## Project 20-1-1-2187

# CNN Acceleration and Simulation on an FPGA

### Chaim Gruda 312562721

### Shay Tsabar 208723627

## Instructor: Yoni Seifert

#### Tel Aviv University

##### Faculty of Engineering

## **Abstract**

In recent years image analyzing and recognition has become a vital feature in ever more applications, ranging from autonomous vehicles, security, healthcare and more. Convolution neural networks (CNN) algorithms are widely used in many such applications since its high accuracy for image recognition. However, with the high accuracy come very high computational complexity. The need to include such image recognition capabilities in systems with tight real-time and power constraints, lead the design for hardware accelerators for CNNs. Such accelerators may be implemented on an FPGA, GPU, or ASIC.

In this project we explored the potential of FPGA-based CNN acceleration, by implementing and simulating the CNN operation, and comparing it to software-based implementation of the same CNN.

The CNN accelerator was synthesized using High Level Synthesis (HLS), and the FPGA platform used is the Zynq-7000 based ZedBoard™.

## **Contents**

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 4 |
|  | 1.1 | Artificial Neural Networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 4 |
|  | 1.2 | Convolutional Neural Networks . . . . . . . . . . . . . . . . . . . . . . . . . . | 5 |
|  | 1.3 | CNN Acceleration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | 6 |
|  | 1.4 | Field-Programmable Gate Arrays . . . . . . . . . . . . . . . . . . . . . . . . . . | 6 |
|  |  |  |  |
| 2 | Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . | | 7 |
|  | 2.1 | High Level Synthesis |  |
|  | 2.2 | Sliding Window |  |
|  | 2.3 | Processing Element |  |
|  |  |  |  |
| 3 | Implementation | | 11 |
|  | 3.1 | FIFO |  |
|  | 3.2 | AXI interface |  |
|  | 3.3 | ZedBoard |  |
|  |  |  |  |
| 4 | Results | | 16 |
|  | 4.1 | CNN Simulation |  |
|  | 4.2 |  |  |
|  |  |  |  |
| 5 | Discussion | | 18 |
|  | 5.1 | Conclusion |  |
|  | 5.2 | Future Work |  |

## **1 Background**

**1.1 Neural Networks**

Neural networks are a family of computation architectures inspired by the biological nervous system. That system consists a large amount[[1]](#footnote-1) of neurons, which are electrically excitable cells, that are connected to each other in junctions named synapses[[2]](#footnote-2), forming a massive network. Each neuron receives input signals and produces output signals which branch out and connect to the other neurons. The synapses connecting the neurons influence the transfer of information from one neuron to the other by amplifying, attenuating, or inhibiting the signals transfer. Together, the billions of simple neurons form an incredibly complex interacting network which enables all the brain activity.

The basic building block of an Artificial Neural Network (ANN) is the artificial neuron. The artificial neuron receives several input signals from other neurons. These input signals are multiplied with weights to simulate the synaptic interaction. The weighed input signals are summed, biased, and fed into a non-linear activation function, which produces the neuron’s output signal. A neural network is formed by interconnecting many artificial neurons. Usually, the neurons are grouped into layers, and connections are only allowed between neurons of adjacent layers. Such layered connections are called Fully Connected layers.

When neural networks are employed for image-related tasks, their input usually consists of pixel data. Even for images with modest resolutions, the input consists of large amounts of elements (especially when dealing with multiple channel images, such as RGB). Large input data results in a need with amounts of neurons and connections that are a challenge for computing systems, and place a burden on the efficiency, both in time and in power, of analyzing images effectively in such neural networks.

**1.2 Convolutional Neural Networks**

Convolutional Neural Networks (CNNs) are a special class of neural networks, suited for operation on 2D input data such as images. The idea behind the CNN is the locality of information in images, which means that important information in images  
can be captured from local neighborhood relations. To decide whether there is a car in the center of an image, one does not need to consider the top-right corner pixel, and the bottom-right pixels usually do not influence the class assigned to the top-left pixels. Strong contrasts indicate edges, aligned edges result in lines, combined lines can result in circles and contours, circles can outline a wheel and multiple nearby wheels can point to the presence of a car. This locality of information in images is exploited in convolutional neural networks by adding new layers before the fully connected layers, namely convolutional layers, and pooling layers.

The purpose of convolutional layers (CONV) and pooling layers (POOL) is to reduce the spatial data of 2D images into smaller data sets, that contain the most distilled data from within the image. After that process is done, the refined data can be passed through fully connected layers (FC), which will now be much smaller and tailored for the purified information. A CNN may have multiple CONV and POOL layers.

* **Convolutional layer** – This layer preforms the mathematical 2D convolution. This operation is described by Eq. (1):

|  |  |  |
| --- | --- | --- |
| (1) |  |  |

The operation basically slides filter of size , where each pixel in the filter has its own weight , over the 2D image, and computing the sum of the product of all overlapping pixels. The filter is also known as a kernel, and the convolution output is called a feature map. The operation is described in **Figure 1**.

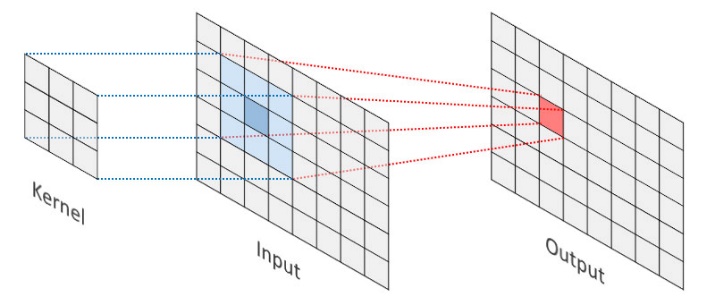


Figure : 2D convolution illustration

* **Activation Layer –** This layer applies a non-linear activation over the CONV layer, introducing non-linearity into the CNN. The non-linearity increases the power of deepening the network (moreover, if the CNN would be linear then a set of N layers could be represented in a single layer). One of the most common non-linear activation functions used in CNNs is the Rectified Linear Unit (ReLU) function: , seen in **Figure 2**.

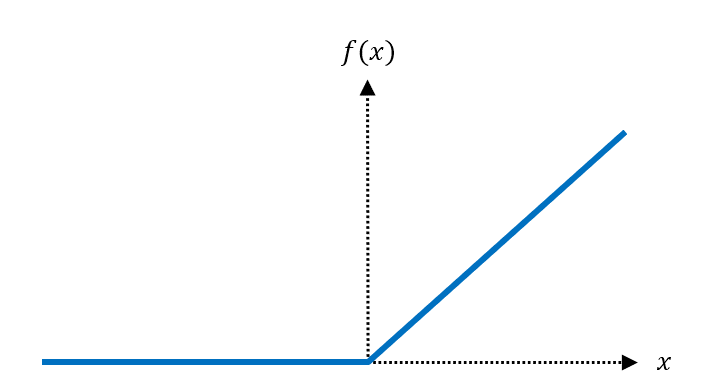


Figure : ReLU function

* **Pooling layer** – This layer reduces the spatial area of the given input, usually being the output of a convolutional layer. This is done by selecting the max value or calculating the average value from a 2D section. These are called Max-Pool and Avg-Pool, respectively. Max polling can be visualized in **Figure 2**.

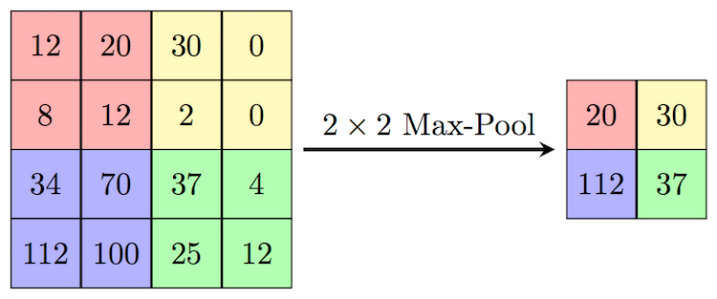


Figure : Max pooling illustration

* **Fully Connected layer** – The operation of FC layer can be expressed by Eq. (2). Input features are fully connected to an output feature vector by weights (). The and are the lengths of the input and output feature vector and the number of weights is . is the corresponding bias term of the th output feature vector. FC layers usually turn the input feature maps into a feature vector.

|  |  |  |
| --- | --- | --- |
| (2) |  |  |

**1.3 CNN Acceleration**

As CNNs achieve very high accuracy in image understanding and classification, they get embedded in more applications, some of which require tight real-time constraints. This leads to the need to accelerate the CNN calculations.

Most of the computations done in the CNN is in the convolutional layers, around 85-90%[[3]](#footnote-3). Since all layers: CONV, POOL and FC, basically deal with matrices, the software implementation of those layers consists of nested loops. This means all computations are done serially, and thus the time the network operates on a given output may be very long.

To reduce the computation time, and thus to accelerate the CNN performance, the different layers can be implemented in hardware. Since hardware can preform computations in parallel, this basically allows to expand the software loops so that a CONV single operation can be done in a single clock cycle.

Potential hardware platforms that can be used for accelerating the CNN are DSPs, GPUs, ASICs, and FPGAs. The advantages for using an FPGA are discussed in the following section.

**1.4 Field-Programmable Arrays**

Field-Programmable Gate Arrays (FPGAs) are semiconductor devices consisting of a 2D array of configurable logic blocks which are connected via programmable interconnects. The interconnect can be thought of as a network of wire bundles running vertically and horizontally between the logic slices, with switchboxes at each intersection. Modern high-end FPGA generations feature hundreds of thousands of configurable logic blocks, and additionally include an abundance of hardened functional units which enable fast and efficient implementations of common functions. The logic blocks, the fixed-function units as well as the interconnect are programmed electronically by writing a configuration bitstream into the device. The configuration is typically held in SRAM memory, and the FPGAs can be reprogrammed many times

The advantage of FPGA-based systems over traditional processor-based systems including GPUs, is the availability of freely programmable general-purpose logic blocks. These can be arranged into specialized accelerators for very specific tasks, resulting in improved processing speed, higher throughput, and power savings. This advantage comes at the price of reduced agility and increased complexity during the development, where the designer needs to carefully consider the available hardware resources and the efficient mapping of his algorithm onto the FPGA architecture. Further, some algorithmic problems do not map well onto the rigid block structures found on FPGAs. Compared to ASIC (Application-Specific Integrated Circuits) implementations, the FPGA advantages are re-programmability, that allows fast prototyping and short development cycles

## **2 Design**

**2.1 Vivado HLS**

Vivado HLS (VHLS), is a High-Level Synthesis tool for Xilinx FPGAs. It is a standalone application based on the Eclipse development environment. VHLS can be used to write and debug C, C++ and SystemC specifications of FPGA designs. Its most important component is the *HLS compiler*, which analyzes and translates the high-level code into an intermediate low-level representation of all the necessary instructions to run the program. It then optimizes and parallelizes these instructions into a *schedule* and a *resource allocation* scheme, and generates suitable RTL code in either Verilog or VHDL. A very important capability of HLS software is the automated *verification* of the generated RTL code, which enables designers to use the original high-level software specification as a *test-bench*. If the software model includes enough test cases, and the automated verification passes, the generated RTL code can be assumed to be correct. This feature works by means  
of *co-simulation* in Vivado HLS: In a first step, the software model is executed and all input and output data consumed and generated by the function-to-be-synthesized are recorded. Then the RTL code is simulated with the recorded data as input stimuli, and the output from both the software model and the RTL simulation are compared. All data values consumed and emitted must match in order for the co-simulation to be successful. A unique characteristic of high-level synthesis with C and C++ is the complete absence of the concepts of timing and clock cycles in the software specification (which can be both a curse and a blessing, as further explained in section 4.4.4 on the limitations of the HLS approach). The HLS design is constrained, shaped and optimized using a number of *compiler directives* (either as in-code *pragmas* or using a separate TCL-based script). The directives can be used to specify the implementation and partitioning of memories, the unrolling of loops, function-level pipelining, etc. More details on compiler directives follow in section 4.4.3.  
Vivado HLS also has an astonishingly good support for object-oriented C++. There is full support for C++ classes, private and public member variables, and even (compile-time resolvable) inheritance.7 Pointers and even double-pointers are also supported, albeit with some limitations: Pointers can only be casted between native C types, arrays of pointers may only point to scalars or arrays of scalars, and all functions which use a double-pointer are inlined together [63]. After each synthesis run, Vivado HLS estimates the device utilization and the maximum achievable clock frequency of the design. The tool also provides a number of different  
analysis views that visualize the resources allocated for each code section as well as the exact schedules for each loop and function. Handing the tedious process of writing register transfer level code off to the compiler can heavily speed up the development of FPGA designs. Xilinx talks about average 4*×* speed gains in the development of new components, and speed gains of up to 10*×* when adapting previous designs, while reaching between 70 % to 120 % of the quality of results with respect to hand-coded RTL [113]. This speedup, combined with a more agile development style and increased flexibility are especially important with regard to the ever-growing design complexities and the increasing capacities of newer FPGA generations.

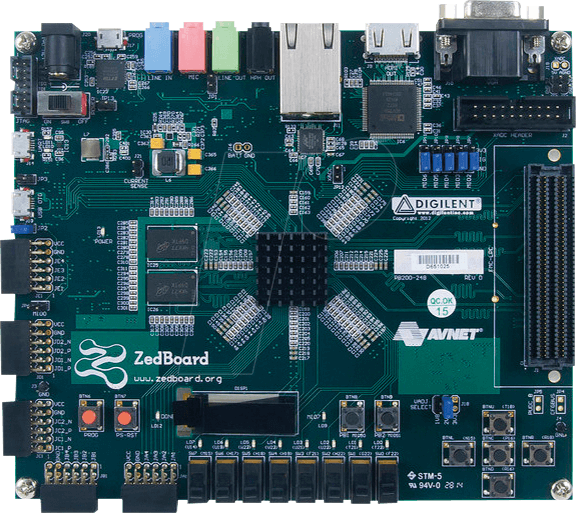
Of course, HLS compilers get better at understanding different algorithms and coding  
styles with every new version, and their coming can be compared to the rise of logic  
synthesizers which required very specific description styles initially, and are capable of  
generating relatively efficient logic from almost any specification today. One key aspect is to  
accept a design that may not be a *perfect* solution, but which does the job *well enough* [67].  
With RTL designs, the goal was often to optimize an architecture down to the last bit. This  
may no longer be feasible with HLS – but maybe also no longer necessary, thanks to the  
abundance of resources available in modern FPGAs.  
Initially, we found it very difficult to find a satisfactory coding style, given the short code  
examples and the different styles used in the two HLS guidelines. This section describes the  
experiments made and reports their successes and failures.8  
**Unstructured, Monolithic**The first software implementation of the ZynqNet FPGA Accelerator algorithm was designed  
to be as simple as possible:  
• straightforward implementation of algorithm 1  
• 7 nested loops (layers, height, width, input channels, output channels, kernel y and x)  
• all arrays and most variables declared as global  
This seemed like an adequate representation of the FPGA hardware (where all resources  
necessarily exist “globally”) with minimum complexity of the control flow (no function calls,  
no handshakes, just a number of interconnected counters for the loop indices). The C++  
software model eventually compiled and worked well. However, the HLS synthesis got  
stuck in the middle of the *design analysis* phase without any error or indication of what was  
8More code examples can also be found in the Vivado High-Level Synthesis Tutorials (UG871) [114].  
**44 Chapter 4** FPGA Accelerator Design and Implementation  
currently being analyzed. Many variants and changes to the code were tried without success,  
and overall this coding style proved to be:  
• hard to constrain using pragmas (scopes of application are not well defined)  
• hard to read and maintain due to the unstructured “spaghetti code” nature  
• very hard to debug due to the lack of indicative error messages  
**Object-Oriented**Having learned from the previous approach, the software model was rewritten from scratch.  
The new version included extensive testing, logging, debugging and verification facilities.  
Additionally, a conversion tool for .prototxt network description files and .caffemodel  
weight files as well as support for strided convolution and on-the-fly padding were added.  
This enables a very easy adaptation and reconfiguration of the architecture for different  
network topologies. This time, the accelerator core was written in an object-oriented  
manner:  
• hardware blocks modeled as class instances (MemoryController, ImageCache, WeightsCache, OutputCache, ProcessingElement)  
• arrays and variables encapsulated as private class members  
• data movement via high-level member functions (data\_t ICache::getPixel(y,x,ci),  
**void** PE::macc2D(data\_t pixels[9], data\_t weights[9], data\_t& result). . . )  
• control flow still via nested loops in top-level function (layer, height, width, input  
channels) and inside **class ProcessingElement** (output channel, kernel y and x)  
This coding style gave better results and was more pleasant to work with. By splitting the  
code into separate functional units, problems during synthesis became easier to trace and  
isolate. The usage of compiler directives was simplified, because the lexical scopes to which  
the directives apply now coincide with functional blocks (e.g. pipelining can be explicitly  
applied to the postprocessing function and to the pixel writeback loop).  
However, it was still not possible to complete synthesis. This time we experienced multiple  
fatal crashes of the HLS compiler process during the *RTL generation* phase. Closer inspection  
suggested that the compiler automatically inlined multiple hierarchical levels of function  
calls in order to avoid double or even triple pointers, and tripped somewhere in that  
process. Double and triple pointers are very easily created in object-oriented code. For  
example, assume a class ProcessingElement, which includes a reference to an instance of  
another class WeightsCache \*ProcessingElement::WCache, which *itself* contains an array  
data\_t WeightsCache::BRAM[] (the variable BRAM may be hidden behind an interface  
data\_t WeightsCache::read(**int** addr), but due to various reasons, this type of functions  
tend to be inlined by Vivado HLS). BRAM is then accessed as **this**->WCache->BRAM from  
inside class ProcessingElement (double pointer), and is itself a pointer to elements of type  
data\_t (triple pointer). The HLS compiler tries to avoid these double and triple pointers,  
and may for example try to inline the whole instance WCache into ProcessingElement, but  
this quickly gets messy (imagine for example that a member function of another class *also*accesses the BRAM array). Therefore, the object-oriented coding style had mixed success:  
• much easier to read, modify and debug  
**4.4** High-Level Synthesis Design Flow **45**  
• much easier to apply compiler directives  
• still no successful synthesis (triple pointers due to class instances)  
**Block-Structured**The third approach was finally designed to be a compromise between the flat spaghetti  
code approach and the fully hierarchical object-oriented approach. This coding style uses  
*namespaces* to structure the code conceptually, while avoiding the need for references  
and pointers. With namespaces, modular and object-centric code can be written (such as  
data\_t px = ImageCache::getPixel(y,x,ci) or OutputCache::reset()), but the actual  
hierarchy stays flat (when OutputCache is simply a namespace and not an object, no  
references or pointers to it are needed to access data\_t OutputCache::BRAM[]). The  
software model for the ZynqNet FPGA Accelerator has been partially rewritten to fit this  
*namespace-based* or *block-structured* coding style:  
• use namespaces to structure code into modules  
• arrays and variables are encapsulated in namespace-scopes  
• data movement is done via high-level namespace-scoped functions  
• control flow still via nested loops in top-level function (layer, height, width, input  
channel) and inside **namespace** ProcessingElement (output channel, kernel y and x)  
This approach worked very well, except for one flaw: Global pointers are not supported for  
synthesis in Vivado HLS, and variables defined within namespaces are also considered global.  
Therefore, the declaration of a pointer into main memory via data\_t\* MemoryController::  
SHARED\_DRAM is not synthesizable, and accesses into main memory can not be properly  
hidden behind an interface (such as data\_t MemoryController::loadNextWeight()). Instead the pointer into main memory (which comes in as an argument to the top-level  
function) has to be dragged through all affected functions as an additional argument (such  
as data\_t MemoryController::loadNextWeight(data\_t\* SHARED\_DRAM), and therefore  
also WeightsCache::loadFromDRAM(data\_t \*SHARED\_DRAM)). While this solution is not  
very elegant, it works and this last coding style finally resulted in a synthesizeable design.  
The *namespace-based* coding style combines the advantages of both previous attempts, and  
we would describe our next HLS design in this coding style again:  
• straightforward, close to hardware description  
• easy to read, modify and debug code  
• easy to apply compiler directives  
4.4.3 Compiler Directives  
The high-level languages C and C++ by themselves do not allow the designer to specify  
concurrency in the code. Frameworks which enable the explicit parallelization of C and C++  
programs typically use either the concept of *kernels* or *threads* which are launched in parallel  
(e.g. CUDA, OpenCL or Pthreads), or they allow designers to annotate the source code with  
*compiler directives* that specify the desired type of parallelization (e.g. OpenMP).  
**46 Chapter 4** FPGA Accelerator Design and Implementation  
As already indicated earlier, Vivado HLS uses this second approach and supports the annotation of the high-level source code using *#pragma* directives.9 The compiler directives affect  
all code in the lexical scope in which they have been placed (such as a function, loop, or the  
branch of an if-clause), and can influence e.g. the synthesis of FPGA memories from arrays,  
the derivation of control and data flows, and the parallelization and pipelining of individual  
code sections. However, in comparison to directly writing RTL code where the structure  
and timing of the design can be exactly controlled, shaping an architecture using compiler  
directives can feel more like trying to thread a needle while wearing fireproof gloves.  
In this section we introduce the most important *#pragma HLS* compiler directives which  
have been used for the ZynqNet FPGA Accelerator.  
**Interfaces**Vivado HLS usually synthesizes C/C++ functions into different functional entities. All blocks  
automatically receive clock and reset ports (ap\_clk, ap\_rst). The function arguments are  
turned into RTL ports of different types. The compiler directive *#pragma HLS INTERFACE  
<mode> [register] [depth=<D>] port=<P>* allows the specification of the *function-level  
interface protocol* and the *port-level interface protocol* for each argument.  
**Function-Level Interface Protocols** The function-level interface protocol is set by applying  
the *#pragma* to the return port. The choices are ap\_none, ap\_ctrl\_hs and ap\_ctrl\_chain,  
where the *handshake* protocol ap\_ctrl\_hs is the default and creates ap\_start, ap\_done,  
ap\_ready and ap\_idle signals which let the blocks negotiate data transfers.  
**Port-Level Interface Protocols** When the *#pragma* is applied to individual arguments to set  
the port-level interface protocol, there are many modes available depending on the type of  
argument, and both inputs and outputs can be automatically registered. ap\_none is the default mode for scalar pass-by-value and pass-by-reference inputs and corresponds to a simple  
wire. The ap\_vld, ap\_ack, ap\_ovld and ap\_hs modes add increasingly complex handshake  
protocols to the individual signals, with the *output-valid* protocol ap\_ovld being standard  
for pass-by-reference outputs. Arrays on the function interface are normally synthesized into  
ap\_memory ports, which creates *data, address* and *RAM control* ports. Alternatively, the port  
can be turned into an ap\_fifo interface if the access patterns correspond to a first-in-first-out  
buffer behavior.  
**AXI4 Interfaces** On the top-level, Vivado HLS also supports the axis (AXI4-Stream), m\_axi  
(AXI4-Master) and s\_axilite (AXI4-Lite) interfaces which strongly simplify the connection  
of the design into a larger system architecture. This project uses the AXI4-Master interface to  
connect to the main memory via the AXI4 bus in the Zynq XC-7Z045. An AXI4-Lite interface  
is used for configuring, starting and stopping the accelerator. Vivado automatically generates  
C/C++ driver files for accessing the AXI4-Lite ports from software running either on the  
Zynq’s ARM cores or on Soft Processor Cores in the FPGA fabric.  
9Alternatively, TCL-based scripts can be used, which allows a separation of the optimization directives and the  
code. The scripts support the same compiler directives, but have not been used in this project.  
**4.4** High-Level Synthesis Design Flow **47**  
**AXI4 Depth Settings** When proceeding to the co-simulation, it is crucial to set the depth of  
the AXI4-Master ports correctly (i.e. to the exact number of elements in the array connected  
to this port on the test-bench side). Setting the depth too small results in the simulation  
getting stuck. Setting the depth too large results in ambiguous error messages or even  
segmentation faults in Vivado HLS 2016.2. The depth can also be passed to the *#pragma*using a **const int** variable in C++.  
**Data and Control Flow**There are a number of *#pragma* directives that affect the control flow in hardware, and are  
therefore very important for parallelizing algorithms.  
**Loop Unrolling** *#pragma HLS UNROLL [factor=<N>]* instructs the compiler to unroll the  
loop in which the *#pragma* is placed, either completely or partially by a factor of N. Because  
Vivado HLS by default schedules all operations as soon as they are ready for execution, these  
unrolled iterations are then executed in parallel. Of course, unrolling only works if there  
are no dependencies between the loop iterations, and complete unrolling requires known  
loop bounds at compile time. Besides the opportunity for parallel execution of the loop body,  
unrolling also removes the loop entry and exit overhead, which otherwise adds two clock  
cycles to every iteration.  
**Dependencies** *#pragma HLS DEPENDENCE variable=<var> <intra|inter> [false]* allows to override the automatic (and relatively conservative) dependency analysis. This  
directive needs to be applied when loops cannot be unrolled because a (false) inter-iteration  
dependency is detected by the compiler. For example, a loop which executes a read-modifywrite operation on every individual element of an array cannot be unrolled by default,  
because the compiler sees read-after-write operations on the same array variable. However,  
the designer knows that the operations target different elements in the array in every loop  
cycle, and can therefore assert a *false dependency* to re-enable loop unrolling.  
**Loop and Function Pipelining** *#pragma HLS PIPELINE [II=<N>]* is a very important optimization directive for loops as well as for functions. This *#pragma* enables pipelining for the  
context in which it is placed, and for all entities in the hierarchy below. Vivado tries to build  
a pipelined design with an initiation interval II=<N> (default: N=1), which means that a new  
data element is accepted into the pipeline every N clock cycles. The necessary depth of the  
pipeline (and the corresponding latency) are automatically determined by the compiler. An  
important caveat is the fact that pipelining *forces all loops in the hierarchy below to be fully  
unrolled*. Full unrolling requires fixed loop bounds, and therefore this requirement can often  
prevent the pipelining of higher-level loops and functions, even if the lower-level loops are  
themselves pipelined and would be fully compatible with e.g. II=1.  
**Resource Specification and Pipelining of Arithmetic Operations** *#pragma HLS RESOURCE  
variable=<var> core=<string> [latency=<N>]* specifies the resource (core) that should  
be used to implement variable var in the RTL. This can be useful to select a certain type of  
memory for an array (e.g. dual-ported block RAM RAM\_2P\_BRAM or single-ported distributed  
ROM ROM\_1P\_LUTRAM), but it is also very useful to pipeline arithmetic operations:

**Function Inlining** *#pragma HLS INLINE* forces a function to be inlined into all its callers,  
which effectively creates copies and additional hardware, and thereby avoids the overhead  
of the function-level handshake (which is typically around 2 to 3 clock cycles). Vivado HLS  
often inlines functions automatically, e.g. to increase throughput. This can be prohibited by  
specifying *#pragma HLS INLINE off* .

4.4.4 Limitations and Problems

**2.2 ZedBoard FPGA**

The FPGA platform used for this project is the Xilinx ZedBoard Zynq-7000 based FPGA, seen in **Figure X**.



**Zynq-7000**

**DDR3**

The Zynq-7000 contains two sub-systems:

1. Processing System (PS) – 2 ARM Cortex A9 CPUs
2. Programmable Logic (PL) – an FPGA

The two subsystems are utilized and work together in the CNN implementation. The PL containing the CNN acceleration cores, and the PS running the cores drivers – initializing and configuring the CNN, feeding it data, and receiving the results. The communication between the systems is by AXI interface. Both systems also have access to the DDR3 memory using an AXI-DMA core controlled by the PS.

**2.3 Fixed Point Calculations**

To simplify HW implementation, data in HW is represented in fixed point format. Out of 32 bits of word length, 10 bits are used to represent the fraction. This gives a resolution of . The downside of using fixed point is the loss of accuracy in calculations, which leads to less accuracy of the accelerated network. The accuracy loss can be seen in the simulation results.

The reason for using 10 bits only for the fraction, is since when multiplying 2 fixed point numbers, the resulting word will have double the bits representing the fraction. Thus, using more bits for the fraction would lead to overflows in the integer values of the represented numbers.

**2.4 Sliding Window**

Since the input data is held in PS memory as an ordered array, when preforming CONV or POOL operations on the data, out of order pixels need to be extracted and operated on. Preforming the extraction in software is quite simple since the array is stored in RAM. In HW this is more complex, and thus there are 2 possible solutions. The simple one is to utilize the SW to extract the needed data per operation and feed it to the HW to calculate. However, this means data pixels are passed from SW to HW multiple times, causing a bottle neck that may reduce the acceleration affect.

The more advanced way to preform the operation, is to have a sliding window mechanism implemented in HW. That way the ordered array can be passed in an ordered stream to HW, and HW will know what pixels to operate on at any given cycle. The sliding window concept is show in **Figure X**.

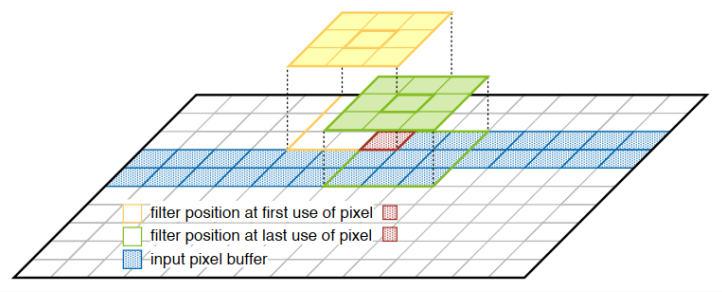


Figure :

The main advantage of the sliding window mechanism is that each pixel is passed to HW only one singe time. Another advantage is that the CONV and POOL layers can start preforming the operation on the full image, even as soon as the first needed data pixels arrive and pass their results to the next layers, which in their turn can start calculating even while prior layers are still calculating. This increases the acceleration affect and allows high data throughput in the CNN.

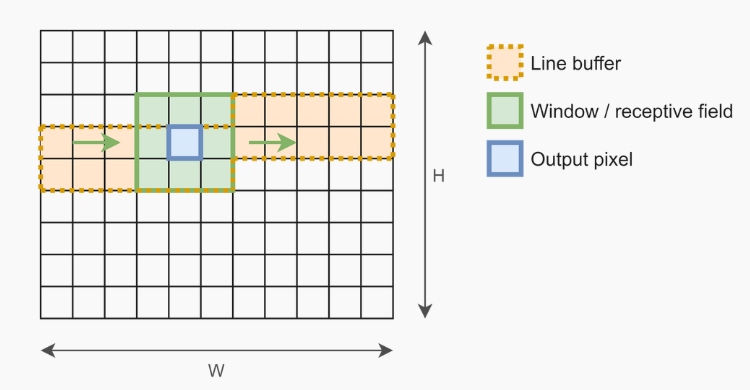
**2.4 System Flexibility**

To increase the systems flexibility, we have considered implementing HW blocks that can handle dynamic sizes of data input and operation (CONV/POOL) dimensions. This proved to complicate the system beyond the capabilities of the Vivado HLS and the Zedboard capabilities. moreover, one may argue that the FPGA would be better utilized if the HW blocks were exactly matched to the CNN data and operation dimensions. Allowing the resources to utilized to the maximum per application.

Instead of that, the solution we decided on was to completely automate the creation of the CNN, both HW and SW. the user need to only configure a settings file, setting the desired network layers data dimensions, and run a script. The script generates all the code, build the block design in Vivado and exports the bitstream to SDK with the needed drivers.

## **3 Implementation**

**3.1 Concepts**



**3.1 Sliding Window**

**3.2 Processing Elements**

Each processing element

Three layer types were implemented

Conv Elements

Also include activation

Pool elements (support max and avg)

**3.2 Software Design**

## **4 Results**

Simulation explanation

Simulation results

Simulation results discussion

## **5 Discussion**

**5.1 Conclusion**

The

**5.2 Future work**

Fit for an actual CNN

Dynamic sizes

## **References**

## **Learning Materials**

* [Vivado HLS Course Training](https://www.youtube.com/playlist?list=PLo7bVbJhQ6qzK6ELKCm8H_WEzzcr5YXHC)

1. For example: a mouse brain has approx. 70 million neurons, human brain approx. 100 billion neurons [↑](#footnote-ref-1)
2. Mouse: approx. 1 trillion synapses, human: approx. 125 trillion synapses [↑](#footnote-ref-2)
3. link [↑](#footnote-ref-3)