The vbias instruction sets the elements of a vector register (vd) to a specified bias value (rs1) for the first imm elements, while the remaining elements are set to zero, ensuring the specified bias is applied only to a portion of the vector.

Algorithm 3 Vector Bias

```
1: procedure VBIAS(vd, rs1, imm)
2:  for i = 0 to V32LEN − 1 do
3:   if i < imm then
4:    vrf_i32[vd ∗ V32LEN + i] ← rs1
5:   else
6:    vrf_i32[vd ∗ V32LEN + i] ← 0
```

C. xadac.vmacc

The vmacc instruction performs a multiply-accumulate operation on elements of two source vector registers (vs1 and vs2) and accumulates the result into a destination vector register (vd). For each element in vd, it multiplies corresponding imm elements from vs1 and vs2 and sums them.

Algorithm 4 Vector Multiply-Accumulate

```
1: procedure VMACC(vd, vs1, vs2, imm)
2:  for i = 0 to V32LEN − 1 do
3:   for j = 0 to imm − 1 do
4:    vrf_i32[vd ∗ V32LEN + i] +=
5:    vrf_i8[vs1 ∗ V8LEN + i ∗ imm + j]×
6:    vrf_u8[vs2 ∗ V8LEN + i ∗ imm + j]
```

D. xadac.actv

The vactv instruction applies an activation function to the elements of a vector register (vs3) and stores the result to a memory address (rs1). For each element in vs3, it shifts each value right by rs2 bits if it is positive, otherwise sets it to zero, and stores the result in the memory location pointed to by rs1.

Algorithm 5 Vector Activation

```
1: procedure VACTV(vs3, rs1, rs2, imm)
2:  ptr ← CAST rs1 TO uint8_t
3:  for i = 0 to V32LEN − 1 do
4:   if i < imm then
5:    sum ← vrf_i32[vs3 ∗ V32LEN + i]
6:    sum ← ( sum > 0 ) ? ( sum ≫ rs2 ) : 0
7:    ptr[i] ← CAST sum TO uint8_t
```

VII. Refactored Application

By understanding the operation of the MNIST CNN application and with the instructions defined for the co-processor, we were able to refactor the MNIST CNN Application to leverage the XADAC co-processor. Therefore, enhancing parallelism.

The algorithm 1 is the baseline refactored, with each input, output and kernel defined in each position. Algorithm 7 replicates the functionality of Algorithm 6 with the addition of vectorization. It initializes biases and applies activation functions using the custom instructions discussed in Section VI, effectively leveraging parallelism and optimizing the use of the accelerator.

Due to the poor optimization of convolution operations by the GCC RISC-V compiler, which used macros for each layer's convolution functions, GCC failed to optimize these functions effectively and merely unrolled the entire convolution. Although the code became efficient, the built hardware was not being fully utilized.

Algorithm 6 Convolution Baseline Refactored

```
1: procedure CONV(L, O, I, W)
2:  for oy = 0 to OY − 1 do
3:   for ox = 0 to OX − 1 do
4:    for oa = 0 to OA − 1 do
5:     for ob = 0 to OB − 1 do
6:      O[oy][ox][oa][ob] = BIAS
7:    for oa = 0 to OA − 1 do
8:     for wy = 0 to WY − 1 do
9:      for wx = 0 to WX − 1 do
10:      for ia = 0 to IA − 1 do
11:       for ob = 0 to OB − 1 do
12:        for ib = 0 to IB − 1 do
13:         O[oy][ox][oa][ob] +=
14:         I[iy][ix][ia][ib]×
15:         W[wy][wx][ia][oa][ob][ib]
16:    for oa = 0 to OA − 1 do
17:     for ob = 0 to OB − 1 do
18:      O[oy][ox][oa][ob] =
19:      ACTIVATION(O[oy][ox][oa][ob])
```

To address this, we had to refactor the convolution once more. This time, we wrote the critical parts of the code in assembly, which would have resulted in many lines of code. To manage this complexity, we used Python to generate our own assembly code.

With these modifications, we were able to better utilize the hardware.

Algorithm 7 Convolution Baseline Vectorized-ASM

```
1: procedure CONV(L, O, I, W)
2:  for oy = 0 to OY − 1 do
3:   for ox = 0 to OX − 1 do
4:    for oa = 0 to OA − 1 do
5:     VBIAS(SUM, BIAS, OB)
6:    for oa = 0 to OA − 1 do
7:     for wy = 0 to WY − 1 do
8:      for wx = 0 to WX − 1 do
9:       for ia = 0 to IA − 1 do
10:       VLOAD(VEC_I, &W[wy][wx][oa], OB ∗ IB)
11:       VLOAD(VEC_W, &I[iy][ix][ia], IB)
12:       VMACC(VEC_S, VEC_W, VEC_I, IB)
13:    for oa = 0 to OA − 1 do
14:     VACTV(VEC_S, &O[oy][ox][oa], SHIFT, OB)
```

To achieve a clean and efficient implementation, we refactored the fully connected layers as convolutions. This is based on the principle that any fully connected layer can be considered a convolution with a 1x1 kernel.

Then, we could explore the organization of the weights on the memory to express the parallelism.

The top blocks in Figure 2 represent their initial definitions in memory. The block labeled "u8" is the

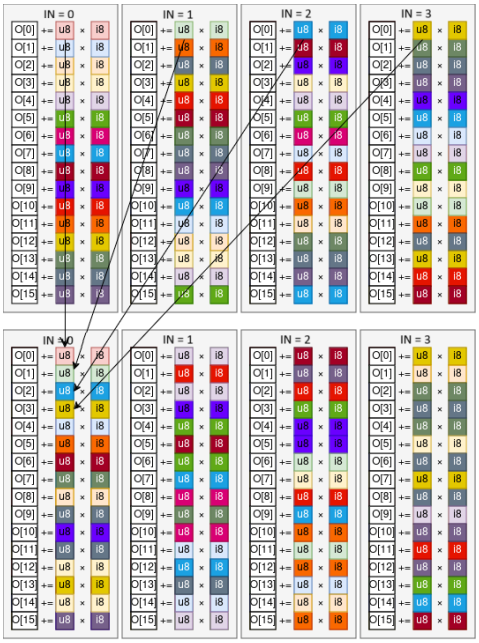


Figure 2. VMACC.

input, and ”i8” is the weight. After multiplication, the results are summed, completing the MACC operation.

To optimize the memory positions of the weights, we used a software layer function to identify frequently accessed weights and reorganize them in memory, revealing the memory access pattern. During each convolution operation, defined by ”CONV,” the initial MACC uses the red, green, blue, and yellow blocks. The objective was to change the execution order of weights and inputs to enable parallel operations.

Once the access pattern was identified, we could rearrange the memory layout, as illustrated at the bottom of Figure 2. The execution sequence of each MACC was then optimized by placing the rows of execution into the first four MACC executions, thus expressing parallelism.

The application input remains unchanged. The inputs for each stage of the convolution process still use the same data, precision, and types, ensuring consistency.

The code utilizes a cyclic redundancy check (CRC) by comparing the Baseline values and the output of each layer. The refactored parallelism version produces the same output for each layer as the Baseline, ensuring that the code faithfully reproduces the results without altering the layers or data.

VIII. Co-Processor XADAC

The Co-Processor XADAC is central to our optimization strategy, using two critical communication protocols: XADAC-IF for instruction offloading and AXI for memory access. The XADAC-IF is specifically developed to handle the reception and processing of offloaded instructions efficiently, ensuring optimal co-processor performance. The AXI protocol, in contrast, facilitates access to data cache memory. Notably, the AXI interface is later converted into the CV32A6 data cache protocol to allow for data cache access.

We initially used CV-X-IF to interface CVA6 with our co-processor but found that it could hinder our performance. Our analysis suggests that our co-processor could be optimized through pipelined instruction offloading, a process that proves to be challenging with CV-X-IF. Specifically, when the issue channel’s ready signal remains low for a clock cycle due to prolonged decode times, it prevents the issuing of any instructions during that period. Our proposed extension effectively decouple this dependency. Furthermore, this interface is also used internally in the co-processor to connect all the different modules, creating a modular and flexible design.

In Figure 3, we have a high-level illustration of the co-processor’s architecture. Where the arrows represents one of the three interfaces used: XADAC-IF, AXI or the Data Cache Interface.

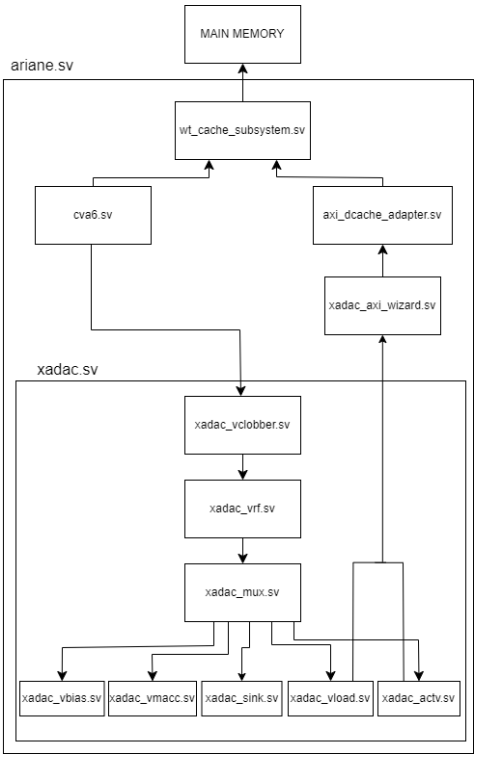


Figure 3. Integration of XADAC.

- **xadac_vclobber.sv:** Manages dependencies among vector registers.
- **xadac_vrf.sv:** Serves as the Vector Register File.
- **xadac_mux.sv:** Directs each instruction to the appropriate execution module.

- **xadac_vbias.sv:** Implements the xadac.vbias opcode.
- **xadac_vload.sv:** Implements the xadac.vload opcode.
- **xadac_vmacc.sv:** Implements the xadac.vmacc opcode.
- **xadac_actv.sv:** Implements the xadac.vactv opcode.
- **xadac_vsink.sv:** Addresses illegal instructions.

Additionally, beyond the XADAC top module but within the scope of ariane.sv, there are several auxiliary modules apart from the CVA6. These serve as bridges between different protocols. Moreover, while the cache is actually situated within the CVA6 module, for clarity and simplicity in our diagrams, we have chosen to represent it externally.

IX. Test and Verification

To ensure the accuracy and reliability of our modifications to the CV32A6 processor, we conducted extensive tests using the Questa advanced simulation tool. Questa confirmed the correctness of new instructions and prevented functional errors, ensuring the XADAC co-processor produced correct results. Additionally, we integrated the Spike RISC-V ISA simulator into our Docker environment, streamlining the verification process and enhancing test efficiency by ensuring consistency across development setups. After addressing potential instabilities with Questa, we employed Verilator for thorough verification of the accelerator. Verilator's rapid simulation and C++ integration capabilities were crucial in detecting and correcting design errors, refining, and validating the XADAC design before its final hardware implementation.

X. Docker

To incorporate the updated instructions, we modified the Docker image due to changes in the toolchain. Additionally, we integrated Verilator to facilitate isolated simulation of the XADAC accelerator and introduced CMake for compiling the revamped MNIST CNN application. To reconstruct our setup, please follow the detailed instructions available in our repository [3].

XI. Results

A. Recognized digits

1) Before: Results of Baseline.

```
Expected  = 4
Predicted = 4
Result : 1/1
credence: 82
```

2) After: Results of ADAC.

```
expected: 4
output: 4
credence: 82
end
```

B. Number of Cycles in Simulation

1) Before: Results of Baseline in Simulation.

```
# [UART]: image env0003: 1731593 instructions
# [UART]: image env0003: 2316326 cycles
```

2) After: Results of ADAC in Simulation.

```
# [UART]: INSTRUCTIONS,            81236
# [UART]: CYCLES,                 394287
```

C. Number of cycles on the FPGA board

1) Before: Results of Baseline in FPGA.

```
Expected  = 4
Predicted = 4
Result : 1/1
credence: 82
image env0003: 1731593 instructions
image env0003: 2353806 cycles
```

2) After: Results of ADAC in FPGA.

```
Current Configuration:
    IMAGE = img0003
    GLOBAL_PERF

INSTRUCTIONS,             81236
CYCLES,                  429700

expected: 4
output: 4
credence: 82
end
```

D. Maximal frequency of design

1) Before: Results of Baseline.
F_{max} = 1 / (20ns - 0.600ns) = 51.546 MHz

2) After: Results of ADAC.
F_{max} = 1 / (25ns - 0.298ns) = 40.482 MHz

E. Number of LUTs and flip-flops

1) Before: Results of Baseline.

Resource	Utilization	Available	Utilization %
LUT	20047	53200	37.68
LUTRAM	941	17400	5.41
FF	14277	106400	13.42
BRAM	66.50	140	47.50
DSP	4	220	1.82
IO	9	125	7.20
MMCM	1	4	25.00

Figure 4. Number of resources used.

2) After: Results of ADAC.

Resource	Utilization	Available	Utilization %
LUT	39239	53200	73.76
LUTRAM	941	17400	5.41
FF	26711	106400	25.10
BRAM	62.50	140	44.64
DSP	8	220	3.64
IO	9	125	7.20
BUFG	4	32	12.50
MMCM	1	4	25.00

Figure 5. Number of resources used.

XII. Interpretation of Results

The initial design recorded 2,353,806 cycles on the FPGA. After our optimizations, the cycle count reduced to 429,700 achieving a 5.47X speedup factor, corresponding to a 81.744% reduction in cycles. In simulation, we achieved a 5.87X speedup factor, corresponding to an 82.978% reduction in cycles.

It is important to highlight the changes in cache memory. We modified the size of Data Cache and Instruction Cache to 8,192 bytes and 7,168 bytes, respectively. We removed 1,024 bytes from the Instruction Cache to compensate for the addition of the Vector Register File.

The use of pipelining to address clock issues and the addition of parallel operators in our accelerator led to an increase in Flip Flops (FFs) and Look-Up Tables (LUTs). Despite this increase, our design still fits comfortably on the Zybo board.

XIII. Conclusion

In this work, we have successfully optimized the CV32A6 processor for accelerating AI applications, particularly the MNIST digit recognition task, by integrating a dedicated co-processor named XADAC. Our optimizations focused on vectorized execution techniques, custom instruction creation, and effective hardware-software co-design strategies.

The integration of XADAC and the refactoring of the MNIST application code resulted in a reduction in execution cycles where, decreased from 2,353,806 to 429,700 on the FPGA, yielding an acceleration factor of 5.47. By incorporating vector-specific instructions and leveraging the power of SIMD hardware, we were able to handle parallel computations more efficiently. The use of pipelining and parallelism in the accelerator led to an increase in the utilization of Flip Flops (FFs) and LUTs. However, this trade-off was justified by the substantial gains in processing speed and efficiency.

In conclusion, our project highlights the significant performance improvements achieved through the integration of a specialized co-processor and comprehensive optimization of both hardware and software components. Our approach, which includes the creation of custom instructions, not only enhances computational efficiency but also establishes a scalable framework for future enhancements.

References

[1] C. Lohr, "mini-rv32ima: A tiny C header-only RISC-V emulator," https://github.com/cnlohr/mini-rv32ima, 2024, accessed: 2024-05-13.

[2] A. Yasin, "A top-down method for performance analysis and counters architecture," *Intel Corporation, Architecture Group*, 2014.

[3] "ADAC Team: Accelerating AI applications on a RISC-V processor," https://github.com/alencar-felipe/cva6-softcore-contest, Branch: adac. Commit: d580966fd03cff871f008f84e028af589cce77c3. Accessed: 2024-05-13.