POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Hardware acceleration of a convolutional neural network through high-level synthesis

COMPUTER SCIENCE AND ENGINEERING

Davide Preatoni **939259**
Federico Sarrocco **936199**
Alessandro Vacca **935588**

Advisor:
Prof. Davide Zoni

Co-advisors:
Dr. Andrea Galimberti

Academic year:
2021-2022

Abstract: A Convolutional Neural Network (CNN) is a learning model that is increasingly used in multiple fields of engineering, especially in image processing. High-level synthesis (HLS) is the process of automatically designing the hardware component around the software code. The goal of this project was to implement this type of network in a hardware accelerator (HA) through HLS. HA is a specialized physical component that performs a specific process and, when inserted into a system, has the purpose of improving its overall performance.

Key-words: convolutional neural network, hardware accelerator , high-level synthesis

1. Introduction

Deep Learning is a technology that has been under development since the 1980s, but it is only in the last decade that its true potential has been revealed; in fact, nowadays we have overcome the lack of data that was the main problem for a long time. Analysing the large amounts of data flooding the modern world requires a corresponding increase in computing capacity. One possible solution is the hardware acceleration of special machine learning algorithms.
The aim of this project was to learn and understand the flow of HLS by applying it concretely to a specific case. For this reason, in this report we will explain how we implemented a CNN in a Xilinx FPGA. Specifically, the CNN is trained on the MNIST [1] dataset and the project uses the translation of C/C++ code by a High-level synthesis tool called `Vitis HLS`. The report is organised as follows. In Section 2, an overview of the background of the project will be given. In Section 3 we will assess the development methodology of such project. Section 4 will present the final results of the project. The last section is where the conclusions will be taken. At the end is located a small appendix and our references.

[1] The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples

2. Background

2.1. Convolutional neural network

In the area of Deep Learning, a convolutional neural network is a class of artificial neural networks (ANNs), typically used in the fields of computer vision or language processing. The fundamental difference from fully connected neural networks is the convolutional step that extracts features from the input image. This makes the process much more efficient than the non-convolutional equivalent.

Convolution layer

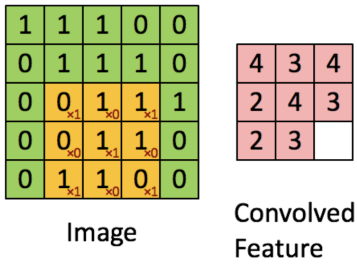

Figure 1: Example of the convolution step

This layer is defined by the operation of convolution, which is an operator between the input data (e.g. an image represented by pixel weight) and a matrix, called filter or kernel. This filter slides over the image and the output is represented by the dot product of the kernel with the sub-matrix. The length by which the filter slides is called the *stride length*. This process must be performed for each channel of the underlying input. (e.g. an RGB image has a matrix for each of the 3 colours).

Max Pooling layer

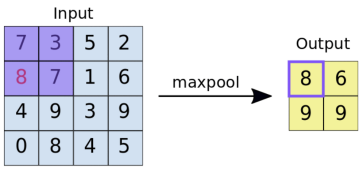

Figure 2: Example of the max pooling step

The goal is to reduce the computational complexity and to make the features of the input more robust. Similarly to the convolution layer, the filter slides over the input and extracts a subset of it. The output is the maximum value of this subset and consequently represents the input's most prominent features. The kernel moves across the image each time by the *stride length*.

Fully Connected layer

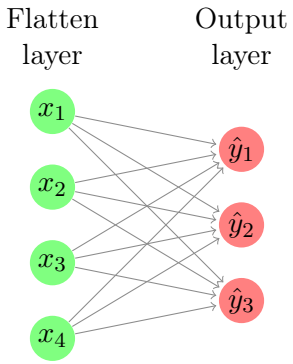

Figure 3: Example of fully connected step

First, the previous output matrix is flattened to an array. Then the array is processed by a standard fully connected layer; the general equation can be summarized as:

$$o_j = \sum_{i=0}^{I} (w_i \cdot x_i) + b_j$$

Where:

- I is the input length;
- w_i is the weight for i-th input x_i;
- b_j is the bias of j-th neuron;

The end result will be $output = [o_0, .., o_9]$ representing the probability $[P(image = 0), .., P(image = 9)]$.

Activation function

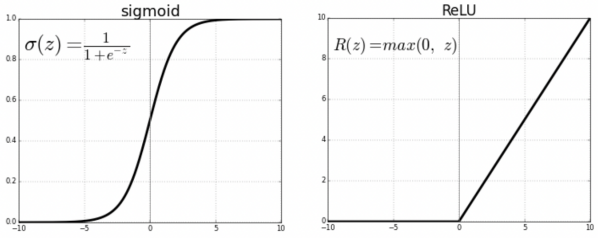

Figure 4: Sigmoid and ReLu functions

An activation function in a neural network determines how the weighted sum of the input is converted into an output. There are several activation functions, which are generally non-linear. The choice of activation function has a large impact on the capability and performance of the neural network, and different activation functions can be used in different parts of the model.

2.2. High-level synthesis

High-level synthesis tools transform a C/C++ specification into a register transfer level (RTL) implementation that can be synthesized into a field programmable gate array (FPGA). This chapter provides an overview of high-level synthesis. [2]

[2] For more information on FPGA architectures and Vivado HLS basic concepts, see the Introduction to FPGA Design Using High-Level Synthesis (UG998).

Accelerators

Hardware acceleration is the use of computer hardware designed to perform specific functions more efficiently than the software equivalent running on a general purpose central processing unit (CPU). FPGAs are a kind of hardware accelerators (semiconductor devices) based on a matrix of configurable logic blocks wired through programmable interconnects. They can be phisically manipulated using a hardware description language (HDL, such as VHDL/Verilog).

Design Workflow

As we mentioned earlier, FPGA programming uses HDL code to describe the hardware. Writing this type of code is considered a low-level approach, otherwise it's possible to use specific tools to write the code in a high-level programming language. It is possible to translate a C or C-like sequential code into the low-level HDL equivalent through an automated process called high-level synthesis. HLS design mainly consists of 5 steps:

- Write C code and test its functional behaviour;
- Adapt the code to fit the HLS workflow;
- Specify top function, constraints, clock and directives;
- The synthesizes C/C++ to RTL/VHDL;
- Verify RTL using the C test bench;

Optimization Directives

HLS tools provide several synthesis directives or pragmas to create different hardware implementations from the same C/C++ source code. The performance of the generated hardware circuits varies with the configuration of directives. We present five frequently used directives:

1. **Loop Unrolling**: Allows iterations of one loop to run in parallel, and they can be fully or partially unrolled. Fully unrolling the loop creates a copy of the body in the RTL for each iteration, so the entire loop can be run concurrently. Partially unrolling a loop lets you specify a factor N, to create N copies of the body and reduce the iterations accordingly.
2. **Loop Pipelining**: Pipelining a loop allows its operations to be implemented in a concurrent manner as shown in the following figure. By default, every iteration of a loop only starts when the previous one has finished. Pipelining the loop executes subsequent iterations in a pipelined manner. This means that subsequent iterations of the loop overlap and run concurrently, executing at different sections of the body.

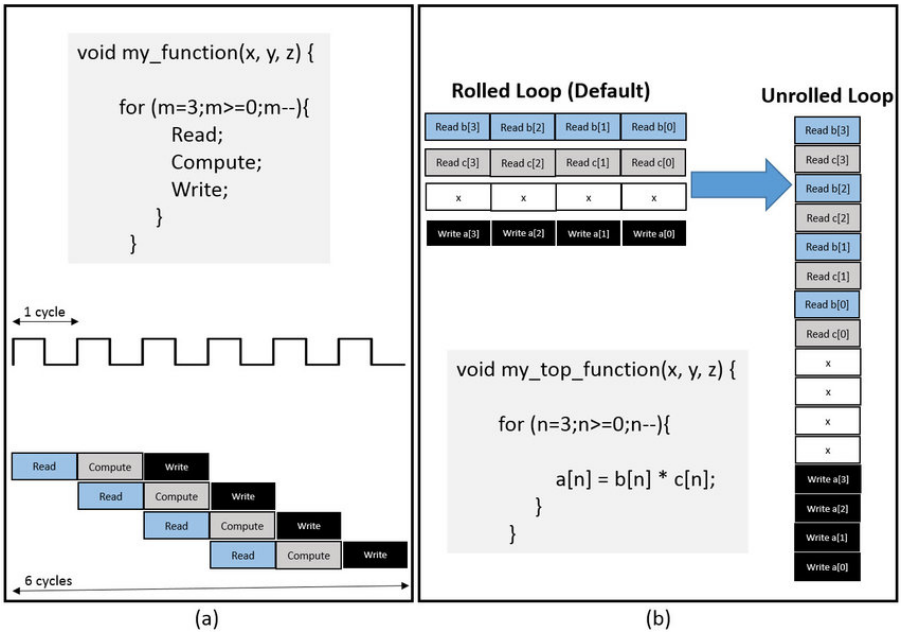

Figure 5: (a) Loop Pipelining and (b) Loop Unrolling

3. **Dataflow**: Dataflow is pipelining at the task level, allowing functions and loops to operate concurrently. It does not require sub-functions to be pipelined and sub-loops to be unrolled, but can only be applied to functions at the top level. Also, it aims at applications executing in a producer–consumer model, where the first processed output of a function is immediately propagated as the input of the next function. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the design.

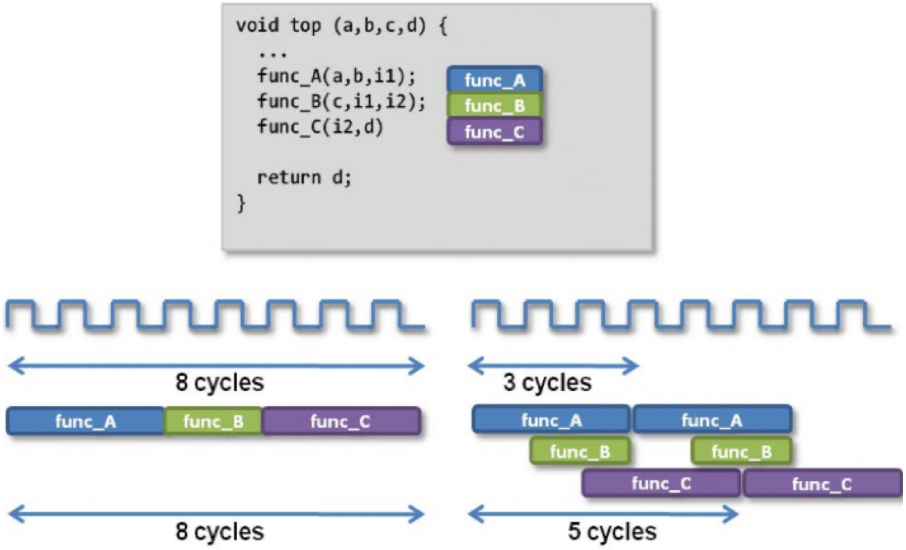

Figure 6: (a) Without Dataflow and (b) with Dataflow

4. **Array Partitioning**: Partitions an array into smaller arrays or individual elements. It results in RTL with multiple small memories or multiple registers instead of one large memory. It effectively increases the amount of read and write ports for the storage. It requires more memory instances or registers.

3. Our methodology

In this section we will discuss the development of the project. We have developed a CNN with a convolutional layer, a max-pooling layer and a fully connected layer. Every input image has a shape of 28x28x1 pixels.

3.1. Python and C/C++ implementation

We used TensorFlow's Keras API as this is an effective way to setup a working CNN prototype. The convolution layer takes the image as input and applies 32 3x3 filters. The activation function chosen for this layer is the ReLu. The max pooling layer has a filter size of 2x2 and the resulting output is then flattened and passed to the fully connected layer. It has 10 output units, one for each corresponding digit. Finally, a softmax activation function is applied. We trained the network until we achieved above 98% accuracy; then we stored all the weights for all the layers. To create an effective test bench for the C++ and HLS implementation, we also stored the output values when we received a particular test pattern as input.

Algorithm 1 Pseudo - code of the network, extracted from Python

1: $conv\_out \leftarrow layers.Conv2D(32, kernel\_size = (3,3), activation = "relu")(image)$
2: $pooling\_out \leftarrow layers.MaxPooling(pool\_size(2,2))(conv\_out)$
3: $flatten\_out \leftarrow layers.Flatten()(pooling\_out)$
4: $drop\_out \leftarrow layers.Flatten()(flatten\_out)$
5: $out \leftarrow layers.Dense(units = 10, activation = "softmax")(drop\_out)$

The next step after the development and consequent training of our CNN network was to rewrite the Python code in the C++ language. The motivation behind it is that the HLS design workflow generally requires code written in C/C++. To ensure that the newly implemented C++ code worked fine, it was tested against the Python inference using the same trained weights.

Algorithm 2 Pseudo - code of convolution layer, extracted from C++

```
out_index ← 0
for i < (IMG_SIZE − CONV_KERNEL_SIZE + 1) do
  for filter < CONV_FILTERS do
    for r_of < CONV_KERNEL_SIZE do
      for c_of < CONV_KERNEL_SIZE do
        for chns_of < CONV_CHANNELS do
          conv_buff[r_of ∗ (IMG_SIZE) ∗ (IMG_CHANNELS) + (c_of) ∗ (IMG_CHANNELS) + (chns_of) ∗ (i) ∗ (IMG_SIZE) + j]
          sum ← conv_buff[] ∗ weight[r_of][c_of][chns_of][filter]
        end for
      end for
    end for
  end for
  out[out_index] ← relu(sum + bias[filter])
  out_index + +
end for
```

Algorithm 3 Pseudo - code of max-pooling layer, extracted from C++

```
BUFFER_SIZE ← (P_SIZE ∗ P_CHANNELS)
pool_buff[BUFFER_SIZE][1]
for i < P_SIZE do
  for l < P_KERNEL_SIZE do
    for j < P_SIZE do
      for m < P_KERNEL_SIZE do
        for p < P_CHANNELS do
          read ← in[z]
          z + +
          if ((l == 0)&(m == 0)) then
            pool_buff[j ∗ P_CHANNELS + k][0] ← read
          end if

          (pool_buff[j∗P_CHANNELS+k] > read)?(pool_buff[j∗P_CHANNELS+k] : read)
          if (l == (P_KERNEL_SIZE − 1))&(m == (P_KERNEL_SIZE − 1)) then
            out[out_index] ← pool_buff[j ∗ P_CHANNELS + k][0]
            out_index + +
          end if
        end for
      end for
    end for
  end for
end for
```

Algorithm 4 Pseudo - code of fully connected layers, extracted from C++

$output[FC\_ACT\_SIZE] \leftarrow 0$
for $j < FC\_WEIGHTS\_H$ **do**
  $read \leftarrow in[j]$
  for $i < FC\_WEIGHTS\_W$ **do**
    $output[i] \leftarrow output[i] + weight[j][i] * read$
  end for
end for
for $i < FC\_WEIGHTS\_W$ **do**
  $out[i] \leftarrow output[i] + bias$
end for
$softmax(out, FC\_ACT\_SIZE)$

3.2. Code adaptations

In order to take the most advantage of FPGAs by providing highly parallel, pipelined, and customized applications, it is important to adapt the C source code before HLS.

Top function

High-level synthesis requires a single top function that performs all the necessary operations. We implented this as the `nnet()` function, that loads all the weights in the corresponding layers, and performs all the necessary computations.

Data structures

In order to have a synthesizable design the network needs to contain only data structures of static size as it is not possibile to have dynamic behaviour. In particular, we changed the `array` buffers to be of bounded size.

Fixed point arithmetics

The initial C/C++ code uses the `float` data type for parameters, which comes at a high computational cost. Fixed point operations are less precise but they use less hardware and they won't affect the global accuracy of the network. We used the `ap_fixed` HLS data type, declared as:

```
#include "ap_fixed.h"
typedef ap_fixed<24, 18> fp24;
```

Data streams

Restructuring the information to be processed between to layers was a key optimization needed for enhanced parallelization of tasks. Streaming data is a type of data transfer in which data samples are sent in sequential order starting from the first sample. Streaming requires no address management. This was combined with the `ap_fixed` data type in order to create a fast and easy to process data structure, perfectly suited for parallelization. The streams were declared as follows:

```
#include "hls_stream.h"
hls::stream<fp24> stream;
```

4. Results

For the HLS workflow we used `Xilinx Vitis HLS 2021.2` with the `Zynq 7000` board and clock of `10ns` as target. The entire project was developed on an `Intel i7-6700K` workstation running `Manjaro Linux`.

Name	Pragmas	Notes, functions
Initial	None	This is the result of auto-optimization
Base	HLS PIPELINE II = 1	`fc_mul_weight_outer`
Opt. 1	HLS PIPELINE II = 1	`conv_label3`, includes BC
Opt. 2	HLS UNROLL	`pool_maxsearch4`, includes BC
Opt. 3	HLS DATAFLOW	nnet(), includes BC

Table 1: Test cases

Table 2: Test results, notable changes in red

	CONV		POOL		FC		NNET			
	CC	FF	CC	FF	CC	FF	CC	FF	LUT	DSP
Initial	22434	1814	21638	355	54195	700	98283	6452	8251	30
Base	22438	1761	21636	491	5539	4097	49617	6349	8273	30
Opt. 1	22434	1814	21638	355	5539	4097	49620	6478	8187	30
Opt. 2	22434	1814	21646	22910	5539	4097	49627	6509	79880	30
Opt. 3	22438	1814	21638	356	5540	4188	22438	6509	8236	30

CC: Clock Cycles, FF: Flip Flop, LUT: Look Up Table, DSP: Digital Signal Processor

In general, the initial results of synthesis need to be optimised by the designer in order to be properly synthesizable. We then explored the various optimizations that can be applied to syntesisable code using `Vitis HLS`.

Initial result

When no directive is selected, `Vitis HLS` tries to auto-apply them on the code. This is generally done with `HLS PIPELINE` pragmas applied through all the code, with the iteration interval defined by incrementing it until the design satifies the timing constraints. In this case all the interval set by the tool couldn't satisfy the timing requirements, thus causing a *timing violation*, as shown by the clock cycles of the fully connected layer.

Base case

In order to have a working design, we began by fixing the timing problem of the fully connected layer by applying the `HLS PIPELINE` directive to the outer loop of said function, with the iteration interval set to 1. This reduced greatly the cycles of the fc layer, but it also increased the necessary flip-flops by a modest amount. By reducing the computational time, a corresponding increase in area used is to be expected.

Optimization 1

We tried to apply the same directive to the convolutional layer, but we could not observe any significant change in performance or area. This is likely due to the fact that by automatically optimizing the design, this layer was already pipelined.

Optimization 2

The next step was to apply `HLS UNROLL` to the max-pooling layer, because of its low complexity and high latency. We observed that the used area greatly increased, while the latency hardly changed. This is probably due to the fact that the pooling layer needs all the data from the previous layer in order to complete.

Optimization 3

By applying different directives to the functions, we found that the optimal solution is the `HLS DATAFLOW` pragma, which allows the functions to be executed in parallel, massively increasing performance.

4.1. Final results

In order to make a comparison that provides consistent data, we used the same machine with the following characteristics: `Intel i7-4700HQ CPU, macOS 12.1`:

- Python running time average: 33.461 ms
- C++ running time average: 4.081 ms
- Accelerator running time average: 0.24 ms

5. Conclusions

High-level synthesis is still a new technology. In fact, FPGA design is mainly done directly in hardware description languages. As we have seen, CPUs are several orders of magnitude slower than an FPGA in performing forward propagation. The ability to use almost the same code running on CPU for an equivalent hardware implementation is a dramatic breakthrough. As a result, software developers might be able to develop hardware descriptions without really knowing the low-level HDL languages. While working on our project, we realised that the choice of HLS directives is a very critical part, as it requires deep knowledge of the C++ code structure and the general FPGA architecture. We also understood that the choice of directives not only enables better performance, but also trade-offs between throughput and required area. Since synthesis is generally a slow step, we found that performing random tests on the position and type of directives results in a large loss of time. Therefore, an automated method that finds the best combination while trying to minimise the number of trials could be a future development, not only in relation to our project but to the HLS development in general. You can find all the material regarding our project on our GitHub repository.

A. HLS library inclusion problem

We begun our project on the `Intel i7-6700K` workstation previously mentioned, initally on `Windows 10`. In order to use mathematical functions with the `ap_fixed` data type, we had to include the `hls_math.h` library. Then a compilation problem occurred: `Vitis` reports that the compiler in set to an error state (`"File too Big"`) because it could not close an object file. After investigating the possible causes, we observed that when we used more than one function (in our case `hls_log` and `hls_exp`), `gcc` tried to include the whole library to the object file. We assessed this by observing the sizes of the object files: if we include more than one function, the size reaches almost 10 times the previous size, that was in line with other object files. On the surface, this was fixed by switching to the `Manjaro Linux` OS but the file sizes have stayed the same, hinting to the fact that the `Linux` version of `Vitis` is just more forgiving, because the issue is still there.

References

[1] AMIQ Consulting. *How to Implement a Convolutional Neural Network Using High Level Synthesis*, 2018.

[2] M.V.Valueva; N.N.Nagornov; P.A.Lyakhov; G.V.Valuev; N.I.Chervyakov. Application of the residue number system to reduce hardware costs of the convolutional neural network implementation.

[3] Grant Martin; Gary Smith. High-level synthesis: Past, present, and future.

[4] Xilinx. *Vitis High-Level Synthesis User Guide (UG1399)*, 2021.