Implementing a CNN in High Level Synthesis with Vitis HLS

Lorenzo Bossi Federico Mandelli

May 2022

Contents

1 Introduction

This paper contains the findings and our first experience approaching to *High Level Synthesis* (HLS). During this project we created and synthesized a Convolutional Neural Network capable of recognising hand written digits using HLS. In the last semester we:

- Learnt the structure and implemented a *Convolutional Neural Network*(CNN) in python;
- Trained said network;
- Implemented the CNN in c/c++ using the weights obtained from the python training;
- Revisited the code to make it more "HLS friendly";
- Developed the neural network with *Vitis HLS*;
- Applied different *Vitis* pragmas;
- Implemented the solution.

In the next sections we will discuss:

- Background: A brief explanation of the technology used in this project,
- Methodology: How we used said technology in this project,
- Results: The results obtained by applying our methods,
- Conclusion.

This project is not to be considered a proper research paper but rather a *dip* into the HLS world and its rules. A first look into hardware development useful to better understand a lower level to which we are not used to think about.
This paper will contain our findings, the problems that occurred, how we solved them, and the opinions that formed while trying to open the *Black Box* of HLS

2 Background

In this section we will explain everything that will be used as a tool in the next chapters.

2.1 Convolution Neural Network

A convolutional neural network is a feed-forward neural network inspired by a biological processes, used mainly in image and video recognition. They are usually made by several layers which can perform various operations. Commonly CNN are composed by a first portion, which applies different filters to the input data, and a second which is similar to a *classical* neural network.
In the following sections we will describe each layer used in our convolutional neural network.

2.1.1 Convolution

The Convolution layer applies a filter (also called kernel) to the input data. Filters are matrices with a smaller size than the image, they are composed by values determined during the training process. The input image is divided in sub-matrices with the size of the kernel. The resulting output is a matrix composed by the scalar product between the filter and a sub matrix plus a bias. During the training process kernels and biases are determined in order to recognize and accentuate meaningful patters for the specific application. For example, the application of a filter created to recognize straight vertical lines, will increase the value of pixels in vertical lines and decrease the others. However the training process usually generates kernels which recognise patterns that seem meaningless to human eyes.

2.1.2 Pooling

The Pooling layer divides the input image in disjoint sub-matrices of a predetermined size (called *strife*), and calculate an aggregate value depending on the pooling strategy. The most commonly used method in image recognition is max pooling, in which the aggregate value is the maximum value found in the sub-matrix. Other types of less used pooling techniques are min pooling and average pooling.
The purpose of this layer is to reduce noise, to avoid over fitting, and to reduce the size of the input. For example, the max pooling can take as an input the result of a convolution layer, transforming it in a less detailed matrix while showing the recognized pattern.

2.1.3 Fully connected

The fully connected or dense layer is the main component of traditional neural networks. Each output node j (also called neuron) is influenced by every input node k through the following relation:

$$node_j = activation((\sum_{k=0}^{n} weight_{k,j} \times node_k) + bias_j)$$

Every connections between input and output neurons is associated to a weight which determines the influence of each input node on each output node.
The most common activation function is $ReLU$, which will normalize the input to zero if negative, doing nothing otherwise.
Another example of a more complex activation function is *Softmax*, which calculate and normalize the input based on an exponential function.

Weights and biases are determined during the training process in order to create meaningful associations. For example, a fully connected layer can associate specific compositions of patterns, found by a convolution layer, to the activation of an output neuron which correspond to a recognizable figure. However it is usually impossible to extrapolate a meaning from weights and biases of a dense layer.

2.2 High Level Synthesis

High Level Synthesis(HLS) is a tool that allows us to develop hardware with little to no knowledge of hardware design while giving more experienced users a powerful tool for easy and fast evaluation.
Synthesis begins with a C/C++ implementation of the program, that is then translated by HLS into an hardware description.
In this project we used Vitis 2021.2, that allows us to improve our code using *Vitis Pragmas*. Pragmas can be used to optimize the design, reduce latency, improve throughput performance, and reduce area of the resulting RTL code.
During this project we experimented with a couple of pragmas,the ones that gave us the more interesting results are:

2.2.1 Pipeline II

Pipline II (Initialization Interval) tells how many cycle are needed before we can process a new input. A pipelined function or loop can process new inputs every <N> clock cycles, where <N> is the II of the loop or function. An II of 1 processes a new input every clock cycle. You can specify the initiation interval through the use of the II option for the pragma.
Pipelining a loop allows the operations of the loop to be implemented in a concurrent manner as shown in the following figure. In the figure, (A) shows the default sequential operation where there are three clock cycles between each input read (II=3), and it requires eight clock cycles before the last output write is performed. (B) shows the pipelined operations that show one cycle between reads (II=1), and 4 cycles to the last write.

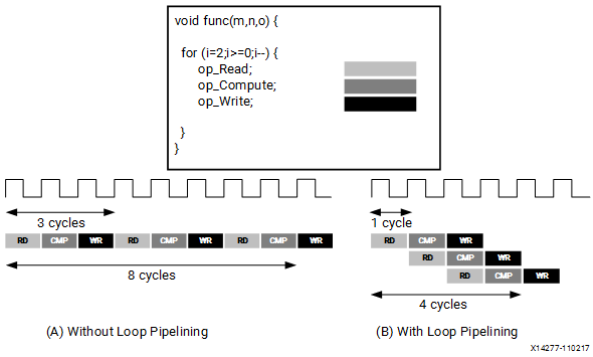

Figure 1: Pipeline visual explanation

2.2.2 Unroll

Unrolls loops to create multiple independent operations, rather than a single collection of operations. The unroll pragma transforms loops by creating multiples copies of the loop body, which allows some or all loop iterations to occur in parallel. We can partially unroll a loop by a specified factor (in this example by 2). This allows us to get a faster code, if it is formatted in the right way.
In this example, we first have a loop that requires 4 cycles Then we partially unroll it requiring only 4. Finally we unroll it fully, requiring only one cycle to compute.
This boost in performance is not free as the last loop requires the ability to perform 4 reads, 4 writes and 4 multiplications at the same time.

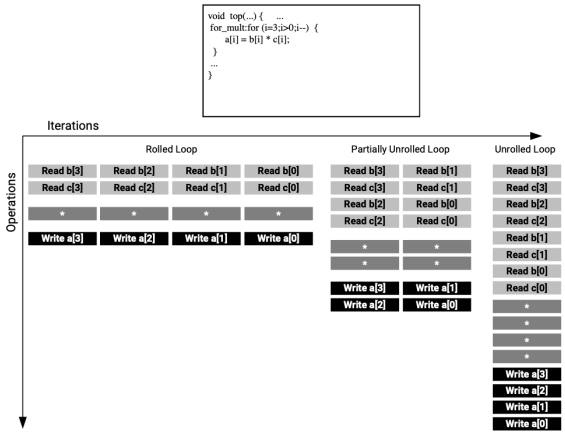

Figure 2: Unroll visual explanation

2.2.3 Dataflow

Allows functions and loops to overlap in their operation, increasing the concurrency of the RTL implementation and increasing the overall throughput of the design.
For the Dataflow optimization to work, the data must flow through the design from one task to the next.
We must not use this coding styles:

- Single-producer-consumer violations;
- Feedback between tasks;
- Conditional execution of tasks;
- Loops with multiple exit conditions.

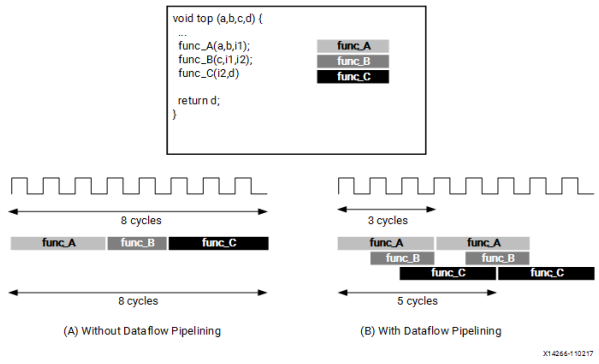

Figure 3: Dataflow visual explanation

2.2.4 Array Partition

It partitions an array into smaller arrays or individual elements. This could be useful for allowing multiple read and write at the same time.

Figure 57: **Array Partitioning**

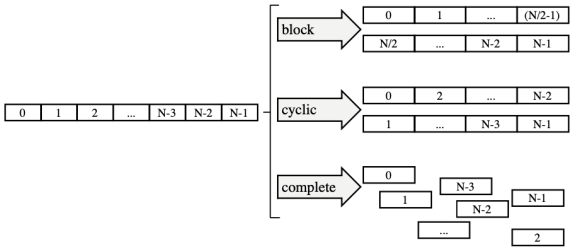

Figure 4: Array Partition visual explanation

2.3 Solution Evaluation

During this paper we will use different quantities to evaluate our solution, here we will explain what we will be using and their meaning.

2.3.1 Cycles

The clock cycle is the amount of time between two pulses of an oscillator. It is a single increment of the clock during which the smallest unit of processor activity is carried out, and it is considered the basic unit of measuring how fast an instruction can be executed. In this project we used a clock cycle of 10 nano seconds.

2.3.2 Flip Flops

A flip-flop is a device which stores a single bit (binary digit) of data. One of its two states represents a "one", and the other represents a "zero". Such data storage can be used for storage of state, and such a circuit is described as sequential logic in electronics. They represent the *Area* of our project.

2.3.3 Look Up Table

In computer science, a lookup table (LUT) is an array that replaces runtime computation with a simpler array indexing operation. The savings in processing time can be significant, because retrieving a value from memory is often faster than carrying out an "expensive" computation or input/output operation. The tables are stored in hardware.

3 Methodology

In this section you will read about the methods that we applied.

3.1 CNN

After understanding the basic principles of convolutional neural networks, we tried to implement one. The goal was to create a CNN able to recognize hand written number digits from the MNIST data-set, which contains 70000 examples labeled with the real value. Every image is a 28x28 black and white picture, and every pixel is a number between 0 and 255. We had to normalize every pixel, dividing them by 255.
We planned to split the CNN in two main parts: the former to apply several operations on the image, and the latter to find the right output value. In the first part we used a convolution layer followed by a max pooling layer. In the second part we added a fully connected layer to determine the correct output. In this way convolution and max pooling can recognize different patterns of numbers, like straight lines or circles, and then fully connected can find the right number according to the pattern combinations. In addition we had to add a flatten layer to convert the 2 dimensional output of max pooling into a 1 dimensional input for the fully connected layer. Here we will briefly show in pseudo code (the full code can be found on GitHub) the different function of our CNN on which we based our HLS implementation.

3.1.1 Main

The main is the core of our program, it reads the input from a file, for each kernel is applies *Convolution()* and *MaxPooling()* saving the results into an array of arrays for further processing. It then calls the *FullyConnected()* were it gets the probability of each number to print it out.

Algorithm 1 Main

```
GetInputFromFile();
for i<NumberOfKernel do
  Convolution();
  MaxPooling();
end for
FullyConnected();
PrintResult();
```

3.1.2 Convolution

It verlaps a kernel to each component of the input matrix, it then execute a member to member product, sums the results with the bias and saves it in the result matrix after applying the *Relu* activation function. This is done for each element until the kernel fits in the matrix.

Algorithm 2 Convolution

```
for i<ExpectedSize do
  for f<ExpectedSize do
    for z<KernelSize do
      for j<KernelSize do
        sum += tensor[] * kernel[];
      end for
      convoluted[] = relu(sum + bias);
    end for
  end for
end for
```

3.1.3 MaxPooling

Similarly to *Convolution()*, we overlap a matrix of dimension *StrifeSize x StrifeSize* to each element of the input matrix. We then find the max value inside said matrix and write it in MaxPooled

Algorithm 3 MaxPooling

```
for i<InputSize do
  for f<InputSize do
    for z<StrifeSize do
      for j<StrifeSize do
        if max<tensor[] then
          max=tensor[];
        end if
      end for
    end for
    MaxPooled[] = max;
  end for
end for
```

3.1.4 FullyConnected

Firstly we take a matrix as an input, the matrix has the result of *MaxPooling()* as rows, the different rows represent the output obtained from different kernels. We then flatten the matrix, rewriting it as an array, and multiply each non zero member to his corresponding weight, adding the result to the previous values.
We do this for each element of the array, we then apply *Softmax* as our activation function changing the values into probability of each final node (numbers 0-9). Finally we return the probability obtained to the main.

Algorithm 4 FullyConnected

```
flatten();
for i<FlattenSize do
  if flatten[i]!=0 then
    for f<InputSize do
      layer[] += flatten[] * weight[];
    end for
  end if
end for
for i<FlattenSize do
  layer[] += bias[];
end for
Softmax()
```

3.2 HLS

After implementing the same algorithm in vitis we started *"playing"*. At first there were warnings that we resolved. Then, following the official guide, we tried refactoring the code, seeing which changes would provide the better results, founding an optimal configuration. Finally we applied pragma that we tough would have the best result.
The code will not be shown here since it is logically identical to the c++ implementation.
In the next section this operations will be explained in details with experimental data.

4 Results

In this chapter we will discuss the results of the methodologies described in the previous section. In particular in the next chapters you will find results about:

- CNN : network designs, training and C implementation;
- Vitis HLS : code refactoring, directives, final solution.

4.1 CNN

In this section we will discuss the results, problems and solutions of our attempts to create and train a CNN. In particular our first design was very precise, but had an excessive number of weights. In the second generation we managed to reduce significantly the number of weights, slightly lowering the network precision.

4.1.1 First design

The first solution consisted of the following layers:

1. Convolutional layer, applying 32 kernels 3x3;
2. Max Pooling layer, with a strife of size 2;
3. Fully Connected layer, with 100 neurons;
4. Fully Connected layer, with 10 neurons (the same number of the possible outputs).

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_4 (MaxPooling 2D)	(None, 13, 13, 32)	0
flatten_4 (Flatten)	(None, 5408)	0
dense_8 (Dense)	(None, 100)	540900
dense_9 (Dense)	(None, 10)	1010

Figure 5: Model specifications printed by keras model.summary() of the firs model.

We trained the network with 10 epochs obtaining a mean accuracy of *98.6%*.
Our first design achieved an acceptable accuracy, however the training generated 542230 weights, which were difficult to handle and could have caused problems in the following steps of the project.

4.1.2 Second design

After the first attempt we tried to create a model with less weights without losing too much accuracy. We removed the first fully connected layer, which generated the bigger number of weights, and adjusted the others in the following way:

1. Convolutional layer, applying 10 kernels 7x7;
2. Max Pooling layer, with a strife of size 3;
3. Fully Connected layer, with 100 neurons;
4. Fully Connected layer, with 10 neurons (the same number of the possible outputs).

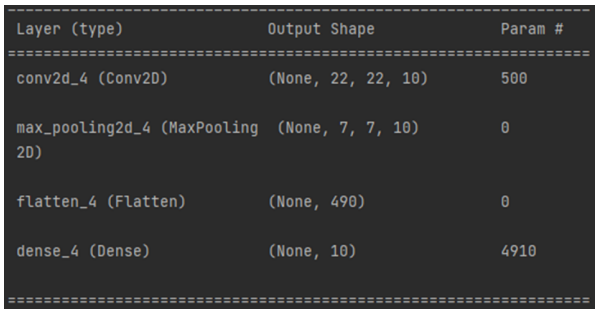

Figure 6: Model specifications printed by keras model.summary() of the second model .

With the increase of kernels and strife size, convolution and max pooling generated a smaller output, reducing the input of the final layer and therefore his weights.
In total our second design generated just 5410 weights instead of 542230.
We trained the network with 10 epochs, as we did with the first one, obtaining a mean accuracy of *97.8%*.
Even if we reduced drastically the number of weights needed by the CNN, we found a slight decrease in the accuracy of the network. We were satisfied by the structure of the network, so we tried to increment accuracy, without changing its configuration, with a deeper training.
We trained the network with 100 epochs and increased the batch size. As a result the time needed by the training increased significantly, but we obtained a mean accuracy of *98.5%* which was almost the same as the first design.

	weights	**Accuracy**	**deeper training**
Design1	542230	98.6%	
Design2	5410	97.8%	98.5%

Table 1: CNN designs comparison

4.1.3 C implementation

The *C* implementation of our second design needed around *2.8 milliseconds* to successfully recognize an image.The time value was obtained averaging 1000 CNN run on different samples. We will use this time as our standard to beat in our HLS implementation. Tests run using:

- CPU : i7 6700k boosted @4.2ghz
- RAM : 16 GB RAM @3000 mhz

4.2 HLS implementation

In this chapter we will discuss the results obtained while trying to optimize our code.
Every presented result has been found using the following Vitis HLS settings:

- target clock period: *10ns*
- target board: *xa7z030fbg484-1I*

4.2.1 Forcing pipeline II=1

As previously explained Pipline II (Initialization Interval) tells how many cycle are needed before we can process a new input.
Before any optimization we got II of 98 in *Convolution*, that gave us unreasonable times as we computed a new input every 100 clock cycles (a new input every microsecond), especially for a function that is supposed to be called multiple times.
Our first approach was to ignore this problem and try to find different options elsewhere (especially in pragma unrolled). Then we tried forcing a lower pragma; in our case we tried setting II = 1.
By doing so *Convolution* lowered the clock cycles of the synthesis from 2456 to 1300. However it gave us an enormous area.
The following table shows resources usage of the synthesis compared to our board availability.

DSP	FF	LUT
1080557 (270139%)	91625192 (58285%)	155209664 (197467%)

Table 2: Area generated forcing II=1 in *Convolution*

We think that, since Vitis was unable to respect the constraint, it tried to hard-translate everything to observe the pragma, thus occupying an enormous area.
In the end we removed Pipline II and we tried to reduce the time optimizing other parts of the code.

4.2.2 Adding buffers and splitting operations

We found out that saving array values into buffers is a good practice if they were needed multiple times, so we tried to reduce the number of reading and writing from arrays as much as possible.
We noticed that buffers could also solve timing violation and negative slack problems. By analyzing arrays usage in our code, we noticed we could split some operations needed to find indexes.
An example of those practices can be found in our *FullyConnected* function.

Algorithm 5 FullyConnected

```
for j<FlattenSize do
  tempFlatten = tensor[j];
  weightBiasIndex = j * BIASSIZE;
  for (f<flattenSize) do
    tempWeight = dense[][];
    result[] = result[] + (tempFlatten * tempWeight);
  end for
end for
```

4.2.3 Max and Sum refactoring

As the standard implementation of the *sum* function generated a time violation and a negative slack we tried other possible implementations in order to fix it. In the end we found an interesting solution.

Algorithm 6 Standard implementation of sum()

```
sum = 0
for each element do
    sum = sum + element
end for
```

Algorithm 7 Second version of sum()

```
#define SIZE 10
float input[SIZE];
float tempsum1[SIZE/2];
float tempsum2[SIZE/4];
float tempsum;
float sum;
for (j = 0, i = 0 ; i < SIZE/2 ; i++, j = j+2)  do
    tempsum1[i] = input[j] + input[j + 1];
end for
for (j = 0, i = 0 ; i < SIZE/4 ; i++, j = j+2)  do
    tempsum2[i] = tempsum1[j] +tempsum1[j + 1];
end for
tempsum = tempsum2[0] + tempsum2[1];
sum = tempsum + tempsum1[4];
```

After rewriting *sum* functions we managed to remove negative slack and timing violation. The new solution needs more area, since it uses a lot of buffers, but it is faster.

SUM	Clock Cycles	Latency	FF	LUT
Standard	44	$639\mu s$	351	483
Refactored	32	$320\mu s$	1821	890
$\Delta\%$	-27.3%	-49.9%	$+418\%$	$+84.3\%$

Table 3: Comparison between new and old Sum implementation

Applying the same changes to *max()* we obtained a considerable increase in area, but a smaller latency decrease. That's why we decided to maintain the first implementation, as it did not give us any errors and the small latency decrease of the new solution was not worth the increase in area.

Algorithm 8 First version of max()

```
max = 0
for each element do
    if element>max then
        max=element
    end if
end for
```

MAX	Clock Cycles	Latency	FF	LUT
Standard	24	$240\mu s$	152	209
Refactored	23	$230\mu s$	921	1022
$\Delta\%$	-4.2%	-4.2%	$+506\%$	$+389\%$

Table 4: Comparison between new and old Max implementation

In our opinion the problem of *Sum* was the instruction: *sum = sum + vector[i];* as it tries to both read and write into sum in the same iteration.
In our new solution we avoided that situation by writing the partial sum in a buffer. On the contrary *Max* function only reads the input vector and write in a variable. That's why it didn't give us any problems, and the new solution did not lead to significant improvements.

4.2.4 Changing function order

As suggested by Xilinx official user guide, we tried to separate different tasks in different functions, allowing us to run them in parallel. However we couldn't use *#pragma HLS dataflow*, because of the data dependencies of the called functions. So we tried using *#pragma HLS unroll* in the main loop to run them at the same time. Furthermore we noticed that sometimes the function order could change performances. We tried to rearrange our main structure to find the best scheduling order that would allow different functions to run together. In the end we found better results in the following configuration.

Algorithm 9 Best function calls order

```
1: init(result,tensor,input);
2: Convolution(input, 0, tempConvoluted);
3: pool(tempConvoluted, tempMaxPooled);
4: FullyConnected(tempMaxPooled, 0, fc[0]);
5: for (i = 1; i < NUMBEROFKERNELS; i++) do
6:     Convolution(input, i, tempConvoluted);
7:     updateResult(result,fc[i-1]);
8:     pool(tempConvoluted, tempMaxPooled);
9:     FullyConnected(tempMaxPooled, i, fc[i]);
10: end for;
11: updateresult(result,fc[NUMEROFKERNELS-1]);
```

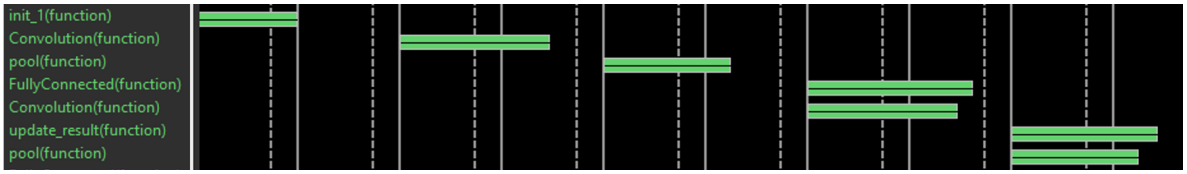

Figure 7: Schedule of setup and first iteration.

This order, combined with *#pragma HLS unroll*, allowed the tool to schedule *Convolution* to run in parallel with *FullyConnected*, and *updateResults* to execute at the same time as *pool*, nine times out of ten.
As *Convolution* and *FullyConnected* are the slowest functions, and *updateResults* and *pool* are the fastest, we found this was the optimal execution order. It is not possible to run more functions in parallel because of data dependencies, so we kept this as our final solution, even if it might seem an illogical order at first sight.
In the following table is shown the comparison between the application of a *#pragma HLS unroll* applied to the main for loop allowing parallelism and the same same code without it.

	Clock Cycles	FF	LUT
No unroll	34073	65049	51540
unroll	28683	65325	52210
$\Delta\%$	-15.8%	$+0.4\%$	$+1.3\%$

Table 5: Performance emprovements by using pragma unroll

Even with optimal parallelism, performances are limited by *Convolution* latency, which is much higher than other functions causing a bottleneck.

4.2.5 Convolution

In *Convolution* we applied *#pragma HLS unroll* combined with *#pragma HLS pipeline*. However the minimum II of the function was very high, causing it to be the slowest. In the following table is presented a comparison between the synthesis of *Convolution* with and without the application of directives.

Convolution	Clock cycles	Latency	FF	LUT
No opt	12336	123.36 μs	9126	5088
optimized	2456	24.560 μs	53951	23002
$\Delta\%$	-80%	-80%	$+491\%$	$+352\%$

Table 6: Convolution performances

Algorithm 10 Convolution

```
1: for i<ExpectedSize do
2:    #pragma HLS pipeline II=98
3:    for f<ExpectedSize do
4:       #pragma HLS unroll
5:       for z<KernelSize do
6:          for j<KernelSize do
7:             sum += tensor[] * kernel[];
8:          end for
9:          convoluted[] = relu(sum + bias);
10:      end for
11:   end for
12: end for
```

4.2.6 Max Pooling

In *Pool()* we applied *#pragma HLS unroll* combined with *#pragma HLS pipeline*. As the II was lower than *Convolution*, we obtained a big jump in performance, at cost of a significant increase in area.
In the following table is presented a comparison between the synthesis of *Pool* with and without the application of directives.

Max Pooling	Clock cycles	Latency	FF	LUT
No opt	2261	22.61 μs	184	698
optimized	302	3.020 μs	4186	6440
$\Delta\%$	-87%	-87%	$+2175\%$	$+851\%$

Table 7: Max Pooling performances

Algorithm 11 Max Pooling

```
for each element of result do
    result[i] = 0;
end for
for i<InputSize do
    #pragma HLS pipeline II=35
    for f<InputSize do
        #pragma HLS unroll
        for z<StrifeSize do
            for j<StrifeSize do
                resultBuffer=result[];
                tensorBuffer=tensor[];
                if resultBuffer<tensorBuffer then
                    result[]=tensorBuffer;
                end if
            end for
        end for
    end for
end for
```

4.2.7 Fully connected

In *FullyConnected()* we tried applying *#pragma HLS pipeline*, other directives did not get better performances. As *Vitis HLS* applies pipeline directive by default, writing #pragma HLS pipeline explicitly didn't cause any changes in performances. By writing the right II explicitly, we removed the II violation warning generated by the default optimization.
The main changes made in*FullyConnected()* consisted in removing the flatten function and adding an *updateResults()* function for better parallelism (4.2.4 Changing function order).This gave us a net gain of 250 cycles every function call (2500 overall).

Algorithm 12 update_result

```
for i<NUMBEROFKERNELS do
    result[i] += fc[i];
end for
```

Algorithm 13 FullyConnected

```
for each element of result do
    result[i] = 0;
end for
for j < TENSORSIZE  do
    #pragma HLS pipeline II=10
    tempFlatten = tensor[j];
    if tensor[j] != 0 then
        for y < BIASSIZE do
            tempWeight = dense[][];
            result[] = result[] + (tempFlatten * tempWeight);
        end for
    end if
end for
```

Fully Connected	Clock cycles	Latency	FF	LUT
First	7404	74.040 μs	2174	5610
without Flatten	10∗517	10∗5.170μs	1342	1743
updateResults	10∗18	10∗0.180μs	455	555
$\Delta\%$	-27.7%	-27.7%	-17.3%	-58.8%

Table 8: Comparison between our first and second version of FullyConnected.

Fully Connected	Clock cycles	Latency	FF	LUT
No pragma	517	5.170 μs	1342	1743
With pragma	517	5.170 μs	1342	1743
$\Delta\%$	0%	0%	+0%	+0%

Table 9: Fully Connected performances

4.2.8 Final Solution

Here is our final solution optimized with directives, compared to our final solution without any directives.
In the end *Convolution* is the bottleneck of our solution as its latencies and resources are several times higher than other functions.
Our final solution, compared to our first working synthesis, shows a significant decrease in latency and increase in area, reducing the cycles needed by almost ten times, while increasing by eight and a half times the number of flip flops needed, and by four and a half times the number of look up table.
Our solution is almost ten times faster than our first C implementation.

	Clock cycles	Latency	FF	LUT
Convolution	2456	24.560 μs	53951	23002
Max Pooling	302	3.020 μs	4186	6440
Fully Connected	517	5.170 μs	1342	1743
CNN	28680	286.80 μs	64577	51468

Table 10: Final solution performances devided by functions

	Clock cycles	Latency	FF	LUT
First	288564	2885.64 μs	6936	9690
Final	28680	286.80 μs	64577	51468
$\Delta\%$	-90%	-90%	+831%	+431%

Table 11: Performance comparison between our first and our final solution

	C	HLS	C/HLS
Latency	2.8000ms	0.2868ms	9.86

Table 12: C vs HLS performance comparison