POLITECNICO DI TORINO

Master degree in Computer Engineering

Master Degree Thesis

Quantifying the figures of merit of MAC architectures for Deep Learning Accelerators

Supervisors:
Prof. Andrea Calimera
Dr. Valerio Tenace

Candidate
Ignacio Goldman

Academic Year 2018

Contents

List of Figures

List of Tables

Summary

Artificial intelligence is moving ahead at a staggering speed in applications and is spreading rapidly in many aspects of daily life such as face and gesture recognition, vision, autonomous cars, remote sensing and robots, agriculture, augmented reality, and bio-metrics, just to name a few.

The potential is even greater since modern approaches of artificial intelligence, such as Machine Learning or Deep Learning, can be applied onto smaller devices such as smartphones or even smaller ones like embedded systems with severe performance constraints. One of the main problems of these new approach to artificial intelligence is the resource usage. Convolutional Neural Networks (CNNs), for instance, need high amounts of data to work, thus implying heavyweight computations during the training phase, as well as during inference stages.

For these reasons, many companies and research groups are working on new dedicated hardware solutions for accelerating CNN operations. In particular, NVIDIA has released, and it is still working on, an open source architecture of a CNN accelerator called NVDLA. This architecture has some interesting points such as that a convolution pipeline working with 16-bits floating point operations (also referred to as the *full-precision* implementation), and a high reconfigurability and modularity. Indeed, modules and cores are fully-independent between them, so they can be removed, replaced, or modified as needed.

Taking advantage of those characteristics, the main objective of this thesis is to analyze and compare the figures of merit of the NVDLA architecture under different working conditions. Tested configurations include a full-precision 16-bit, and a reduced 8-bit implementation. Comparisons have been carried out in terms of area, power, and speed for each configuration. More in detail, the investigation has been done taking into account the most important component of a CNN, the *convolutional module*, where more than 90% of the operations are represented by matrix-vector multiplications, or multiply-and-accumulate (MAC) operations.

Therefore, to improve the yields of the NVDLA, two *inaccurate* multiplier architectures geared towards efficient mathematical operations were also included in the analyses. Preliminary results suggest that although inaccurate multipliers introduce errors in MAC operations, this error is not sufficient enough to affect the prediction

results. In other words, trading accuracy for area and power saving is possible and the prediction accuracy does not vary abruptly.

The work has been carried out on the basis of two work platforms. A *hardware platform* where the accelerator is described in Verilog and in another *software platform* described in SystemC.

The first one, allows to fully-modify each block of the architecture, run single layer simulations of CNN such as Googlenet or AlexNet and synthesize the model to obtain detailed information in terms of area, power, and speed of the architecture. It's also the access point to dump the model to an FPGA.

Otherwise, the software platform is optimal for faster and more complex simulations such as complete inferences of real CNN models. Accelerators descriptions (HDL and SystemC) are directly related between them, since modules have the same logical/behavioral description both in hardware as in software. For instance, if the multiplicator is described as a booth multiplier in Verilog, the same behavior is replicated in SystemC.

NVDLA open source provides many scripts for different applications such as RTL model automatic builds, hardware compilation, simulation, and synthesis. RTL models and synthesis are generated without problems, no errors appears with tools. Otherwise, compilation and simulation is done through a Synopsys Tool called VCS. Due the absence of this tool, the accelerator has been adapted to Modelsim. So, a specific *environment* has been created to automatically correct and modify many files for the compilation of NVDLA in Modelsim. Different RTL models can be inserted in this environment and should compile with little or no modification. Googlenet and AlexNet single layer simulations have been run using a default NVDLA without internal modifications. Data-flow, of features and weights, inside the convolution core particularly inside the MAC, has been extracted to use them later.

All these works mentioned above has been done at *hardware platform* level, specifically they are the *NVDLA hardware platform* possibilities. The other half of the platform is called *MAC hardware platform* and are detailed below.

The NVDLA multiply-and-accumulate unit has been extracted and used as a base for generating custom MAC units using different multiplicators.

NVDLA has described it's multiplicator in a standard way (*dot* symbol between two inputs). Obviously, is an *accurate* multiplier, it not introduces error. Otherwise, inaccurate multipliers has been tested inside the NVDLA MAC unit. It is thought that even if an error is introduced in multiplication, it is not enough to change the outcome of a prediction.

Thus, 8-bits and 16-bits MAC architectures using every multiplier has been generated. Using the extracted data-flow, every MAC has been simulated to check the correctness of the architecture, tested with a single layer Googlenet and AlexNet test-bench. Every architecture has been synthesized and compared in terms of area,

power, and speed and then the netlist has been used for a deeper analysis of the power consumption calculating the switching activity for every circuit.

The software framework has been used for fully-layer and more complete inferences simulations. NVDLA software runs inside a Virtual Platform environment with its own kernels and Linux version. Once the NVDLA software has been installed inside the platform, the platform was able to compile the accelerator and run inferences. For running inferences inside the accelerator, compilation and run-time tools must be configured. Compilation tools converts pretrained CNNs models into loadable files for NVDLA, while the run-time tools loads it into the accelerator with an image for an inference.

In hardware framework, several architectures have been generated and tested. The same architectures have been generated for the software environment in SystemC language.

The architectures has been tested using a LeNet as CNN model to predict handwritten digits (accomplished with a MNIST dataset) and vehicles and animals (CIFAR10).

The accelerator quantifies the incoming 32-bits data (or 64-bits) generated by software tool such as TensorFlow and Caffe down to 16-bits as maximum since the accelerator pipeline possibles data-types are FP16, INT16 and INT8. Despite of this, for a handwritten digits prediction using a LeNet neural network and the MNIST dataset, the prediction accuracy is 98.5%, similar value of inferences without data quantization [7].

If MAC unit is reduced to 8-bits, the accuracy of prediction still at 98.5% but comparing the architectures in terms of area, power, and speed, working at 8-bits has several advantages: 70.6% reduction in terms of area (comparison between 16-bits and 8-bits MAC units), 84% less power consumption and is able to work up to 1GHz frequency.

Which multiplicator unit is convenient? It is possible to trade accuracy with power using inaccurate multipliers inside a CNN? The performance of the inferential multiplier in terms of prediction accuracy is not as it was expected, it has an accuracy of 68% while the approximate multiplier has an accuracy of 98.5%, same value of using the NVDLA multiplicator. When comparing the MAC modules, when the approximate multiplier is inserted, it has 24.1% less area and a minimal reduction in power consumption in comparison with the NVDLA MAC module with the NVDLA multiplier. Hence, another optimization in MAC unit is possible using inaccurate multipliers without loosing accuracy in predictions results.

Acknowledgements

I would like to express gratitude to thank Professor Andrea Calimera, who after having one of his courses has given me the opportunity to work at Electronics Design Automation laboratories of Politecnico Di Torino.

They deserve my thanks also Valerio Tenace for continuously following me with my work, helping me whenever I need it, and Roberto Rizzo, with whom I have worked a large part of the thesis.

Chapter 1

Introduction

1.1 Motivation

Artificial Intelligence is spreading rapidly in many aspects of daily life. Smartphones and all types of computers are using techniques for identifying objects, recognizing voice or face, intelligent marketing and statistics just to name a few.

Approaches for reaching artificial intelligence are evolving and improving in terms of performance every year. For instance, Machine Learning, widely used technique for *Computer Vision* is being overcome in terms of performance with Deep Learning structures.

Machine Learning approach needs a feature extraction step before training the model, Deep Learning algorithms automatically reach this task with backpropagation techniques. Thus, the advantages of using Deep networks are: higher performances since the features are set on their own, and no human error is introduced with the cost of higher number of computations. To obtain better performances with Deep Neural models, a high amount of images is needed for training it.

Convolutional neural networks used in deep learning execute a high amount of computations whence sometimes to train a model are needed from weeks to months, not always arriving to the expected results. To increase the training and inference speed, many companies are improving the performances of CNNs at software or hardware level. NVIDIA has developed an open source hardware accelerator called NVDLA for accelerating deep learning operations.

The accelerator is highly configurable, so different models can be generated trading-off performance and savings. Starting from a full version based on performance and speed without taking into account area and power consumption the accelerator can be reduced down to a scalable small version for being programmed into Field Programmable Gate Arrays. This version is also suitable for smaller devices such as smartphones or embedded systems due the complexity reduction.

NVDLA can be shaped, so every internal module can be modified or replaced with a custom one. This allows the user to test custom hardware architectures in real inferences of CNN in terms of prediction accuracy, area, power and delay.

1.2 Objective

TensorFlow, Caffe and Pytorch are some of the deep learning frameworks used for training and testing deep neural networks. Graphics Processing Unit (GPU) is usually the hardware to solve deep neural operations. GPUs are general purpose architectures used for many tasks. Otherwise, NVDLA is a single purpose accelerator used for increasing the performance when solving deep neural networks operations. Therefore, one of the main differences between them is that NVDLA pipeline data-types are FP16, INT16 or INT8, reducing the precision of data for faster operations and/or hardware reductions in terms of area, power and delay (data-type of GPUs are commonly FP64 or FP32). The first objective of this work is to compare inferences results using the accelerator and other software platforms (such as Caffe) running in regular GPUs and analyze how a reduction in precision affects to final inferences results.

In Convolutional Neural Networks, more than the 90% of the computations are done in convolutional layers [5]. The key operation is the multiply-and-accumulate (MAC): multiplications between inputs activations and weights ending up in a sum using an accumulator. The MAC unit is composed by a set of "n" multipliers depending on the parallelism, so the multiplication is crucial in deep learning applications.

Special architectures of multiplicators such as an approximate multiplicator [6] and an inference multiplicator [7] trade accuracy for area and power. The second objective is to insert these inaccurate multipliers into the accelerator and compare them in terms of precision. If the accuracy of predictions do not vary drastically, smaller and less power hungry architectures can be generated from these multipliers.

1.3 Organization

Chapter 2 introduces background concepts such as Deep Learning, Computer Vision, Convolutional Neural Networks, and inaccurate multipliers architectures.

Chapter 3 contains a study of the accelerator architecture, interfaces, pipeline, and more.

Chapter 4 describes the *Hardware Framework* from the set-up up to its usability and tests. This section is divided in two: an *NVDLA Framework* where is possible to compile, test and synthesize the full accelerator architecture and a *MAC framework* where the same operations can be applied to the MAC unit specifically.

Chapter 5 is related to the *Software Framework*. A description of the accelerator in SystemC language is provided for faster and larger simulations. This chapter contains information starting from the set-up of the virtual environment needed up to inferences with custom MAC architectures with the different multiplicators units.

Chapter 6 describes the architectures of each multiplier both in software and in hardware and how to correctly interface them with the accelerator.

Chapter 7 describes the results obtained in terms of the accuracy, area, power and speed of each architecture.

Chapter 8 summarizes the most general results of the thesis and raises possible future work that could be done, based on the work done in this document.

Chapter 9 contains the links to all the files and platforms in which you have worked.

Chapter 2

Background

2.1 Deep Learning

Artificial Intelligence is defined as the ability of a computer or electronic device to perceive its environment and takes actions that maximize its chance of success at some goal. It is related with the ability of computers to reason as humans.

Machine Learning is one of the moderns approach of artificial intelligence. Considered as a subfield of Machine learning, Deep Learning has algorithms inspired in the structure and behavior of human brain. The relation between these concepts is shown in figure 2.1.

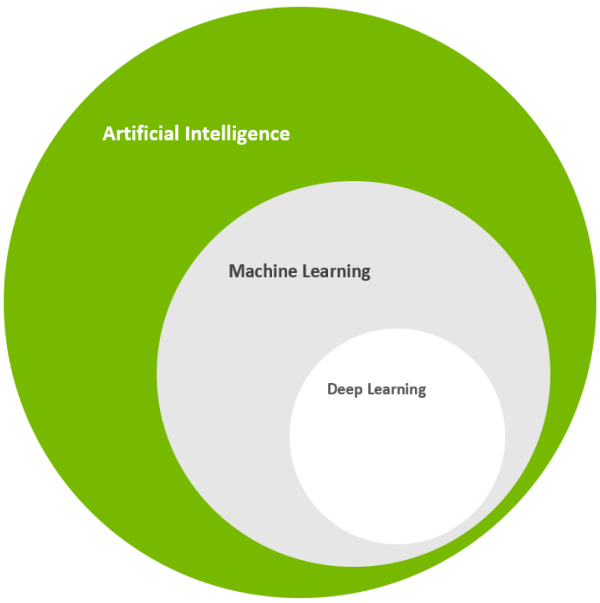


Figure 2.1: Relation between AI, ML and DL

With Machine Learning algorithms it is possible to identify patterns in images. The result of identifying pattern is for instance, predict if a certain object is a cat or a dog, if a person is crossing or not a street in case of an autonomous cars or block the access to a person to a certain area using face recognition.

ML techniques need a set of images to extract it's features. Feature extraction is a programmer/human task. After the feature extraction, the model start being intelligent to new incoming images at input, the model start predicting them.

Otherwise, in Deep Learning, features are extracted directly from the input data using backpropagation techniques. In the training step of the network, using backpropagation technique, it is possible to adjust weights of each kernel via it's gradients.

In Deep Learning, the feature extraction is not a human task, there are generated by the network by itself.

Deep Learning approach has greater performances rather than Machine Learning when the amount of training data is large, the accuracy of the network improves. Another advantage is that the human work of setting the features is replaced by an own adjustment of weights due backpropagation. The trade-off is the high computational cost. Thus, the disadvantage of Deep Learning is the large amount of computations that are needed to train a network.

2.2 Computer Vision

Computers can easily solve mathematical problems such as equations, task that for humans sometimes results difficult. Otherwise, for a human, it is simple to recognize if an object is a cat or a bird, humans are trained from early ages to put labels to objects to differentiate them.

Computer Vision is the science of extracting information of images, giving the ability to computers of differentiate objects, recognize patters. Is the field dedicated to give the computers the ability of processing, analyzing, and understanding images as the human being. Nowadays is used in autonomous cars, face and gesture recognition, image search, machine vision, character recognition, remote sensing and robots, agriculture, augmented reality, biometrics, forensics, industrial quality inspection, geoscience, medical image analysis, pollution monitoring, process control, security and surveillance, transport.

2.3 Convolutional Neural Networks

2.3.1 Definition

Convolutional Neural Networks (CNN or ConvNet) are related with human neurons due the structure and behavior. They are mostly used in computer vision for image recognition due their ability to detect patterns. A CNN is composed of a set of layers connected one with each other in a sequential way: the output of a layer is the input of the following one and so on. Main layers are: convolutional, pooling, activation, normalization, fully connected and softmax and they are combined in different ways, depending of the network. There are several architectures used for different tasks. Some recognized networks architectures are: LeNet, AlexNet, VGG, GoogleNet, ResNet (later, some architectures are explained in details).

Inputs of CNNs are images. The image passes through several filters, preserving the spatial structure, until a fully connected layer where a prediction is generated. For example, using a LeNet architecture, pretrained with a MNIST dataset, this network receives a handwritten number as input (from 0 to 9) and the output of the network is a prediction about which number could be. Figure 2.2 shows how the input image proceeds through different layers and how the spatial structure is preserved until the fully-connected layer.

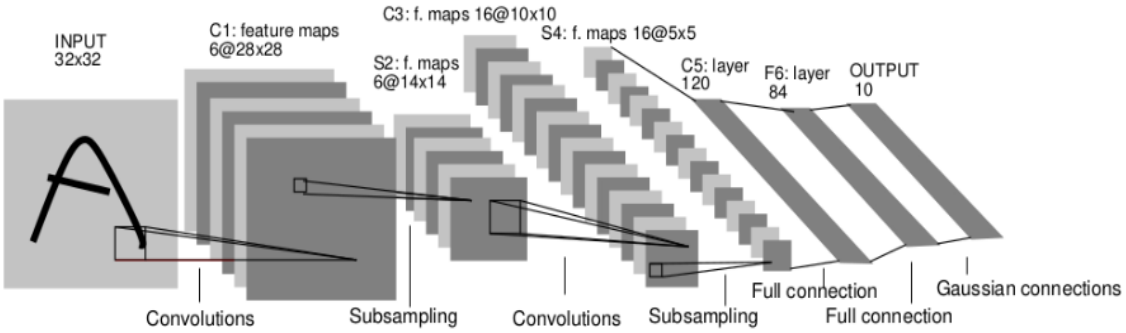


Figure 2.2: Spatial domain preservation of an input image. Extracted from [8].

Firsts convolutional layers of a CNN are usually low level filters. They are capable to detect borders, edges, lines, curves of input images. In a general way, they can detect simple images compositions. In the last layers of a CNN, the featured image pass thought high level filters that are capable to detect more complex compositions like eyes, bodies, beaks and more (in case that the network is trained to detect animals). Figure 2.3 shows the difference between filters used in different layers of a CNN. How filters values are generated? The answer is backpropagation. After being

initialized, at training step, the input image prediction result is compared with the label of the image and gradients are corrected with backpropagation techniques.

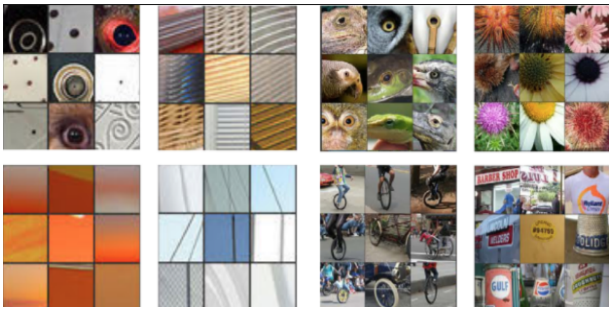


Figure 2.3: Low, Mid and High level filters. Source: *https://devblogs.nvidia.com/deep-learning-computer-vision-caffe-cudnn/*

2.3.2 Layers

Convolutional layer

This layer is composed of a set of filters (also called kernels). An input feature map is convolved by each filter and an output feature map is generated. Given an input image, suppose a CNN with a convolutional layer composed with 16 filters. There will be 16 outputs images.

Which is the operation between the input image and the filter? How the output image is generated? suppose an input image of 5x5 pixels where the pixel is represented as a numerical value from 0 to 256 (8-bits). The same for kernels (filter) but consider them of size 3x3. To understand the convolution operation, first is defined the filter operation. A filtering operation is a multiplication and accumulation between pixels of input and weight. Figure 2.4 shows a filter operation between them.

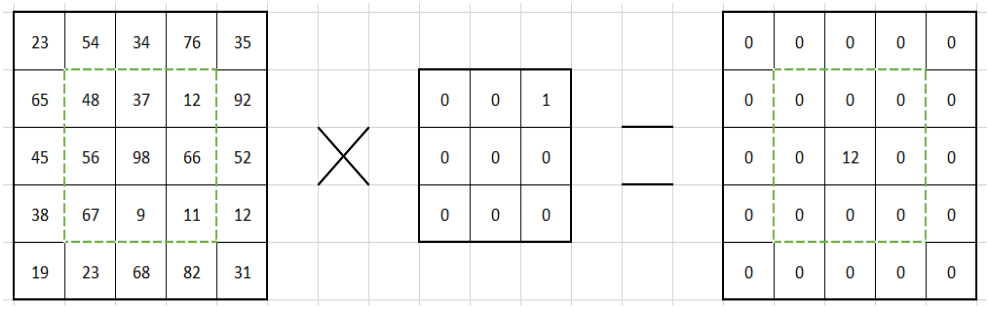


Figure 2.4: Filtering operation

Defined the filtering operation, a convolution operation occurs when passing the filter through the whole image. The result is an output image with the following size:

$$OutWidth = \frac{imgWidth - filWidth}{stride} + 1 \tag{2.1}$$

Activation layer

After a convolutional layer, there's an activation layer (usually a ReLU) where all negatives numbers change to zero. The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the convolutional layers. Is a way of data discretization and the prediction accuracy have better results. Figure 2.5 shows the ReLU mathematical function. For negative inputs the output goes to zero while if the input is positive the output is equal to it.

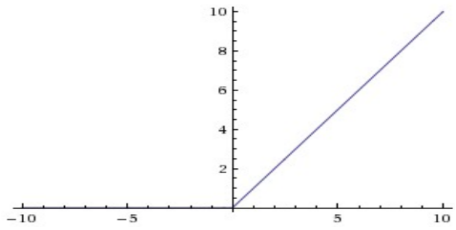


Figure 2.5: ReLU used in Activation Layers

Pooling layer

In pooling layer, the input feature suffers a shrink, a down-sampling. The most common pooling functions are: max, min, or average while max pooling is the most used technique. Figure 2.6 shows an example of using max-pooling resize to a 4x4 input image. From each colored block is extracted the max value so the image is reduced to 2x2.

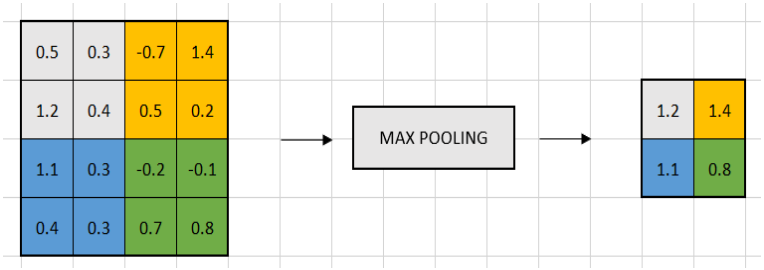


Figure 2.6: Max Pooling function

Once it is known that the image has a certain feature (while passing through a convolutional layer), its exact location is not important as its relative location. For this reason, down-sampling do not affect in performance and provides advantages such as less computation due the reduction of spatial dimension and control over-fitting. A symptom of overfitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.

Fully-connected layer

Takes as input the last feature map (after passing through all filters of CNN) and outputs a probability. It looks at the output of previous layer and determines which features most correlate to a class.

2.3.3 Architectures

The CNN architecture used is LeNet. The network is described in a *prototxt* file. The description of each layer type, connections and parameters are described in this file. A pretrained model is used in NVDLA. The file generated after the training step of the network is a *caffemodel* file and contains information about the weights (kernels) of each layer.

LeNet

This CNN combined with MNIST dataset is used to predict handwritten digits. According to [9] the error is 0.9%. Also, this network can be used also combined with CIFAR10 dataset for predicting animals and vehicles. LeNet has a precision of 76.27% when working with CIFAR10. [1] The network architecture layer by layer is:

$$Conv \rightarrow Pool \rightarrow Conv \rightarrow Pool \rightarrow InnerP \rightarrow ReLU \rightarrow InnerP \rightarrow FullyConnected$$

Detailed information about each layer can be found in appendix .1.

[1] https://github.com/BIGBALLON/cifar-10-cnn

2.4 Multipliers

NVDLA Multiply and Accumulate unit contains standard multipliers, without inaccuracies. In Verilog, an accurate multiplier is described as a booth multiplier or basically with a "*" between two inputs.

The objective is replacing it with inaccurate multipliers without losing prediction accuracy. Inaccurate multipliers generate multiplications introducing a certain error that should not be significant if the final prediction result still correct.

2.4.1 Approximate

According to [6] the core of this inaccurate multiplier is a 2x2 multiplier with a reduced architecture. Instead of throwing a 4-bit results, last bit output is not considered 0. Figure 2.7 shows the difference at gate-level between an accurate (standard) multiplier and this inaccurate multiplier version.

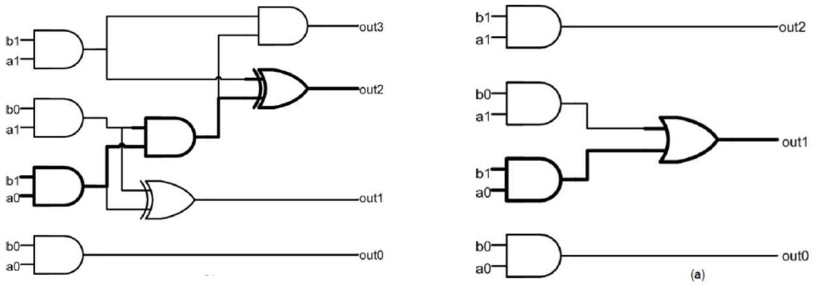


Figure 2.7: Standard multiplier (left), Approximate multiplier (right). Extracted from [6].

Mathematically, the 2x2 inaccurate multiplier is defined with the following equations:

$$out0 = a0 \wedge b0 \tag{2.2}$$

$$out1 = (a0 \wedge b1) \vee (a1 \wedge b0) \tag{2.3}$$

$$out2 = a1 \wedge b1 \tag{2.4}$$

$$out3 = 0 \tag{2.5}$$

Largers multipliers such as 4-bits or 8-bits inputs, can be generated combining them. Figure 2.8 shows an example of a 4x4 input multiplier generated with the 2x2 as base.

Each 2x2 multiplier produces partial products for then being added together.

This architecture have several benefits such as an average power savings of 31.78% to 45.4% with an average error of 1.39% to 3.32%. The max error magnitude is 22.22% and its remain constant for larger multiplier architectures.

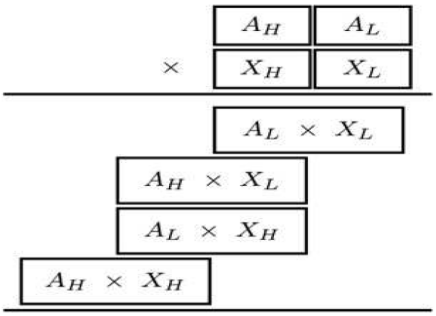


Figure 2.8: Larger multipliers from smaller ones. Extracted from [6]

2.4.2 Inferential

Machine learning techniques mimic the human brain in a certain way, which works as a statical inference engine, solving problems based on past experiences. The author explains in [7] as an example, how a person usually multiply a number by 100. Instead of doing an arithmetic operation, the human simply adds two zeros to the other number when multiplied. Thus, this is the basis of the inferential multiplier: the traditional logical function of the multiplier is replaced by an inferential way of solving the problem.

For that, an initial RTL model, before the synthesis is trained for extracting a classification tree which is described again in a RTL model (for then being synthesized). This is called *ML-driven synthesis* and can be applied in the design stage of other logic circuits.

Resuming, the inferential multiplier is a tree-based multiplier that works like an inference engine. Comparing it with a classical radix-4 multiplier, it has average accuracy of 76%, with a reduction of area of 22% and 2x latency reduction.

2.5 Ristretto

It is a tool that has been used in a first step, to generate a quantized pre-trained model of a CNN. The compilation of the same was not possible due to the limitations of the compiler of the accelerator (software framework compiler). It is a compiler that is still in process and many commands still do not recognize them. Beyond that, it is a tool that allows to check whether it is feasible or not to reduce the data-type since it makes an analysis of the accuracy of the quantization of the model.

To install Ristretto, is recommended to follow this tutorial: Ristretto installation.

Chapter 3

NVDLA

3.1 Introduction

Deep learning inferences are being more accurate than machine learning ones when the training dataset is large [1]. The problem of Deep Convolutional Networks is that the number of computations increase drastically due the required large amount of data for training.

NVDLA Accelerator is an open source hardware architecture to address these computational demands, solving commons inference operations as convolution, activation, pooling, and normalization at higher performances. Other advantages of NVDLA are:

- Provides a scalable version for FPGA.
- Highly configurable.

The high configurability allow the programmer to generate it own version of the accelerator only with the required blocks or add custom blocks into the entire architecture. NVIDIA describes this accelerator in two ways:

- Hardware description: in Verilog, RTL form useful for synthesis, single-layer simulations and accurate power calculations.
- Software description: in SystemC for inference simulations and complete CNN simulations

3.2 Architecture

The aim of the accelerator is to solve CNN inferences faster, accelerating their main operations: convolution, activation, pooling, and normalization. For each one it has an specific module:

- Convolution core: for convolution operations.
- Single Data Point operations (SDP): for activation operations.
- Planar Data operations (PDP): for pooling operations.
- Multi-Plane operations (CDP): for normalization.
- Reshape and Bridge DMA: for memory and reshape operations.

Figure 3.1 shows the complete NVDLA core structure:

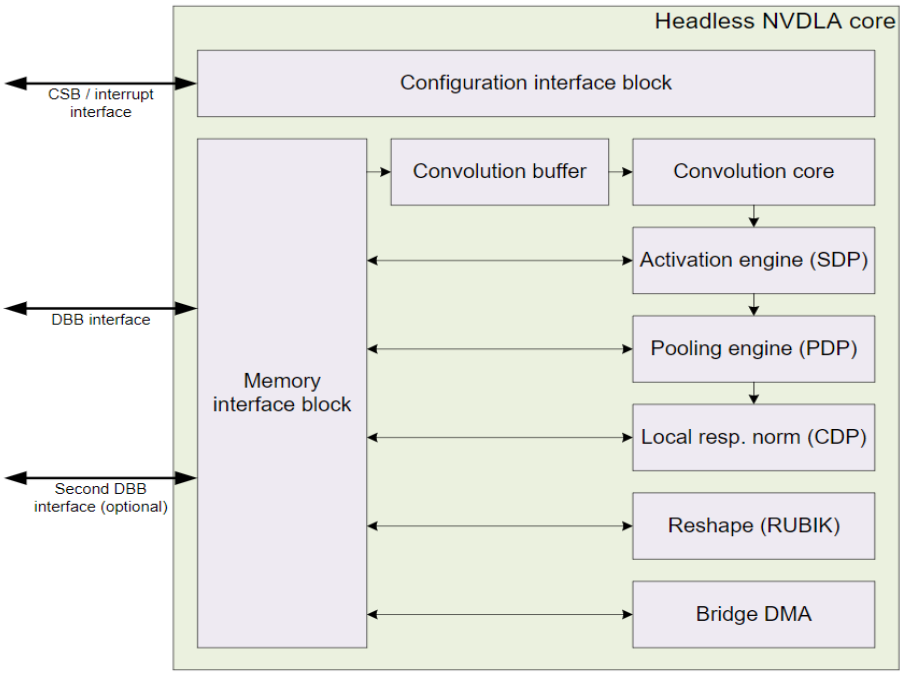


Figure 3.1: NVDLA core. Source: *https://github.com/nvdla/doc.*

Blocks inside the core work independently one with each other. If a system does not need a certain block, it can be removed entirely without affecting the rest. The independence between blocks generate two ways of work with the core:

- Independent, an external scheduler starts/ends different modules without relating its operations and data.
- Fused, pipelined activities and data connection between blocks.

For instance, if NVDLA is working in independent mode within a convolutional layer, this operation ends with a writting in memory. Otherwise, if NVDLA is working in fused mode, the output data of the convolution module moves towards the SDP block, for activation operations. Between blocks, the core provides small FIFOs to pass data between them when working in this mode.

Architecturally is built in as a pipeline. The full pipeline is as follows:

- CDMA: convolution DMA.
- CBUF: convolution buffer.
- CSC: convolution sequence controller.
- CMAC: convolution MAC array.
- CACC: convolution accumulator.
- SDP: single data processor.
- SDP_RDMA: single data processor, read DMA.
- PDP: planar data processor.
- PDP_RDMA: planar data processor, read DMA.
- CDP: channel data processor.
- CDP_RDMA: channel data processor, read DMA.
- BDMA: bridge DMA.
- RUBIK: reshape engine.

3.2.1 Convolution Pipeline

The first five stages of the convolution core belong to the "convolution pipeline". Figure 3.2 shows its structure.

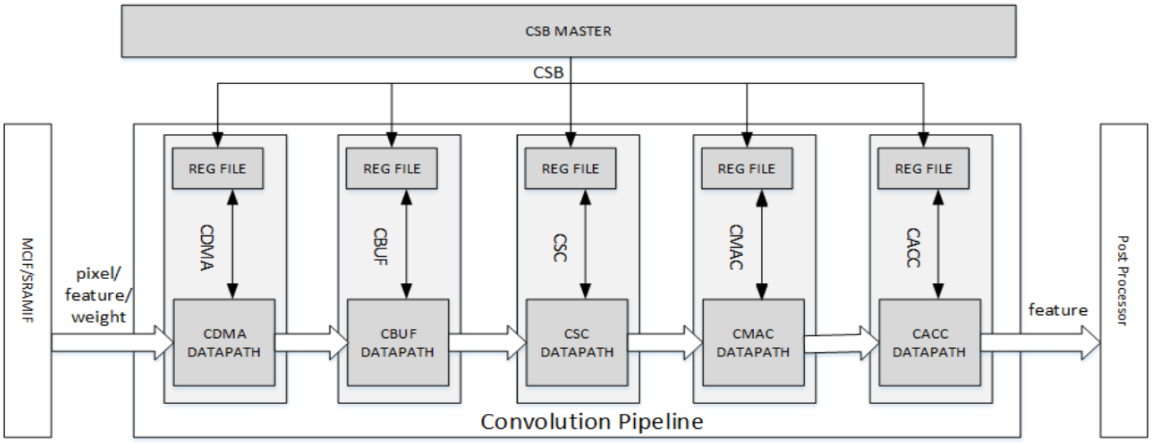


Figure 3.2: Convolution Pipeline. Source: https://github.com/nvdla/doc.

The inputs of the convolution pipeline are: the data extracted from the image (feature map) and the weights of the kernel related to the convolutional layer which is being processed. First convolutional layer receives the image as input and then the feature map are always being modified by each layer while weights remains constant since it's a pretrained model (data read from .caffemodel file).

It can be configured in a standard convolution mode (called direct convolution) or in optimized modes such as image-input (used at the 1st convolution layer), Winograd convolution and Batching option(optimized techniques). Other optimization feature is called sparse weight compression and optimizes fetching operations of weights from memory.

Atomic-C and Atomic-K are the parameters used to determine the number of MAC units. Specifically, the multiplication between them: *Atomic-C * Atomic-K*. For example, setting Atomic-C=16 and Atomic-k=64 generates a hardware architecture with 1024 MAC units. It is fundamental to consider the neural network structure and convolutional layer parameters. Suppose an input feature data channel equal to 8 and a weights data channel equal to 16. Thus, 128 MAC are required. If *Atomics* are set to 16 and 64, only 128 of 1024 MACs would be used, achieving a performance of 12.5% . All previous configurations are loaded before each layer execution.

CSB master does not belong to the convolution pipeline, it is not a pipeline stage. It configures each NVDLA sub-units registers using a ping pong synchronization technique (explained in detail in interfaces and interconnection section). The stages

of the convolution pipeline are:

- **CDMA**: Between the memory interface and the convolution core there's a buffer (RAM) reserved for weights and input storage. It s optimal to avoid repeated accesses to system memory. This unit is dedicated to fetch data from the internal SRAM (or external DRAM) and store it in the buffer. This operation usually is accompanied by a conversion operation of input feature data depending the convolution mode (direct, Winograd, image-input). the information about the conversion is sent to the sequence controller (CSC). Figure 1 shows the architecture of this block.

- **CBUF**: Is a 512Kb SRAM memory composed by 16 32KB banks of two 512-bit-wide and 256-entry two-port SRAMs. Data is pre-allocated in this block before the convolution operation. Figure 2 shows the architecture of this block.

- **CSC**: CDMA works as an interface between the SRAM memory and the convolution buffer. The *convolution sequence controller* has the same purpose between the buffer and the CMAC. It's composed by two data loaders, one for feature map and other for weights respectively, and they are configured according the convolution mode, by the signal coming from CDMA stage. Figure 3 shows the architecture of this block.

- **CMAC**: It is composed by a configurable number of MAC cells for parallel multiply-and-accumulate calculations. The input feature map is the same for all cells, varying the kernel weights. In other words, each MAC cell operates with a particular kernel generating its partial sums.

  Internally, each MAC cell generates a parallel multiply-and-accumulate calculation due the variable number of multiplicator units working in parallel. All the multiplications are added together and the output of the MAC is finally the partial sum between the feature data and the weight.Figure 4 shows the architecture of this block.

- **CACC**: In the last pipeline stage, partial sums are accumulated by the *convolution accumulator*. After the accumulation the result is rounded/saturated. When working at INT8 the accumulative sum is stored in 34 bits and truncated to 32. For INT16 the accumulative sum is stored in 48 bits and truncated to 32. Otherwise at FP16 the accumulative sum is stored in 44 bits (8-bits exponent, 38-bits signed decimal) and truncated to FP32. Figure 5 shows the architecture of this block.

3.2.2 Post-Convolution Pipeline

- **SDP**: core dedicated to post-processing operations. The allowed operations are: bias addition, non-linear functions and batch normalization. All of them are widely used after a convolution layer. The *bias addition* operation traduced with the following mathematical formula: $y = x + bias$. *Non-linear* allowed functions are: ReLU, Sigmoid and Hyperbolic and they are used to accomplish activation layers.

  Besides the post process operation, SDP also performs format conversion. After a convolution layer, output data has higher data-types than the commons data-types of NVDLA pipeline (FP16, INT16, INT8) thus a conversion is necessary.

- **PDP**: core dedicated to accomplish pooling functions. NVDLA support max, min, and average pooling. Neighboring input elements within a plane are sent to a non-linear function to compute one output element. Figure 2.6 shows a graphical example of a max-pooling function.

- **CDP**:Core dedicated to local response normalization.

- **RUBIK**:Is the data reshape core. Splitting, slicing, and merging of data are some of the functions that this core is capable to do.

- **BDMA**: images and processed results are stored in a external DRAM memory. Bandwidth and latency of this memory are insufficient to fully utilize NVDLA. Bridge DMA is the core dedicated to move data between memories.

3.3 Interfaces & Connections

Figure 3.3 shows the NVDLA core, the external blocks and its interfaces and connections.

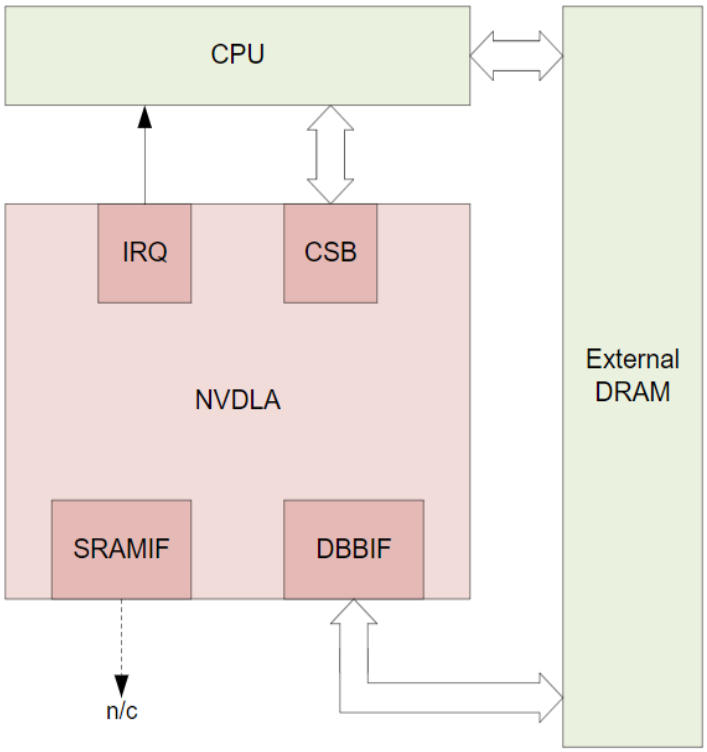


Figure 3.3: Interfaces and Connections of NVDLA. Source: https://github.com/nvdla/doc.

The CPU oversees scheduling operations; sends signals to certain blocks (for instance convolutional or SDP) to start the execution. When the block ends it's respond with an interrupt signal to the CPU. The process repeats until the inference of the entire network in complete. The interfaces between the accelerator and the CPU are: Configuration Space Bus (CSB) and interrupt interface (IRQ).

3.3.1 CSB interface

Host systems configure NVDLA registers via this interface. It uses a "ping-pong" Synchronization mechanism.

"ping-pong" Synchronization mechanism: Every subunit of the accelerator has its own registers (status, configuration and more). For instance, a CNN composed with a convolutional layer followed with an activation layer. The workflow, at register configuration level is the following:

1. CPU program NVDLA via CSB the convolutional unit.
2. Execution.
3. Finish execution, IRQ signal from NVDLA to CPU.
4. CPU program NVDLA via CSB the activation unit (SDP).
5. Execution.

Every sub-unit of NVDLA has a set of configuration and control registers **duplicated**. This *duplicated-register architecture* is the key of the mechanism. When a subunit is executing using the configuration of the first set of registers, the CPU is allowed to program the second set. So, CPU reprogramming latency do not affect to NVDLA performance.

3.3.2 Interrupt interface

NVDLA hardware includes a 1-bit level-driven interrupt. The interrupt line is asserted when a task has been completed or when an error occurs.

3.3.3 System Data Interfaces

The accelerator accesses to external data through two data interfaces: DBBIF and SRAMIF. DBBIF is the interface with the external memory while SRAMIF is the interface with an optional external SRAM memory. Both uses AXI4 interface protocol and has a configurable data bus from 32 up to 512 bits.

3.4 Implementations

Depending on the application, the accelerator can be configured in different ways. NVIDIA has released two versions: a large one based on performance and a small based on savings.

Large accelerator, also called full or headed, has primary emphasis on high performance and versatility, it can resolve many tasks at once and include an optional second memory interface.

Small one has good fit for cost-sensitive connected Internet of Things (IoT) devices, AI and automation-oriented systems that have well-defined tasks for which cost, area, and power are the primary drivers. It can handle only one task at a time, and as such, sacrificing system performance while NVDLA is operating is generally not a strong concern. This generates inexpensive context switches so there's no need of an additional microcontroller (headless). This version does not include the optional second memory interface.

3.5 Configurability

Setting different parameters, will conclude in different versions of the NVDLA RTL and synthesis description with differences in performance, area, power, etc. Figure 3.4 shows different configuration examples:

- Data type supporting = INT8
- Feature supporting - Winograd = No
- Feature supporting - Second Memory Bus = No
- Feature supporting - Compression = No
- Image input support = R8, A8B8G8R8, A8R8G8B R8G8B8X8, Y8___U8V8, Y8___V8U8
- SDP function support = Single Scaling
- BDMA function support = No
- Rubik function support = No
- Atomic - C sizing = 8
- Atomic - K sizing = 8
- SDP throughput = 1
- PDP throughput = 1
- CDP throughput = 1
- BUFF bank # = 32
- BUFF bank size = 4KB

- Data type supporting = FP16/INT16
- Feature supporting - Winograd = Yes
- Feature supporting - Second Memory Bus = Yes
- Feature supporting - Compression = Yes
- Image input support = A8R8G8B8/YUV16 Semi-planar
- SDP function support = Scaling/LUT
- BDMA function support = Yes
- Rubik function support = No
- Atomic - C sizing = 64
- Atomic - K sizing = 16
- SDP throughput = 16
- PDP throughput = 4
- CDP throughput = 4
- BUFF bank # = 16
- BUFF bank size = 32KB

Figure 3.4: Different configurations of the accelerator. Source: https://github.com/ nvdla/doc.

Each parameter affects a certain or a group of blocks of NVDLA. For example, setting the data type will affect the entire pipeline while Winograd feature only

affects to CMAC block. Most of the parameters have been mentioned in section 3.2.1.

Chapter 4

NVDLA hardware framework

4.1 Introduction

NVIDIA describes the accelerator in two different ways:

- Hardware description: using Verilog as HDL language, in a RTL form useful single-layer simulations and synthesis to obtain information of the module (or sub-modules) in terms of area, energy and speed.
- Software description: in SystemC for faster simulations. Instead of a single layer, a complete CNN can be simulated.

Moreover, the hardware framework is divided into two parts:

- NVDLA: different RTL versions of the accelerator are generated varying certain parameters, proceeding with single-layer simulations for testing. Finally, a synthesis of the models to obtain information about area, power and speed. Compilation and simulation build tools of NVDLA are in VCS (Synopsis tool), thus lot of modifications has been done to adapt it to Modelsim. The top unit in this environment is the accelerator.
- MAC: a separated environment for synthesis with custom scripts (not the NVDLA provided ones), pre and post-synthesis simulations of each MAC module and switching activity calculations are the main tasks of this section. The top unit in this environment is the MAC one.

4.2 NVDLA Framework

NVDLA Hardware framework files can be downloaded and used by the following link https://github.com/IgnacioGoldman

4.2.1 RTL build

NVIDIA provides a hardware description of the accelerator. The peculiarity is that files can not be directly compiled, they are not described entirely. Before the compilation, a build step is needed. Figure 4.1 shows an instance of a Verilog file of a NVDLA module that is not fully-described.

```
57  wire signed [CMAC_RESULT_WIDTH-1:0] sum_out;
58  wire [CMAC_ATOMC-1:0] op_out_pvld;
59  //: my $mul_result_width = 18;
60  //: my $bpe = CMAC_BPE;
61  //: my $rwidth = CMAC_RESULT_WIDTH;
62  //: my $result_width = $rwidth * CMAC_ATOMC * 2;
63  //: for (my $i=0; $i < CMAC_ATOMC; ++$i) {
64  //:     print "assign op_out_pvld[${i}] = wt_actv_pvld[${i}] & dat_actv_pvld[${i}] & wt_actv_nz${i} & dat_actv_nz${i};\n";
65  //:     print "wire signed [${mul_result_width}-1:0] mout_$i = (\$signed(wt_actv_data${i}) * \$signed(dat_actv_data${i})) & \$signed
66  //: }
67  //:
68  //: print "assign sum_out = \n";
69  //: for (my $i=0; $i < CMAC_ATOMC; ++$i) {
70  //:     print "    ";
71  //:     print "+ " if ($i != 0);
72  //:     print "mout_$i\n";
73  //: }
74  //: print "; \n";
75  `endif
```

Figure 4.1: Pre-built code example of accelerator module. Source: https://github.com/ nvdla/doc.

The final architecture of a module will depend on a certain amount of parameters such as *CMAC_ATOMC* that can be configured before the build. For instance, with Atomic-C and Atomic-K it is possible to determinate the number of MAC units of the accelerator thus, the RTL build files bases on these parameters to generate the final verilog with the amount of modules needed and the correct bit-widths.

Previous lines of codes belong to the MAC unit, but all the modules are described in this way. The commands for building an RTL model are the following:

1. *cd hardware_framework/NVDLA*

2. *make rtl*

The resulting architecture is saved at *hw/outdir*. The execution command has two variables: vmod and a *spec* file. *vmod* is the directory where all the Verilog files, plus libraries and memories of the accelerator are located while the spec file contains all the configuration parameters as number of MACs, widths, throughput of blocks, data-type, Winograd mode and other special features are enabled or disabled. This file is crucial for the final accelerator size and performance.

The most common errors in this step can be: in paths or with perl modules. To solve the first type is a matter of modifying the file tree.make which contains all paths to java, gcc, python, and more. Then, the missing perl modules have to be installed.

4.2.2 Compilation & Simulation

The customized RTL model generated in previous step is inserted inside the NVDLA hardware framework to be compiled, simulated, and synthesized it. For inserting the RTL model, the vmod generated directory should be copied into *container_NVDLA_RTL/NVDLA/outdir/nv_full*.

Then, for compilate and simulate the model, the default command is *make simulate*. For a custom compilation and simulation, some parameters should be configured:

- TESTBENCH=<name>. Provided testbenches given by NVDLA are sufficient to prove different MACs. They are located at */verif/traces/traceplayer/*. Some interesting's testbenches are the Googlenet and AlexNet convolution layers since they simulate originals CNN layers.

- WAVES=-c. Adding this parameter generates a console level simulation without showing waves which is faster.

- MAC=<nvdla, approx, inference>. Specify the mac unit to be used. MAC units are described in the following chapters.

- SYNTH=<0 or 1>. Set in 1 for a post-synthesis simulation.

NVDLA open source uses VCS (tool of Synopsys) for testbench verification. The simulation environment has been moved into Modelsim due the absence of this tool.

When running the command *make simulate*, the script automatically solves many incompatibilities migrating verilog files to systemVerilog. From line 4 to 7 at .3 there's an extraction of the script that shows the file adaption. The rest of the errors appears due other reasons. Many files have been modified at specific lines for correcting those errors thus in a second stage of the script, original files of the RTL containing errors, are replaced with the same ones, but corrected. All correct files

can be located at https://github.com/IgnacioGoldman/. From lines 9 to 11 at .3 there's an extraction of the script that shows the files replacing.

After Modelsim adaptation step, the NVDLA should compile without errors, so the compilation and simulation can be started. Before that, inputs files are formatted when running the *inp_txn_to_hexdump.pl* script.

For every simulation there are related three inputs files. The script inp_txn_to_hexdump.pl converts those files into a readable format for the accelerator. First two files are called: *image.dat* and *weight.dat* and they contain the image and kernels data. There's a third one called *input.txt*, containing all the configurations of NVDLA.

Sequentially, CSB master reads the *input.txt* that starts with two load operations (weights and data are loaded into the memory). The rest of the operations are related to the configuration of the accelerator registers. The following link directs to the files related to the single-layer testbench of a Googlenet https://github.com/IgnacioGoldman/.

After a compilation and simulation of a CNN layer in the NVDLA hardware framework, a transcript file is generated. This file contains information about: write/read of NVDLA registers, write/read of NVDLA internal and external memories, inputs and outputs of every Multiply and Accumulate unit are displayed, inputs and outputs of every multiplicator unit are displayed.

This information is collected with "$display" operations. At MAC level, every time the output changes it values show the it's values also with the inputs that are entering to the module at this moment. MAC unit has a pipeline of 3 stages so for being clear, those inputs are not directly related with the output. In other words, the inputs generate an output after three rising edge clocks. After the transcript is generated, there are several scripts to calculate the relative error of MAC operation and multiplication operation. Those errors are used to ensure that every MAC is working as they are expected to work. .4

4.2.3 Testing

Figure 4.2 shows the main accelerator connections with: external memory (violet) and CSB master (orange) and some inputs to MAC units (green). Blue signals are related to the accelerator clock.

With the figure, it can be deduced the flow of operations that occur when running a testbench. In particular, it is Googlenet's single-layer testbench.

Observe that the process is as follows:

1. CBS configures the registers corresponding to each stage of the pipeline (*csb2nvdla_wdat*).

2. The accelerator begins to bring data from the memory (*nvdla_core2cvsram_r_rdata*).

3. The accelerator operates with the data (green signals).

4. The output feature data is saved again in memory (*nvdla_core2cvsram_w_wdata*).

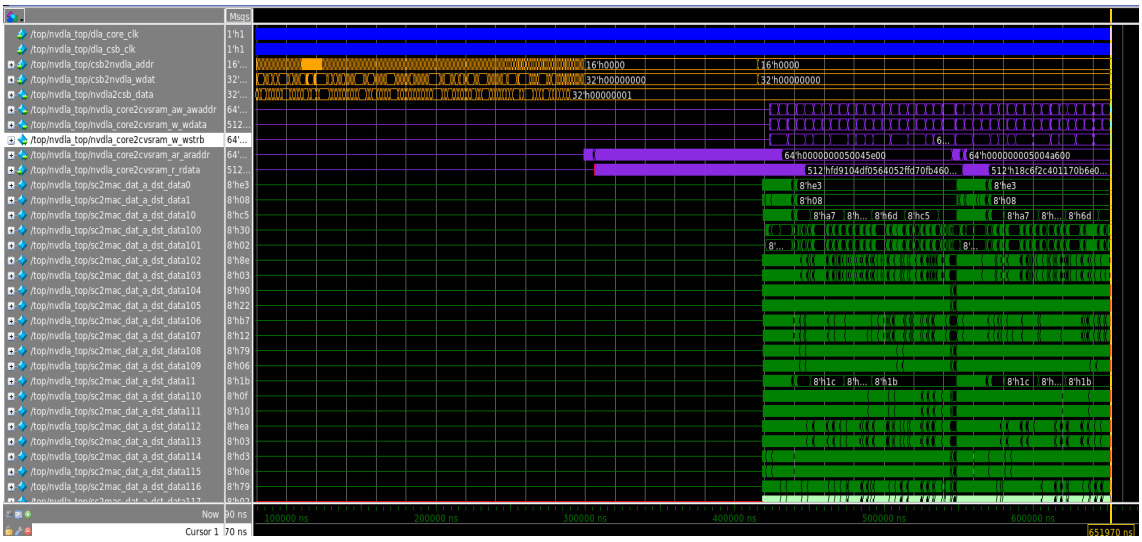


Figure 4.2: GoogleNet testbench waves

4.2.4 Synthesis

For running a synthesis the command is *make synth*. Errors appears, specifically in file *config.sh* that contains all the configurations for the synthesis. The following modifications has been applied to this file:

- Changes in erroneous tools paths.
- Export TOP_NAMES=<module name>. For synthesizing only the mac unit: NV_NVDLA_CMAC_CORE_mac. Any module can be synthesized separately, indeed it is recommended to synthesize the accelerator partitions separately.
- Export TARGET_LIB and LINK_LIB with own libraries CMOS045_SC_14_CORE_LS_nom_1.10V_25C.dc.

4.3 MAC Framework

The MAC module has been separated from the whole NVDLA architecture for fast simulations, accurate synthesis results using own scripts and switching activity calculation of the post-synthesis circuit. In the following paragraphs the functionalities that this environment allows are explained.

4.3.1 Input data for MAC testbench

When a layer simulation occurs (in previous framework) the NVDLA read the input data from weights and feature files and operates with them until the MAC cells. Running testbenches of real layers in the complete NVDLA architecture allows to generate a file with the correct data-flow inside NVDLA MAC cells.

In other words, after running each single-layer testbench the following files have been obtained: *fc_layer.txt*, *googlenet_layer.txt*, *alexnet_layer.txt*. They contain the exact data-flow of a real CNN, but only for the MAC unit testbenches. Thus, running the command:

- Make testbench TESTBENCH=<value> BITS=<value>

The script *format_tb_MAC.pl* format the data into two files called *data_file.txt* and *weight_file.txt* for being used in this environment.

4.3.2 Compilation & Simulation

The command for compile is:

- make compile UNIT=<mac_architecture>

After modifying the MAC architecture ensure that there's not any error in compilation step with previous command.

Once testbench has been generated and mac architecture compiles without errors, a simulation can be executed. The command is:

- make simulate TESTBENCH=<testbench> BITS=<bits> UNIT=<mac>

For instance, running *make simulate TESTBENCH=<googlenet> BITS=<8> UNIT=<approx_int8>* means using and configuring the accelerator with a GoogleNet single-layer testbench and the approximate 8-bits MAC unit.

How the results of a testbench are checked? How is it verified that the MAC is working as it should? A transcript file is generated after the simulation containing information about inputs and outputs of the MAC. Thus, a script called *results_generator.pl* generate two files: *exact_result.txt* and *MAC_result.txt* containing as indicated by their names, the exact MAC results and the approximate

MAC result generated by the accelerator (or also exact if an accurate multiplier is used).

Previous generated files are the inputs of a script called *compare.pl* which compare the relative error of the exact MAC with the approximate one. Both scripts codes can be found at .5 and .6 with detailed explanations.

4.3.3 Synthesis

For synthesizing a certain MAC unit, the procedure is the following:

1. Set the MAC unit to be synthesized in file *logicSynthesis/core_settings.tcl* modifying the 3rd line.
2. Set correctly the source file in file *logicSynthesis/synthesis.tcl*
3. run *make synthesis*

All the results of the synthesis including reports and the netlist are saved in *saves* directory.

4.3.4 Switching Activity

For an accurate power consumption calculation, a set of scripts have been configured in the environment to calculate the switching activity of a certain MAC unit. The command for running a switching activity is *make switching_activity*.

Chapter 5

NVDLA software framework

5.1 Introduction

NVIDIA provides a software description of the accelerator (in SystemC language) useful for complete CNN inferences. While in the hardware description there are single layer testbenches, in this platform it is possible to run a full CNN network with an image as input and check the prediction results. As in hardware description, modules of software NVDLA are fully-modifiable.

5.2 Software Tools

The software associated with NVDLA is divided into two parts:

- **Compilation tools**: converts a CNN model (stored in a *.prototxt* file) in combination with its pre-trained weights (stored in a *.caffemodel* file) into a loadable file capable of running on NVDLA. It is smart enough to pro-actively recognize those neural network operations that are not suitable for the NVDLA implementation and report back to the application.
- **Runtime environment**: run-time software to load and execute neural networks into NVDLA. The loadable file generated in compilation step in combination with an input image are the inputs of the runtime environment, being the prediction the result.

The runtime environment must run inside a *Virtual Platform* while for compilation tools this is not necessary. Figure 5.1 shows the general flow of NVDLA software.

Before running the accelerator software, User Mode Driver (UMD) and Kernel Mode Driver (KMD) kernels must be installed inside the virtual platform. UMD

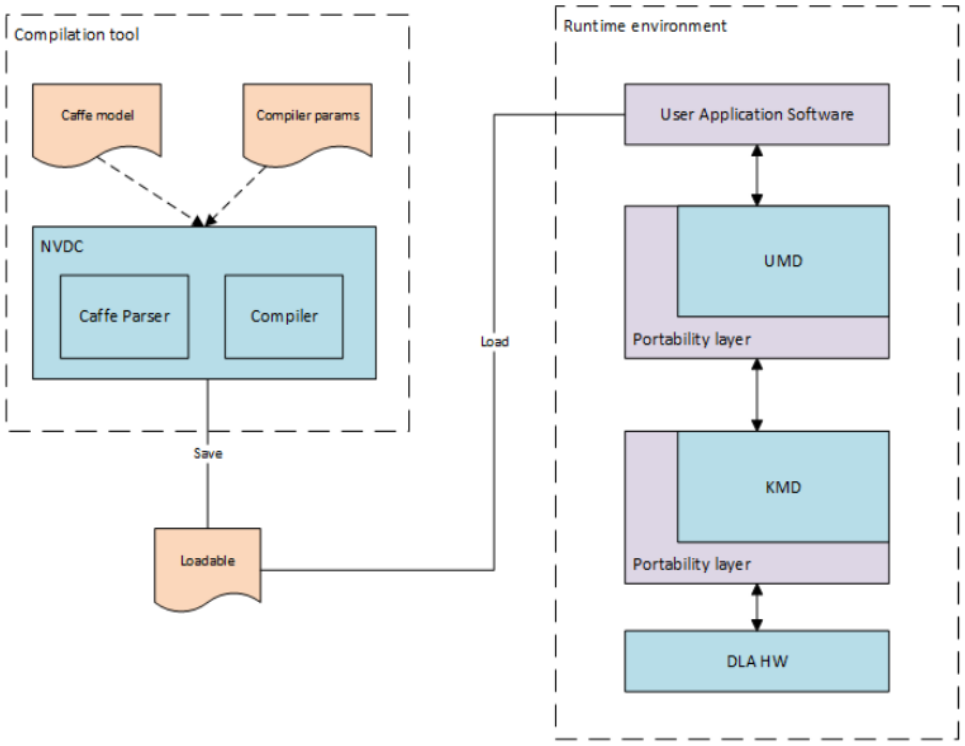


Figure 5.1: NVDLA Software tools. Source: https://github.com/nvdla/doc.

provides standard Application Programming Interface (API) for processing loadable images, binding input and output tensors to memory locations, and submitting inference jobs to KMD. KMD is main entry point; receives an inference job in memory, selects from multiple available jobs for execution (if on a multi-process system) and submits it to the core engine scheduler.

NVDLA provides examples of these kernels at */prebuilt/linux*. Kernel files are *opendla.ko* and *drm.ko* and for testing, these kernels are sufficient, but it is possible to compile and generate custom kernels. Using custom kernels inferences it is possible to arrive to higher accuracies in predictions due bugs corrections and modify them freely for showing necessary information such as the registers writting and reading or score of each class when predicting.

5.3 Virtual Platform

The virtual platform is based on GreenSoCs QBOX, co-simulator with QEMU and SystemC. QEMU is a free and open-source hosted hypervisor that performs hardware virtualization. In this case, QEMU is an ARMv8 emulator. Figure 5.2 shows how the software version of the accelerator is embedded inside the virtual platform.

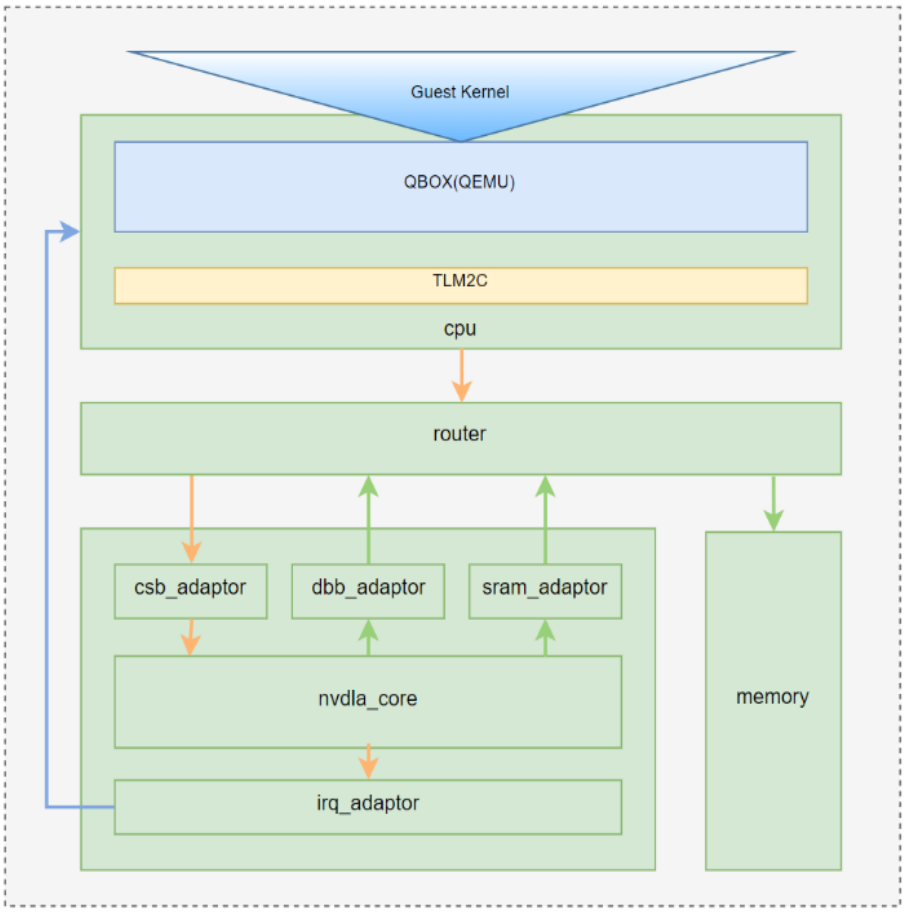


Figure 5.2: NVDLA Virtual Platform. Source: https://github.com/nvdla/doc.

5.4 Configuration

In the following section is described the whole installation process of the virtual platform and the accelerator software inside of it. The process is divided into three different configurations: virtual platform configuration, buildroot configuration and NVDLA software insertion.

5.4.1 VP

1. Clone the VP repository and move into the directory.
2. Type the following command: *git submodule update –init –recursive*. Many modules will throw errors, probably all of them starting from the dtc module. To solve them type:
   (a) *gedit ∼/.gitconfig*
   (b) Configure the file such as .7.
3. Install the required tools and libraries with the following command: *sudo apt-get install g++ cmake libboost-dev python-dev libglib2.0-dev libpixman-1-dev liblua5.2-dev swig libcap-dev libattr1-dev*
4. SystemC 2.3.0 version is required (fails with 2.3.1 and 2.3.2). Follow the next tutorial for the installation [1]
5. Clone the NVDLA hardware inside the virtual platform
6. Move into the hw directory and run *make* to set up the environment paths.
7. Run *tools/bin/tmake -build cmod_top* to build a generic SystemC version of the accelerator.
8. Move again into the virtual platform directory and type *cmake -DCMAKE_INSTALL_PREFIX=build -DSYSTEMC_PREFIX= /usr/local/systemc-2.3.0/ -DNVDLA_HW_PREFIX=/home/ignacio/vp/hw -DNVDLA_HW_PROJECT=nv_full* for cmake build under the vp repository directory.
9. git submodule update –init –recursive
10. make

[1] http://rkrara.blogspot.it/2012/12/install-systemc-230-on-linux.html

11. make install

At this point, the virtual platform should be correctly installed with a standard version of the accelerator.

5.4.2 Buildroot

Buildroot is a tool used to generate embedded Linux systems through cross compilation. It is composed by a set of scripts that make easy the process of building a bootable Linux. It can be downloaded from *https://buildroot.org/download.html*. Once downloaded, the following steps should be done:

1. Decompress it and move into the buildroot directory.
2. Run the following command: *make qemu_aarch64_virt_defconfig*
3. Then, *make menuconfig*

5.4.3 Software Insertion

Clone the software repository inside the virtual platform directory. Then, compile kernel modules:

1. **UMD compilation**
   (a) In UMD directory, export TOP=<path to umd>
   (b) Then, export TOOLCHAIN_PREFIX==<search path in buildroot>/bin/aarch64-linux-gnu-
   (c) make
2. **KMD compilation**, move into KMD directory and type make KDIR=<path to buildroot>/buildroot-2017.11-rc1/output/build/linux-4.13.3/ ARCH=arm64 CROSS_COMPILE=<path to buildroot>/buildroot-2017.11-rc1/output/host/bin/aarch64-linux-gnu-
3. Move the generated kernels *opendla.ko* and *drm.ko* to the */prebuilt/linux* directory.

5.5 Testing

This part of the document is useful also as a tutorial to use the NVDLA software correctly. Up to here, all the required tools have been installed and configured. So, a LeNet with a set of MNIST images is used for testing. These files can be found at thesis repository in the following link IgnacioGoldman Github.

The process is divided into two parts:

- Loadable file generation.
- Running loadable in NVDLA.

5.5.1 Compilation

The convolutional neural network model plus the pre-trained weights file are the inputs to the compiler. The first one described as a *prototxt* file and the second one as a *caffemodel* file. This stage does not necessarily have to be executed within the virtual platform. The commands are:

- Change directory up to *sw/prebuilt/linux*.
- *./nvdla_compiler –prototxt <prototxt_file> –caffemodel <caffemodel_file>*

The resulting loadable file "default.nvdla" should be generated.

5.5.2 Runtime

SystemC model of the accelerator has been generated before when running *tools/bin/tmake -build cmod_top* at the hardware environment. If modifications were introduced in the architecture, it must be recompiled using the same command. Then, change directory to virtual platform and login into the kernel using the following command:

- *export SC_SIGNAL_WRITE_CHECK=DISABLE && ./build/bin/aarch64_toplevel -c conf/aarch64_nvdla.lua*. 'root' for the account and 'nvdla' for the password.

Once inside the virtual platform, software files are mounted and kernels should be compiled, all in one with the following command:

- mount -t 9p -o trans=virtio r /mnt && cd /mnt/sw/prebuilt/linux/ && export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/mnt/sw/prebuilt/linux/ && insmod drm.ko && insmod opendla.ko

To run a single inference the command is the following: *./nvdla_runtime –loadable default.nvdla –image ../../regression/images/testSample/img_1.jpg –rawdump.*

Inputs of the runtime command are the loadable and a random image. As the loadable used is a LeNet trained with handwritten digits, the input image is also a digit.Then, to run multiples inferences, a simple one-line script is done:

- n=0; while [[$n -lt 1]]; do ./nvdla_runtime –loadable default.nvdla –image ../../regression/images/testSample/img_$((n+1)).jpg –rawdump && cat output.dimg » result.txt && echo “” » results.txt; n=$((n+1)); done

An inference in this environment takes approximately 2 minutes and increases the duration while the neural network is more complex. An inference in programs of the type of tensorflow or caffe takes some milliseconds.

The following image shows how the environment shows the results after inference.

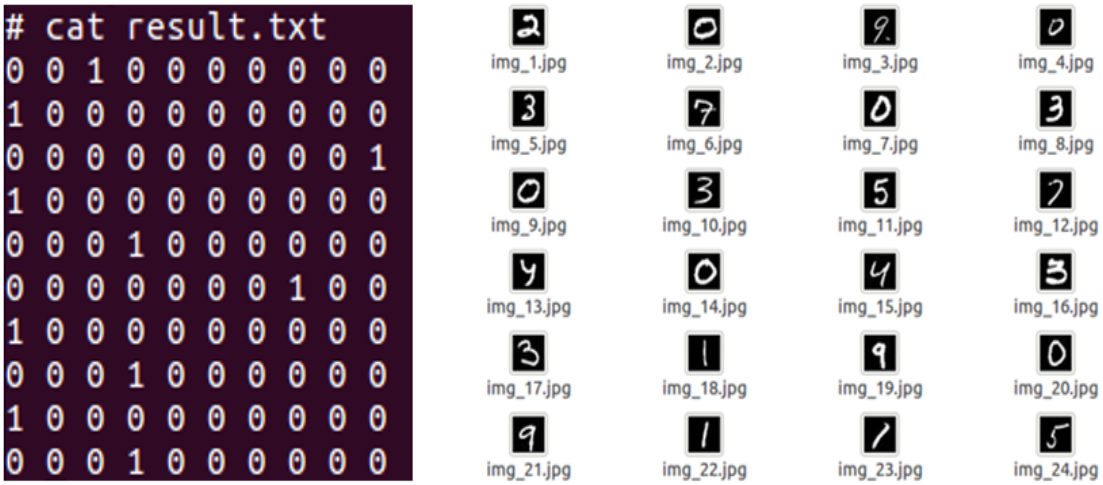


Figure 5.3: Prediction results

Each line represents the result of each inference and each columns is related to each class. For instance, the line *0 0 1 0 0 0 0 0 0* means that the prediction result is number two, while line *1 0 0 0 0 0 0 0 0* is number 0.

After running the runtime command two files are generated. *results.txt* shows the result of the predictions in a list form while *scores_results.txt* include the scores of each class when predicting. An example of both files after running a single inference can be seen in the following link Google Drive link.

To automate the process of checking results the predictions file have been compared with a golden file using a script. Furthermore, the scores of each prediction are useful to understand how each multiplier affects differently to each MAC operation,

and how the MAC operation affects to the final prediction. The values of the scores are inserted in an excel file "prediction_behavior.xlsx" which shows the behavior of each architecture.

Chapter 6

MAC architectures

6.1 NVDLA MAC unit

NVDLA CMAC stage of the convolution pipeline has been described in 3.2.1 section. In the following section is described the MAC cell internally to understand the behavior and correctly generate other architectures with different types of multipliers. Figure 6.1 shows the CMAC pipeline stage, containing multiples MAC cells. Many of the figures displayed in this chapter are a simplified version of them. For a full comprehension check the Verilog files at https://github.com/IgnacioGoldman.

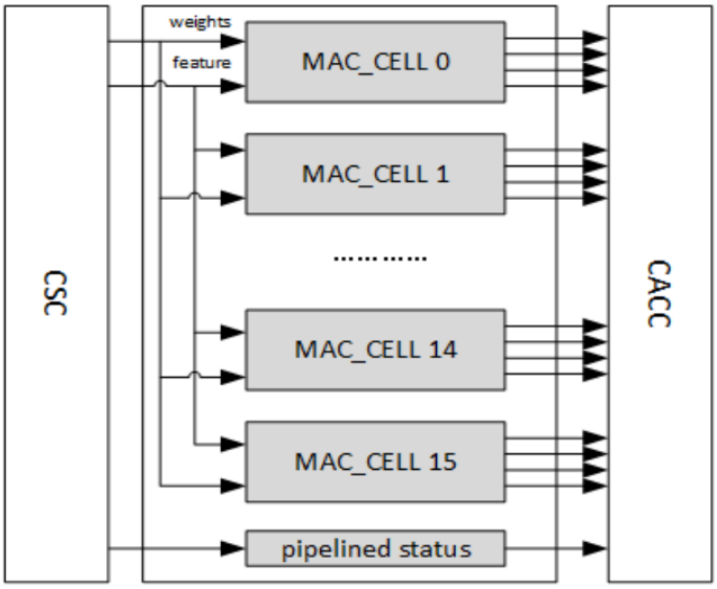


Figure 6.1: CMAC stage architecture

NVDLA still in continuous development, therefore, the architecture of the multiply-and-accumulate unit has undergone changes, improvements and optimizations during the course of the thesis.

6.1.1 MAC with Booth multipliers

It is the architecture of the *MAC cell* during the first release of the accelerator. The operation is explained in the following lines.

In stages prior to the convolution, the accelerator organizes the data in memory in a particular way. The image and weights are stored in memory in a private format called *cube format*, which consist in divide the full data-set into small cubes of 32-bytes data aligning them in weights, height and channel.

From the point of view of the CMAC stage, it is enough to understand that each clock cycle a single cube of image and kernels enter inside of the CMAC. In other words, in a clock cycle, a cube of the input image enters to every MAC cell and a cube of each kernel enter inside each MAC cell. Thus, each MAC cell works generating the partial results of each kernel.

The package (small cube of image and kernel) incoming to a MAC cell , is decomposed in each atomic FP16 data (INT16 or INT8, depending the data-type) and stored in a set of registers. Then, each pair of registers (respective to image and kernel) enter into set of booth multipliers working in parallel. The result of each multiplication are finally added together in a third stage with a tree adder. The figure 6.2 graphically describes the process.

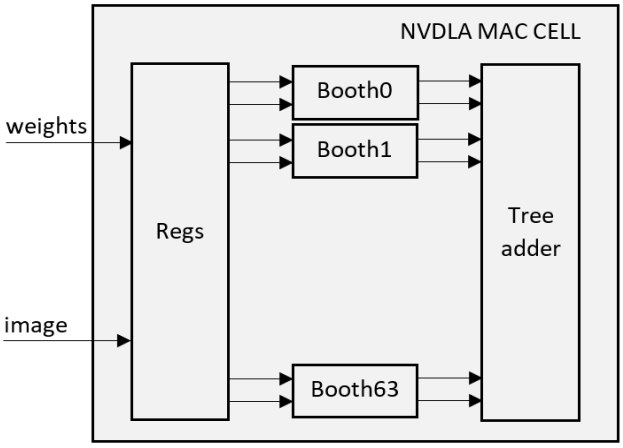


Figure 6.2: MAC cell architecture

Booth Multiplier

Optimized hardware that multiplies two signed binary numbers in two's complement notation using an algorithm different to the traditional way of solving multiplication by the human being. Takes as input both 16-bit weight and feature data and uses one as a *source data* and the other for *encoding*. Then, the source data is shifted accordingly the encoder [1] and all the results are added in a Design-Ware tree adder. A figure of the booth multiplier is shown at 6.3.

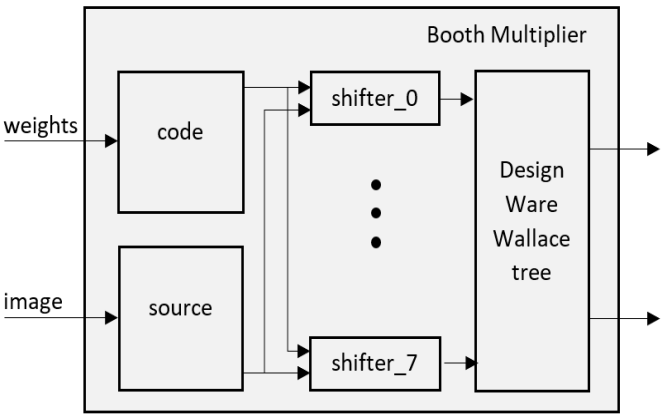


Figure 6.3: Booth Multiplier architecture

The multiplier booth has its inputs to 16-bits. So what happens if the accelerator is configured to work at 8-bits? Would 16-bit multipliers be used for operations at 8? The answer is yes, but the booth would solve two multiplications in a clock cycle, doubling the performance. In other words, the architecture of the MAC and the multipliers do not change, but the configuration makes the hardware to work in different ways and with different sets of data.

The output of each booth multiplier are the inputs of another DesignWare tree adder dedicated to sum the result of each multiplication 6.3. Consider that while a traditional MAC unit multiply and then accumulate serially it's values, the NVDLA MAC performs all the multiplications in parallel, concluding with a final sum of all the previous results.

[1] the booth multiplier algorithm has a better performance due these shift multiplications that can be done in parallel

6.1.2 MAC optimized architecture

The first NVDLA MAC architecture using booth multipliers, behaviorally works fine but it was not optimal in terms of area and power due the complexity of the booth multiplier. It is not described as a simple booth multiplier since it has the possibility to be configured, for instance to double the speed when reducing the data-type from 16-bits to 8-bits.

Hence, in following releases, NVIDIA has described the MAC cell in a different way, using a simpler multiplier. As reported in listing .8 is the Verilog file of the MAC cell after an RTL build of a small accelerator working at 8-bits.

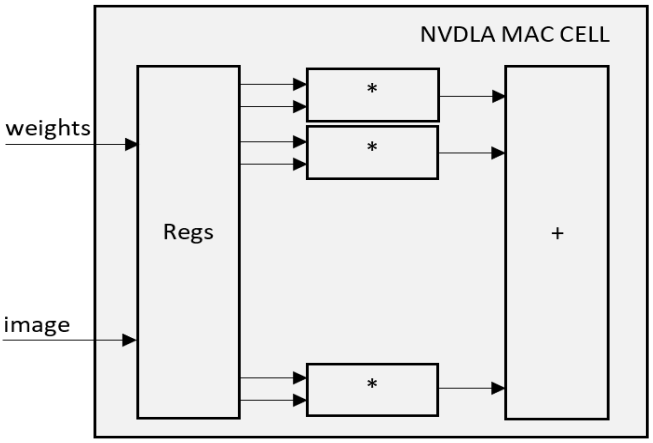


Figure 6.4: NVDLA MAC cell Architecture

In section 4.2.1 it is explained that different RTL models of the accelerator can be generated by changing certain parameters. In particular, the Atomic-C and Atomic-K parameters determine the number of MAC. Moreover, it can be configured the data-type. In this case the generated MAC cell works at 8-bits, thus the multiplication is at 8-bits.

When trying to generate the MAC cell at 16-bits, after the build, the multipliers still multiplicating at 8-bits. *This may be due to a possible NVIDIA bug (in the RTL generation) or simply due to its own quantization at the MAC level.*

Therefore, the 16-bit MAC NVDLA has been generated manually starting from the 8-bit version, applying small modifications.

6.2 Approximate and Inferential MAC Architectures

From the NVDLA architectures, inaccurate MACs have been generated by removing the exact multiplier of the accelerator and inserting the approximate 16-bits and 8-bits version. The respective files to each architecture can be found on github repository.

6.3 Inaccurate MAC optimizations

The input image and kernels values are represented in 2's complement, therefore the booth multiplier is an optimal architecture since the algorithm is directly based on this notation.

For the inaccurate multipliers this is a problem since they work in Sign-and-Magnitude notation. Specifically the problem is that in order to insert those multiplications for replacing the NVDLA one there must be a previous conversion to the inputs of the multiplier from 2's complement to Sign-and-Magnitude and viceversa at the output, which adds area and power consumption to the circuit and its also reduce the working frequency. If the multipliers are compared (without converters), the inaccurates have better performances in terms of area and power, but by introducing these extra hardware they are no longer convenient.

Figure 6.5 shows the structure of the inaccurate multiplier with the converters and compared graphically with the NVDLA multiplier that do not need them.

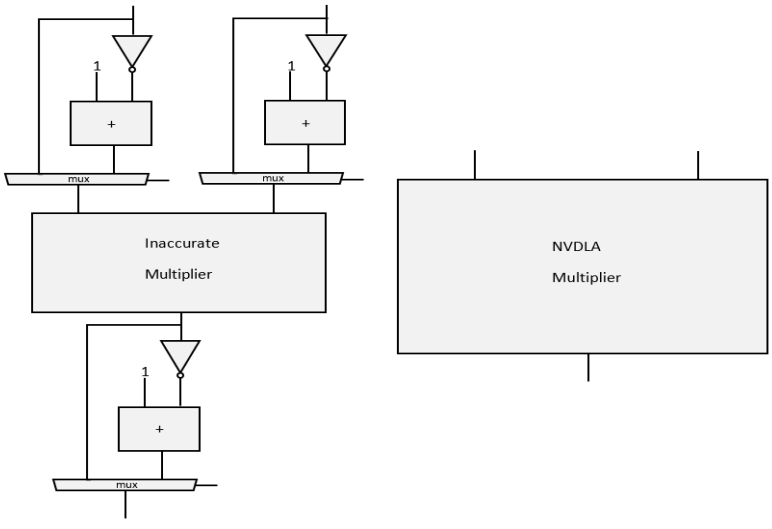


Figure 6.5: Multipliers with/without converters

Leftmost picture in 6.5 shows the inaccurate multiplier with the obligatory converters. The multiplexer selects the input depending on the MSB (not shown in the graph to simplify the understanding). In case of a 0 as the MSB, it means that the value is positive, thus it does not need to be converted. Otherwise, if the number is negative, therefore the module of the negative number is calculated by an inverter following by a +1 addition. For instance, if the input has a value of 0x0045, the MSB is 0, therefore the representation in 2's complement does not change to the representation in Sign-and-Magnitude. On the contrary, if the input number is 0xA0FE, its MSB is 1, therefore the number must be negated, resulting 0x5F01 and added +1 for getting the exact module of the number. In decimal representation, 0xA0FE is not +41214 but is -24320.

To the right of the image you can see the NVDLA multiplier that does not need conversion stages.

The biggest problem with the converters are the adders. **Removing them, the circuit is abruptly improved in terms of area, power and speed, losing a minimum accuracy that in theory should not affect the predictions of the neural network**. Using the previous values as an example, the conversion will throw -24321 instead of -24320. Then the relative error between these values is calculated:

$$relative\_error = \frac{exact - approx}{exact} * 100\%$$

Finally 0.0041% is the error introduced in the converters when removing the adders.

6.4 SystemC

The same architectures are developed in SystemC. While Verilog allows comparing the architectures in terms of area, power and speed, SystemC allows high level simulations for testing the accuracy of each MAC by running inferences of complete neural networks. The accelerator is described in software in a modular form, similar to the hardware model. Therefore there is a module directly related to the CMAC stage of the convolution pipeline which contains the multiply-and-accumulate operations, therefore, the multipliers.

Files related to the MAC modules are:

- NV_NVDLA_cmac.cpp for definitions
- NV_NVDLA_cmac.h for declarations

Inputs of MAC module are explained at next, starting from the inference running command:

- *./nvdla_runtime –loadable default.nvdla –image ../../regression/images/testSample/img_1.jpg –rawdump*

When running the runtime command, the parameters are: the loadable file and the input image. The first one is the pretrained convolutional neural network, so it contains information about the accelerator configuration and weights while the input image has the feature data. All these inputs (for sure processed and with a certain format given in previous steps) comes into the cmac when convolutions has to be done.

Depending on the configuration, the calculation is done at INT8, INT16 or FP16 with the Winograd option enabled or not. As neural networks are developed at software level in programs such as TensorFlow or Caffe, all of them works in a high precision (FP32 or FP64) so minimally NVDLA quantize the data down to FP16.

The MAC operation is usually done at FP16 (at least with the pretrained networks used) since no networks have been found that after the compilation generate a loadable file with a configuration at INT8 or INT16. This has been attempted with specific tools to quantize pretrained models such as *Ristretto* but the compiler fails to interpret the lines related to the quantization.

Diagram 6.6 explain graphically the main functions of the CMAC module in SystemC.

As explained above, due to the configuration of CNN, in a switch-case statement, enter the function calculation_fp16. Enter the data related to the image and the weights in array forms (replicating the cube format) where they pass through two stages: in a first (cal_fp16_mul) where all the multiplications are performed in parallel and stored in an array called *mts_product[ch_iter]* and the second where

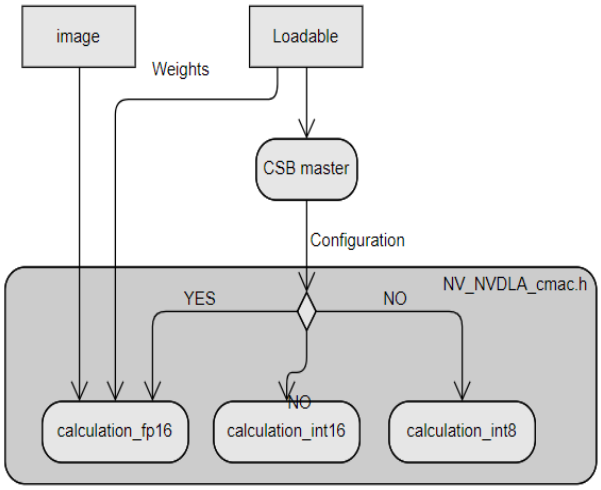


Figure 6.6: NV_NVDLA_cmac.h

the products are added together. Figure 6.7 graphically shows the behavior of the function explained above.

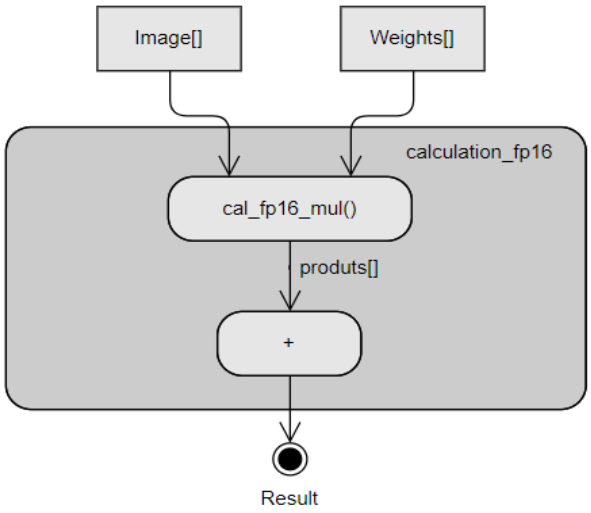
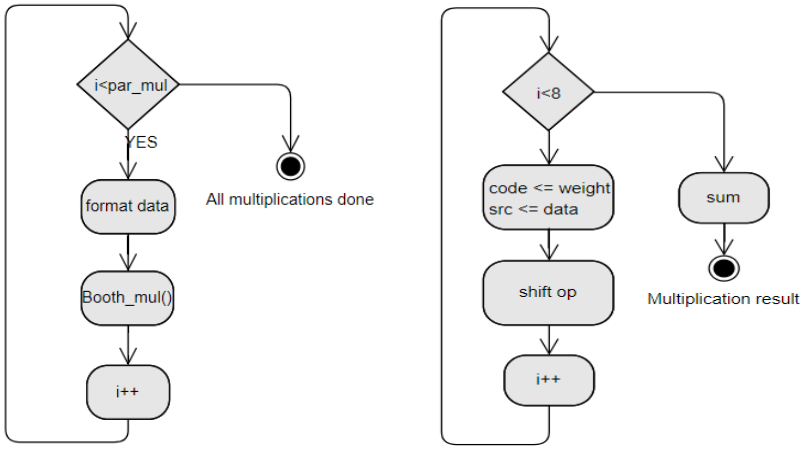


Figure 6.7: calculation_fp16

Within the function cal_fp16_mul, a process occurs as shown in 6.8(a). The array is decomposed in each atomic data of images and weights and they enter in pairs into the cal_fp16_booth function which works as shown in the figure 6.8(b).

The booth multiplier sequentially decomposes weights into codes and shifts the image according to them, finally resulting in a sum. Behaviorally, it acts like a hardware described booth multiplier.

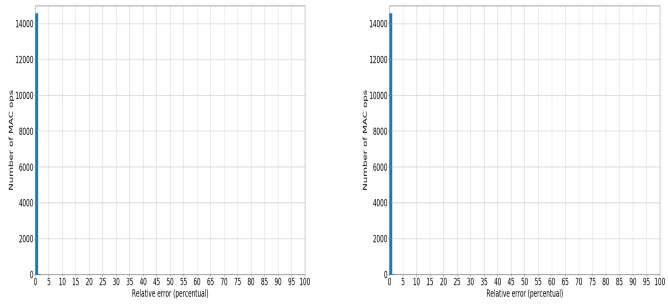


(a) cal_fp16_mul (b) cal_fp16_booth

6.4.1 Interfacing

Internally, the performance of the CMAC stage at the software level was decomposed until the multiplication function. The next task is to replace the booth multiplier with a standard one (using the asterisk symbol between two variables). Therefore, a function called *MULTIPLICATION* is generated, which has as inputs the incoming values to the booth multiplier and as output the *mts_product[]* array where results of the booth multipliers are saved.

As booth multiplier is exact, there are no differences in the results of the multiplications, therefore the results at the MAC level do not vary. Finally the inference predictions are the same. The relative error of each MAC operation using the NVDLA multiplier and the ideal one can be seen in the figures 7.3(c) and 6.8(d) respectively.



(c) MAC error using NVDLA multiplier (d) MAC error using ideal multiplier

6.4.2 Custom multipliers in SystemC

The interface created previously allows to insert custom multipliers, in particular, the approximate and the inferencial architectures.

Approximate

A hierarchical description has been replicated using functions. The appendix .9 contains the software description of the inaccurate 2x2 multiplier. The same appendix contains the 4x4 multiplier conformed with 2x2 multiplier functions. For the construction of bigger multipliers the same logic was used until arriving at the 16 bits version. At the top function, are added to inputs and outputs the respective converters from 2's Complement to Sing-and-Magnitude and vice versa.

To replicate the behavior of the hardware-optimized versions of the multipliers, the "+1" adders used after the negation of the value have been removed.

Inferential

The inferential multiplier, is described in verilog as a huge set of assignments that represent the classification tree.

When trying to replicate the behavior in SystemC, appears a problem due the difference between the runtime of a Verilog code and a SystemC one. While the assings in verilog are done in parallel, the programs in C work sequentially, therefore replicating the code also requires an ordering of assignments which is cumbersome.

Therefore, the most viable option is to represent the inferential multiplier with a Look Up Table with the information of inputs and outputs. When trying to generate a LUT for the 16-bits inferential multiplier the sizes are the following:

$$2^{64} = 1.8446x10^{19}$$

Finally, working with an 8-bits inferential multiplier, the size of the LUT is viable:

$$2^{32} = 4.294.967.296$$

6.4.3 Quantization

To try the accelerator operating at 8-bits, the idea was the following: using *Ristretto*, quantize a pre-trained model and compile it with the nvdla software tools. The problem is that the compiler does not recognize the quantization commands described in the model which leads to the loadable file can not be generated. Therefore, it is not possible to configure externally the NVDLA SystemC model to work with the 8-bits function 6.6.

The compiler continues in development, since it is not only the only bug it presents, for example, some more complex CNNs can not be compiled either.

In one of the last releases, the accelerator allows the build of a small SystemC accelerator model, which would be the optimal solution since it has only the 8-bits function, which means that the accelerator quantize the data from FP32 to INT8. The release was in the final stages of development of the thesis, therefore, this solution is left for future work.

The solution considered was to quantize the data manually upon entering the multiply-and-accumulate function. Half-LSB of the image and weights is truncated, followed by a shift right of the data. Then the multiplication is on 8-bits and the 16-bits result is shifted again accommodating it into 32-bits.

6.4.4 Configuring the MAC

In summary, different versions of the multipliers have been developed within the MAC architecture. All of them are in the same file (*NV_NVDLA_cmac.h*) to speed up simulations and avoid moving files.

If you want to run an inference using a certain MAC, the steps are as follows:

1. In the function *cal_fp16_mul* of the file *NV_NVDLA_cmac.h* select the function that represents the correct multiplier. Uncomment and leave commented on the rest (follow the instructions given in the file).

2. Follow the process described in 5.5.2.

Chapter 7

Results

7.1 Inference Results

The accelerator architecture has been tested using several MAC architectures each one composed by different multipliers:

- NVDLA 8-bits and 16-bits.
- Approximate 8-bits and 16-bits.
- Inference 8-bits and 16-bits.

The original NVDLA MAC has been compared with inaccurate MAC architectures in both 16-bits and 8-bits.

7.1.1 MNIST

The LeNet convolutional neural network pretrained with a MNIST handwritten digits set, expects a precision of 98.9% according to [9]. The inference results of the accelerator using different MAC architectures are shown in table 7.1.

Architecture	Accuracy
NVDLA 16-bits	98.5%
approx 16-bits	98.5%
NVDLA 8-bits	98.5%
approx 8-bits	98.5%
inferential 8-bits	68%

Table 7.1: MNIST Inference results using different multiplicators.

For each test, the number of handwritten digit was 200 thus, 98.5% means 197/200 hits and 3/200 misses.

7.1.2 CIFAR

Using the same LeNet convolutional neural network pretrained with the CIFAR10 dataset of animals and vehicles expects a precision of 76.26% [1]. The inference results of NVDLA using different MAC architectures are shown in table 7.2.

Architecture	Accuracy
NVDLA 16-bits	86%
approx 16-bits	83%
NVDLA 8-bits	86%
approx 8-bits	83%
inferential 8-bits	10%

Table 7.2: CIFAR10 Inference results using different multiplicators

The test set where these results were obtained is composed of 30 random images of vehicles and animals thus 86% means 26/30 hits and 4/30 misses.

It is worth mentioning that inferences in the environment usually take a long time compared to a normal processor or GPU, since it is simulating the behavior of the accelerator.

The MNIST is taken as the study set and the CIFAR10 as proof that the accelerator recognizes multiple configurations. Some interesting facts according to previous results:

- Quantification of data do not affects severely to prediction result. For instance, there is no difference in precision between the 16-bit and 8-bit NVDLA.
- The error introduced by the approximate multiplier is not large enough to affect predictions in big terms.
- The error introduced by the inferential multiplier it is sufficient to affect drastically in prediction results.

[1] According to https://github.com/BIGBALLON/cifar-10-cnn

7.2 Inference Scores

From the results shown in table 7.1, the behavior of each architecture using different multipliers is analyzed at label's scores. Figure 7.1 shows the scores of each class using different architectures.

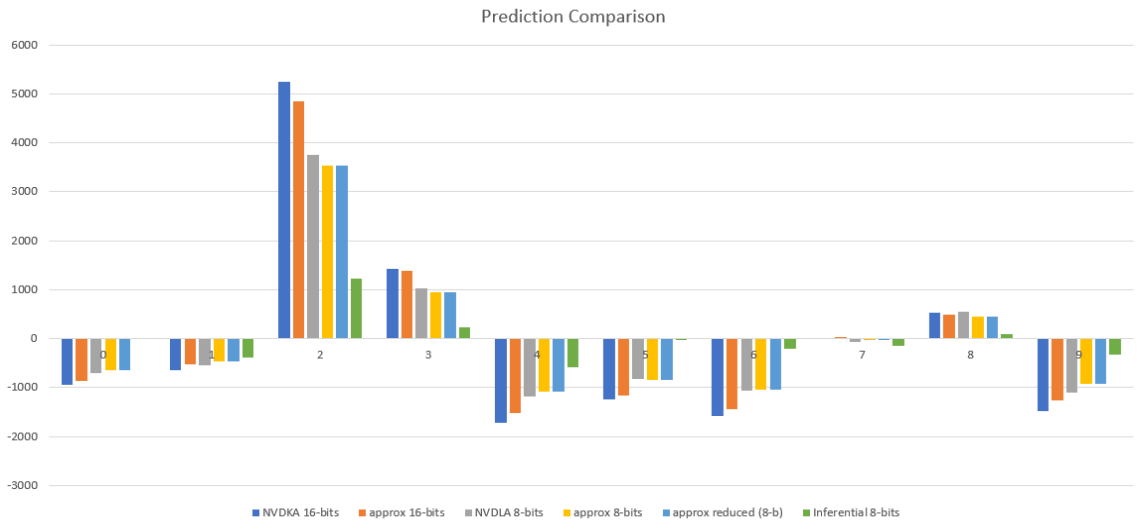


Figure 7.1: Label's scores, "2" as input digit

In *x-axis* are all possible classes (labels), thus handwritten digits from 0 to 9 are represented while in *y-axis* is the value of the final score, the unnormalized probability. Each color represent a different architecture using a different multiplier. This particular figure shows the scores of an inference using a LeNet network pretrained with a MNIST dataset with a *handwritten number 2* as input. All the architectures predict correctly the number, all of them have the label corresponding to number two with the highest score.

NVDLA 16-bits (blue) has the highest score, thus is the most accurate of all of them while approximate 16-bits (orange) is the second one. Quantizing data generate a small fall in scores but still predicting well: gray, yellow and light blue are 8-bits architectures. At last, the green one is the architecture with the inferential. It has a big fall in scores but keep hitting correctly.

Figure 7.2 shows an example of an inference using a LeNet pretrained model where the input image is a handwritten 9. More examples can be found in file *prediction_behavior.xlsx*.

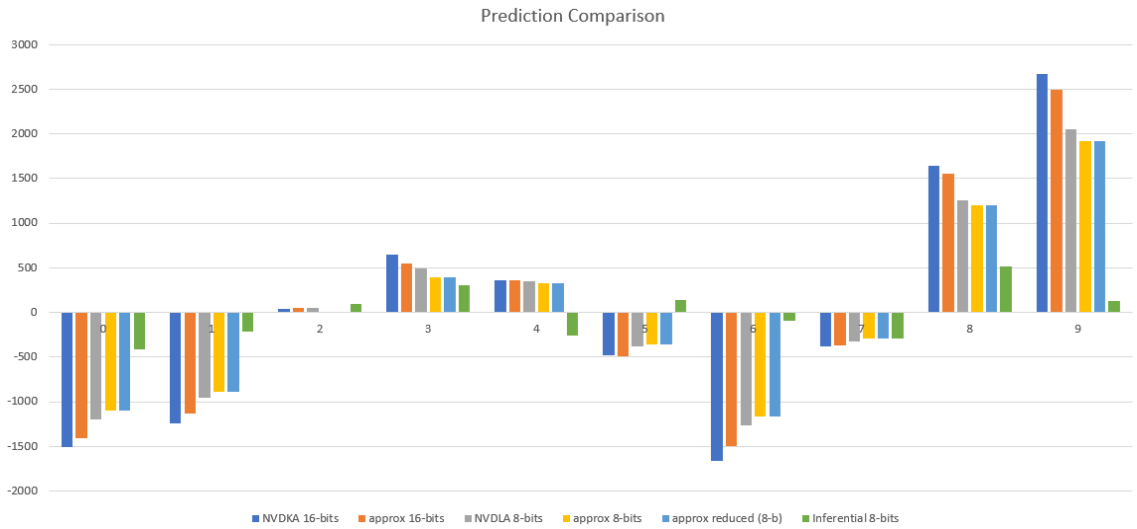


Figure 7.2: Label's scores, "9" as input digit

All the architectures predict correctly the number having the highest score at label 9, except for the inferential multiplier that predicts an "8". Thus, if the multiplier is imprecise enough, predictions will start to fail. In the next section, multipliers are compared directly at the operation level.

7.3 MAC Error

7.3.1 Introduction

A deeper comparison is carried out at MAC level. The relative error of a MAC multiplication can be calculated using the following formula:

$$relative\_error = \frac{exact - approx}{exact} * 100\%$$

exact value represents the multiply-and-accumulate operation result using an ideal multiplier (without introducing errors) while *approx* represents the result of the operation using an approximate multiplier and/or quantizing the data to 8-bits, in both cases introducing an error.

MAC operations have been extracted from an inference of a LeNet CNN pre-trained with a MNIST dataset, using a random handwritten number as input. The file *MAC_SystemC.txt* contains a list of all the multiplications done by the accelerator during the inference. Indeed, the exact number of multiplication is 169.353.

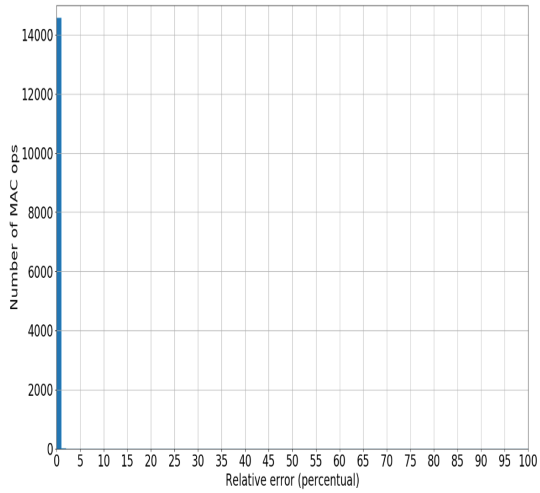
Using this file as input, a script called *plots.pl* generates a list of all the MAC operations results (and it's error) in an output file. The number of MAC operations done by the accelerator during the inference is approximately 15,000.

7.3.2 NVDLA vs Approx 16-bits

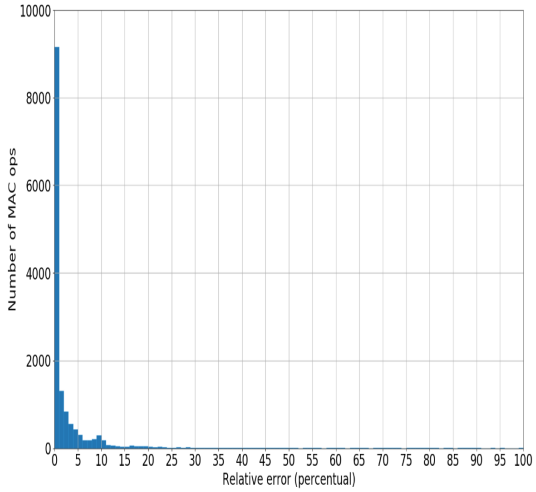
Figure 7.3(c) shows shows the relative error of each MAC operation when using 16-bit NVDLA MAC. Being an exact multiplier, there is no difference between the exact and approximate result, therefore the relative error is always 0.

x-axis represents the relative error in percentage that goes from 0% to 100% (or more). For instance, if the result should be 10 but the approximate throws 12, it means that the relative error is 20%. On the other hand, y-axis is an accumulation of results with this relative value.

Figure 7.3(b) shows the error when using the approximate multiplier at 16-bits. The behavior is quite similar, being wrong in a very low percentage. 11% is the last relative error value that is repeated, although to a very low quantity regarding to 0%.



(a) Error introduced by NVDLA 16-bits multiplier.

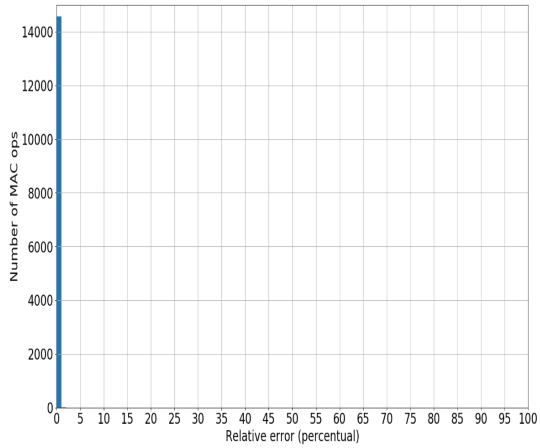


(b) Error introduced by Approx 16-bits multiplier.

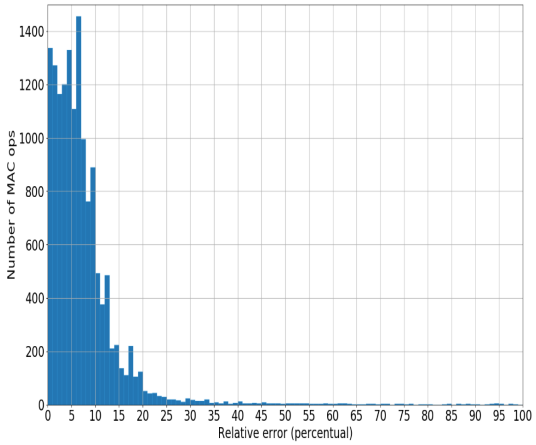
7.3.3 NVDLA 16-bits vs 8-bits

In the previous section it has been compared the exact multiplier with the approximate one. Now it is compared the NVDLA at 16-bits 7.3(c) with NVDLA at 8-bits 7.3(e).

It can be appreciated that when quantifying the data, the greatest number of errors are maintained between 0% and 10%. Then between 10% and 20% there is another group of accumulated errors, but in smaller quantity and abruptly decreases since 20%.



(c) Error introduced by NVDLA 16-bits multiplier.

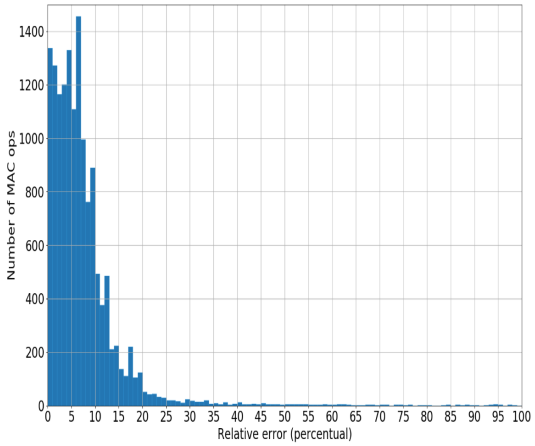


(d) Error introduced by NVDLA 8-bits multiplier.

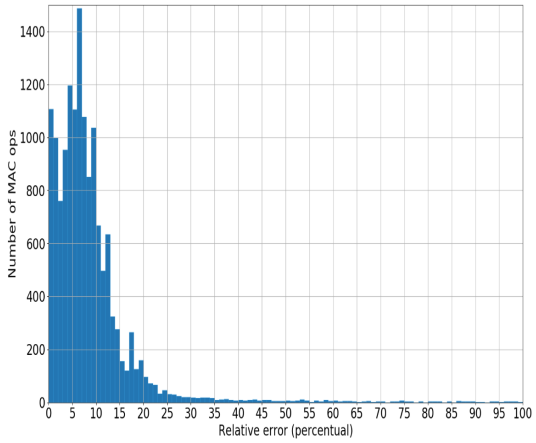
7.3.4 NVDLA vs Approx vs Inference 8-bits

In this section the 8-bits architectures are compared. Figure 7.3(e) is related to the NVDLA, 7.3(f) to the approx and 7.3(g) to the inference.

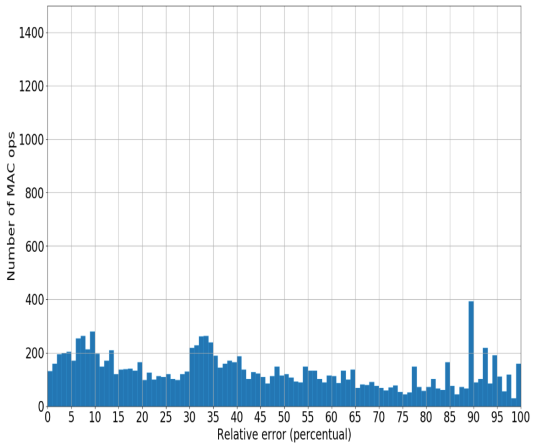
The approximate multiplier behavior is similar to the NVDLA while the inference has high relative errors, greater than 20%. From these results it can understood the result of the imprecise predictions generated by this multiplier.



(e) Error introduced by NVDLA 8-bits multiplier.



(f) Error introduced by Approx 8-bits multiplier.



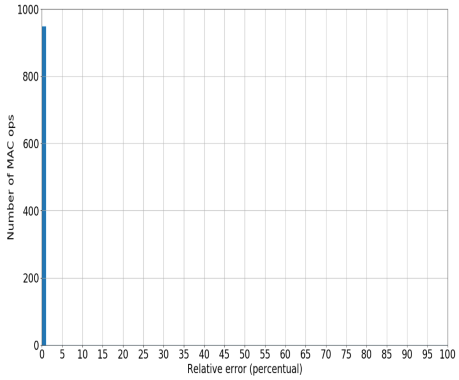
(g) Error introduced by Inference 8-bits multiplier.

7.3.5 MAC error at hardware framework

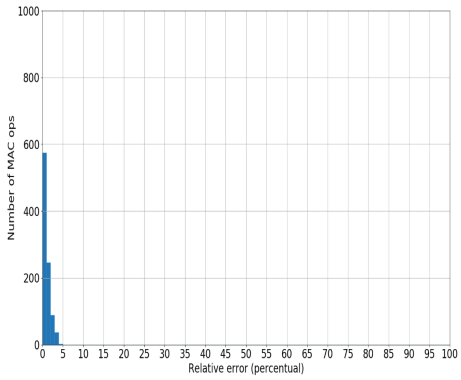
The behavior of the architectures so far was documented in the software framework. The next step is to migrate to the hardware framework to compare them in terms of area, power and speed, but first, before beginning the synthesis of the models to get this data, it is proved the behavior to check the correctness of them. For this task it has been used the single layer Googlenet testbench provided by NVDLA. In other words, the relative error of the MAC operations is extracted and analyzed using the testbenchs of the hardware framework.

Figures 7.3(h), 7.3(i), 7.3(j), 7.3(k) and 7.3(l) are the related relative error distributions for each MAC architecture. The amount of MAC operations is less since it is only a layer what is being tested and not the whole network as it was in the software framework.

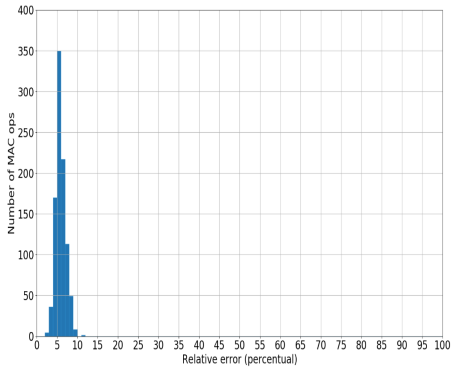
It is worth mentioning that the behavior does not necessarily have to be similar, since they are diverse neural networks. Anyway, there are some similarities between software and hardware simulations. This ensures that the architectures are working correctly is their Verilog descriptions.



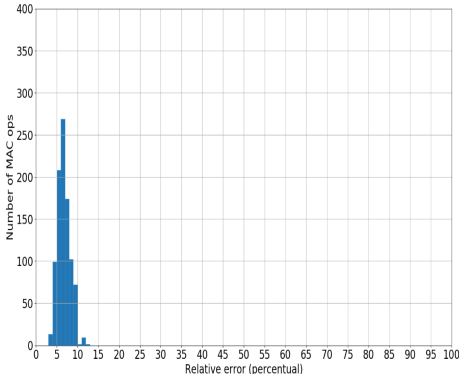
(h) Error introduced by NVDLA 16-bits multiplier



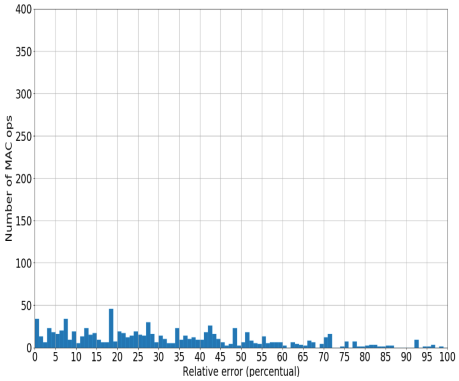
(i) Error introduced by Approx 16-bits multiplier



(j) Error introduced by NVDLA 8-bits multiplier



(k) Error introduced by Approx 8-bits multiplier



(l) Error introduced by Inference 8-bits multiplier

7.4 Synthesis

7.4.1 16-bits vs 8-bits architectures

Among 16-bit architectures (NVDLA, Approx and Inference) there are no major differences when synthetizing them, but comparing 16-bit architectures with the same to 8-bits the results are as shown in figure 7.3.

Reducing operations to 8-bits reduces the area of the circuit by approximately 70% and the power consumption reduction is of 84%. In addition, it allows working at higher frequencies. 8-bit architectures allow working at 1GHz of frequency while 16-bit does not.

Therefore, as long as CNN allows it, in the sense that the quantization of the data does not affect the result of the predictions it is very convenient to work with 8-bits architectures.

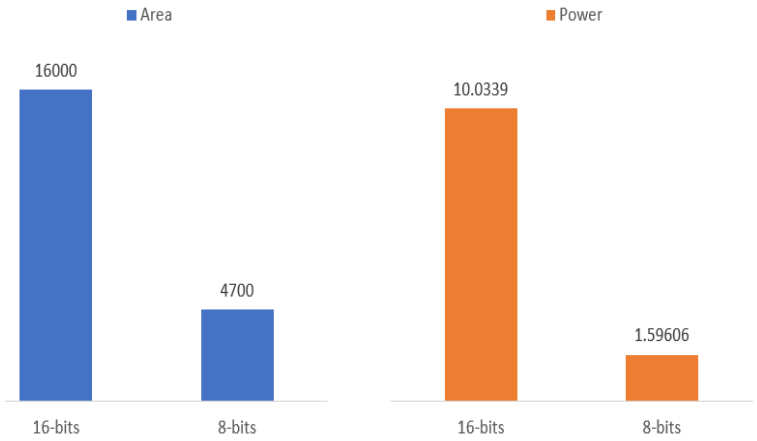


Figure 7.3: Area $[um^2]$ and Power $[mW]$ comparison between 16-bits and 8-bits architectures

For NVDLA 16-bits values are the following

- Area: $16000um^2$
- Power: $10.0339mW$

For NVDLA 8-bits values are the following

- Area: $4700um^2$
- Power: $1.59606mW$

7.4.2 NVDLA, approx and inference at 8-bits

It has been seen that whenever is possible it is very convenient to reduce the architecture to 8-bits. Then, three architectures with different multipliers are possible: NVDLA, approx and inferential. The results in terms of area are shown at figure 7.4.

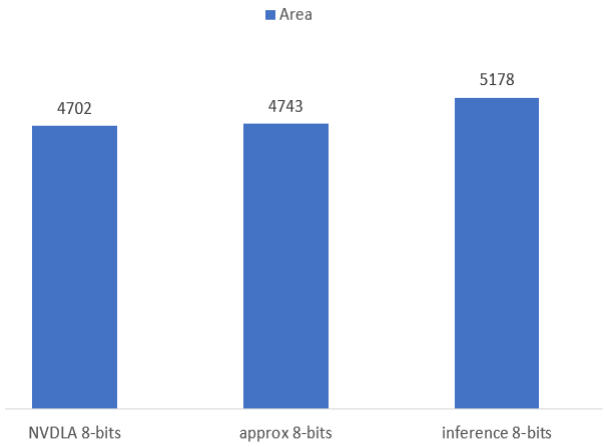


Figure 7.4: Area comparison between 8-bits architectures

According to the results in 7.1 the inferential multiplier is discarded since it strongly reduces the precision in the predictions but could be used with re-training phases to mitigate the accuracy drop. In other words, it can be used as a tool for fine-tuning a CNN model. On the other hand, the results of the NVDLA and the approx are 4702 and 4743 respectively.

The results seem to show that the NVDLA architecture offers better solutions. The question is: why a reduced version of the multiplier is bigger than the original version? The answer has been explained in the section 6.3 and it is because of the converters that are included.

It has been proposed to remove the adders from all converters at the cost of an additional error aggregation, which by the way is minimal. The prediction results when removing the adders are shown in table 7.3.

Architecture	Accuracy
NVDLA 8-bits	98.5%
approx 8-bits	98.5%
inferential 8-bits	68%
approx reduced 8-bits	98.5%

Table 7.3: Inference results including the reduced version of the approximate MAC

Removing the adders, does not worsen the accuracy, it remains the same since as explained in the section, the error introduced when removing them is minimal.

Synthesis results removing the adders is shown at figure 7.5.

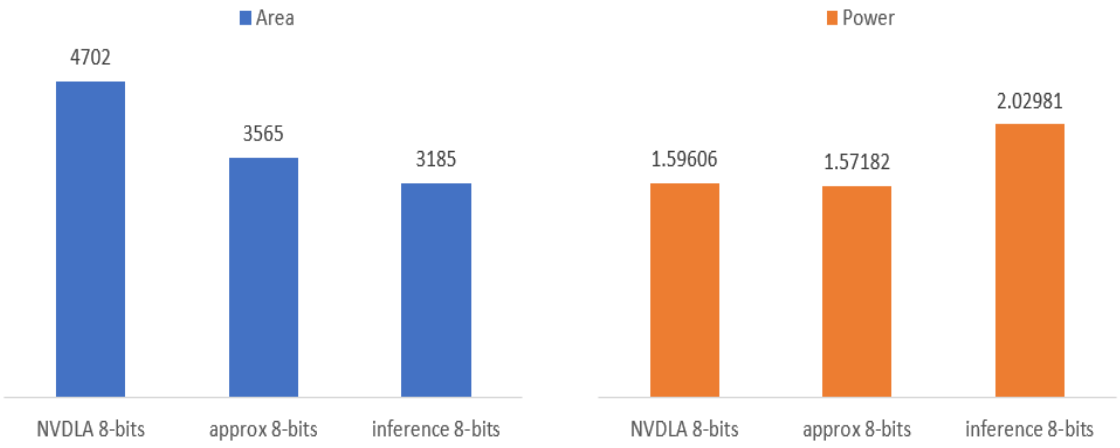


Figure 7.5: Area $[um^2]$ and Power $[mW]$ comparison between 16-bits and 8-bits architectures

For NVDLA 8-bits values are the following

- Area: $4702um^2$.
- Power $1.59606mW$.

For approx. 8-bits values are the following

- Area: $3565um^2$.
- Power $1.57182mW$.

The approximate architecture reduces the area by 24% while the power consumption remains practically the same. If it is a circuit in which savings are fundamental, a reduction of 24% of area is a good point as long as it does not affect the accuracy as in this case tested.

Chapter 8

Conclusion and Future works

In the field of artificial intelligence and machine learning, this type of accelerators are beginning to be of vital importance either for improving performances and computing times or to reduce the architecture in terms of area, power and speed for being able to insert them into smaller devices. As a conclusion, the accelerator architecture is working correctly in both descriptions (Verilog and SystemC) and as a future work, it would be interesting to carry it out as a prototype on an FPGA.

Secondly, it has been shown that the precision in operations is not of crucial in neural networks. Reducing it down to 8-bits generates great improvement in terms of area and power consumption and increases the maximum clock frequency. Related to this topic, the future works should be more testing. Thesis work has been developed when the accelerator was being developed, whereby for instance, a large number of CNNs could not be compiled (because of bugs or future-releases) and therefore tested. Today there are the necessary bases to continue easily testing with new neural networks and architectures.

Regarding the multipliers and the different generated MAC architectures, the conclusion is the following. If the area is not a determining factor in the circuit and high precision is required in the calculations, the standard multiplier is recommended, whereas if the area of the circuit is crucial, using an approximate multiplier is very convenient.

On the other hand, the inferential multiplier can not yet be inserted to replace an exact one, but can be used as a tool for fine-tuning a CNN model. For instance, for re-training phases to mitigate the accuracy drop.

The power consumption of a single multiply-and-accumulate unit has been calculated (via the switching activity) by simulating a single-layer of a convolutional neural network.

Without doing extra work, it can be calculated the power consumption of the entire accelerator during a single-layer. Even more but as a future job, new single-layer configurations at hardware level (Verilog) could be generated from the CNN

simulations in software platform since the KMD has been customized to show register writing to the NVDLA. Therefore, the total power consumption of a complete inference can be calculated.

Chapter 9

Files

In the next chapter, the files related to the thesis are briefly explained.

In https://github.com/IgnacioGoldman. it can be downloaded all the files related to the thesis:

- hardware framework (NVDLA and MAC level), verilog files, scripts, post-synthesis files, results.
- software framework contains the results obtained in this environment, the pre-trained models but not the framework itself. The framework can be described from Google Drive link.

Both the hardware and the software framework can be downloaded and used with little or no modification.

Bibliography

[1] Yann LeCun, Yoshua Bengio Geoffrey Hinton, *Deep learning*, New York University, 2015.

[2] Valentina Arrigoni Beatrice Rossi Pasqualina Fragneto Giuseppe Desoli, *Approximate operations in Convolutional Neural Networks with RNS data representation*, University of California, April 2017.

[3] Alex Krizhevsky Ilya Sutskever Geoffrey E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, University of Toronto.

[4] Christian Szegedy1, *Going Deeper with Convolutions*, University of North Carolina, 2015.

[5] Yu-Hsin Chen, Tushar Krishna, Joel Emer, Vivienne Sze, *Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks*, Massachusetts Institute of Technology, 2016.

[6] Parag Kulkarni, Puneet Gupta, Milos Ercegovac, *Trading Accuracy for Power with an Underdesigned Multiplier Architecture*, University of California, 2011.

[7] Roberto Giorgio Rizzo, Valerio Tenace, Andrea Calimera, *Multiplication by Inference using Classification Trees: a Case-Study Analysis*, Politecnico di Torino, 2018.

[8] Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, *Gradient-Based Learning Applied to Document Recognition*, 1998.

[9] Yann LeCun, Jackel, Bottou, Brunot, Cortes, Denker, Drucker, Guyon, Muller *Comparison of Learning Algorithms for Handwritten Digit Recognition.*

Appendices

.1 CNN LeNet

```
name: "LeNet"
layer {
  name: "data"
  type: "Input"
  top: "data"
  input\textunderscore param { shape: { dim: 1 dim: 1 dim: 28 dim: 28 } }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr\textunderscore mult: 1
  }
  param {
    lr\textunderscore mult: 2
  }
  convolution\textunderscore param {
    num\textunderscore output: 20
    kernel\textunderscore size: 5
    stride: 1
    weight\textunderscore filler {
      type: "xavier"
    }
    bias\textunderscore filler {
      type: "constant"
    }
  }
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling\textunderscore param {
    pool: MAX
    kernel\textunderscore size: 2
    stride: 2
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
```

```
47   param {
48     lr\textunderscore mult: 1
49   }
50   param {
51     lr\textunderscore mult: 2
52   }
53   convolution\textunderscore param {
54     num\textunderscore output: 50
55     kernel\textunderscore size: 5
56     stride: 1
57     weight\textunderscore filler {
58       type: "xavier"
59     }
60     bias\textunderscore filler {
61       type: "constant"
62     }
63   }
64 }
65 layer {
66   name: "pool2"
67   type: "Pooling"
68   bottom: "conv2"
69   top: "pool2"
70   pooling\textunderscore param {
71     pool: MAX
72     kernel\textunderscore size: 2
73     stride: 2
74   }
75 }
76 layer {
77   name: "ip1"
78   type: "InnerProduct"
79   bottom: "pool2"
80   top: "ip1"
81   param {
82     lr\textunderscore mult: 1
83   }
84   param {
85     lr\textunderscore mult: 2
86   }
87   inner\textunderscore product\textunderscore param {
88     num\textunderscore output: 500
89     weight\textunderscore filler {
90       type: "xavier"
91     }
92     bias\textunderscore filler {
93       type: "constant"
94     }
95   }
```

```
96 }
97 layer {
98   name: "relu1"
99   type: "ReLU"
100   bottom: "ip1"
101   top: "ip1"
102 }
103 layer {
104   name: "ip2"
105   type: "InnerProduct"
106   bottom: "ip1"
107   top: "ip2"
108   param {
109     lr\textunderscore mult: 1
110   }
111   param {
112     lr\textunderscore mult: 2
113   }
114   inner\textunderscore product\textunderscore param {
115     num\textunderscore output: 10
116     weight\textunderscore filler {
117       type: "xavier"
118     }
119     bias\textunderscore filler {
120       type: "constant"
121     }
122   }
123 }
124 layer {
125   name: "prob"
126   type: "Softmax"
127   bottom: "ip2"
128   top: "prob"
129 }
```

.2 Convolution Core Stages

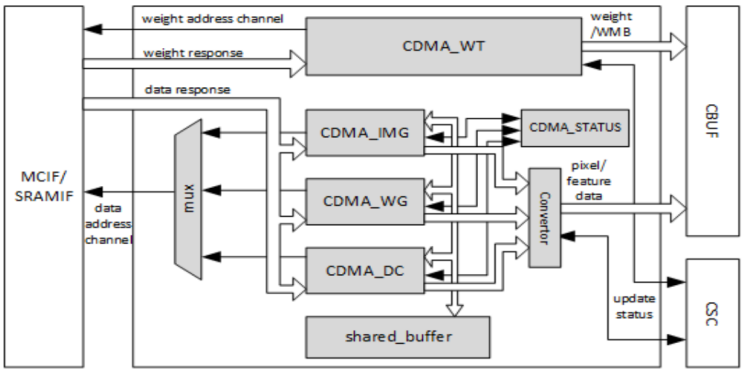


Figure 1: CDMA. Source: https://github.com/nvdla/doc.

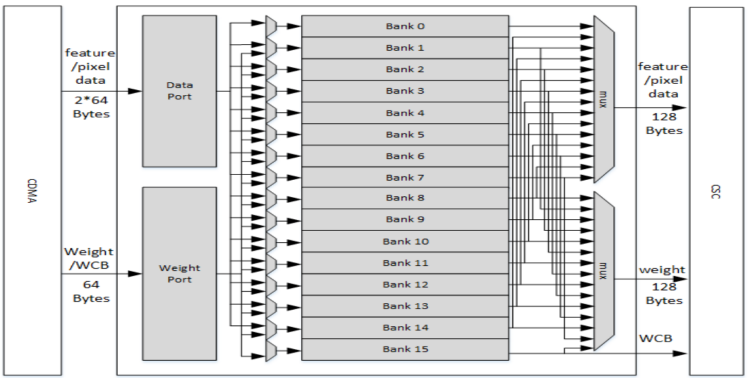


Figure 2: CBUF. Source: https://github.com/nvdla/doc.

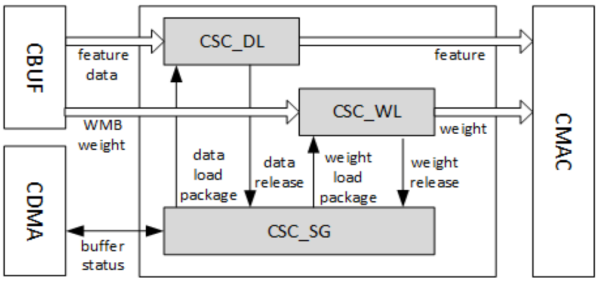


Figure 3: CSC. Source: https://github.com/nvdla/doc.

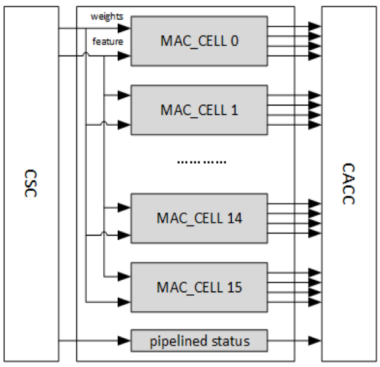


Figure 4: CMAC. Source: https://github.com/nvdla/doc.

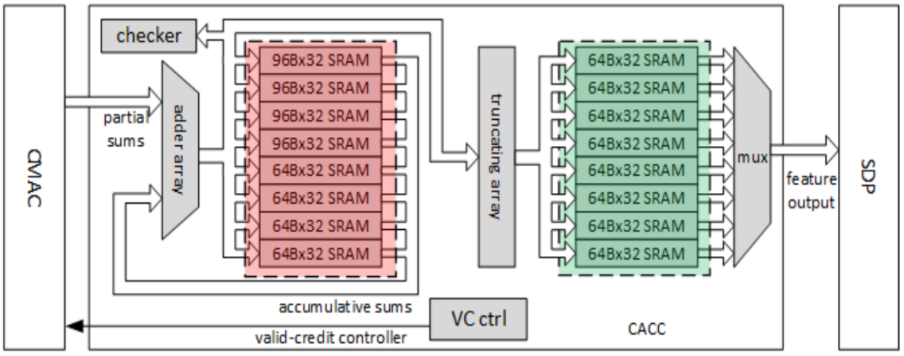


Figure 5: CACC. Source: https://github.com/nvdla/doc.

.3 Modelsim Adaption

```
...
make simulate:
    ...
    @#Change files from .v to .sv in /rams/synth
  @[ -f nv_ram_rws_64x116.v ] && mv nv_ram_rws_64x116.v
    nv_ram_rws_64x116.sv || echo "nv_ram_rws_64x116.sv exists"
  @[ -f nv_ram_rws_16x256.v ] && mv nv_ram_rws_16x256.v
    nv_ram_rws_16x256.sv || echo "nv_ram_rws_16x256.sv exists"
  @[ -f nv_ram_rwsp_80x14.v ] && mv nv_ram_rwsp_80x14.v
    nv_ram_rwsp_80x14.sv || echo "nv_ram_rwsp_80x14.sv exists"
    ...
    @cp files_without_errors/NV_HWACC_NVDLA_tick_defines.vh
    NV_HWACC_NVDLA_tick_defines.vh
  @cp files_without_errors/RANDFUNC.vlib RANDFUNC.vlib
    NV_NVDLA_BDMA_csb.v NV_NVDLA_BDMA_csb.v
    ...
    @cd ~/hardware_framework/NVDLA/verif/synth_tb/sim_scripts && ./
    inp_txn_to_hexdump.pl
    ...
    @vsim ${WAVES} -do "vlog vlibs/*; vlog rams/model/*; vlog rams/
    synth/*; vlog nvdla/apb2csb/*; vlog bdma/* ; vlog cacc/* ; vlog car
    /* ;vlog cbuf/* ;vlog cdma/* ; vlog cdp/* ;vlog mac_units/${MAC}
    _mac/cmac/* ;vlog csb_master/* ; vlog csc/* ;vlog glb/* ;vlog nocif
    /* ;vlog pdp/* ;vlog retiming/* ;vlog rubik/* ;vlog sdp/* ;vlog
    synth_tb/*;vlog top/* ;vsim -novopt work.top +incdir+synth_tb/ +
    incdir+top/ run -all;exit;"
```

.4 Script flow NVDLA framework

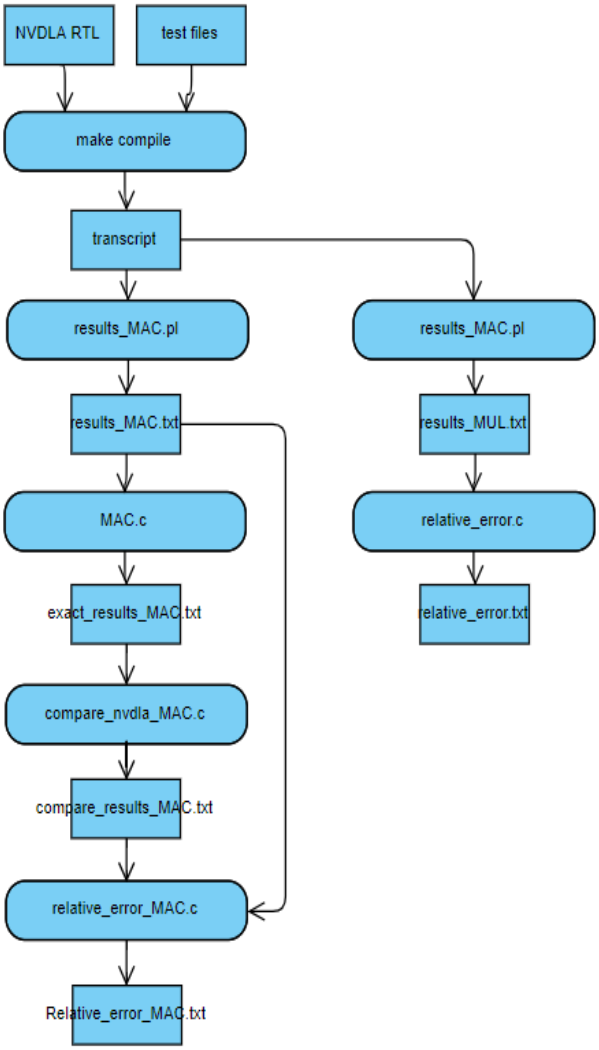


Figure 6: Pre-built code example of accelerator module

.5 After simulation results_generator.pl script

```
#!/usr/bin/env perl
use strict;
use File::Compare;
my $bits      ="$ARGV[0]";
my $flag      ="$ARGV[1]";
my $input\textunderscore file = "./transcript";
my $output\textunderscore file = "./error\textunderscore check/MAC\
    textunderscore result.txt";
my $exact\textunderscore file;
if($flag == 1){
  $exact\textunderscore file = "./error\textunderscore check/exact\
    textunderscore result.txt";
}
my $inf; my $ouf; my $exac; my $check;
my $input\textunderscore lenght = 0;
if($bits == 8){$input\textunderscore lenght = 2;}
if($bits == 16){$input\textunderscore lenght = 4;}

open ($inf, "<", $input\textunderscore file) || die "Cannot open $input
    \textunderscore file";
open ($ouf, ">", $output\textunderscore file) || die "Cannot open
    $output\textunderscore file";
if($flag == 1){
open ($exac, ">", $exact\textunderscore file) || die "Cannot open
    $exact\textunderscore file";
}
my $C = 0;my $B = 0;my $A = 0;my $k = 0;
while(my $line = <$inf>){
  if(index($line,"weight:")>0){
  if ($k < 300000){
    #Inputs and outputs are not in the same line!!!
    #line 1: A0*A1 = X
    #line 2: B0*B1 = X
    #line 3: C0*C1 = A result...
    $C = $B;
    $B = $A;
    $A = $line;
    my $data = (split (/ /, $C, 9))[2];
    my $weight = (split (/ /, $C, 9))[5];
    my $result = (split (/ /, $A, 9))[7];
    my $accumulation = 0;

    #Fill with zeros if data is incomplete
    while(length($data)<(8*$input\textunderscore lenght)){
      $data = "0" . $data;
    }
    while(length($weight)<(8*$input\textunderscore lenght)){
```

```
43        $weight = "0" . $weight;
44      }
45      #convert to decimal approximate result
46      my $res =  hex($result);
47      my $i = 0;
48      #calculate exact result
49      while($i<(8*$input\textunderscore lenght)){
50        my $sub\textunderscore data = substr($data,$i,$input\
      textunderscore lenght);
51        my $sub\textunderscore weight = substr($weight,$i,$input\
      textunderscore lenght);
52        #print $sub\textunderscore data . " * " . $sub\textunderscore
      weight . " = \n";
53        $i = $i + $input\textunderscore lenght;
54        #exact MAC calculation
55        my $dat = hex($sub\textunderscore data);
56        my $wt =  hex($sub\textunderscore weight);
57        if($bits == 8){
58          $dat -= 256 if $dat >= 128;
59          $wt  -= 256 if $wt >= 128;
60        }
61        else{
62          $dat -= 65536 if $dat >= 32768;
63          $wt  -= 65536 if $wt >= 32768;
64        }
65        my $mul\textunderscore res = $dat*$wt;
66        $accumulation = $accumulation + $mul\textunderscore res;
67      }
68        if($k>1){
69        if($bits == 8){
70          $res  -= 65536 if $res >= 32768;
71          $res = $res*65536;
72        }
73        else{
74          $res  -= 4294967296 if $res >= 2147483648;
75        }
76        #print results of output files
77        print $ouf $res . "\n";
78        #print exact result only if working with MAC NVDLA 16-bits
79        if($flag == 1){
80          print $exac $accumulation . "\n";
81        }
82            }
83    }
84    $k++;
85    }
86  }
```

.6 After simulation compare.pl script

```
#!/usr/bin/env perl
use strict;
use File::Compare;
#input and output files
my $approx = "./error\textunderscore check/MAC\textunderscore result.
    txt";
my $exact = "./error\textunderscore check/exact\textunderscore result.
    txt";
my $output = "./error\textunderscore check/relative\textunderscore
    error.txt";
#variables related to I/O
my $appr;
my $ouf;
my $exac;
#open in R or W mode
open ($appr, "<", $approx) || die "Cannot open $approx";
open ($exac, "<", $exact) || die "Cannot open $exact";
open ($ouf, ">", $output) || die "Cannot open $output";

while(my $res\textunderscore appr = <$appr>){
  #get lines of both files
  my $res\textunderscore exac = <$exac>;
  #exact result of MAC operation
    my $exact = $res\textunderscore exac;
  #approximate result o MAC operation
  my $approx = $res\textunderscore appr;
  #calculate error
  my $absolute = $exact - $approx;
  my $relative = 0;
   if ($absolute != 0 && $exact != 0){
     $relative = ($absolute/$exact)*100;
   }
   if($relative == 0){
     print $ouf "relative error is: 0%\n";
   }else{
     print $ouf "relative error is: " . $relative . "%\n";
   }
}
print "exiting file\n";
close $exac;
close $appr;
close $ouf;
```

.7 Gitignore file configuration

```
[url "https://github.com/qemu/skiboot"]
    insteadOf = git://git.qemu.org/skiboot.git
[url "https://anongit.freedesktop.org/git"]
    insteadOf = git://anongit.freedesktop.org
[url "https://github.com/qemu"]
    insteadOf = git://git.qemu-project.org
[url "https://github.com"]
    insteadOf = git://github.com
[url "http://git.qemu.org/git/QemuMacDrivers.git"]
    insteadOf = git://git.qemu.org/QemuMacDrivers.git
[url "https://github.com/qemu/skiboot"]
    insteadOf = git://git.qemu.org/skiboot.git
[user]
    name = IgnacioGoldman
    email = ignig22@gmail.com
```

.8 NVDLA MAC cell description

```
1  `define DESIGNWARE_NOEXIST 1
2  /*MAC NVDLA INT8*/
3  module mac_unit (
4     nvdla_core_clk
5    ,nvdla_wg_clk
6    ,nvdla_core_rstn
7    ,cfg_is_wg
8    ,cfg_reg_en
9    ,dat_actv_data
10   ,dat_actv_nz
11   ,dat_actv_pvld
12   ,wt_actv_data
13   ,wt_actv_nz
14   ,wt_actv_pvld
15   ,mac_out_data
16   ,mac_out_pvld
17   );
18 input nvdla_core_clk;
19 input nvdla_wg_clk;
20 input nvdla_core_rstn;
21 input cfg_is_wg;
22 input cfg_reg_en;
23 input [8*8 -1:0] dat_actv_data;
24 input [8 -1:0] dat_actv_nz;
25 input [8 -1:0] dat_actv_pvld;
26 input [8*8 -1:0] wt_actv_data;
27 input [8 -1:0] wt_actv_nz;
28 input [8 -1:0] wt_actv_pvld;
29 output [19 -1:0] mac_out_data;
30 output mac_out_pvld;
31
32 wire [8-1:0] wt_actv_data0 = wt_actv_data[7:0];
33 wire [8-1:0] dat_actv_data0 = dat_actv_data[7:0];
34 wire wt_actv_nz0 = wt_actv_nz[0];
35 wire dat_actv_nz0 = dat_actv_nz[0];
36
37 wire [8-1:0] wt_actv_data1 = wt_actv_data[15:8];
38 wire [8-1:0] dat_actv_data1 = dat_actv_data[15:8];
39 wire wt_actv_nz1 = wt_actv_nz[1];
40 wire dat_actv_nz1 = dat_actv_nz[1];
41
42 wire [8-1:0] wt_actv_data2 = wt_actv_data[23:16];
43 wire [8-1:0] dat_actv_data2 = dat_actv_data[23:16];
44 wire wt_actv_nz2 = wt_actv_nz[2];
45 wire dat_actv_nz2 = dat_actv_nz[2];
46
47 wire [8-1:0] wt_actv_data3 = wt_actv_data[31:24];
```

```
48 wire [8-1:0] dat_actv_data3 = dat_actv_data[31:24];
49 wire wt_actv_nz3 = wt_actv_nz[3];
50 wire dat_actv_nz3 = dat_actv_nz[3];
51
52 wire [8-1:0] wt_actv_data4 = wt_actv_data[39:32];
53 wire [8-1:0] dat_actv_data4 = dat_actv_data[39:32];
54 wire wt_actv_nz4 = wt_actv_nz[4];
55 wire dat_actv_nz4 = dat_actv_nz[4];
56
57 wire [8-1:0] wt_actv_data5 = wt_actv_data[47:40];
58 wire [8-1:0] dat_actv_data5 = dat_actv_data[47:40];
59 wire wt_actv_nz5 = wt_actv_nz[5];
60 wire dat_actv_nz5 = dat_actv_nz[5];
61
62 wire [8-1:0] wt_actv_data6 = wt_actv_data[55:48];
63 wire [8-1:0] dat_actv_data6 = dat_actv_data[55:48];
64 wire wt_actv_nz6 = wt_actv_nz[6];
65 wire dat_actv_nz6 = dat_actv_nz[6];
66
67 wire [8-1:0] wt_actv_data7 = wt_actv_data[63:56];
68 wire [8-1:0] dat_actv_data7 = dat_actv_data[63:56];
69 wire wt_actv_nz7 = wt_actv_nz[7];
70 wire dat_actv_nz7 = dat_actv_nz[7];
71
72
73 `ifdef DESIGNWARE_NOEXIST
74 wire signed [19 -1:0] sum_out;
75 wire [8 -1:0] op_out_pvld;
76
77 assign op_out_pvld[0] = wt_actv_pvld[0] & dat_actv_pvld[0] &
      wt_actv_nz0 & dat_actv_nz0;
78 wire signed [18-1:0] mout_0 = ($signed(wt_actv_data0) * $signed(
      dat_actv_data0)) & $signed({18{op_out_pvld[0]}});
79 assign op_out_pvld[1] = wt_actv_pvld[1] & dat_actv_pvld[1] &
      wt_actv_nz1 & dat_actv_nz1;
80 wire signed [18-1:0] mout_1 = ($signed(wt_actv_data1) * $signed(
      dat_actv_data1)) & $signed({18{op_out_pvld[1]}});
81 assign op_out_pvld[2] = wt_actv_pvld[2] & dat_actv_pvld[2] &
      wt_actv_nz2 & dat_actv_nz2;
82 wire signed [18-1:0] mout_2 = ($signed(wt_actv_data2) * $signed(
      dat_actv_data2)) & $signed({18{op_out_pvld[2]}});
83 assign op_out_pvld[3] = wt_actv_pvld[3] & dat_actv_pvld[3] &
      wt_actv_nz3 & dat_actv_nz3;
84 wire signed [18-1:0] mout_3 = ($signed(wt_actv_data3) * $signed(
      dat_actv_data3)) & $signed({18{op_out_pvld[3]}});
85 assign op_out_pvld[4] = wt_actv_pvld[4] & dat_actv_pvld[4] &
      wt_actv_nz4 & dat_actv_nz4;
86 wire signed [18-1:0] mout_4 = ($signed(wt_actv_data4) * $signed(
      dat_actv_data4)) & $signed({18{op_out_pvld[4]}});
```

```
87 assign op_out_pvld[5] = wt_actv_pvld[5] & dat_actv_pvld[5] &
       wt_actv_nz5 & dat_actv_nz5;
88 wire signed [18-1:0] mout_5 = ($signed(wt_actv_data5) * $signed(
       dat_actv_data5)) & $signed({18{op_out_pvld[5]}});
89 assign op_out_pvld[6] = wt_actv_pvld[6] & dat_actv_pvld[6] &
       wt_actv_nz6 & dat_actv_nz6;
90 wire signed [18-1:0] mout_6 = ($signed(wt_actv_data6) * $signed(
       dat_actv_data6)) & $signed({18{op_out_pvld[6]}});
91 assign op_out_pvld[7] = wt_actv_pvld[7] & dat_actv_pvld[7] &
       wt_actv_nz7 & dat_actv_nz7;
92 wire signed [18-1:0] mout_7 = ($signed(wt_actv_data7) * $signed(
       dat_actv_data7)) & $signed({18{op_out_pvld[7]}});
93 assign sum_out =
94     mout_0
95     + mout_1
96     + mout_2
97     + mout_3
98     + mout_4
99     + mout_5
100    + mout_6
101    + mout_7
102 ;
103 `endif
104
105 wire pp_pvld_d0 = (dat_actv_pvld[0] & wt_actv_pvld[0]);
106 wire [19 -1:0] sum_out_d0 = sum_out;
107 reg [19-1:0] sum_out_d0_d1;
108 always @(posedge nvdla_core_clk) begin
109     if ((pp_pvld_d0)) begin
110         sum_out_d0_d1[19-1:0] <= sum_out_d0[19-1:0];
111     end
112 end
113
114 reg pp_pvld_d0_d1;
115 always @(posedge nvdla_core_clk) begin
116     pp_pvld_d0_d1 <= pp_pvld_d0;
117 end
118
119 reg [19-1:0] sum_out_d0_d2;
120 always @(posedge nvdla_core_clk) begin
121     if ((pp_pvld_d0_d1)) begin
122         sum_out_d0_d2[19-1:0] <= sum_out_d0_d1[19-1:0];
123     end
124 end
125
126 reg pp_pvld_d0_d2;
127 always @(posedge nvdla_core_clk) begin
128     pp_pvld_d0_d2 <= pp_pvld_d0_d1;
129 end
```

```
130
131 reg [19-1:0] sum_out_d0_d3;
132 always @(posedge nvdla_core_clk) begin
133     if ((pp_pvld_d0_d2)) begin
134         sum_out_d0_d3[19-1:0] <= sum_out_d0_d2[19-1:0];
135     end
136 end
137
138 reg pp_pvld_d0_d3;
139 always @(posedge nvdla_core_clk) begin
140     pp_pvld_d0_d3 <= pp_pvld_d0_d2;
141 end
142
143 wire [19-1:0] sum_out_dd;
144 assign sum_out_dd = sum_out_d0_d3;
145
146 wire pp_pvld_dd;
147 assign pp_pvld_dd = pp_pvld_d0_d3;
148
149 assign mac_out_pvld=pp_pvld_dd;
150 assign mac_out_data=sum_out_dd;
151 endmodule
```

.9 Approximate multiplier SystemC

```
void mult2x2(int x1, int x0, int y1, int y0, int *p3, int *p2, int *p1,
    int *p0){
    *p3 = 0;
    *p2 = x1 && y1;
    *p1 = x1 && y0 || x0 && y1;
    *p0 = x0 && y0;
    }

    void mult4x4(int *out, int a, int b){
    int wirex_out;
    int wire_pp1, wire_pp2, wire_pp3, wire_pp4;

    int a_0 = (a & ( 1 << 0 )) >> 0;int a_1 = (a & ( 1 << 1 )) >> 1;int
    a_2 = (a & ( 1 << 2 )) >> 2;int a_3 = (a & ( 1 << 3 )) >> 3;
  int b_0 = (b & ( 1 << 0 )) >> 0;int b_1 = (b & ( 1 << 1 )) >> 1;int
   b_2 = (b & ( 1 << 2 )) >> 2;int b_3 = (b & ( 1 << 3 )) >> 3;

  int wire_pp1_0, wire_pp1_1, wire_pp1_2, wire_pp1_3;
  int wire_pp2_0, wire_pp2_1, wire_pp2_2, wire_pp2_3;
  int wire_pp3_0, wire_pp3_1, wire_pp3_2, wire_pp3_3;
  int wire_pp4_0, wire_pp4_1, wire_pp4_2, wire_pp4_3;
  mult2x2(a_1, a_0, b_1, b_0, &wire_pp1_3, &wire_pp1_2, &wire_pp1_1, &
    wire_pp1_0);
  mult2x2(a_3, a_2, b_1, b_0, &wire_pp2_3, &wire_pp2_2, &wire_pp2_1, &
    wire_pp2_0);
  mult2x2(a_1, a_0, b_3, b_2, &wire_pp3_3, &wire_pp3_2, &wire_pp3_1, &
    wire_pp3_0);
  mult2x2(a_3, a_2, b_3, b_2, &wire_pp4_3, &wire_pp4_2, &wire_pp4_1, &
    wire_pp4_0);

  wire_pp1 = (wire_pp1_3 << 3) + (wire_pp1_2 << 2) +(wire_pp1_1 << 1) +
     (wire_pp1_0);
  wire_pp2 = (wire_pp2_3 << 3) + (wire_pp2_2 << 2) +(wire_pp2_1 << 1) +
     (wire_pp2_0);
  wire_pp3 = (wire_pp3_3 << 3) + (wire_pp3_2 << 2) +(wire_pp3_1 << 1) +
     (wire_pp3_0);
  wire_pp4 = (wire_pp4_3 << 3) + (wire_pp4_2 << 2) +(wire_pp4_1 << 1) +
     (wire_pp4_0);

  int conc_wire = (wire_pp4 << 4) + wire_pp1;
  wirex_out = (wire_pp2 << 2) + (wire_pp3 << 2) + conc_wire;
  //wirex_out = { 2'b0, wire_pp2, 2'b0} + {2'b0, wire_pp3, 2'b0} + {
    wire_pp4, wire_pp1};

  *out = wirex_out;
    }
```