Comparison of 3 Convolution Neural Networks (CNN) hardware accelerators on CVA6 Risc-V core.

BABIN-RIBY Hugo
Engineering apprentice
ISAE - SUPAERO
Toulouse, France
Hugo.BABIN-RIBY@student.isae-supaero.fr

DION Arnaud Phd.
Associate Professor, DISC
ISAE - SUPAERO
Toulouse, France
arnaud.dion@isae-supaero.fr

***Abstract*—AI algorithms are thriving, and the execution of those algorithms is very power and time consuming. This can lead to important operating costs for some Artificial Intelligence (AI) models. This paper dives into an exploration of strategies aimed at operating convolution neural networks faster using simple hardware acceleration, with a particular emphasis on the acceleration of the convolution operation using a Multiply Accumulator (MAC) and also adopting a certain strategy for running calculations in parallel. By looking into existing methodologies and proposing novel approaches though the customisation of the CVA6 Risc-V core, our objective is to present a comprehensive overview of the computer approach to AI convolution operations (like the MNIST database example) and what strategies we can use to accelerate it.**

This paper serves as valuable insights for researchers, practitioners, and enthusiasts on cost effective hardware for embedded AI solutions. We will dive into the multiply-accumulation operation and how it can make convolution faster, we will then make 3 different designs in systemVerilog based on the "Core-V Application 6 stages" Core (CVA6), a RISC-V processor (32 bits version), and see what approach was the most effective and thus paving the way for designing accelerators for larger and broader applications.

The design flow will also be detailed through an in-depth examination of convolution neural networks, software, and finally hardware.

I. INTRODUCTION

A. Context

The tech industry is moving fast and the most recent waves has been caused by AI. The models are getting bigger and bigger and thus take more and more resources to train but also to operate [1]. Every AI Model is unique and each application requires a very specific set of operations. One of the most used application in AI is image recognition [7]. Image recognition uses convolution layers that are able, once trained, to recognise specific features within an image, thus being able to classify this image depending on how the engineers trained the model [7]. Though the convolution is very powerful, it also is a very demanding operation. A powerful way to make those calculation is creating threads in a Graphical Processing Units (GPU). Designed to run multiple math operations at once, GPUs are very useful for matrix-wise operations [12].

The very core idea of the GPU is to get the most calculations done on each clock cycle. In fact the main usage for GPUs are games graphics and shaders, that mostly involves matrix and vector wise operations. But AI Engineers are using this general-purpose hardware to train convolution-based AI models [2]. GPUs are very powerful but this power comes at a cost, this translates into energy consuming chips and involves a very important investment [5]. The goal will be to take inspiration from this idea of making multiple operations in one single cycle to make very efficient piece of hardware optimized for convolution operations and very cost efficient by making it very specialized.

This paper has been elaborated in the context of an R&D session at ISAE-SUPAERO, a project aimed at giving us (FISA program students) an introduction the the research world. My work was supervised by Mr A. Dion PhD, associate professor at ISAE-SUPAERO. The main constraint of this R&D project was the inability to intervene on the algorithm's logic nor the AXI bus (when it comes to CVA6 architectural modifications). These constraints will lead us to take very specific paths concerning our design strategies.

B. Scope of the paper

This paper will mainly focus on discussing the improvement of the forward propagation of data through convolution. We will provide MNIST forward propagation execution results from 3 different designs, each answering to a specific set of constraints. Those result will be presented (in section V) as :

- The number of clock cycles necessary to complete execution.
- The number of instructions necessary to complete execution.
- Timings, max frequencies & FPGA ressources utilisation.

With the algorithm executed staying exactly the same. The maximum delay authorized is 25ns and the CVA6 designs will be implemented on the Zybo Z7-20 FPGA. These designs will be based on CVA6 architectural improvements

C. State of the art

The CVA6 RISCV core is pretty recent [15] researchers already customized this core for hardware acceleration [3]. These researches paved the way for the first two designs detailed in the fourth section of this paper (involving creating a custom instruction on the first and modifying the binary tool chain on the second).

A lot of work was done on RISC-V to accelerate AI networks [4], but no significant work has been done on CVA6. Researchers already did work on leveraging simple MAC hardware for AI acceleration on RISC-V [4] and this paper aim at contributing to CVA6 the same way these researchers contributed to other RISC-V Cores providing adapted designs.

For the parallel computing design, huge amounts of work were already done in the field of parallel computing for broad and specific applications [12]. However the strategy use in this paper (3rd design) is very specific and serves as a solid base and proof of concept for researchers and RISC-V enthusiasts.

D. A Word on RISC-V & CVA6

RISC stands for ”Reduced Instruction Set Computing”. This design philosophy emphasizes simplicity by providing a streamlined set of core instructions with well-defined behavior. By minimizing the complexity of its instruction set, RISC-V aims to achieve efficient execution, simplified hardware design, and improved compiler optimization [14]. This approach combined with its completely open-source philosophy makes it more than just an alternative to closed source architectures and our number-one choice to develop custom hardware.

The Core-V Application class 6 stage processor (CVA6) is a RISC-V processor developed by the Open Hardware group. It is part of the CORE-V family which regroup all the RISC-V core they develop [15]. This is the core we’ll use to implement for the custom designs.

E. A Word on MNIST

MNIST is a database on which we can run AI models. It is considered the ”hello world” of AI algorithm. It consists in a set of images representing hand drawn numbers, each 28x28 pixels of grayscale values encoded on 8bits (ranging from 0 to 255) [7]. In this paper, a simple use case of a MNIST C program will serve as a support to study the computer approach to operating a convolution neural network.

II. Understanding Convolution Neural Networks (CNN) and convolution

A. Neural networks, convolution operation and CNNs

The core technology is based on forward and backwards propagation : we propagate the input data forward in a network of weighted nodes, organized in layers, and then determine gradients to adjust the weights of the different operations made by the model (back propagation) [7]. See example in Figure 1 with a fully connected layer.

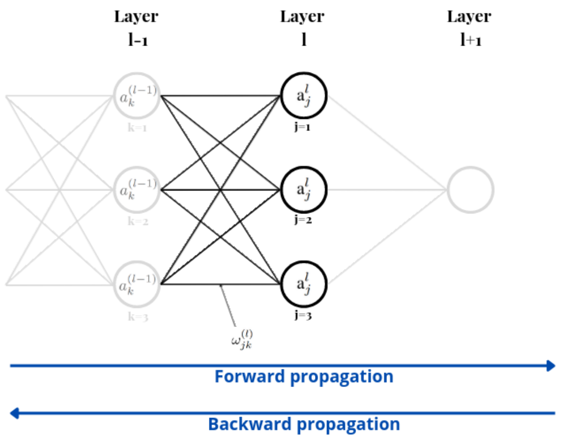


Figure 1 : Forward and backward propagation

This fully connected layer computes new data using weights on the input data from the previous layer. We call this new data ”activation” and will get passed to the next layer in the exact same way it got passed from the previous one [13]. Layers like this stack on top of each other in a certain engineered way depending on the data we use for training but also the type of result we want [7], [13]. We can describe the activation of a node j in the layer l like so [8]:

$$a_j^l = \sigma\left(\sum_{k=0}^{N-1} \omega_{jk}^{(l)} . a_k^{(l-1)}\right)$$

With:

- σ The activation function that get the activation in a range between 0 and 1.
- $\omega_{jk}^{(l)}$ the weights of the node j in layer l. (one for each previous node k)
- $a_k^{(l-1)}$ The activation of the previous node k.

these networks of nodes and weights are a good candidate for a range of basic data sorting but it is not very efficient for image recognition [13]. Introducing : the convolution layers (see Figure 2):

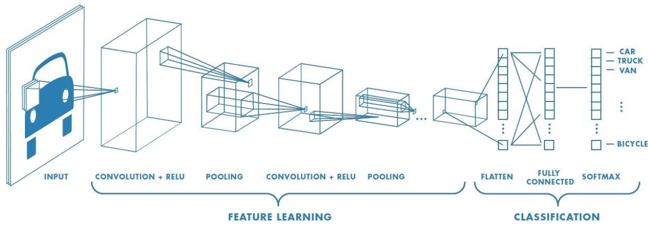


Figure 2 : The convolution neural network. Source : A Study on CNN Transfer Learning for Image Classification: [10]

The CNNs introduces convolution filters instead of simple nodes. [13], [7] If you are familiar with image processing, you know how powerful convolution can be to pick on certain patterns in an image, especially edges [11]. The CNN will use kernels optimized through training in order to recognize patterns in the data that is propagated (an image for example) [11], [7]. It is very important to note that the convolution operation is very demanding in terms of computing power as basic convolution algorithms are a $O(n^2)$ complexity (See section II. B. For matrix-wise convolution formula), thus making it our target for further optimisations. Let us dive into the convolution operation :

$$\text{out}_j^l(x) = I^{(l-1)} * K_j^{(l)}$$

With:

- $out_j^l(x)$ The output matrix (a new image in the image recognition context for example).
- $I^{(l-1)}$ The input image of layer l-1.
- $K_j^{(l)}$ The convolution kernel, comparable the a node in *"traditional"* fully-connected layers.

As we can see, the operation is pretty simple at a high level and we can visualize how this operation works to detect edges in an image (See Figure 3):

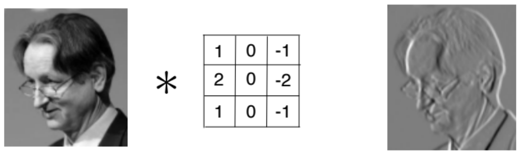


Figure 3 : The convolution applied to an image. Source : Hardware Acceleration for Neural Networks, Sahar Eslami

We can see that the kernel picks on a certain pattern [11], especially vertical edges in this example, which makes sense if we pay close attention the the kernel value that are symmetrical along a vertical axis (with the exception of adding a negative sign) [7].

B. Breaking down the convolution operation

We are now going to break down this convolution operation to get a deeper understanding of how it works and how to optimise it. Here is the mathematical definition of the discrete convolution:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m].g[n-m]$$

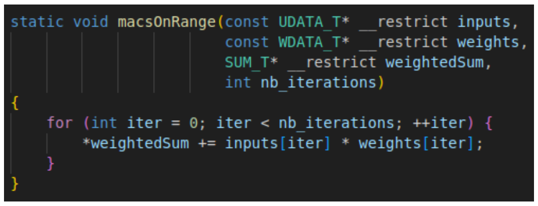
Let us also apply this formula to determine the output of one pixel in the output image of index (i,j) in layer l from an input image I from layer l-1 convoluted with the associated kernel K.

$$\text{out}_{ij}^{(l)} = \sum_{k=1}^{S}\sum_{l=1}^{S} I_{(k+i)(k+j)}.K_{kl}$$

As we can see, each output pixel is a sum with a number of members equals to S^2 (maximum kernel size). But more importantly, this operation has to be repeated for each and every output (i,j) with (i,j) each ranging from 0 to $(size(I) - S)^2$. This can very quickly sum up to enormous amounts of micro operations (multiplications and sums). We can now understand where the power of GPUs comes from : the ability to make all these operation in parallel makes it a lots faster [12]. In this paper, we will focus on finding a way to optimise this operation with and without parallel computing. We will not create a "GPU" per say though, as creating such a piece of hardware involves creating general purpose ALUs with separate controllers and custom libraries to handle threads. Instead, we will focus on understanding the computer approach to the MNIST program and create dedicated hardware that will be specialized in accelerating this specific algorithm.

III. Software study : Breaking down the RISC-V computer approach to convolution

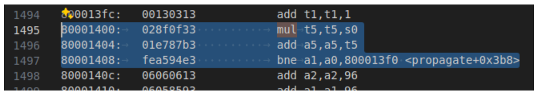
Lets take a look at a C program that operate a MNIST forward propagation. We quickly spot the heart of the convolution operation :



```
static void macsOnRange(const UDATA_T* __restrict inputs,
                        const WDATA_T* __restrict weights,
                        SUM_T* __restrict weightedSum,
                        int nb_iterations)
{
    for (int iter = 0; iter < nb_iterations; ++iter) {
        *weightedSum += inputs[iter] * weights[iter];
    }
}
```

Figure 4 : Convolution code snippet

Here we can see the "macsOnRange" function. The name already gives us a hint on what it does : Multiply and accumulate. The value of $weightedSum$ is the value $out_{ij}^{(l)}$ that we described earlier using formulas. Now knowing the name of the function and what it looks like, we can compile and disassemble the MNIST program to RISC-V assembly and investigate the resulting code to get an overview of how the computer will approach this problem at a lower level. [18]:



```
800013fc:   00130313            add t1,t1,1
80001400:   028f0f33            mul t5,t5,s0
80001404:   01e787b3            add a5,a5,t5
80001408:   fea594e3            bne a1,a0,800013f0 <propagate+0x3b8>
8000140c:   06060613            add a2,a2,96
```

Figure 5 : Resulting RISC-V Assembly pattern

We notice a pattern where we multiply values, add (or accumulate) this value in the a5 register and branch earlier in the sequence if a certain condition is not met... *In other words : a loop of multiplication and accumulation.* we can now imagine a design that would execute those multiplications and additions at the same time. This design would concatenate the "mul" and "add" instructions into one, thus making us gain a lot a cycles.

IV. Hardware study

This section of the paper will describe the different designs used to improve CVA6 (Also called Ariane) on MNIST. The results will be compared in the fifth section : "Results". This sections focus on the design flow to really understand where the result comes from and how each solution can be scaled to other applications. For each design, we will take a high-level approach and then provide some details on how to actually implement this in the CVA6 core. All the designs are based on Core-V eXtension Interface (CV-X-IF), a co-processor included in CVA6 that we can use to handle custom instructions [15]. It behaves as a regular unit in the execute stage so we can customize it with any hardware in order to execute any instruction we want :

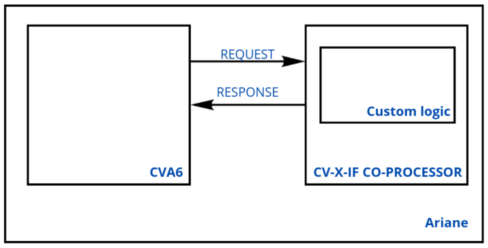


Figure 6 : CV-X-IF & CVA6 Ports block diagram

A. A word on the Multiply-Accumulator (MAC)

Here is a block diagram of the MAC design :

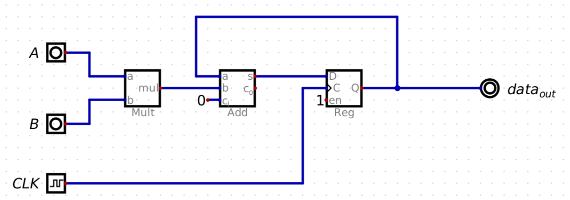


Figure 7 : simple MAC Design (Software : "Digital")

As simple as this design can be, it allows us to feed data for multiplication without having to specify to add afterwards. In this paper, we will design 3 improvements to CVA6, the first one being hardware only and the next two involves custom instructions & parallelism. Yet, all 3 designs rely on the MAC because it allows us to multiply two sources and accumulate the result with only one instruction (and thus only one clock cycle instead of two). This design is fairly simple and can be declined in many ways depending on the strategy we want to use. We can pipeline it if the data gets too large, we can add a dedicated controller to run a thread with one single MAC module, we can create an array of them to use parallel computing (*which we will*), etc..

B. First design : Hardware only

The first design is based on hardware only. Lets take a look at our example assembly code from Figure 5 (re-adapted):

```
Instr @ PC+0x0     (...)
Instr @ PC+0x4    MUL RD1, RS1, RS2
Instr @ PC+0x8    ADD RD2, RD2, RD1
Instr @ PC+0xC     (...)
```

With :

- $Instr@PC + X$ The address in the memory where the instruction lives compared to the Program Counter (PC)
- RDx Destination registers
- RSx Source registers

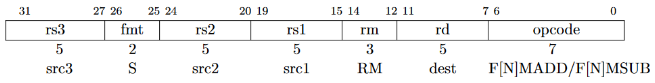
This sequence of instructions multiplies and accumulate a weighted sum. The idea behind this first design will be to add an interceptor that looks at the currently fetched instruction and the one right after. This interceptor would then check if those instruction are a MAC operation, and if so, it will assemble an entirely new custom instruction from those two. This new instruction will be the following.

```
MAC RD, RS1, RS2, RD
```

With:

- RD the final destination register ($weightedSum$).
- $RS1$ & $RS2$ the two sources registers.
- RD (the last one as a source) is the third source register (to read the actual value in RD from the register file).

We add RD as a third source to be able to read the actual value in the destination register and add it to the multiplication result to accumulate the value. This instruction format corresponds to the r4-type [18] :



31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	fmt	rs2	rs1	rm	rd	opcode
5	2	5	5	3	5	7
src3	S	src2	src1	RM	dest	F[N]MADD/F[N]MSUB

Figure 9 : r4 type instruction. Source : RISC-V ISA [18]

We will use a custom OPCODE in order for the instruction to be handled by CV-X-IF. 0010011 is reserved for custom instructions and will be the one we'll use for our MAC instructions. Back to the interceptor (we'll also call it merger) : here is a high level overview of its inputs and outputs (I/Os).

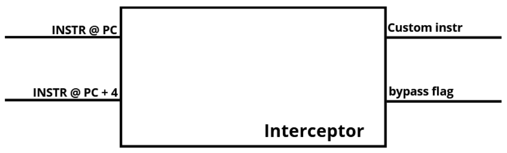


Figure 8 : simplified MAC interceptor / merger

The bypass flag will be raised so a simple MUX can choose either the original instruction or the custom one for fetching. We then test bench it and add it to CVA6. We can first get a broad idea of how to do such an integration using a very basic RISC-V block diagram [14], [16], [18] :

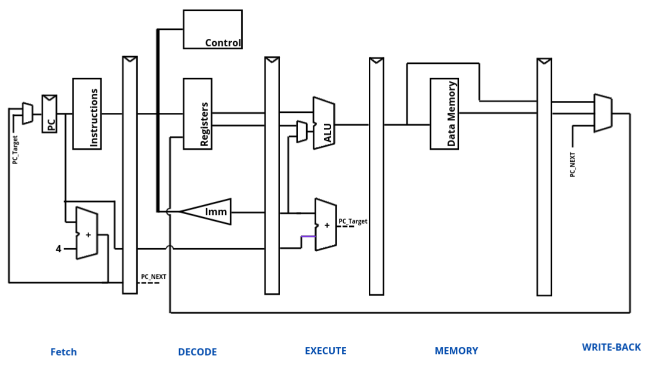


Figure 9 : simplified RISC-V pipelined single issue processor

As you can see in this simplified version of the RISC-V pipelined single issue processor, we have an instruction memory we can try to add the interceptor to. The idea is to fetch two instructions at a time, one to actually fetch and the other would be the instruction living in memory at PC+4 in order to check for an incoming MAC sequence. If a MAC sequence is intercepted, simply raise the bypass flag and increment PC by 8 instead of 4 (in a byte addressed memory) and the instructions are automatically merged using hardwired logic & values. If we translate all these requirements into a simplified block diagram, we get :

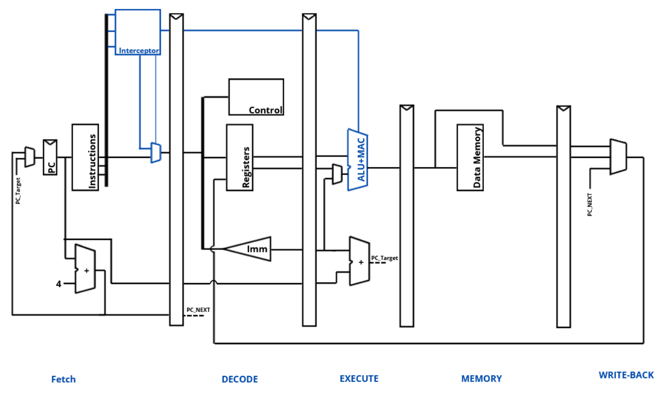


Figure 10 : simplified Interceptor / Merger MAC integration to RISC-V

The main problem with this simplified version is where the interceptor lives in the design. In CVA6, such a place would be tricky to pull of as we would need a new of way of fetching instructions from memory (involving messing with the cache system etc...). In the context of the contest, we decided to go for a simpler way : using the CVA6 instruction queue [15]:

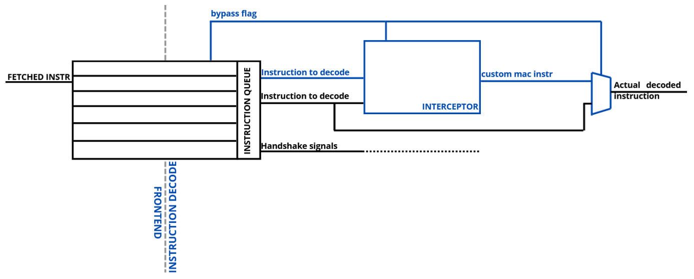


Figure 11 : Interceptor / Merger integration to CVA6's instruction queue

In CVA6, fetch stage feeds instructions in a FIFO called ”instruction queue” [15]. This queue allows us to peek at every instructions with our interceptor (using a modified FIFO). We also add to the FIFO a ”pop” signal. This way we can pop the next instruction (ADD in case of a MUL-ADD sequence) to skip it, this allows us to merge two instructions into one and reduce execution time. We then add a MAC module in CV-X-IF :

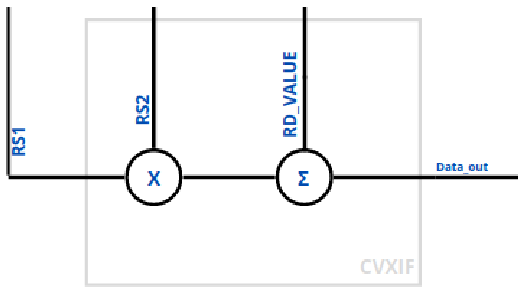


Figure 12 : CV-X-IF basic MAC co-processor

CV-X-IF will compute and return the result in 1 clock cycle only.

C. *Second design : custom compiler instructions*

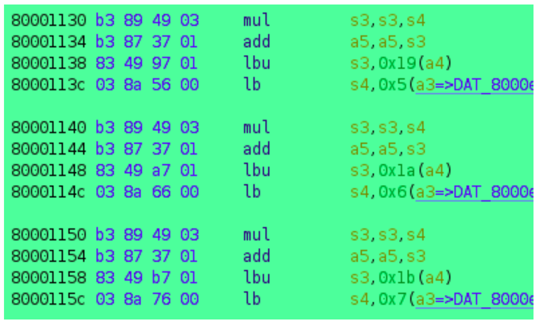
The second design rely more on compiler modification than hardware modification per say. To do so, we will reuse the first design except we remove the interceptor. This is because the ”merging” process (the fusion of MUL and ADD instruction into a single MAC instruction) will now happen at compile time when generating the binaries. To do so the RISC-V GNU tool chain was modified to handle the translation of ”mac” assembly into RISC-V binary. Then we use inline assembly code in the MNIST C programm to tell the compiler to use MAC instead of MULs and ADDs :

```
// ==================================
// macsOnRange function code snippet
// ==================================

// old code :
// *weightedSum=inputs[iter]*weights[iter];

// new code :
asm volatile ("mac %0, %1, %2, %0"
    : "+r" (*weightedSum)
    : "r" (inputs[iter]),
        "r" ((signed)weights[iter])
);
```

Now, the custom MAC will directly be included in the binary without the need of an interceptor. CV-X-IF remains unchanged compared to the first design as it stills need to handle these instructions.

D. Third design : parallelism !

This third design is based on a basic parallel computing implementation in CV-X-IF. The idea is to create an array of registers, load values into it and then read the result only once. This would be a big gain of time because until now, we loaded two values and "*MACed*" them together using our MAC instructions. On the following figure, you can have an idea of the process of loading once and than executing the MAC (or in this case MUL and ADD) every time (using the GHIDRA de-compiler) :



```
80001130 b3 89 49 03     mul        s3,s3,s4
80001134 b3 87 37 01     add        a5,a5,s3
80001138 83 49 97 01     lbu        s3,0x19(a4)
8000113c 03 8a 56 00     lb         s4,0x5(a3=>DAT_8000e

80001140 b3 89 49 03     mul        s3,s3,s4
80001144 b3 87 37 01     add        a5,a5,s3
80001148 83 49 a7 01     lbu        s3,0x1a(a4)
8000114c 03 8a 66 00     lb         s4,0x6(a3=>DAT_8000e

80001150 b3 89 49 03     mul        s3,s3,s4
80001154 b3 87 37 01     add        a5,a5,s3
80001158 83 49 b7 01     lbu        s3,0x1b(a4)
8000115c 03 8a 76 00     lb         s4,0x7(a3=>DAT_8000e
```

Figure 13 : De-compiled MAC loop snippet (Software : ghidra)

1) Overview: The idea is to load all value **before** sending a MAC instruction and end up with a sequence that look more like this :

```
// ASM Pseudo-code example

// Somehow load input
LBU     RD, imm(RS)
// Somehow Load weight
LB      RD, imm(RS)
// Loop until finished
BRANCH @ PC-8

// Get result and send to rd
MAC     RD, RS, range
```

With :

- imm : standing for "immediate".
- RS and RD : source and destination registers
- $PC - 8$: The value of the program counter - 8 (byte addressed)
- $range$: the actual $macsOnRange$ range size (details below).

To achieve this, we run all the computation in parallel (**after** everything is loaded) instead of 1 at a time (once every time a single element is loaded). Note that MAC now only has 2 source being the initial value of the weighted sum, and another operand called "$range$" that will carry the range size from the $macOnRange$ to prevent data hazards (See Section IV. D. 3 : handling data hazards). So how can we remove all those MULL ADD sequences (MACs) in order to only execute once everything has been loaded ? To address this problem, we'll once again use CV-X-IF for the custom logic. The problem is CV-X-IF was not meant to work with main memory nor cache (yet) and does not have any memory interface (yet). We can not use the General Purpose Registers (GPRs) from CVA6 as this is simply not suitable for large parallel computations. We have to create our own memory interface for CV-X-IF as well as its own local registers. The number of registers to use in CV-X-IF is determined by looking at the source code of our MNIST application. By doing so, we can determine the maximum range on which we execute the parallel MAC computations. In our case, this value is 150 registers. Each value is 8 bits but it can be signed (weights) or unsigned (input). This is not an issue when using regular GPRs in CVA6 as 32 bits registers have more than enough room to carry sign extensions for 8 bits signed operations, but for our custom CV-X-IF registers, we have to go for the smallest size possible to avoid timing hazards. So we use 9 bits registers : 8 data bits and 1 sign extension bit. We design this register file in a way that they do not need to be addressed when loads occur to save resources (and work). This is due to the fact that each weight is loaded with its corresponding input in sync every time we load. This means our CV-X-IF custom registers work as a stack (with its own internal pointer) from which we can read all at once. We then add a reset flag that allows us to clear all the values once the MAC computation is complete.

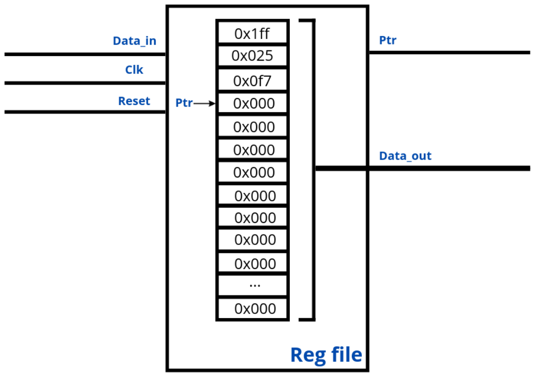


Figure 14 : Custom CV-X-IF Parallel register file

We then create two instances of this register file : one for the weights and one for the inputs. We then add logic between each of the weights and inputs (multiplication and sum). The result is then sign extended to 32 bits for CVA6 response.

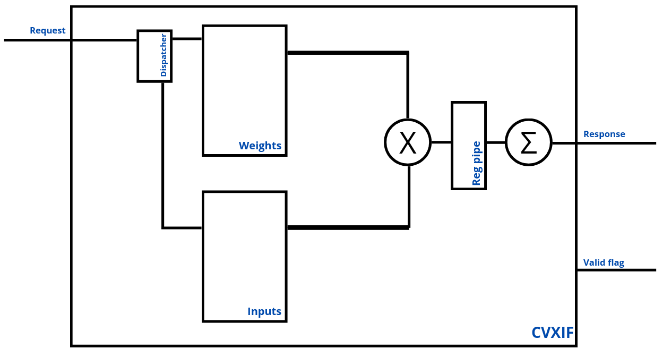


Figure 15 : CV-X-IF logic & Register files block diagram

2) *Adding a "memory interface" to CV-X-IF:* In order to load values in the custom register files, we need to have a way to interface with the main memory / cache that contains all the weights / inputs data. Looking back at figure 13, we notice that the same load instructions are always used :

- Load Byte (LB)
- Load Byte Unsigned (LBU).

The idea is to replace LB and LBU with custom loads instructions that will redirect data to CV-X-IF instead of the COMMIT STAGE (Also called write-back stage). Those added instructions will be called :

- Load Byte in CV-X-IF (LBC)
- Load Byte in CV-X-IF Unsigned (LBCU)

To add those instructions to the GNU compiler, we can re-use LB and LBU masks because we were lucky enough that all F3 combinations were not used in the 32 bits version of CVA6 (32 bits version reference : *CV32A65X*) [14] [15] : 011, 110 and 111 F3 Values are free to use. You can see here in figure 16 how LBC and LBCU are built :

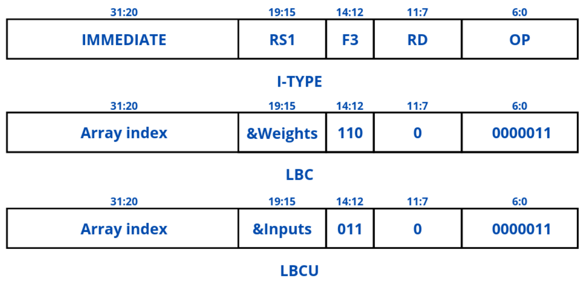


Figure 16 : custom LBC & LBCU instructions base (from I-Type) [18]

Note that we are setting RD to 0. The register 0 is tied to ground and is always zero, we do not want to actually write anything back to CVA6 registers and alter processor state (we want to send value to CV-X-IF) so zero is the default value here. We then proceed to adapt CVA6. This way, LBC & LBCU acts as regular I-Type instructions (load instructions) and can retrieve data from memory as it would with traditional loads like *Load Word* (LW), *Load Byte* (LB), etc... Then the data is redirected to CV-X-IF where its gets stored on the stack-like register file. As inputs are always unsigned and weights always signed, LBC will always get stored in the weights register file and LBCU in the inputs register file (See "*dispatcher*" module in figure 15). If necessary, this design can use the 5 unused address bits (RD in I-TYPE) to give clues on what type of data we are facing (See figure 16 RD unused bits set to 0).

3) *handling hazards (Timing, data & control):* After synthesis, this design was too slow and was the source of timing hazards. A pipeline register was added after the multiplication (150 x 18bits) to make the critical path shorter (See "*reg pipe*" in figure 15). Our design does not use a control module as our strategy could be done without it. This leads to control issues : we send the MAC instruction (to retrieve computed data from CV-X-IF) without waiting for the loads to be finished (committed to the CV-X-IF's register file). CVA6 is an in-order core but loads can take several clock cycles. Meanwhile, MAC instruction gets issued and the result is returned even if the data loads are not finished properly. To address this problem, a "load_done" flag was added and its role is to check whether the CV-X-IF register file internal pointer is equal to the range size on which we are executing the $macsOnRange$ function. The range value is known as it's passed as a source in the MAC instruction operands as we discussed earlier in this subsection.

4) *A word on this subsection & parallelism strategy:* This design flow is important to point out as it helps understanding the strategies used for basic acceleration & parallelism. This very specific strategy was possible because of the software involved. When designing your own accelerator, software might (and will) differ and the strategy might differ with it (and so will the results). Bigger & more general purpose computation often involves threads with deep pipelines, dedicated memory interfaces and more complex control flow involving the creation of custom libraries to actually being able to use it, allowing for larger frequencies and operations per cycle but increasing delays [21] (See VI : Conclusion / future work).

V. Results

A. Overview

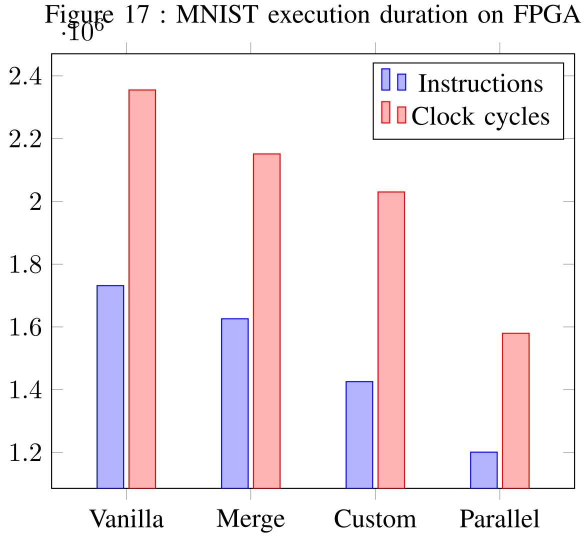
In this section, we will detail results and compare them with one another. We'll then conclude on what design is the fastest but also at what cost in term of hardware resources (FPGA utilisation). Each design will be synthesized and implemented on the Zybo Z7-20 FPGA board using Vivado. We'll examine and compare timings (for max clock frequency), utilizations and execution time (using the number of instructions & clock cycles to complete the forward propagation). Here is a summary of the compared designs :

- CVA6 core "Vanilla" : $Vanilla$
- CVA6 core with instruction interceptor / merger :$Merge$
- CVA6 core with custom instructions : $Custom$
- CVA6 core with parallel computing : $Parallel$

B. Numbers & comparisons

First of all, let's compare the results in term of execution time as it was the determining factor to minimize in this R&D session :

Figure 17 : MNIST execution duration on FPGA



We notice that parallel computing outperforms every other form of acceleration. That was expected as parallel computing crushes sequential approach in terms of operations per cycle [21] and the only remaining bottle neck is our home-brew memory interface. Another point is that the merger (interceptor) runs longer than the custom instruction implementation enven though they work on the exact same principle. This is due to the fact that the interceptor / merger works on hardware only and does not intercept 100% of the MUL ADD (MACs) sequences due to the instruction queue FIFO not always fetching both at the same time or simply because MUL and ADDs are not always neatly place next to one another by the compiler. Here are the improvements each design brings compared to the Vanilla CVA6 (in percentage) :

TABLE I
DESIGNS' IMPROVEMENTS ON FPGA

Design name	Instructions	Clock cycles
Vanilla	1 731 593	2 355 071
Merge	1 625 994 (-6.1%)	2 151 318 (-8.6%)
Custom	1 425 646 (-17.6%)	2 030 052 (-13.8%)
Parallel	1 200 889 (-30.6%)	1 579 589 (-32.9%)

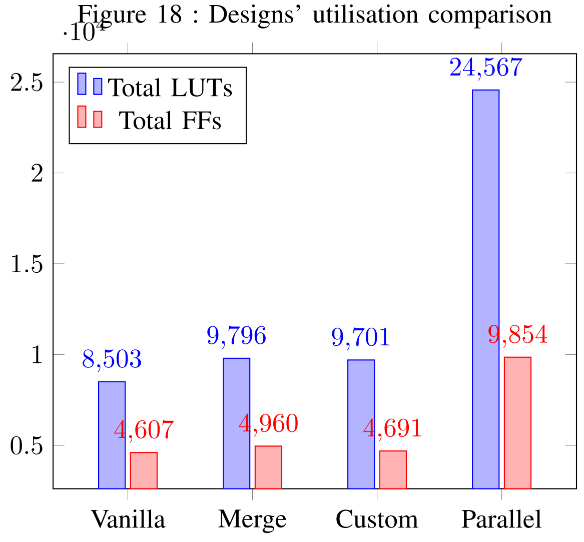
We also gathered data on the improvement on simulation (Software : ModelSim / Questa) :

TABLE II
DESIGNS' IMPROVEMENTS ON MODELSIM

Design name	Instructions	Clock cycles
Vanilla	1 731 593	2 316 005
Merge	1 626 562 (-6.0%)	2 117 194 (-8.5%)
Custom	1 425 646 (-17.6%)	1 992 098 (-14.0%)
Parallel	1 200 889 (-30.6%)	1 541 173 (-33.4%)

Once again, parallel computing really stands out from the crowd with a 32.9% improvement on the number of clock cycles it took to execute MNIST, only throttled by the slow memory loads. But is it worth it ? CVA6 is destined for embedded applications so hardware utilisation plays a key role here. Let us compare FPGA utilisation of Look Up Tables (LUTs) and Flip Flops (FFs) to see how this *"neural engine logic"* was implemented :

Figure 18 : Designs' utilisation comparison



We notice that the parallel design's utilisation skyrockets (the design fits on FPGA). Note that the number of RAM16 stays the same for all the designs : 16. Now do not be fooled, we designed an dedicated co-processor on CV-X-IF designed to compute the $macsOnRange$ function in only two clock cycles (not 1 because of the pipeline) (after loads). This kind of parallel hardware acceleration does not come without a cost. There are still ways to really improve this utilisation issue, see the "future work" section in the conclusion. In terms of timing, every design meet the maximum critical path delay constraint of 25ns. However, timing vary slightly and we had to handle timing hazards for the parallel design so here is a comparison of the different timings :

TABLE III
DESIGN TIMINGS & MAX FREQUENCIES EVOLUTION

Design name	Max delay (ns)	Max frequency (MHz)
Vanilla	19.4	51.55
Merge	19.5	51.28 (-0.5%)
Custom	19.7	50.76 (-1.5%)
Parallel	20.3	49.26 (-4.4%)

Timing variations still respect the 25ns spec and if necessary, we can still improve the CV-X-IF pipeline for shorter delays (See future work for details on parallel strategies).

VI. CONCLUSION

A. Overview

We designed 3 improvements for the CVA6 RISC-V Core after understanding how CNN works and how the computer approaches the problem. Understanding this approach was a key factor to come up with 3 simple yet effective strategies to accelerate the MNIST algorithm. Each design has its pros and cons that we detailled in the result section (V). The reader now has a solid basis for further improvements for its own hardware accelerator using a RISC-V core, especially CVA6 and the CV-X-IF platform. These designs serve as a proof of concept to show how to approach hardware acceleration on CVA6 and what results we can get from it, these design are not very versatile in this current state so, in the next subsection, we'll give the reader some clues on what to do to really empower those improvement for larger and broader applications.

B. Future work

In this sub section, we will focus on how we can adapt the parallel computing strategies to address the different encountered issues but also to make it better for larger and broader applications. The other two designs will not be discussed as they are already simple, versatile and straight-forward which make their benefits for embedded application already clear.

A big source of improvement, especially in terms of resources utilisation would be to add a control module to handle threads. The co-processor actually sits there waiting for data most of the time, this is a waste of resources because we could use less hardware by running a multi-cycle thread on the same core while waiting for data / other instructions. This also allows to make more cores, more threads and thus more operations per second [21]

This principle is the one used in general purpose chips like GPUs or Tensor Processing Units (TPUs) when it comes to threading multiply-accumulators cores [22].

Finally, we can really exploit the power of parallel computing by improving the memory interface between the CVA6 core and the CV-X-IF co-processor as the actual home-brewed solution (using CVA6 cache interface and redirecting to CV-X-IF) is the main bottleneck.

References

[1] Wang, Y., Wang, Q., Shi, S., He, X., Tang, Z., Zhao, K., & Chu, X. (2020, May). Benchmarking the performance and energy efficiency of AI accelerators for AI training. In 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID) (pp. 744-751). IEEE.

[2] Baji, T. (2018, March). Evolution of the GPU Device widely used in AI and Massive Parallel Processing.

[3] MALLASÉN, David, MURILLO, Raul, DEL BARRIO, Alberto A., et al. Customizing the CVA6 RISC-V core to integrate posit and quire

[4] instructions. In : 2022 37th Conference on Design of Circuits and Integrated Circuits (DCIS). IEEE, 2022. p. 01-06. ZHANG, Shuhua, TONG, Jie, ZHANG, Jun, et al. A RISC-V Based Coprocessor Accelerator Technology Research for Convolution Neural Networks. In : Journal of Physics: Conference Series. IOP Publishing, 2020. p. 012002.

[5] Hobbhahn, M. (2022, 27 juin). Trends in GPU Price-Performance. Epoch.

[6] Chi, L. (2020). Fast Fourier convolution.

[7] Keiron O'Shea, Ryan Nash (Dec 2015) An Introduction to Convolutional Neural Networks — arXiv:1511.08458v2 [cs.NE]

[8] deeplizard. (2018, 27 février). Backpropagation explained — Part 2 - The mathematical notation [Vidéo]. YouTube.

[9] deeplizard. (2018a, février 14). Zero Padding in Convolutional Neural Networks explained [Vidéo]. YouTube.

[10] Hussain, Mahbub & Bird, Jordan & Faria, Diego. (2019). A Study on CNN Transfer Learning for Image Classification: Contributions

[11] Kim, S., & Casper, R. (2013). Applications of convolution in image processing with MATLAB. University of Washington, 1-20.

[12] John D. Owens; Mike Houston; David Luebke; Simon Green; John E. Stone; James C. Phillips (2008, 1 mai) GPU Computing — IEEE Journals & Magazine.

[13] Hijazi, S., Kumar, R., & Rowen, C. (2015). Using convolutional neural networks for image recognition. Cadence Design Systems Inc.: San Jose, CA, USA, 9(1).

[14] Ludovico Poli; Sangeet Saha; Xiaojun Zhai; Klaus D. Mcdonald-Maier (2021, 1 december) Design and Implementation of a RISC V Processor on FPGA. — IEEE Conference Publication.

[15] CVA6 : An application class RISC-V CPU core — CVA6 documentation. (s. d.). https://docs.openhwgroup.org/projects/cva6-user-manual/

[16] Sarah Harris. (2021, 12 septembre). DDCA CH7 - Part 13 : Pipelined Processor [Vidéo]. YouTube.

[17] Sarah Harris. (2021b, septembre 12). DDCA CH7 - Part 14 : Pipelined Processor Data Hazards [Vidéo]. YouTube.

[18] Andrew Waterman1, Krste Asanovi´c1,2 1SiFive Inc (May 2017). The RISC-V Instruction Set Manual, Volume I: User-Level ISA : Document Version 2.2

[19] Filsinger, M. (2005, 12 juin). Understanding CPU pipelining through simulation programming.

[20] Sarah Harris. (2021a, septembre 12). DDCA CH6 - Part 15 : RISC-V Machine Instructions : R-Type [Vidéo]. YouTube.

[21] GARLAND, Michael, LE GRAND, Scott, NICKOLLS, John, et al. Parallel computing experiences with CUDA. IEEE micro, 2008, vol. 28, no 4, p. 13-27.

[22] OUPPI, Norman P., YOUNG, Cliff, PATIL, Nishant, et al. In-datacenter performance analysis of a tensor processing unit. In : Proceedings of the 44th annual international symposium on computer architecture. 2017. p. 1-12.