Accelerating AI applications on a RISC-V processor

Alessandro Mandrile
Master SETI
Télécom Paris
Palaiseau, France
alessandro.mandrile@telecom-paris.fr

Chunyu Zhang
Master Robotique
ENSTA Paris
Palaiseau, France
chunyu.zhang@ensta-paris.fr

Ianis Giraud
Master SETI
ENS Paris-Saclay
Gif-sur-Yvette, France
ianis.giraud@ens-paris-saclay.fr

Pierre-Alexandre Peyronnet
Master SETI
ENS Paris-Saclay
Gif-sur-Yvette, France
pierre-alexandre.peyronnet@ens-paris-saclay.fr

***Abstract*—The RISC-V instruction set is receiving increasing attention both in the world of high-performance computing and in that of embedded applications. In both these areas, it is particularly important to reduce power consumption in relation to computing performance.**

This reduction in power consumption can be achieved in two ways: – by reducing the number of transistors used at any given time, and therefore the complexity of the instruction set and microarchitecture, – or by introducing specialised accelerators to reduce the execution time for a given application.

The CVA6 core developed by OpenHW Group, implementing a small part of RISC-V standard, presents a relatively low complexity, making it synthetisable on the majority of FPGA boards on the market. Despite this, the smaller size of the instruction set limits the possibility of efficiently implementing many algorithms. However, the freedom given by the open nature of the project let us customise it at will.

***Index Terms*—Neural network, MNIST, RISC-V, Co-processor**

I. Introduction

The MNIST database is a standard collection of thousands of images that can be used to train and test handwritten digit recognition algorithms. The application considered as part of this competition is a simple convolutional neural network to recognise the handwritten digits in this dataset.

The CV32A6 core is an open-source implementation of the RISC-V standard developed by OpenHW Group and Thales. The core, described in SystemVerilog, can be implemented on FPGA chips as a soft-core.

As part of the competition organised by Thales, we are proposing a series of architectural improvements to the CV32A6 that will speed up the execution of the MNIST application by reducing the number of clock cycles required for its execution. One of the constraints imposed by the competition rules was that we could not design a "coarse-grained" accelerator, or reduce the operating frequency of the core too much.

A. Application MNIST

The application under consideration is a four-layer neural network for solving the MNIST task. The network consists of two convolution layers followed by two fully connected layers (Fig. 1).

Starting with a 24×24 pixel greyscale image, a first convolution `conv1` by a 4×4 kernel produces 16 11×11 images. A second convolution `conv2` by a 5×5 kernel gives 24 4×4 images. Finally, a fully connected layer `fc1` reduces these images to a vector of size 150. A final fully connected layer `fc2` reduces this vector to a vector of size 10, the maximum of which corresponds to the digit recognised in the original image.

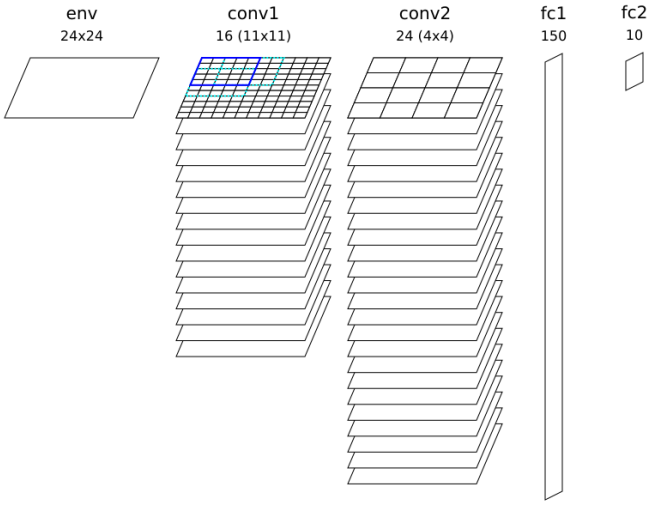

Fig. 1. Neural network architecture.

II. Profilage de l'application

CVA6 implements the hardware performance counters described in the RISC-V[1] standard. These counters can be used to profile the application in detail and determine its hot spots.

Thanks to this profiling, we determined that almost 94% of the computation time was spent in the function `macsOnRange`, the implementation of which is given in Listing 1. This function performs a multiplication-accumulation

operation between two arrays. It is used for both convolutions `conv1` and `conv2`, and for fully connected layers `fc1` and `fc2`.

Listing 1
MACSONRANGE IMPLEMENTATION

```
void macsOnRange(const uint8_t* inputs,
                 const int8_t* weights,
                 int32_t* weightedSum,
                 int nb_iterations)
{
  for (int i = 0; i < nb_iterations; i++)
    *weightedSum += inputs[i] * weights[i];
}
```

A. Performance

The application as initially provided has a number of cycles per instruction (CPI) of 1.35. In particular, pipeline stalls take up 14% of execution time, indicating poor management of data dependencies.

Another area for improvement would be to make better use of the registers' width. Indeed, although the CVA6 operates on 32-bit registers, the operations performed by the application are mostly on 8 bits. Integrating SIMD operators should therefore make it possible to significantly reduce the number of instructions executed.

III. ARCHITECTURE ANALYSIS

A. Starting architecture

CVA6 is a single-issue 6-stage pipelined processor.

The first two stages are responsible for generating the program counter and are so closely linked that they appear as a single unit in the RTL. This is where the instruction fetch logic and the speculative logic are located. The stage contains a RAS (Return Address Stack), a BTB (Branch Target Buffer) and a BHT (Branch History Table). These already present optimisations are important because, otherwise, each branch would only be calculated at the execution stage and executed at the last stage, the commit stage.

The third stage manages the decompression of 16 bits instructions, if necessary, and decodes them.

The fourth stage is the most delicate. It contains a scoreboard, a fifo buffer that stores instructions in program order, so that they can be executed out of order. This stage also contains the register file and the forwarding logic from the functional units and the commit stage.

The execution stage contains an ALU (Arithmetic Logic Unit), possibly an FPU (Floating Point Unit), the branching calculation unit, which is always used to confirm or deny a prediction made by the frontend, and a LSU (Load Store Unit), a complex unit which also contains an MMU (Memory Menagement Unit) and a TLB (Translation Lookaside Buffer).

The last stage is the commit, during which the processor writes the results of the oldest instructions on the scoreboard to the architectural registers, as well as executing the exceptions recorded during the previous stages.

B. Interesting parameters

The processor is highly configurable; each buffer and memory can be modified using a parameter in the configuration file. There is also a data and instruction cache with pseudo-LRU replacement policy, whose size and shape are configurable at synthesis. Among the most interesting parameters we have identified we have: the size of the RAS, BTB, BHT, the size of the scoreboard, because when it is full, it stops the upstream stages, the modification of associativity or the length of a cache line. There is also a modifiable buffer in the load storage unit, at the output of the load module; its omission introduces a combinatorial path between LSU and cache, but potentially reduces the number of cycles required for a load.

C. Profiling analysis

Using HPCs (hardware performance counters), we were able to identify the hot spots in the program, in particular the `macsOnRange` function, which consumes a total of 94% of cpu time. We decided that we needed to speed up this function and the most direct way we could think of was to follow the SIMD philosophy. As the data size is 8 bits, memory accesses are via lb (load byte); the optimisation attempts to force memory access to the more natural 32-bit Word, with a custom instruction that accumulates by multiplication 4 pairs of 8 bits in one cycle. HPC analysis of a version of the algorithm that exploits this optimisation allowed us to deduce that:

- There is no need to increase the size of the RAS, since most of the functions are inline. The call stack therefore does not exceed the RAS' size of 2 ;
- the code has only 9 unconditional jumps and 28 conditional jumps, which is enough to never be eliminated from the BHT and BTB. Despite this, the miss-predict rate is 8%;
- The pipeline stalls 140 000 cycles because of the full scoreboard; and
- There aren't many remaining stalled cycles(25k).

D. Optimised configuration

A first round of optimisation was about blindly increasing these interesting parameters. Since the first objective of the contest was to reach the lowest possible clock count over any other metric, at this stage we ignored resource utilisation, as soon as timing requirements were met and the totality FPGA resources wasn't used. We therefore increased the size of BHT and BTB, but this did not bring any improvement, as the limitation is probably due to the algorithm used and not to access conflicts. We increased the size of the scoreboard to 16 entries, which resulted in 0 scoreboard stalls. Unfortunately, we later realised that such a large size increased the combinatorial path of the scoreboard to the point where it became the critical path and the design did not respect the clock period constraints, with a time more than 20% slower than the starting time, so we limited it to 8 entries, obtaining a still good value of 8500 "Scoreboard Full" cycles. Caches were not explored at first, because of the low miss rate, 4%.

We have also enabled the ability in the commit stage, already implemented in the processor, to retire two instructions in each cycle. Finally, we have removed the buffers in the interface between the load/store unit and the cache memory, observing that the critical path has not deteriorated.

IV. Loop predictor

The final architectural improvement proposed is in the BHT. A mis-predict rate of 8% would already be good, but as the original predictor is rather primitive, we have tried to do better. We've added a loop predictor, which remembers the previous sequences of jumps in the same direction and counts at which point in the sequence we are at a given moment, to know whether or not to continue jumping. Loop counter is composed, for each entry in the BHT, of a counter of how many times the program jumped in the same direction (sequence of N-times taken/not taken). When a change in direction is registered (i.e. actual jump outcome != previous outcome), the value in the counter is transferred to the corresponding register representing the last N-taken/not-taken jumps. After the first training, such a predictor has 100% hit on any branch that respects the sequence $T_0..T_j, NT_0..NT_k$, so also loops, that are simpler. In addition, the new feature do not harm previous correct predictions, as the new predictor works in parallel with the existing 2-bit saturating counter and is only used when it provides better performance than the latter. This is achieved through an arbiter that selects which predictor to use based on which one has given the best results in recent iterations. An overall architecture can be seen in Figure2. This change resulted to a miss-predict rate of 0.22%, saving 35k cycles in the version with a 3-source register coprocessor. We also noticed a significant increase in stalls (10k). We think the cause is that an almost linear behaviour of the pipeline (since we almost never flush) has revealed some data dependencies that were previously hidden by flushing.

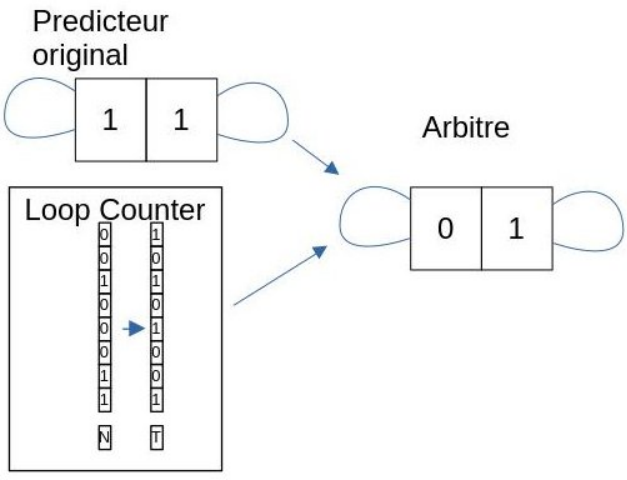

Fig. 2. Predictors and arbiter architecture.

V. Core-V eXtension InterFace

A. Interface analysis

In order to add a MAC compute unit to the CVA6, we opted to implement a coprocessor on the CV-X-IF interface [2]. This architectural choice is justified by the fact that it is much simpler to connect a module to a pre-existing interface than to modify the architecture of the processor ALU.

The principle of the CV-X-IF is to offload to the coprocessor instructions that the processor does not know.

The coprocessor then responds to the processor via various interfaces, indicating among other things whether it is indeed capable of executing the instruction. This interface is very flexible thanks to its set of parameters, which can be defined by the user before synthesis, for example to authorise instructions using 3 operands.

The CV-X-IF is made up of several interfaces, of which only 3 are implemented in the CVA6 :

- The issue interface
- The commit interface
- The result interface

The issue interface allows the processor to initiate an instruction offload, by transmitting information such as the instruction code and operands. In return, the coprocessor will be able to indicate whether it accepts to handle the execution and whether the CVA6 will have to wait for a result. We have seen that for the system to work properly, the coprocessor must validate all transactions initiated by the processor, even if it does not accept the instruction.

The commit interface is supposed to be used to validate or not an instruction. If the CPU is performing speculative execution, it can tell the coprocessor via this interface whether a speculation is bad and whether it should therefore return to a previous state. When the processor knows whether or not to accept the instruction, it will start a commit transaction, specify an instruction identifier that the coprocessor will recognise and indicate whether or not to "kill" the instruction. With CVA6, on the other hand, the transaction is initiated at the same time as the transaction issue and the "kill" signal is always in the low state. There is therefore no need to manage this interface.

Finally, the result interface allows the coprocessor to send the result of the instruction to the processor. This result can be the result of a computation (the MAC operation in our case) or the raising of an exception.

B. Le coprocesseur

The coprocessor architecture is extremely simple, containing :

- A MAC4B module
- An decode module

The first module performs the MAC operation on 4 pairs of 8-bit integers, enabling us to perform 4 accumulative multiplications in a single cycle. The second block tells the processor whether the instruction it has just offloaded is valid or not and whether the coprocessor should send a result.

In the first implementation we added an R-type instruction with 3 registers (two inputs and one output), the format dedicated to arithmetic and logic operations. This format uses 15 bits for register addresses and the remaining 17 bits to describe the operation to be performed. Once implemented, this coprocessor gave a correct result for the MNIST program in around 787 000 cycles.

We then wanted to add a 3rd operand in order to directly sum the new result with the MACs calculated during the previous iterations. So, instead of calculating $\mathrm{rd} = \sum \mathrm{rs1}_n \times \mathrm{rs2}_n$, we would calculate $\mathrm{rd} = \mathrm{rs3} + \sum \mathrm{rs1}_n \times \mathrm{rs2}_n$, where rsi is one of the operands and rsi_n is the n-th byte of rsi. The instruction format to be used here is therefore different. We now use the R4 format, which uses 20 bits for register addresses and the last 12 bits to describe the operation performed. The only other uses of this format are the MAC operation (and its variants) of single and double precision floating point numbers [1]. Having synthesised the new architecture, we were able to run MNIST in 670,000 cycles. The table I summarises the speed-ups we achieved with the coprocessor.

TABLE I
CYCLES AND ACCELERATION GIVEN BY COPROCESSOR (ATTENTION, CYCLE COUNT CORRESPOND TO A SINGLE EXECUTION)

Coprocessor	***Nothing***	***MAC4B***	***MAC4B_RS3***
Cycles	2353693	786726	644106
Speed-up	—	2.99	3.51

C. Code modifications

To add a custom instruction to the compiler, we modified `include/opcode/riscv-opc.h` and `opcodes/riscv-opc.c` in the **binutils-gdb** project, which is compiled on the first build of the sw-docker. This files instruct gcc how to translate the mnemonic for the instruction `mac4b` that we added.

The main modifcation to the code we planned to bring was a 4 times loop unfold on the `MacsOnRange` for loop, to be able to access all 4 values for the `mac4b` in a single iteration. Unfortunately, at first we ignored alignment of the 4-byte words, but later we understood the possible problem we introduced with the new way of accessing data. Indeed, we don't have any assurance on the alignment of arrays in the `MacsOnRange`, for example because kernels in convolutions slides of 1 byte for every element in the result. A first naive version of the unrolled version of `MacsOnRange` is show in Listing2.

Listing 2
MACSONRANGE IMPLEMENTATION

```
int iter = 0;
#ifdef UNROLLED
for (; iter < nb_iterations-3; iter+=4) {
  *weightedSum = mac4b(*weightedSum,
                       inputs[iter],
                       weight[iter]);
}
#endif
for (; iter < nb_iterations; iter++) {
  *weightedSum += inputs[iter] * weights[iter];
}
```

To account for the possibility of unaligned accesses, we designed a new function, `align32` that shifts byte values between 2 4B groups of an offset. The offset is evaluated before looping and depends on the amount of bytes of which the array is misaligned. This means that for each call to `mac4b`, `align32` is called twice. A simplified version of the main `MacsOnRange` loop is shown in Listing3; note that what is a simplified version of what we implemented; hovewer the full code is visible on the git, with comments.

Listing 3
MACSONRANGE IMPLEMENTATION

```
i = 1;
for (; iter < nb_iterations-4; i++, iter+=4) {
    weight2 = base_weights[i];
    input2 = base_inputs[i];

    input = align32(input1, input2,
        offset_inputs);
    weight = align32(weight1, weight2,
        offset_weights);
    *weightedSum = mac4b(*weightedSum,
        input, weight);

    weight1 = weight2;
    input1 = input2;
}
```

VI. PARAMETERS TUNING

As previously mentioned, a lot of micro-architectural blocks can be configured easily with the config files under `core/include`, where we created our own `cv32a6_mac_fpga_config_pkg.sv`.

The main objective of this last part, was to optimise resource utilisation without increasing the total clock count.

Since the miss-predict rate was already low, we decided to reduce FPGA utilisation starting by reducing BTB and BHT sizes. Therefore, we first reduced the size of the BTB and then adjusted the BHT values. In addition, for the instruction cache size(ICACHE), the data cache size(DCACHE) and the scoreboard size, we used a binary search method to determine their minimum effective size. Finally, we identified the optimal configuration as V2 on the TableII, which reduced the use of FPGA resources compared with version one.

During the subsequent optimisation process, we adjusted the ICACHE and DCACHE associativity and line size. Despite several attempts, we found that, no other configuration was able to effectively reduce the number of clock cycles or reduce resource utilisation.

Finally, we explored some values for the lenght of the exit/return buffer of the LSU; Not only we reduced FPGA utilisation, but we also further reduced cycles required to execute MNIST application of an additional 2%. Thus, we decided to keep V3, as our last version. With the application

TABLE II
SUM UP OF MAIN OPTIMISATION STEPS

version	V1	V2	V3
BTB	32	2	2
BHT	128	8	8
ICACHE	8192	512	512
DCACHE	8192	8192	8192
SCOREBOARD	8	8	8
NrLoadPipeRegs	1	1	0
NrStorePipeRegs	0	0	0
NrLoadBufEntries	2	2	2
Cycles	637057	637949	624626
Branches(%)	0.154	0.215	0.223
LUT(%)	45	44.77	44.65
LUTRAM(%)	6.58	6.2	6.2
FF(%)	14.36	14.39	14.32
BRAM(%)	47.5	47.14	47.14
DSP(%)	1.82	1.82	1.82
IO(%)	7.2	7.2	7.2
MMCM(%)	25	25	25

completing in 621699 cycles, it is the configuration chosen on the repository.

VII. TIMING CONSTRAINTS

As previously mentioned, our design violates the initial clock period limit of 20ns. The cause is the longer combinational path introduced when increasing the scoreboard to 8 entries. However, such violation is accepted in the contest as it respects the major period of 25ns, since the most negative skew is -1.48ns.

VIII. FINAL RESULTS

We inform the organiser that we never manage to run a simulation of the MNIST application. All the results we found come from manual tests on the FPGA.

	Before	After	%
Digit	4	4	-
Credence	82	82	-
Cycles (simulation)	-	-	-
Cycles (FPGA)	2354247	624626	-73.47
Max Freq	51.54	46.62	-9.54
LUT	8582	12012	-
FlipFlop	4611	5722	-

IX. FUTURE IMPROVEMENTS

This brings us to the next development: the implementation of dual “issue” in the processor. Unfortunately, we haven’t had time to make this improvement, but according to the study carried out, it shouldn’t be too complicated. It will involve duplicating most of the signals inside the issue stage and at its interfaces with EXE and ID, as well as increasing the depth of instruction fetch. Our hypothesis is that such an improvement, although beneficial for the CPI, will only further worsen the poor management of data dependencies in the processor. The current architecture blocks the ”issue” in the case of WAR, WAW and RAW dependencies, and even if an instruction following the blocked one has no dependencies, it cannot be started before the preceding instructions.

REFERENCES

[1] A. Waterman, K. Asanović, and J. Hauser, eds., *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*. Dec 2021.

[2] *CV-X-IF Interface and Coprocessor*, 2020.