POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

A FPGA-based tensor accelerator for Machine Learning

Supervisors

Prof. Paolo BERNARDI

Prof. Pedro P. M. TRANCOSO

Candidate

Francesco ANGIONE

A.Y. 2019/2020

Abstract

Part of a Neural Network inference execution mainly consists in multiplications and additions, basic operation of tensor convolutions, and across several execution data, especially weight tensors, are reused. Clearly, those operations are executed on a CPU but, as it is well known, they are independent of each other and therefore they can be executed in parallel by the means of parallel architectures, such as GPU or domain specific hardware platform. In the following pages, the state-of-the-art for accelerating Neural Network inference is explored starting from the newest proposed GPGPU architecture by NVIDIA to the domain specific accelerator from Google, NVIDIA, and Habana.
With the state-of-the-art awareness, a hardware accelerator capable of execution tensor convolution, compute and memory intensive operation of a Neural Network, is designed from scratch. It is also designed for accommodating different data type computation request from Neural Network models, ranging from integer8/16/32/64 to floating-point 32 and brain floating-point 16. Starting from the hardware system development, through the software development of a library capable to use the underlying hardware, it ends with integration into a popular Machine Learning framework, Tensorflow.
The work is carried out on a configurable hardware, FPGA, which allows to explore different design points, in terms of latency and number of processing elements, for different Neural Network models and data type. Moreover, the impact of integrating the accelerator into the Neural Network model is measured and compared with different platforms. Energy consumption is also estimated in the case of deployment on mobile devices.

Keywords: Computer, science, computer science, engineering, hardware, accelerator, machine learning.

asas

Acknowledgements

Acknowledgements

It is always hard to write this part of a work. I would say it is the hardest part, more than the technical one.
However, let me try to address it anyway. I am apologizing in advance if i will forget something.

This work is the sum of five years of experiences, from a technical and non-technical point of view, and it has been developed during a terrible event, a pandemic, which has literally stopped the entire world and caused death, issues and debates. However, as the human race has always been, we are resilient to everything, and we tried as much as we could to not let the world stop, especially thanks to technology, Internet and all the related services. We are just human, but we can do whatever we can image, especially in Computer Science.

First, I would like to thank my family for all their support and presence, even when i was going counter current in my life. I would like to thank to both my supervisors Prof. Paolo Bernardi and Prof. Pedro Petersen Moura Trancoso for believing in me without any guarantees on the final work, and their support through this journey.

To the day in which I learned how to read, an important pillar of my life.
To the people who have contributed, in badness and goodness, to make me the person who i am today.
To my past and future failures, where I have built and I will build myself.
To my feelings, which remember us how much we are fragile but at the same time they remind us that we are human being, and we gather our strength from them.
Sapere aude.

Francesco Angione, Gothenburg, September 2020

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine learning is one of the hot technologies today as it is being used to solve complex problems that would otherwise be very hard or costly to solve with traditional methods. Speech and image recognition as well as many other complex decision-making problems such as self-driving vehicles are successfully solved with machine learning and deep-learning. In the last years, the number of published papers regarding Machine Learning have growth exponentially, and the success of machine learning has been driven by the current available hardware which could provide the required demands in terms of storage and compute capacity. But obviously as problems scale so do the demands and thus companies has started to develop, deploy and sell their own hardware platform, such as Tensor Processing Unit [**paper:40**] from Google, NVDLA[**WEBSITE:6**] from Nvidia and Gaudi [**paper:39**] and Goya [**paper:38**], respectively for training and interference, from Habana (acquired by Intel).

The use of commodity hardware is not the most effective and efficient way to execute Machine Learning, so research is looking at flexible hardware solutions [**paper:1**] [**paper:2**] that can satisfy the required demands for different Machine-Learning models but at lower cost and energy consumption in order to be deployed also on mobile devices. Moreover, during the inference process, a model does not need high precision computations [**paper:8**] [**paper:15**] for achieving high accuracy into its outputs. As it is very well-known, hardware accelerators are capable, if designed correctly, of delivering a lot of improvements in terms of the latency but also in terms of energy efficiency [**paper:29**]. Thus, in order to obtain the best solution in every metric a hardware-software co-design is needed, requiring to the hardware designer a basic knowledge of machine learning algorithms.

Machine Learning includes two processes, the training and the inference. The training process is done off the field, on powerful machines, exploiting different algorithms for optimizing the models in terms of memory footprint, data type and feedback mechanisms for fine-tuning the weight values. On the other hand, the inference process is the execution of the trained model, applying the inputs and expecting the correct outputs. It is done on the field, for example a mobile device, which is area and energy constrained. The inference process is massive composed of multiplication and addition and on a normal CPU-based system they are executed sequentially, increasing the latency of the model and the energy consumption due to data movement.

Thus, the goal is to develop a hardware accelerator from scratch, which implements a tensor-based convolution. Exploiting a non Von Neumann architecture and data locality and reuse for weights reduces the CPU workload and boost the models performance. The use of different arithmetic data type can drastically reduce the computations without reducing the final accuracy of the Neural Network [**paper:8**] [**paper:7**]. From a hardware perspective, the use of different arithmetic precision [**paper:14**], such as the use of integer operations instead of floating-point operations, can lead to benefits in terms of area, energy consumption and latency.

In order to have the possibility of exploring different solutions, in terms of size and latency, of the accelerator the work is deployed on FPGA and it is integrated into a common Ml-Framework, Tensorflow. Accuracy of operations, reliability, performance and energy efficiency are evaluated and compared to the implementation of the same models executed on a GPU.

Chapter 2

Background

Can a machine think?
— Alan Turing, Computing Machinery and Intelligence

2.1 Overview

In the past decade many companies have started to advertise the use of AI, even if they are using a subfield of the AI, in their products and software applications. Nevertheless, the recent growth, the AI is not youth.
It takes one of its roots from a theoretical paper of *Alan Turing* published by journal *Mind* in the 1950 [**paper:36**].

The general definition of Artificial Intelligence (AI): *intelligence demonstrated by machines, any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals* [**book:1**].
In general, ”artificial intelligence” is used when machines mimics the cognitive functions of the human mind, i.e. learning and problem solving.
According to the definition, AI is too vast to be studied and simulated [**book:1**]. Therefore, it has been divided into subfields, characterized by different traits, such as knowledge representation, planning, learning, natural language processing, perception, motion and manipulation, social intelligence and general intelligence.

Artificial Intelligence can be seen as a general purpose technology. It does not exist a general task on which it excels neither how to characterize them.

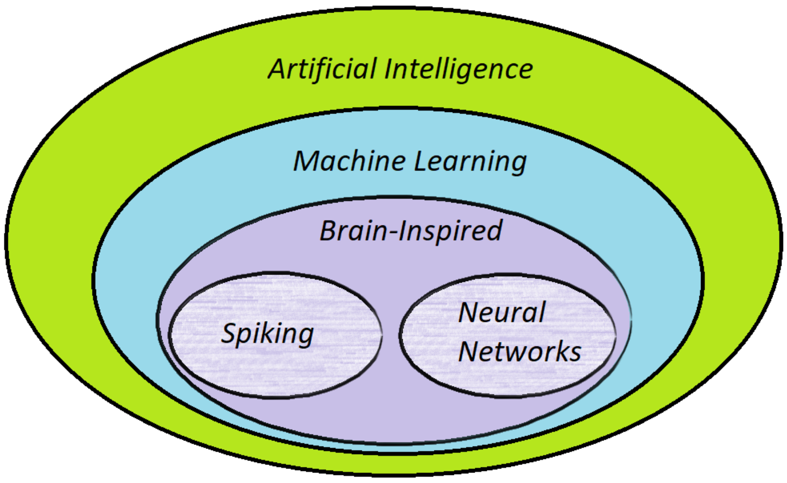


Figure 2.1: Classification of AI with emphasis on Machine Learning and its subclassification

2.2 Machine Learning

A particular interesting subcategory of AI in Computer Science is the machine learning. It is the study of algorithms used to perform a specific task without explicit programming the machine, relying on patterns and inference, in order to make decisions. This approach is used where it is tricky, or unfeasible, to develop a conventional algorithm for solving the task.

A peculiarity of machine learning model is that it is composed of two processes, training and inference.
The inference process is the process in which a conclusion is given at the end of the evaluation process, i.e. the input stimulus are applied to the model and the output is observed.
The training process has to be done before the model is put on the field, before the inference process, otherwise the latter can give wrong results. As the name suggests, in this process the model learns how to behave, adjusting the weight accordingly to the applied inputs and expected outputs. Besides this type of training and according to [**book:1**], other exists, characterized by approach, type of data and tasks:

- Supervised Learning, it builds a mathematical model of a set of data that contains both the inputs and the desired outputs.
- Unsupervised Learning, it takes a set of data that contains only inputs and

find structure in the data.

- Semi-supervised Learning, it falls between unsupervised learning and supervised learning.

- Reinforcement Learning, it concerns how software agents should take actions in order to maximize some notion.

- Self Learning, It is a learning with no external rewards and no external teacher advices.

- Feature Learning, also called representation learning algorithms, often attempts to transform data and preserve at the same time. It is used as a preprocessing step before any classification or predictions.

- Sparse Dictionary Learning, it is a feature learning method where a training example is represented as a linear combination of basis functions, and is assumed to be a sparse matrix.

- Anomaly Detection, also known as outlier detection, identifies rare items, events or observations which are significantly different from the majority of data.

- Association Rules, it is a rule-based method for discovering relationships between variables in large databases.

Machine learning space is also divided into other type of models such as decision tree, support vector machines, regression analysis, Bayesian networks and genetic algorithms. As it can be seen in Figure 2.1 Brain Inspired machine learning is also divided in subcategories.

2.2.1 Brain Inspired

It is based on algorithms which take its basic functionalities from our understanding of how the brain operates, trying to mimic the functionalities.

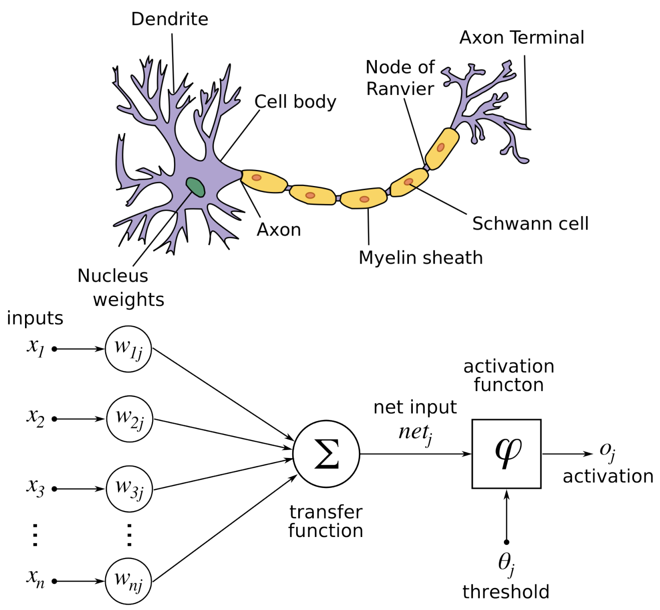


Figure 2.2: A parallelism between a human-brain neuron and a neuron in a Brain Inspired Network[1]

In the human brain, the basic computational unit is the neuron.
Neurons receive input signal from dendrites and produce output signal along the axon which interacts with other neurons via synaptic weights.
The synaptic weights are obtained after a learning process, which can strengthen them or not.

Neural Networks

Neural Networks (or Artificial Neural Networks) are graphs in which every node is interconnected to others using edges, which have a weight properly tuned during the training process.
As mentioned before, each and every node of the neural networks is called artificial neurons (a loosely model of its biological counterpart) and the connections (synapses in biological brain) can transmit information from a neuron to another. In Figure 2.2 the neurons receive signals, which is processed internally, and then they propagate it to the other connected neurons.
The information exchanged between a neuron and another is a real number, a result of a non-linear function of the sum of all its input.

In the Figure 2.3 an implementation of a Neural Network can be appreciated.

[1]Figures under CC license

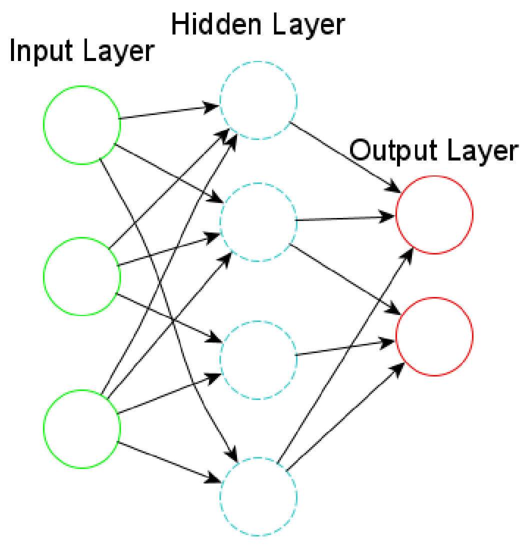


Figure 2.3: Example of a Neural Network

As it can be seen in Figure 2.3, it is always divided in layers in which only the output and input layers are visible from the external world, as consequence the internal layers are called hidden layers. When an input vector is applied, it will propagate from the left side of the network to its right side through the layers and the neurons which compose each layer. It is worth to mention that layers may perform different kind of computation on their inputs. Moreover, the deep neural networks are named after the huge amount of hidden layers.

In the early stages of ANNs the goal was to solve problems as the human brain would do. However, over time, the aim moved to perform specific tasks, leading to a different architecture of the biological brain and brain-inspired networks (Spiking Neural Networks).

Depending on how the edges are connected and the topology, a Neural Network can be classified in several sub-types:

- Feed forward, the data move only from input layer to output layer without cycles in the graph.
- Regulatory feedback, it provides feedback connections back to the same inputs that active them, reducing requirements during learning. It also allows learning and updating much easier.
- Recurrent neural network, it propagates data backward and forward, from later processing stages to earlier stages.
- Modular, several small networks cooperate or compete to solve problems.
- Physical, it is based on electrically adjustable resistance material to simulate artificial synapses.

Spiking Neural Networks

Spiking neural networks (SNNs) are artificial neural networks that more closely mimic natural neural networks [**article:1**].
In addition to neuronal and synaptic state, in their operational model, SNNs adds the concept of time. The idea is that neurons in the SNN do not activate at each propagation cycle but rather activate only when specific value is reached.
The current activation level is modeled as a differential equation and it is normally considered as neuron's state.

In principle, SNNs can be applied to the same application of Artificial Neural Networks. Moreover, SNNs can model brain of biological organisms without prior knowledge of the environment. Thus, SNNs have been useful in neuroscience for evaluating the reliability of the hypothesis on biological neural circuits but not in engineering.

SSNs are still lagging ANNs in terms of accuracy, but the gap is decreasing and has vanished on some task[**article:2**]. However, computer architectures based on SNN have a huge energy footprint compare to other types of architecture [**paper:44**].

2.3 Machine Learning Quantization

The reduction of computation demand, the increase of power efficiency and the memory footprint of machine learning algorithms can be achieved through the quantization.

Quantization is basically a set of techniques which convert, and map, input values from a large set to output values in a smaller set.
The idea of Quantization is not recent, it has been introduced since the birth of digital electronics. Imagine taking a picture with the phone's camera, the real world is analog and the camera is capturing the analog world and converting it into a digital format. Nevertheless, the high quality of nowadays pictures, quantization is not lossless.

A trivial quantization example for Neural Network model is given in the below figure, where a set of potentially infinite value(floating-point) are mapped to finite values (integer).

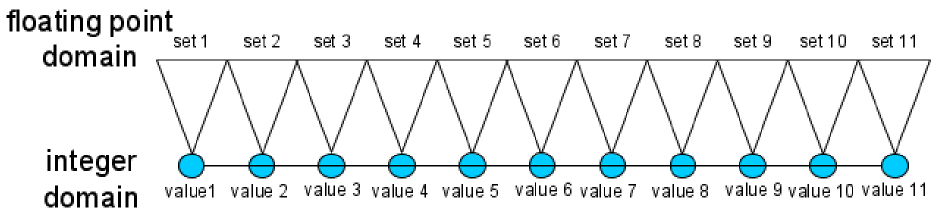


Figure 2.4: Approximation of floating-point values to integer values

It has been proved that even if the model has been quantized, for example from fp32 to integer32, its accuracy is still good and the accuracy drop between the two data representation is negligible [**paper:8**].
Several quantization techniques can be applied, together or separated, to already trained ML models (post-training quantization):

- Linear quantization: data are directly scaled by taking their maximum value and normalizing them to falling in the desired range.
- Outlier channel splitting [**paper:46**] : linear quantization is sensitive by large inputs. The idea of OCS is to reduce the value of outliers (for both weights and activations) duplicating the node with halving the output or the weight. This transformation leaves the node functionality equivalent while at the same

time it narrows the weight/activation distribution allowing a better linear quantization.

- Analytical Clipping for Integer Quantization [**paper:47**]: it represents the state-of-the-art for the post-training quantization techniques. It basically consists into apply a clipping function in a given range in order to reduce the quantization noise.

On the other hand, a quantization-aware training can also be done [**paper:45**]. Quantization has lead to a relief of hardware computation, as it is very well-known floating-point operation are much more expensive than integer operation from a lot of perspectives, and as consequence a reduction into the power consumption of the algorithm. It is also important to mention that the data traffic between the memory and the hardware is reduced due to the compaction of data.

Nowadays, edge devices take advantage of lower precision and quantized operations, including GPUs. Thus, quantization of machine learning algorithms is a defacto standard for edge inference.

2.4 Applications

In principle the AI can be applied to any intellectual tasks [**book:1**]. Focusing on machine learning applications, they can spread through a variety of different domains:

- Healthcare, mainly used for classification purposes.
- Automotive, used in self-driving cars.
- Finance and economics, to detect charges or claims outside the norm, flagging these for human investigation. In banks system for organizing operations, maintains book-keeping, investing in stocks and managing properties.
- Cybersecurity, automatically sort the data in networks into high risk and low-risk information.
- Government, for paired with facial recognition systems may be used for mass surveillance.
- Video games, it is routinely used to generate dynamic purposeful behavior in non-player characters.
- Military, enhancing Communications, Sensors, Integration and Interoperability.

- Hospitality, to reduce staff load and increase efficiency.
- Advertising, it is used to predict the behavior of customers from their digital footprint in order to target them with personalized promotions.
- Art, it has inspired numerous creative applications including its usage to produce visual art.

However, all the Machine Learning applications are characterized by the need of a huge amount of data set for the training process.

Chapter 3

State-of-the-Art

3.1 Overview

The role of Machine Learning has continuously growth in the past few years and a lot of efforts have been done for developing good software APIs in order to address different needs and domains.
In principle, all the machine learning algorithms can be run on the CPU, which already runs the OS and other Application Software. This leads to overheads, especially in terms memory accesses which are expensive in terms of energy and latency.

Analyzing machine learning algorithms comes evident that they massively do the same operations and access to data with some kind of patterns. Thus, with the outcome of the new paradigm for the GPU, the General Purpose GPU programming comes in handy that implementing those algorithms on a GPU, which matches the Machine Learning algorithms requirements regarding the massive operations and the reuse of data, has given a lot of advantages in terms of latency and energy efficiency. However, the capability of GPU of running machine learning algorithms has been pushed almost at the maximum with the increase of computation demands in modern neural networks. Therefore other solutions have been explored, such as the development of specific hardware platform.

3.2 GPU

The Moore's law is reaching the end from the point of view of CPUs. However, it seems that the GPUs can still carry on the Moore's law [**5496638**].
For this reason, improving efforts especially from the companies have been made for developing more and more GPUs with a higher performance per watts.

As already mentioned, with the income of general purpose GPU programming paradigm, more and more machine learning algorithms have been designed for being run on the GPU, gathering the best fruits given by that type of architecture.

As consequences, companies such as Nvidia have started to develop GPU for boosting machine learning applications performance.

3.2.1 Nvidia Ampere A100 Tensor Core GPU

The Nvidia Ampere A100 Tensor Core GPU has been announced recently and it is one of the most performant GPU. The newly added Tensor Core Unit allows massive increases in throughput and efficiency.
It is able to deliver up to 624 TFPLOPS[1] for training and inference machine learning applications.

The GPU is composed of multiple GPU processing clusters (GPCs), texture processing clusters (TPCs) and streaming multiprocessors (SMs). The core of the GPU is the Streaming Multiprocessor, which is built up from the SM of Volta GPU and Turing one.

Composed of integer, FP32, FP64 units and the Tensor Core Units are designed specifically for deep learning. It introduces also new data types in the tensor core for the computation such as binary, integer 8 and 4 bits, floating-point 64, 32, 16 and bfp16 (the throughput of the tensor core computation for fp16 and bfp16 is the same). The Ampere SM can achieve such efficient workload on mixed computation and addressing calculations thanks to an independent parallel integer and floating-point data paths.

Matrix-Matrix multiplication operations are at the core of neural network training and inference, and are used to multiply large matrices of input data and weights in the connected layers of the network. The idea is represented into the Figure 3.2 and compared to previous architectures.

The Ampere A100 GPU contains 108 Streaming multiprocessor, and 432 third generation Tensor Core. According to Figure 3.3 the Tensor Core Units are able to compute multiplications on FP16 and accumulate on FP32, leading to a further reduction of latency and energy consumption.

A novel approach for doubling the throughput of deep neural networks has been

[1]floating-point operations per second

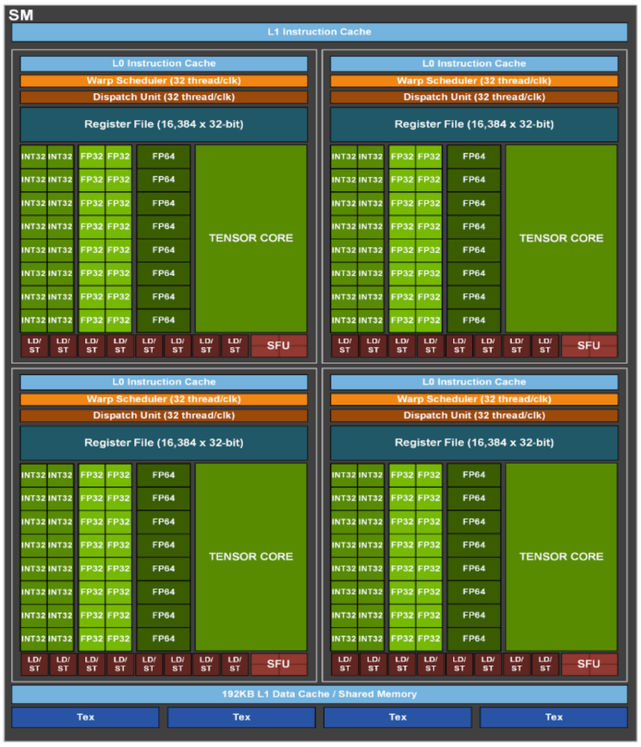

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

Figure 3.1: Streaming Multiprocessor Architecture [**paper:41**]

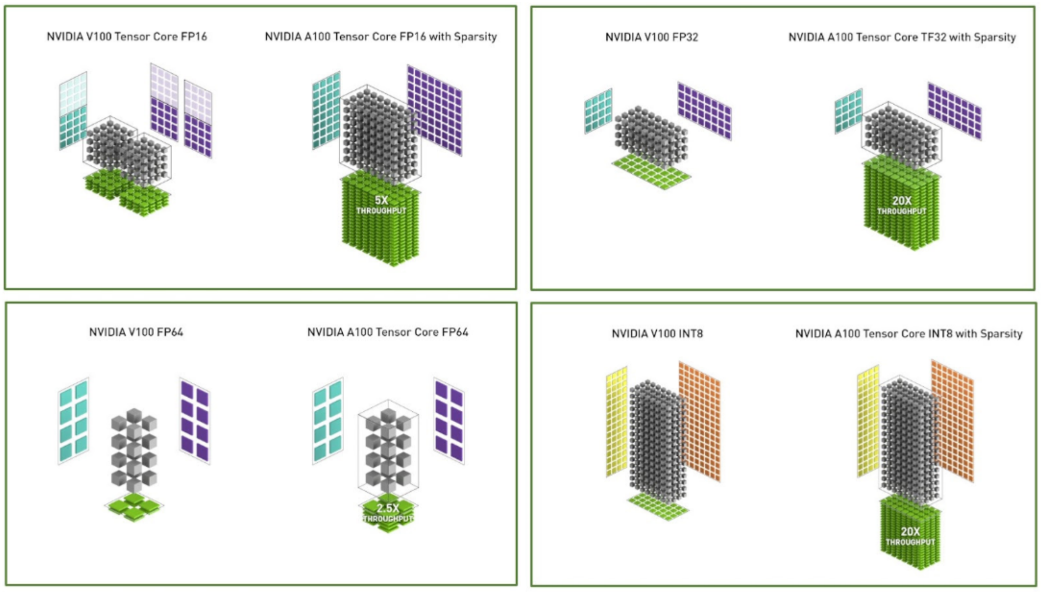

NVIDIA V100 Tensor Core FP16
NVIDIA A100 Tensor Core FP16 with Sparsity
5X THROUGHPUT
NVIDIA V100 FP32
NVIDIA A100 Tensor Core TF32 with Sparsity
20X THROUGHPUT
NVIDIA V100 FP64
NVIDIA A100 Tensor Core FP64
2.5X THROUGHPUT
NVIDIA V100 INT8
NVIDIA A100 Tensor Core INT8 with Sparsity
20X THROUGHPUT

Figure 3.2: Matrix Multiplication in Tensor Core [**paper:41**]

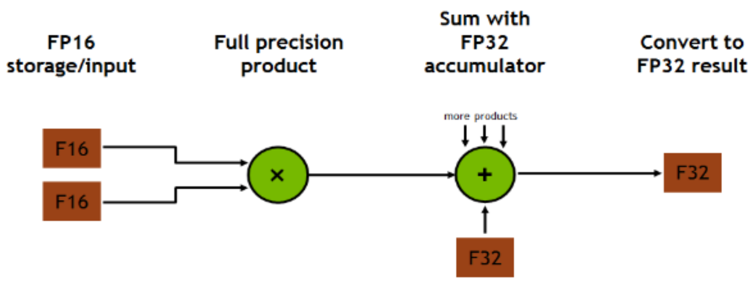


Figure 3.3: Mixed Precision Schema of a FMA unit in Tensor Core Unit [**paper:41**]

introduced in this architecture. At the end of training process, only a subset of the total weights are necessary to execute a neural network correctly. As consequence not all the weights are needed, and they can be removed.
Based on training feedback, weights can be adapted at runtime during the training and this does not have any impact on the final accuracy. Thus, thanks to the sparsity of weight tensors., inference process can be accelerated. In addition, also the training process can be accelerated exploiting the sparsity idea but it has to be introduced at the beginning of the process for achieving some benefits.

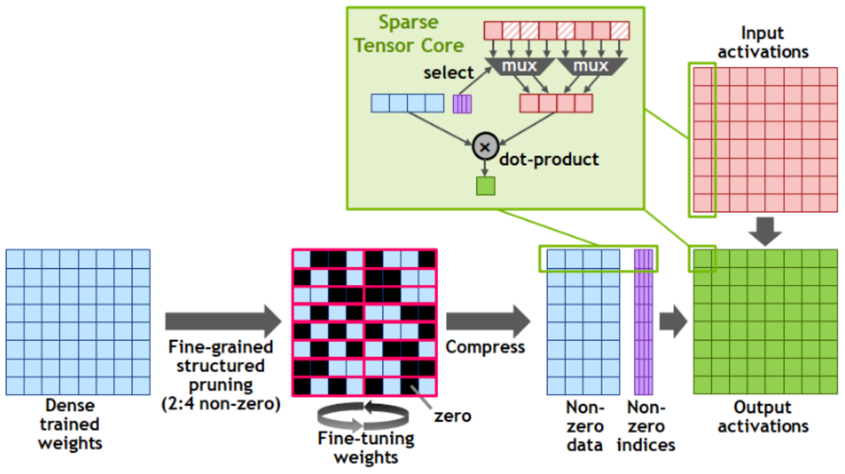


Figure 3.4: Sparsity Optmization of a weight tensor [**paper:41**]

The apporach in Figure 3.4 doubles the throughput by skipping the zeros. It also leads to a reduction of memory footprint and an increase into the memory bandwidth.

Following the idea, NVIDIA has introduced a new set of instruction for inference: sparse Matrix Multiply-Accumulate (MMA). Those instructions are able to skip the matrix entries which contain zero values, leading to an increase of the Tensor core throughput. An example can be seen in Figure 3.5, where the light blue matrix has a sparsity of 50%. It is also important to mention that the non-zero entries of the light blue matrix will be matched with the correct entries of the red one.

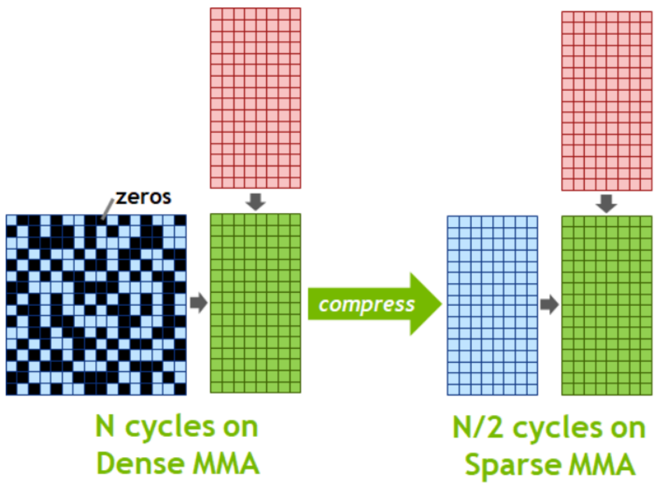


Figure 3.5: Matrix Multiply Accumulate [**paper:41**]

The deep learning frameworks and the CUDA Toolkit include libraries that have been custom-tuned to provide high multi-GPU performance for each one of the following deep learning frameworks in the Figure 3.6.

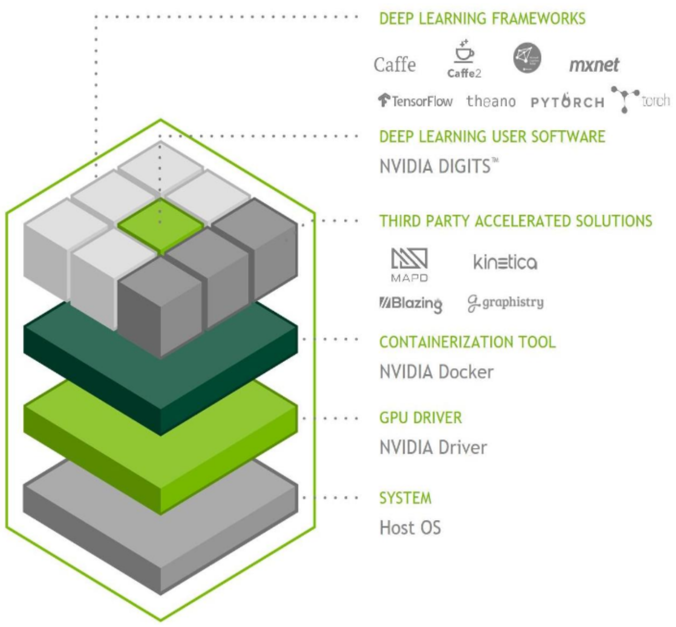


Figure 3.6: Software stack [**paper:41**]

Combining powerful hardware with software tailored to deep learning, it provides to developers and researchers solutions for high-performance GPU-accelerated deep learning application development, testing, and network training.

3.3 Domain Specific Hardware Platform

Instead of developing GPUs also suitable for Machine Learning applications, the companies have designed and deployed special purpose hardware accelerators.

3.3.1 NVDLA

The Nvidia Deep Learning Accelerator is a free open source hardware platform from Nvidia, highly customizable and modular, which allows to design and deploy deep learning inference hardware.

The architecture comes in two configurations:

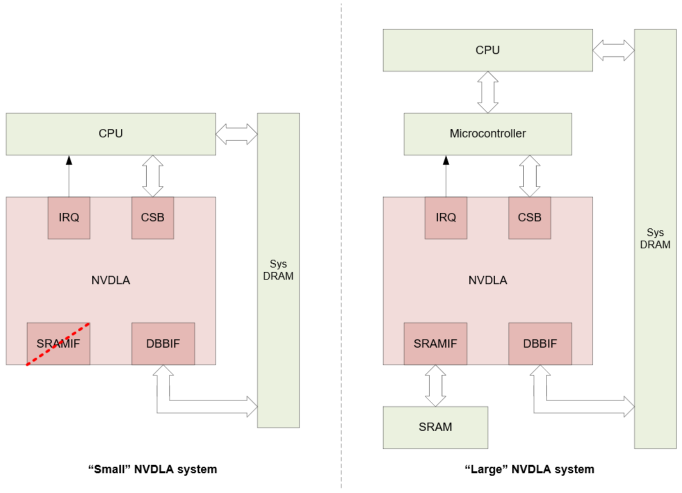


Figure 3.7: Comparsion of two possible NVDLA system [**WEBSITE:8**]

As already mentioned, the aim of the work is to develop a hardware accelerator for machine learning suitable for mobile devices. Therefore from now on the NVDLA small system will be considered and analyzed.
The internal architecture of the NVDLA small system is:

According to Figure 3.7, for the Small configuration of the accelerator, the processor will be in charge of programming and scheduling the operations on the NVDLA and as consequences handles the start/end of operations and possible interrupts, all of them through the CSB (Configuration Space Bus) interface which is AXI Lite compliant[**paper:30**].
The data movement to/from memory are handled by the Internal memory controller through the DBB (Data BackBone) interface, which is AXI [**paper:30**] compliant.

The internal architecture of NVDLA is composed by various engines. Each one of them is able to perform specific Machine Learning operations:

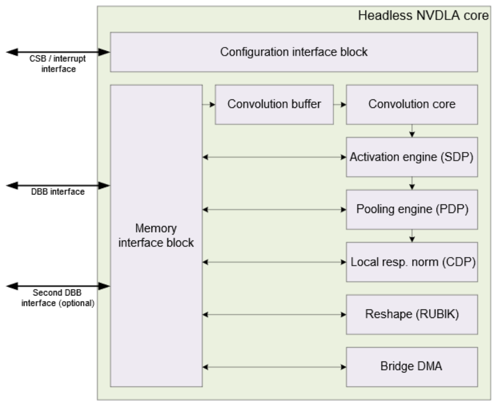


Figure 3.8: Internal architecture of NVDLA small system, Secondary DBB not considered [**WEBSITE:8**]

- Convolution Core: it comes in pair with the Convolution Buffer, its private memory for the data (inputs and weights). It is used to accelerate the convolution algorithms.
- Activation engine (Single Data point Operations): it performs post processing operations at the single data element level such as bias addition, Non-linear function, PReLU (Parametric Rectified Linear Unit) and format conversion when the software requires different precision for different hardware layers.
- Pooling engine (Planar Data Operations): it is designed to accomplish pooling layers, i.e. it executes operation along the width and height plane.
- Local response normalization engine(Cross Channel Data operations): it is designed to address local response normalization layers.
- Reshape(Data memory and reshape operations): it transforms data mapping format without any data calculation.
- Bridge DMA: it is in charge of copying data from the Main Memory to the SRAM of the accelerator, only available into the large configuration of the system.

Another possible configuration which is worth to mention is the possibility to let the engines work separately on independent task or in a fused fashion where all of them are pipelined, working as a single entity.

According to developers the configurability of the cores ranges from arithmetic precision to the theoretical throughput that a single unit can achieve (increasing the number of internal Processing Elements). Moreover, since the engine units are independent of each other, according to the application and the model used they can be safely removed from the design.

NVDLA Software

It is also worth to mention that the accelerator comes already with a basic software stack:

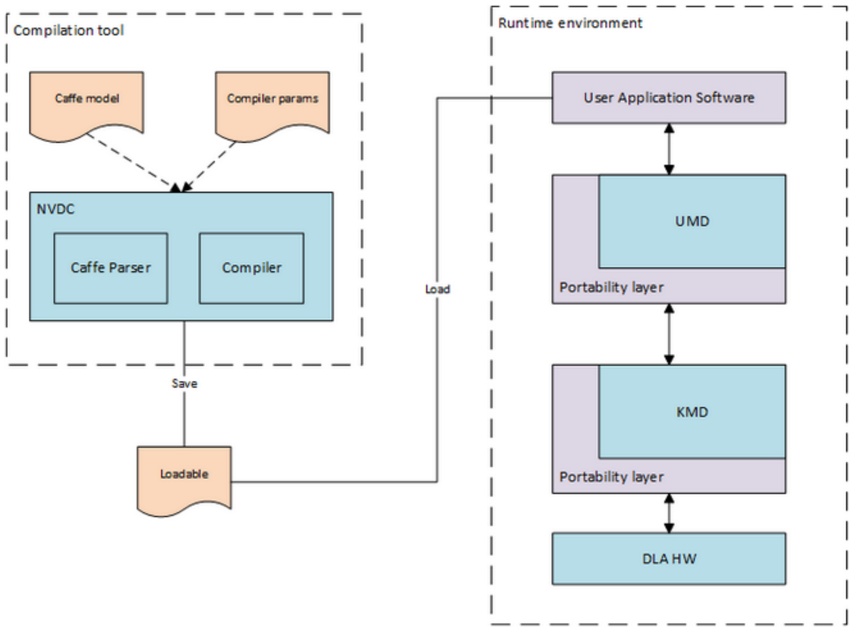


Figure 3.9: NVDLA Software stack[**WEBSITE:7**]

The Compilation tools are in charge of converting existing pretrained model into a set of hardware layers (for the desired precision) and programming sequences suitable for the NVDLA. The output of this process is a Nvidia Loadable file suitable for the runtime environment.
Regarding the runtime environment, it has been designed for a system in which is present an OS. It is composed in two parts: the User Mode Driver (UMD) and the Kernel Mode Driver (KMD).
The User Mode Driver loads the loadable file in memory and submits the operation to the KMD. It is also in charge of data movement from/to the accelerator.
The KMD is in charge of submitting operations to the accelerator through low level functions, scheduling the operations and handling the interrupts.
Both the KMD and the UMD are wrapped into portability layers which are, respectively, hardware dependent and OS dependent. In principle, for migrating the software to another OS or hardware plaftorm it is enough to modify only the portability layers.

3.3.2 Google TPU

Google developed its own application-specific integragrated circuit for neural networks, which is tightly integrated with TensorFlow Software. It includes:

- Matrix Multiplier Unit (MXU): 65,536 8-bit multiply-and-add units for matrix operations
- Unified Buffer (UB): 24 MB of SRAM that work as registers
- Activation Unit (AU): Hardwired activation functions

In Figure 3.10 a general view of TPU architecture is presented.

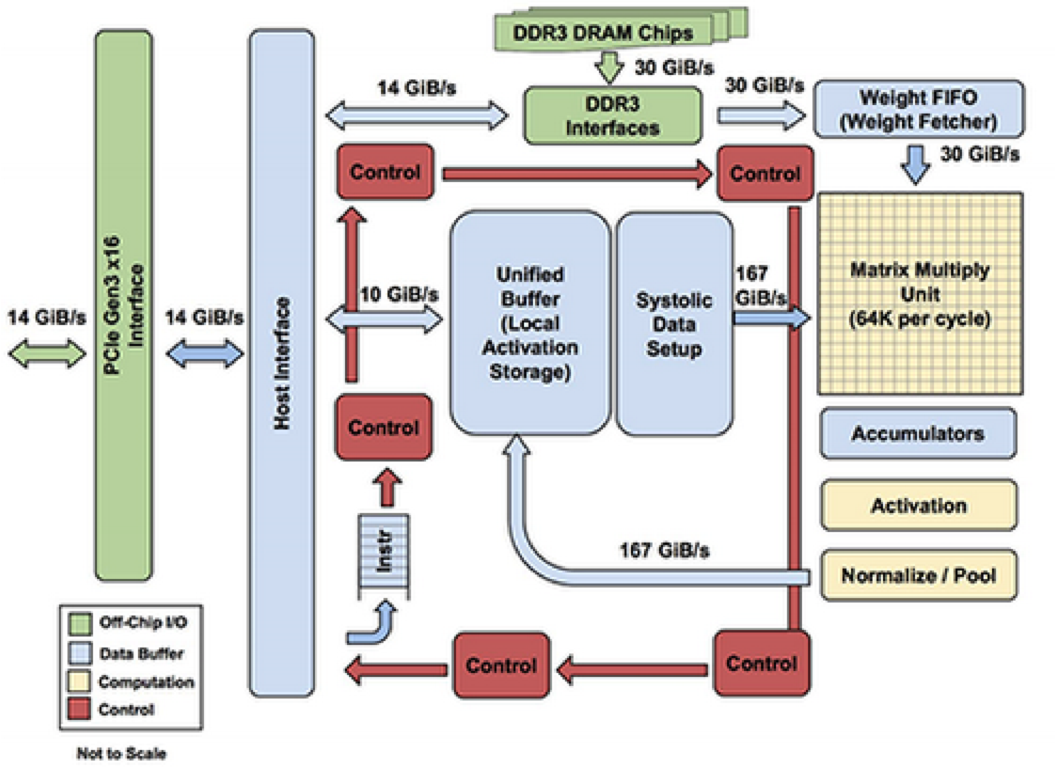


Figure 3.10: Google TPU architecture[**paper:40**]

Rather than be tightly integrated with a CPU, the TPU is designed to be a coprocessor in which the instruction are sent by the host server rather than fetched.

The matrix multiplication unit reuses both inputs many times as part of producing the output, avoiding the overhead of continuously read data from memory.

Only spatial adjacent ALU are connected together, which makes wires shorter and energy-efficient. The ALUs only perform computations in fixed pattern.

As far as concerned the software stack, the TPU can be programmed for a wide variety of neural network models. To program it, API calls from TensorFlow graph are converted into TPU instructions.

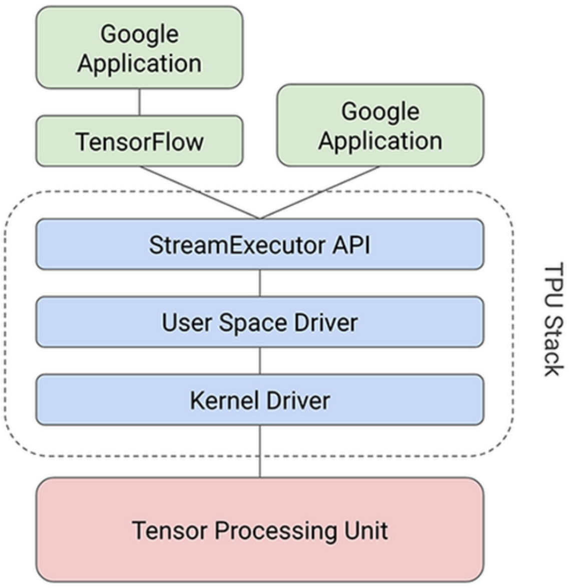


Figure 3.11: Google TPU Software Stack [**WEBSITE:9**]

3.3.3 Habana Goya HL-1000

Habana's Goya is a processor dedicated to inference workloads. It is designed to deliver superior performance, power efficiency and cost savings for data centers and other emerging applications.
It allows the adoption of different deep learning models and is not limited to specific domains. Moreover, the performance requirements and accuracy can be user-defined.

In Figure 3.12 a high level view of the Goya architecture can be appreciated.

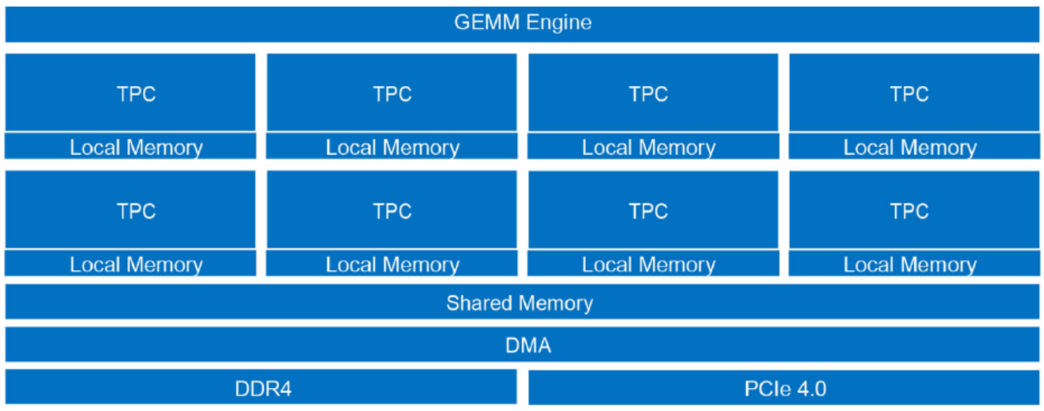


Figure 3.12: High level view of Goya architecture [**paper:38**]

It is based on scalable, fully programmable Tensor Processing Cores, specifically designed for deep learning workloads.
It also provides other flexible features such as GEMM operation acceleration, special functions dedicated hardware, tensor addressing, latency hiding capabilities and different data types support in TPC (FP32, INT32, INT16, INT8, UINT32, UINT16, UINT8).

Regarding the software stack, it can be interfaced with all deep learning frameworks. However, a model has to be first converted into an internal representation, as it can be seen in Figure 3.13.

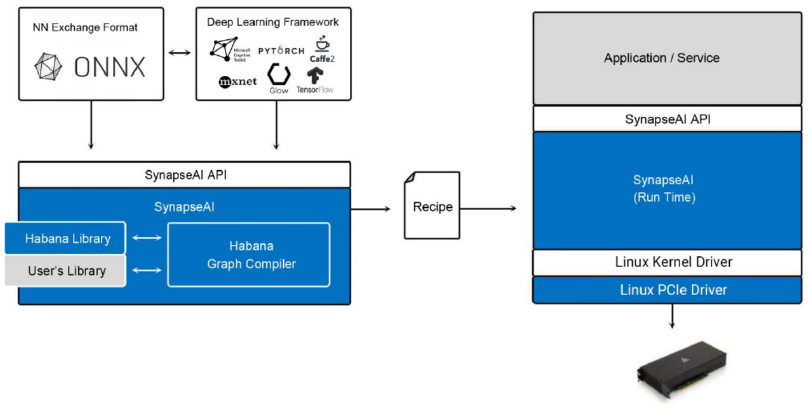


Figure 3.13: Habana Goya Software Stack [**paper:38**]

It also supports quantization of models trained in floating-point format with near-zero accuracy loss.

Chapter 4

System Development

4.1 Overview

As already mentioned, the use of custom hardware for a specific application can have big benefits especially in terms of energy consumption and latency. The inference process of Neural Network is mainly characterized by massive multiply and addition operations. Fetch of data from main memory follows patterns and it has been proved that those data, in particular weight data, are reused for several executions of the Neural Network model. As consequence, executing a Neural Network model on a von Neumann based architecture machine leads to performance degradation, even in a cache-based system, since the CPU has to request the data from the main memory, execute the operation on those data and then save back to main memory before moving to the next data. The introduction of vectored instruction in the modern processors can have a slight impact in the performance benefits. However, the drastically increase of layers in the Neural Network has made them suitable for several applications. This it can be translated into a massive increase of operations for executing them, as it can be also observed in the following Figure:

Following the fast demands of operations into a Neural Network, it becomes evident that executing them on a CPU could not meet real-time application requirements.
Instead, the designed accelerator has a Dataflow architecture, with emphasis on weight data reuse, and it is able to execute a tensor convolution. The basic idea is a computation matrix composed in every entry of processing elements which are able to perform operation between the incoming data and the weights, which have been already loaded for exploiting a data reuse approach.

The custom hardware accelerator is not useful as it is. It has to be integrated into a ML-Framework in order to appreciate its benefits. After a preliminary

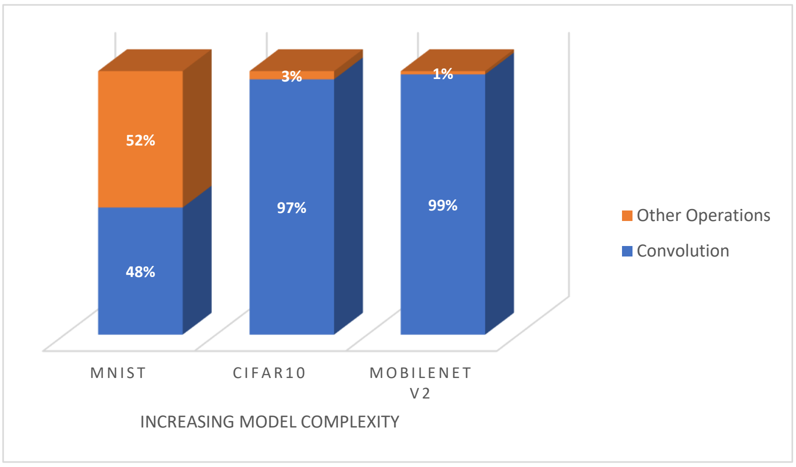


Figure 4.1: Average execution time divided by type of operations

research on which ML-Framework would allow to integrate a custom hardware accelerator minimizing the efforts to change the model code and its definitions, it has been evident that the TensorFlow Framework, an end-to-end open source Machine Learning platform [**WEBSITE:4**], suits the needs.

The workflow of the Hardware-Software development is illustrated in the following:

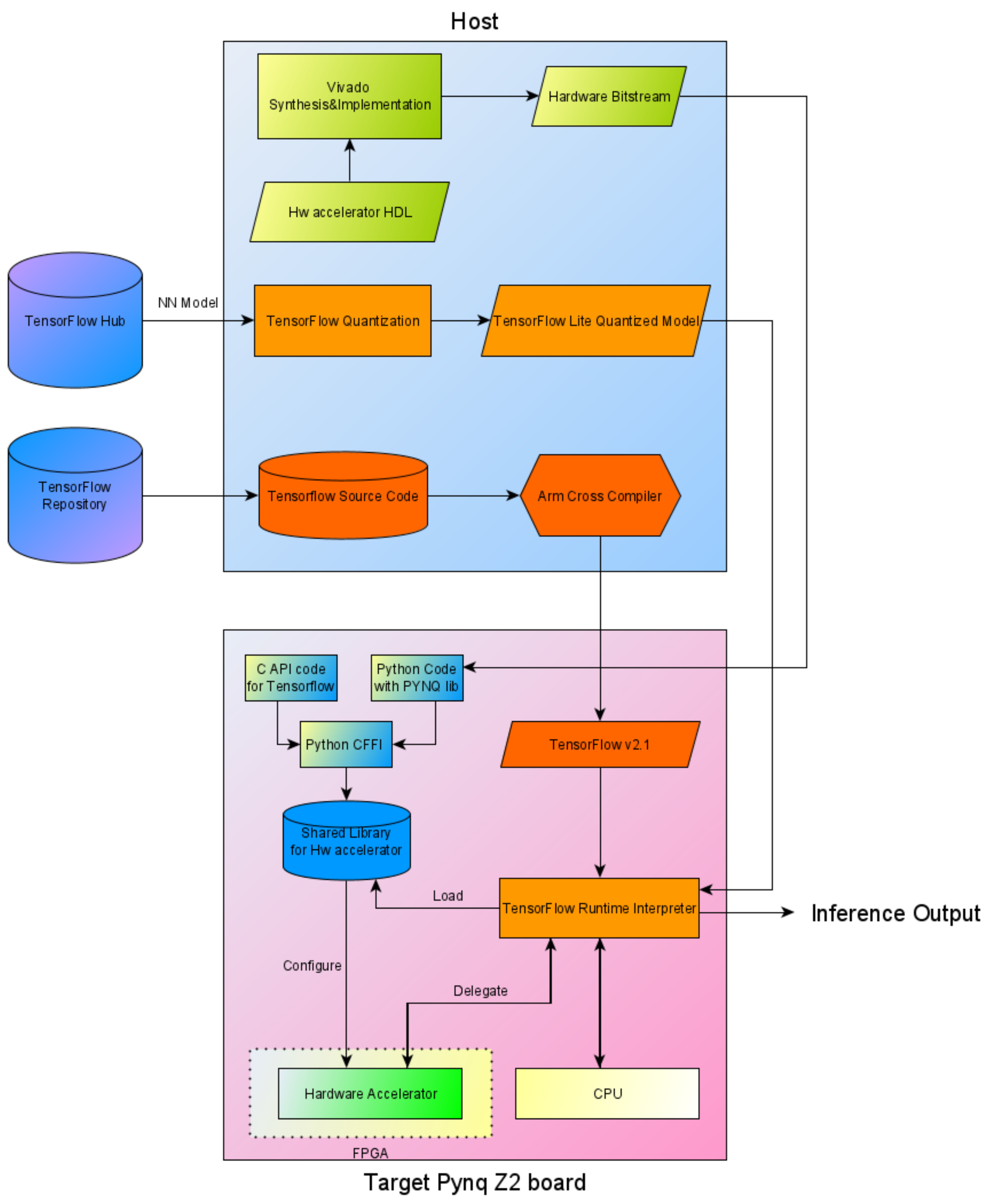
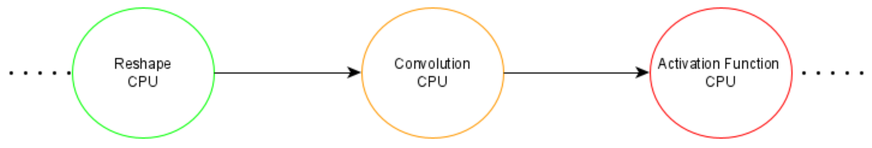


Figure 4.2: Development workflow

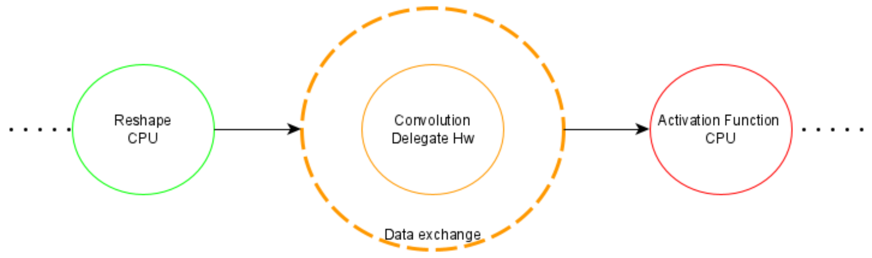
The entire work is implemented on a PYNQ Z2 board from TUL, based on a Zynq-7000 SoC [**paper:31**]. In order to speed-up the development process and use built-in library for the AXI protocol and the DMA transfers, the software is partially carried out through the PYNQ environment of the board [**WEBSITE:2**] based on Python which has became a de facto standard [**paper:37**].
The usage of Python as basic software allows to easily integrate it with high level Machine Learning Framework, such as TensorFlow in this case.

4.2 Software

The focus of the work is the inference process, pre-trained models are needed and TensorFlow Hub [**WEBSITE:5**] comes in handy for this purpose. It provides already pre-trained Machine Learning models for different domains. Moreover, TensorFlow has the feature of quantizing a post-trained model for different arithmetic precision. In the Fig. 4.2 it can be seen that the quantization process has been done offline.
The choice of using the stable release 2.1 of TensorFlow is dictated from the possibility of using Delegates (aka hardware accelerators or GPUs) in its Neural Network model. A delegate is a way to delegate part or all graph execution to another executor. Every model is represented, internally, as a graph (with its relative order of execution for the nodes) and every node of the graph is described as a set of operation that has to be applied to the node's input. As every node is described by a set of operations, it is easy to understand which part of the graph can be executed on the accelerator in advance, and this operation is done at the beginning when both the model and the accelerator library is loaded as it is represented in the following Figures: It is worth to mention that TensorFlow is open-source and



(a) without delegate



(b) with delegate

Figure 4.3: Execution Graph

since no binary installations for its 2.1 release are provided for Arm processor, it has been cross-compiled from scratch for the PYNQ-Z2 board.

TensorFlow demands as library for the accelerator a C Python-API compatible shared library. In addition, the code for using the accelerator was already written using the PYNQ environment in Python. Therefore, for allowing code reuse and decreasing the development time the Python code has been embedded in the C code (from a TensorFlow example of the delegate library), adding callbacks to Python code[1]. This has been possible thanks to the Python library *CFFI* (C Foreign Function Interface) [**WEBSITE:14**], which is also able to provide a shared library Python-API compatible as output. In the following Figure the flow chart between Tensorflow Lite and the accelerator library can be seen:

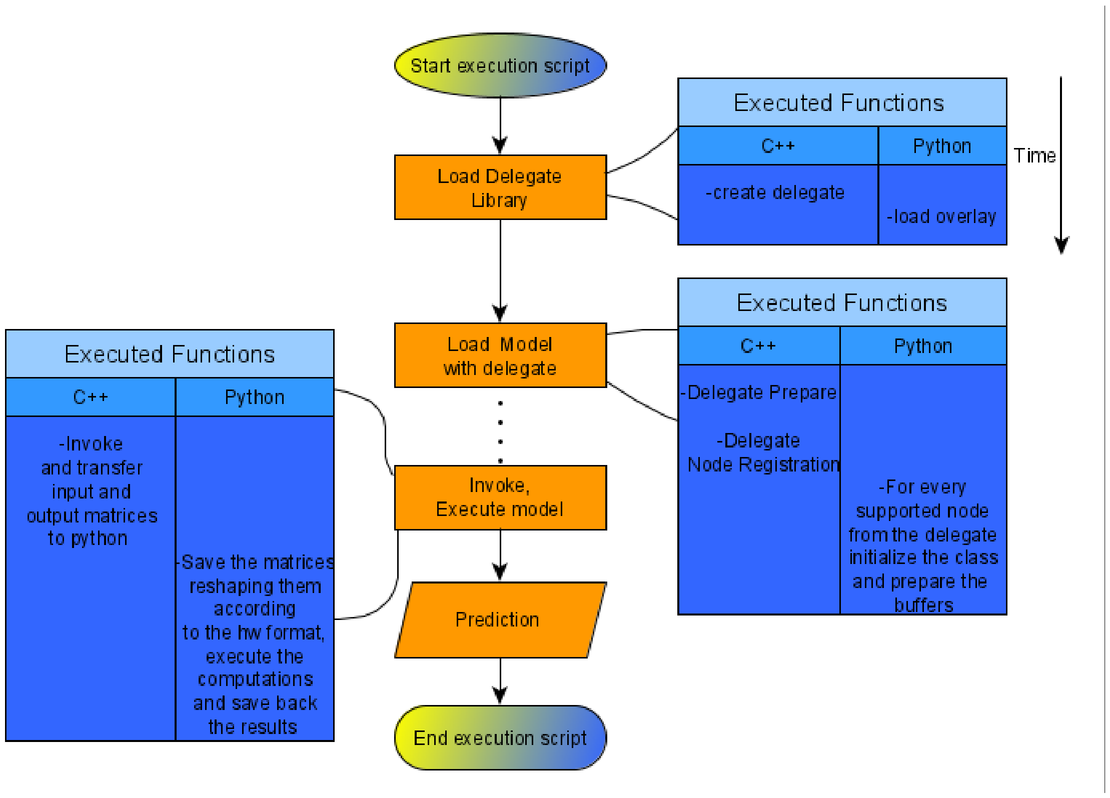

Start execution script
Load Delegate Library
Load Model with delegate
Invoke, Execute model
Prediction
End execution script
Executed Functions
C++
Python
-create delegate
-load overlay
Time
Executed Functions
C++
Python
-Delegate Prepare
-Delegate Node Registration
-For every supported node from the delegate initialize the class and prepare the buffers
Executed Functions
C++
Python
-Invoke and transfer input and output matrices to python
-Save the matrices reshaping them according to the hw format, execute the computations and save back the results

Figure 4.4: Flow Chart with accelerator

[1]See Appendix A

4.3 System Level

As it can be seen from Figure 4.5, it is divided in two big blocks:

- Processing System: The processing system (in Figure 4.6 referred as *processing system7*) is in charge of running the OS and the Machine Learning application. As consequence it also runs the necessary software for programming the accelerator registers and the data movement to/from main memory from/to the accelerator.
- Programmable Logic: The programmable logic (PL) hosts the entire design, from the accelerator itself to the DMAs and the AXI interconnections.

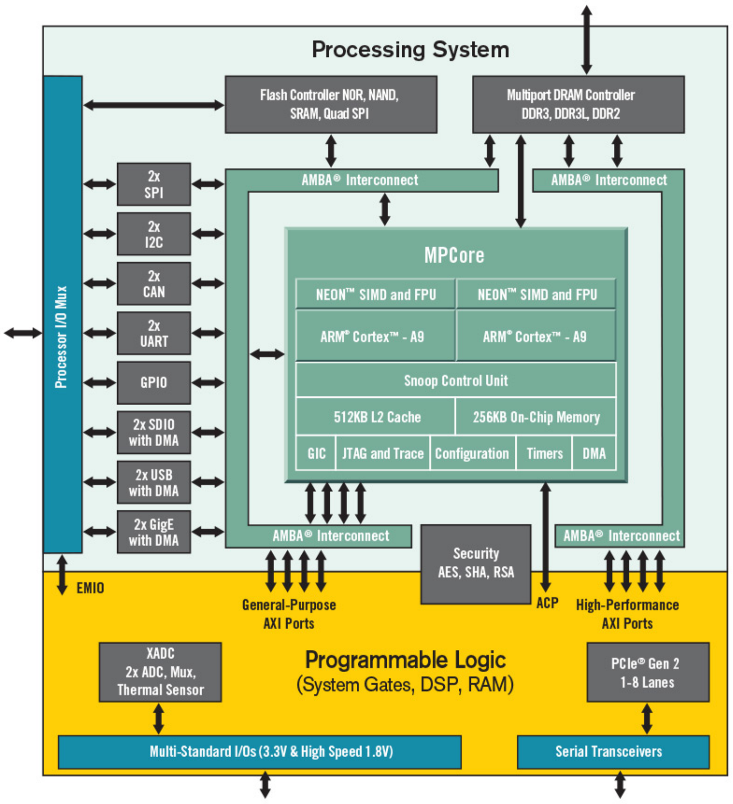


Figure 4.5: Zynq 7000 SoC [**paper:42**]

Furthermore, the Programmable Logic in Figure 4.6 is hosting:

- AXI interconnections: IP cores from Xilinx [**paper:34**] [**paper:35**] in order to connect and correctly address entities in the Programmable Logic.

- AXI DMA: IP core from Xilinx [**paper:33**] which allows data movement between main memory and accelerator memories. Several single channel DMA have been used instead of using a single DMA with multiple channels. The reason is that in the PYNQ environment only the drivers for the single channel DMA are provided.
- DTPU: the actual hardware accelerator.
- XADC: IP core from Xilinx [**paper:32**] which allows to measure the temperature of the SoC, the voltages and the currents at run time.

In the following figure, the schematic of the overall design in the PL is presented.

4.4 DTPU, the hardware accelerator

The hardware accelerator, named *Cogitantium*[3], *The Dumb Tensor Processing Unit*, is in charge of carrying out the tensor convolution of the neural network model, exploiting a data-flow architecture on the input data and a data reuse for the weight data.

Figure 4.7 presents the Logical block diagram of the accelerator.

4.4.1 Real Implementation

The work is not focused on developing embedded memories and AXI interfaces, therefore a Xilinx's IP core, which includes all those necessary sub components, has been used [**paper:43**] leading to the actual block diagram which can be observed in the Figure 4.8.

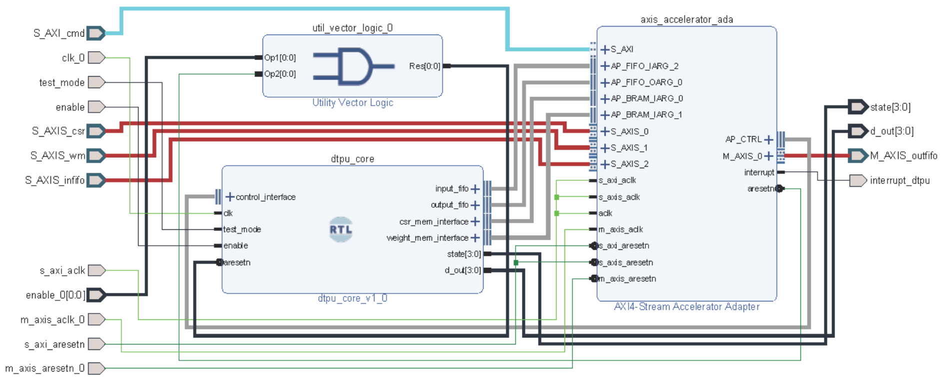


Figure 4.8: Real RTL view of DTPU accelerator

The latter has allowed to completely focus the work on the DTPU core[4], which has become:

[2]Except for the Zynq Processing system

[3] *Thoughtful*

[4]See Appendix B

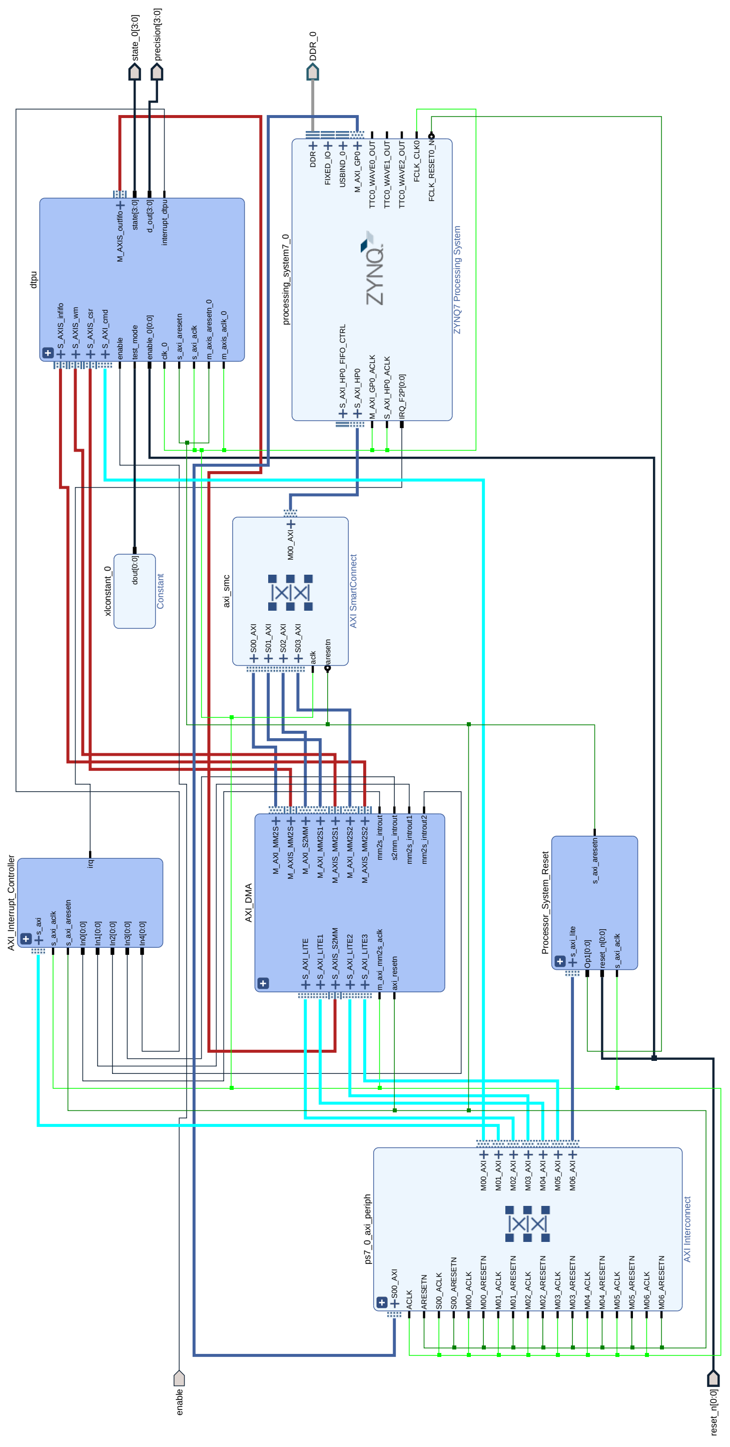


Figure 4.6: System view hosted in the PL [2]

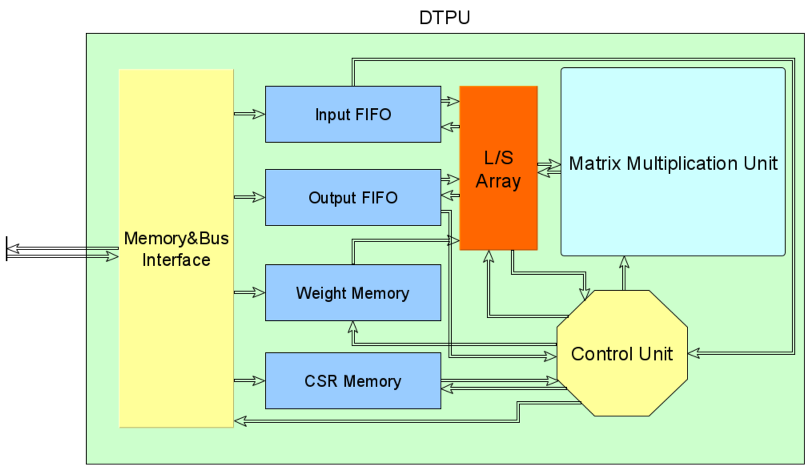


Figure 4.7: Logical view of DTPU accelerator

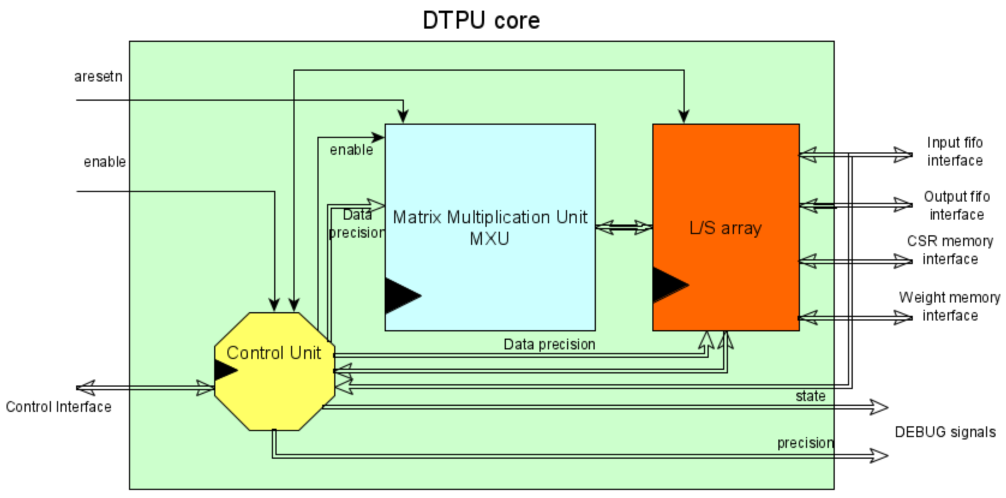


Figure 4.9: RTL view of DTPU core

Where the sub-units:

- L/S array provides the data for the Matrix Multiplication Unit, especially the weight data are reused across several executions and therefore loaded once.
- Control Unit is in charge of handling handshake signals for transferring the ownership of the data (data transferred by the DMA from the Main Memory), load the weights and activation in the respectively units and save the results to the output FIFO. Since it is a Data flow architecture, there is no control flow of the data in the core and this has allowed to keep the Control Unit as simple as possible.
- Matrix Multiplication Unit (Mxu) is the computation unit of the hardware accelerator. It executes the tensor convolution for different arithmetic precision.

4.4.2 High Level State Machine of Control Unit

The Dataflow architecture has allowed to design a Control unit as much simple as possible, presented in the below figure:

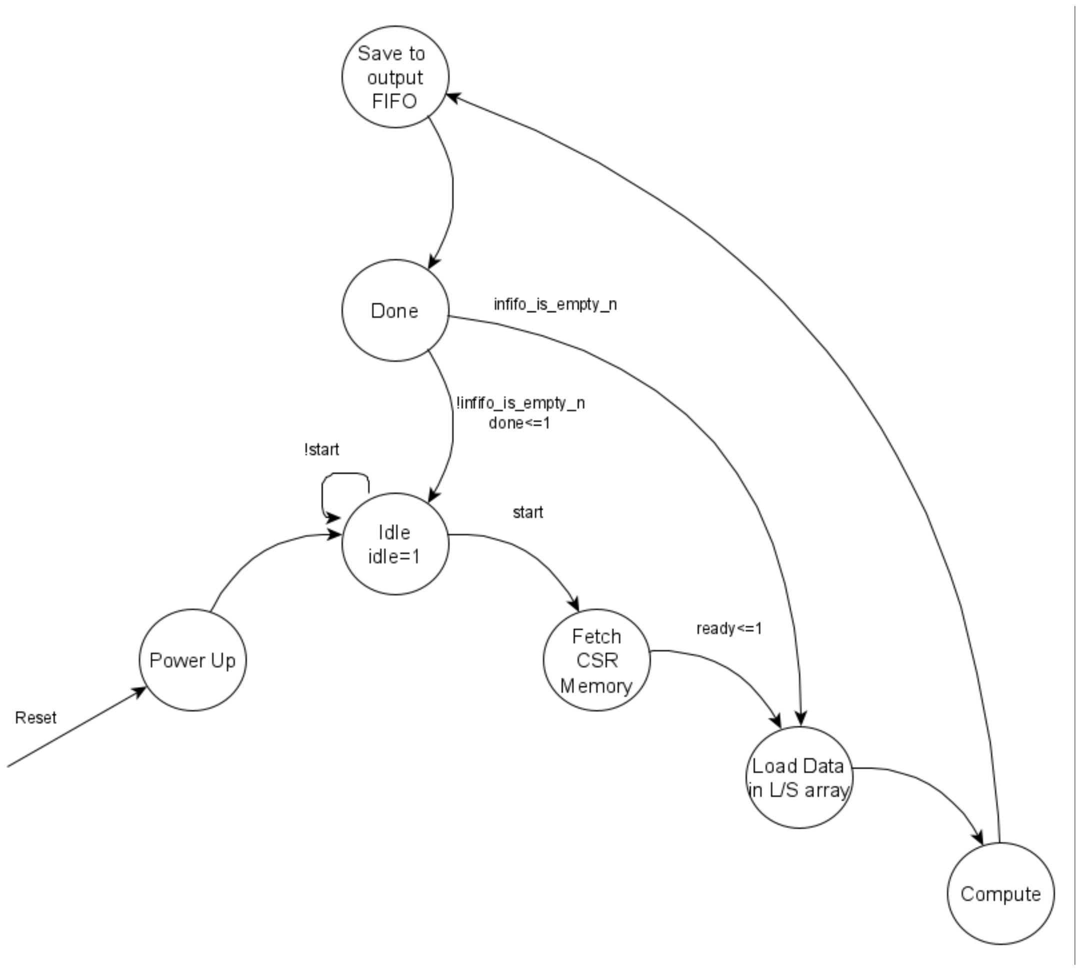


Figure 4.10: A high level view of Control Unit

In which:

- *Idle* state is waiting for the start signal from the *axis accelerator adapter* (generated when all the data have been transferred[5]).
- *Fetch CSR Memory* state is in charge of retrieving from the CSR memory the desired data precision for the computation and the starting address of

[5]Input Data, Weight data and CSR data

the weight memory. It also notifies to the *axis accelerator adapter* that it is ready[6].

- *Load data in L/S array* state loads the correct weight values (retrieved from the weight memory) and the activation data into the correct L/S unit. The number of active L/S unit is computed at run time. It depends on from the current required data precision and the fixed number of rows and columns in the MXU.

- *Compute* state activates the MXU and it waits the end of computation before committing the results to the output FIFO.

- *Save to output FIFO* state saves the data stored in the active L/S units to the output FIFO.

- *Done* state, depending on the input FIFO if it is empty or not, continues the computation for the next activation data or it returns to the idle state, notifying to the axis accelerator adapter the end of the computation[7].

4.4.3 Datapath

As it is well-known, the execution of ML models is memory intensive and it consists in massive multiplication and accumulation operations. In addition, it can be seen that during execution of ML models some memory location are accessed frequently. Therefore, it is evident that a DataFlow architecture which could exploit local data reuse and compute, massively, in parallel multiplications and additions could boost the performance. The DTPU core has been designed according to the previously mentioned ideas. The datapath of the core is presented in Figure 4.11 as block diagram.

[6]The ready signal is used as handshake between the core and the axis accelerator adapter for transferring the ownership of the data

[7]the notification for the end of computation allows the axis accelerator adapter to put the results on the output master axi stream interface in order to be transferred by the DMA

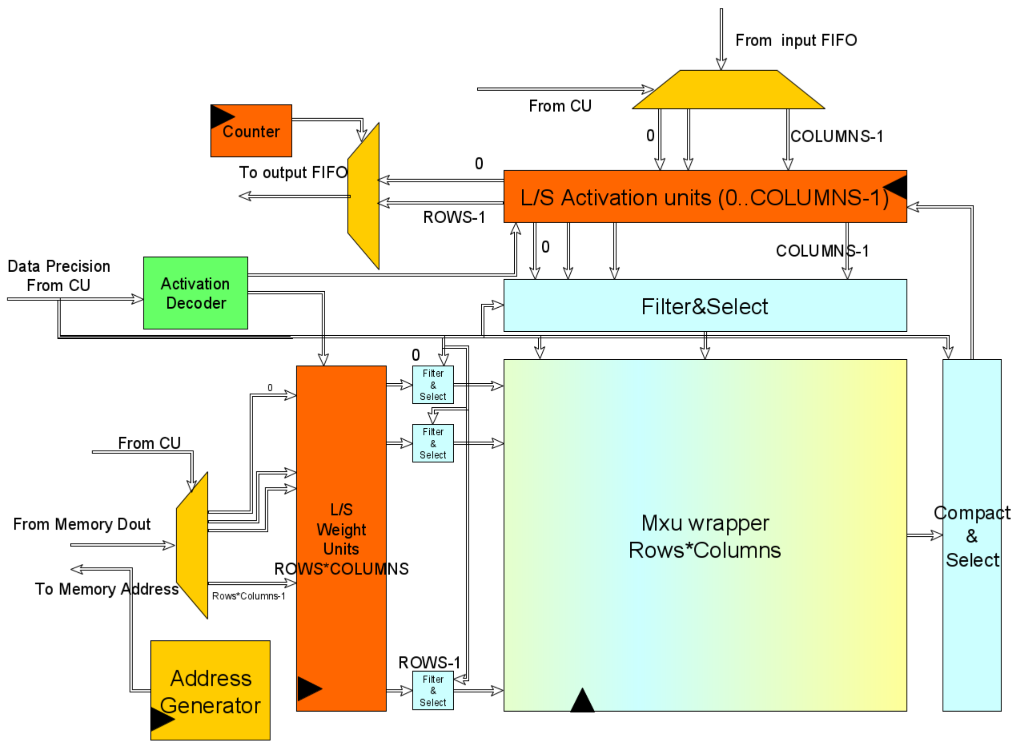


Figure 4.11: A detailed view of the DTPU core datapath. Enable and resets signals for clocked units has been omitted for improving readability.

In Figure 4.11, the brawn of the accelerator is the MXU wrapper, which contains the symmetric matrix of MACs with variable precision. Regarding the other blocks:

- Activation Decoder: It is able to generate the right activation signals for the L/S units, depending on from the current data precision and MXU size.
- Muxes and DeMuxes: Their purpose is to feed the right data from/to memory to/from the right units. The counter (from 0 to ROWS-1) in the Mux for the output FIFO is for saving at every clock cycle a data in the FIFO.
- Filter&Select: depending on the precision it provides the correct data to the correct computation units.
- Compact&Select: it is the complement of the Filter&Select unit. It is able to compact the output data from the MXU wrapper and feed the store registers.

- L/S weight Units: the name L/S has been kept for consistency even if it does not have any store process since the weight are only loaded once (stationary weights) and kept until a next full execution.

- L/S Activation Units: they are in charge of loading the data from the input FIFO into batteries of Flip-Flops while at the same time they can save the results to submit late in the output FIFO.

Filter&Select and Compact&Select

In principle, for each Processing element in the MXU wrapper a weight and an activation has to be provided (and as consequence it has to be provided from its relative Load Units). However, since the data width of memories and FIFO has been fixed to its maximum, 64 bits, it comes evident that during a computation with 8 bit integer it will fetch(and save for the output FIFO), in case of a 8x8 Mxu Size, 8 values from FIFO and 64 values from the weight memory. In this scenario all the Flip-Flops of the L/S units (both activation and weights) would sample values where the 56 upper bits are always unused leading to a waste of time for the memory accesses and energy for unused data.

A clever solution is to pack data before sending them to the accelerator. Nevertheless, the pack of data requires to internally unpack and, before committing to the output FIFO, pack the results. Unpacking and packing are done, respectively, by Filter&Select and Compact&Select units. Retrieving the previous example (computation on 8 bit integer, MXU size of 8x8 and 64 bit memory data) and using the approach of unpacking and packing, this leads to use only one L/S unit for activations (8 for the L/S weight units) for both the load and store operation. With one single L/S active unit and 8 bit integer computation, an 8 bit activation data has to be distributed for each column of the MXU, and this is done by the Filter&Select unit. For committing to output FIFO, results on 8 bit will be compacted in one single data of 64 bit by the Compact&select.

A visual distribution of the data can be seen in Figure 4.12. The same can be applied for each row of L/S Weigth units.

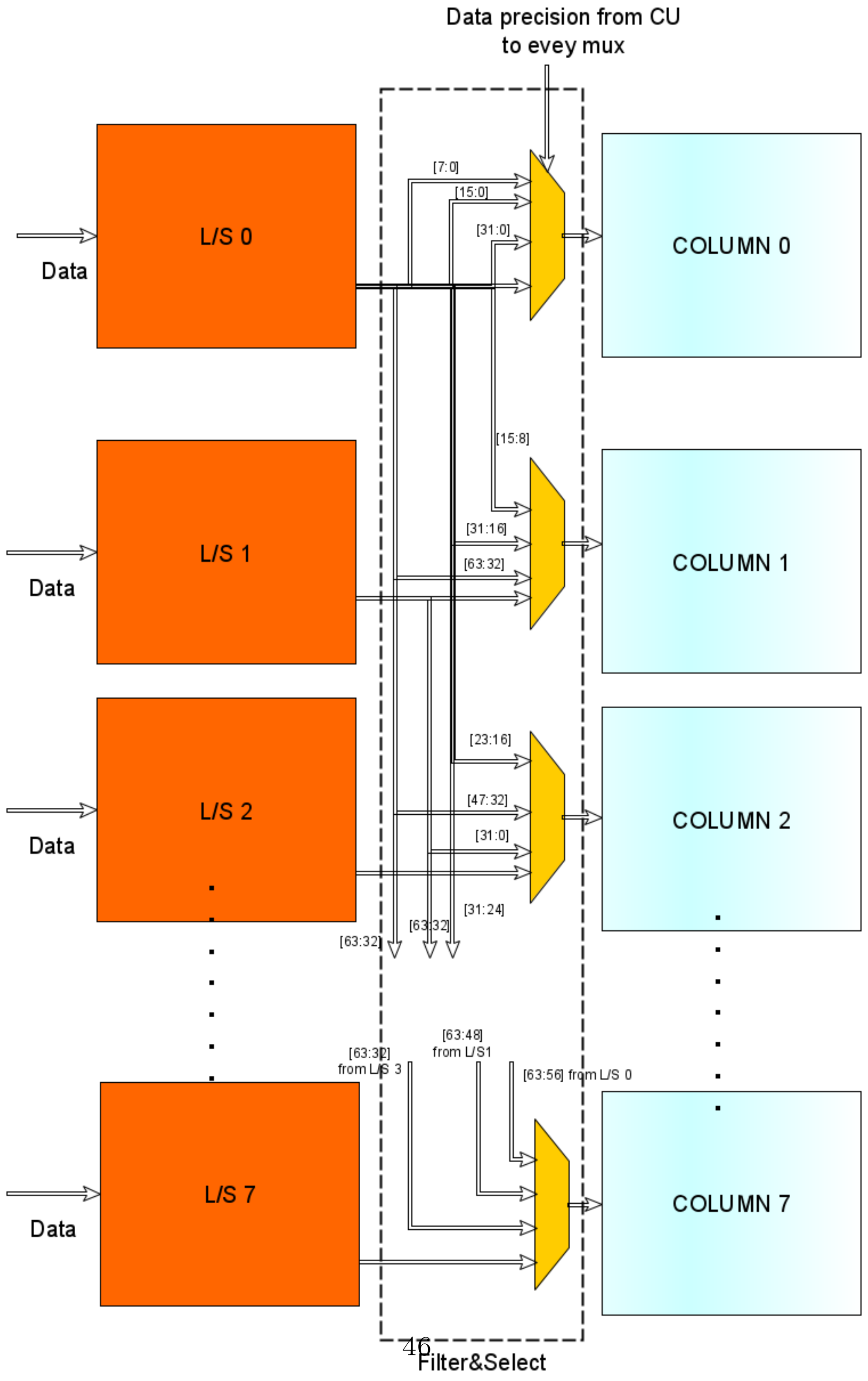


Figure 4.12: Data Distribution of Filter&Select unit for a MXU size of 8x8

In case the required precision is on 16 bit, with the same MXU size, two L/S units for activation are activated (2*ROWS for the L/S weight units) and will feed the respective Columns. The reason behind the two active L/S units is that in 64 bit, only 4 16-bit values can be packed. Increasing the MXU size, the L/S units are activated accordingly. For example, in case of a MXU size of 16x16 and integer 8 bit, two L/S units are activated (in case of integer 16 computation,4 units are activated).

This approach comes also with the overhead of packing and unpacking the data on the CPU but, on the other hand, the memory data movement is reduced and bandwidth increased, with a reduction in the energy consumption (thanks also to the reduced active L/S units).

It is also worth to mention that using size for the MXU which are power of two would maximize the memory bandwidth.

Matrix Multiplication Unit

The Matrix Multiplication Units (referred as MXU) is the muscle part of the accelerator, where the convolution is done. As the name suggest, it is organized as a Matrix:

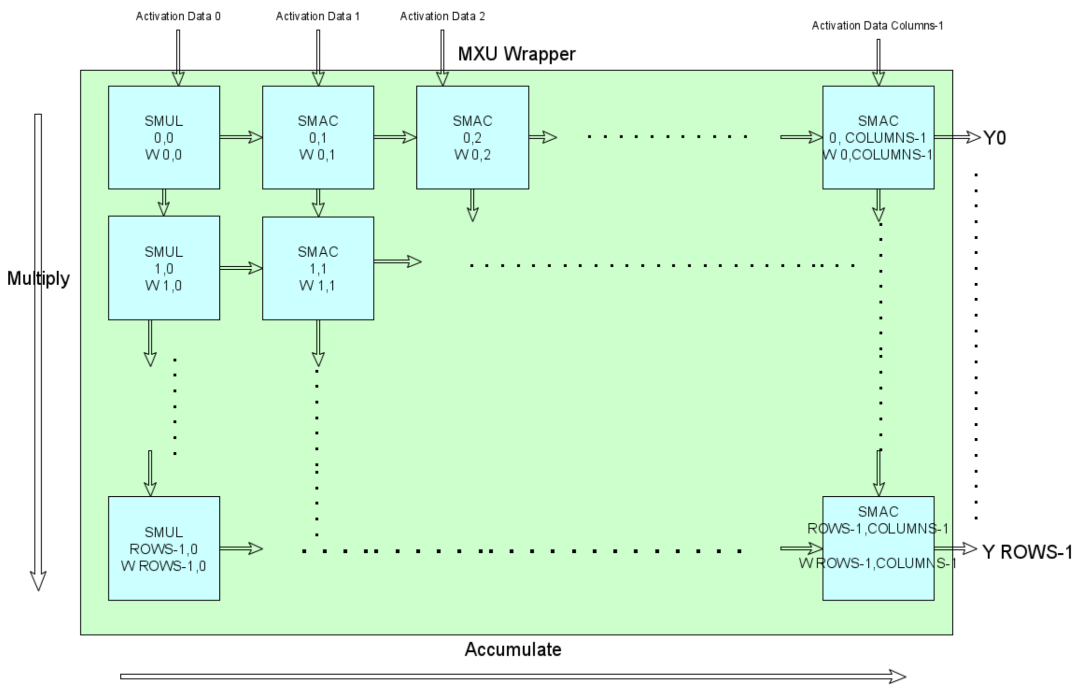


Figure 4.13: MXU interal structure and weights distribution

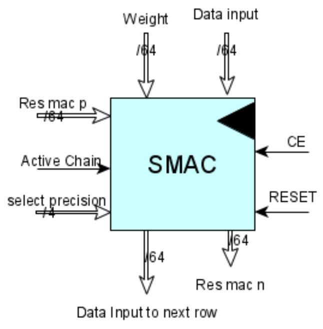
Every sub units has its own weight value (distributed thanks to the L/S weight combined with Filter&Select units, see Figure 4.11). It is a homogeneous unit, except for the first column, which does not accumulate. In addition, as it can be seen from the block diagram, there is no control flow between every Processing units. There is only data exchange from the previous unit to the next one (for both axis). This matrix configuration of the hardware allows to massive multiply and accumulate at the same time, in particular it can compute:

$$MAC_{OPS} = ROWS * COLUMNS \textit{ per \# clock cycle required for a single unit}$$
$$\text{with a } Throughput = Rows$$

The MXU can be synthesized with different criteria.
In particular, the Processing Elements can be independently generated for a single

data precision, from integer 8/16/32/64 to floating-point 32 or brain floating-point 16, or with some precision at the same time. Then data precision is decided, via software, and properly controlled using signal in Figure 4.14.

A detailed view of SMAC (Sub unit Multiply and Accumulate) and SMUL(Sub unit Multiply), the Processing Elements, is given in Figure 4.14.



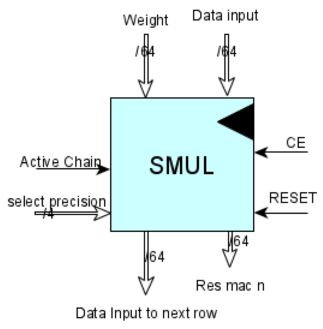


Figure 4.14: SMAC and SMUL details

It is important to mention that the sub units are always receiving data on 64 bits even if internally they may use all of them or not, depending on the value of *select precision* and *active chain* signals. For the full integer configuration (64 bit width operations) beside the possibility of computing for different data width (i.e. choose between 8/16/32/64) the Processing Elements can compute vectorized operations. With the help of *active chain* signal (active low, otherwise it is a 64 bit computation) and data width fixed to 64 bit, it is able to compute at the same time two 8-bit, one 16-bit and one 32-bit operations (multiplication for SMUL and multiplication and addition for SMAC). However, this comes with the overhead of correctly packing and unpacking the data on the CPU before transferring them to the accelerator.

SMAC and SMUL units have been designed, internally, using Vivado DSP primitives [**paper:48**], which a general schema can be appreciated in Figure 4.15:

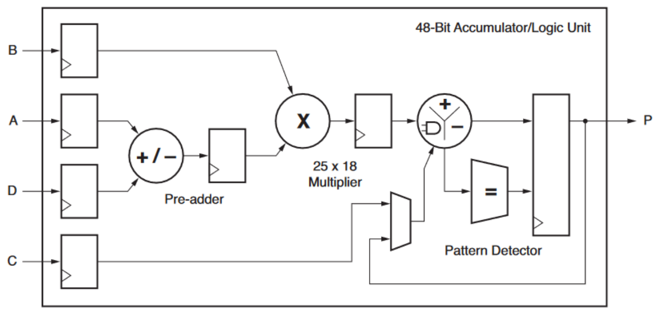


Figure 4.15: DSP Slice Functionality [**paper:48**]

Allowing fitting two computation (referring to SMAC) in one single unit[8] and maximize the resource utilization.

As soon as the Synthesis process reach the maximum value of DSP utilization, it does not switch automatically to use Fabric for those primitives. For maximizing the resource usage of the FPGA, the DSP primitives have been regenerated for both Fabric and DSP blocks. In this way, during the generation algorithm for the MXU, it uses primitives for DSP up to the maximum allowed value for the given board and then it starts to utilize Fabric. This approach has allowed almost a full utilization of the FPGA resources.

[8]Only for integer 8 and 16

Chapter 5

Conclusion

5.1 Discussion

A big portion of inference process for Neural Networks involves massive multiply and add computation, basic operation of tensor convolutions, and across several execution data, especially weight tensors, are reused. As consequence, for speeding-up and reduce the power consumption (especially in mobile devices) of ML models an hardware accelerator has been developed. It is also designed for accommodating different data type computation request from Neural Network models, ranging from integer8/16/32/64 to floating-point 32 and brain floating-point 16.

The approach of the work has been a hardware/software co-design in order to accommodate the high compute intensive request of Machine Learning, the tensor convolution. Therefore, the hardware core for tensor convolution has been developed from scratch, while the common components, such as memories and bus interface, have been chosen from the available ones in the tools. Moving one step at the time above in the abstraction level, the accelerator library has been developed and deployed. In order to accomplish it in a fixed time, the core of the library has been developed in Python, which has been interfaced with a C-code template provided from the developers of thee ML-framework used. This has lead to a hybrid library which encapsulates a frozen Python code layer, called from the C-code, the latter is only in charge of retrieving the data and passing them to the Python layer. Again moving one step above in the abstraction, the ML-framework level is reached. In this level, the most popular ML-framework, TensorFlow, has been chosen. It also offers the possibility of delegate part of the execution graph to coprocessor or GPUs. Moreover, existing Tensorflow pretrained models have been quantized for different bitwidth and data precision.

It is possible to build a custom hardware accelerator for a specific ML operation and then integrate it into a framework without changing the model nor the framework. The bottom up approach and the delegate class available in Tensorflow has allowed to fully tailor a new class of hardware accelerators which can accommodate different needs (i.e. depending on which part of the model has to be accelerated). As it has been organized, changing the core software in the Python code and the core in the hardware, it can be also used for addressing different model's operations.

5.2 Future Works

For every human artifacts, there is always work to do. In addition, for Computer Engineering artifacts there is also an important step which is the software (and in this case also of the hardware) optimization. In particular:

- Software Optimization and migration to a full C code implementation for further reducing the latency.
- A deep software/hardware testing for finding additional bugs.
- Power estimation using the simulation's switching activity in order to obtain a very precise and reliable power consumption.
- Comparison of model execution on different state-of-the-art platforms.

Following the previous recommendation, the work may arrive to a competitive level such as the one of the GPUs or other hardware platforms.

Appendix A

Accelerator library

Script for creating library:

```
import cffi
import sys
sys.path.append('/usr/local/lib')

# ################################################################################
####           The Frankenstein, a mix of C and Python                    ######
####  create .so library from PYNQ python code for DTPU accelerator       ######
####          on board compiling, it requires                             ######
####    to have tensorflow/tensorflow/lite in /usr/include/pythonX.X      ######
####                      from r2.1 branch                                ######
# ################################################################################
ffibuilder = cffi.FFI()

ffibuilder.cdef("""
extern "Python" {
bool destroy_p(void  );
bool CopyFromBufferHandle_p(void);
bool CopyToBufferHandle_p(void);
void FreeBufferHandle_p(void);
bool SelectDataTypeComputation_p(int);
bool Init_p(int  ,int,int);
bool Prepare_p(int);
bool Invoke_p(bool,int);
void  load_overlay(void);
bool ResetHardware_p(void);
void push_weight_to_heap( void  *,int *,int);
void push_input_tensor_to_heap( void  *,int *,int);
void push_output_tensor_to_heap( void  *,int *,int);
bool print_power_consumption_p(void);
bool start_power_consumption(void);
void activate_time_probe_p ( bool);
bool print_python_time_probes(void);
 };
  void * tflite_plugin_create_delegate();
  void tflite_plugin_destroy_delegate(void  * ,void * );
  bool  SelectDataTypeComputation(int);
  bool print_power_consumption();
  bool measure_power_consumption();
  bool print_execution_stats();
  bool activate_time_probe(bool );""")


cpp_file=open("./DTPU_delegate.cpp","r")
ffibuilder.set_source("dtpu_lib", cpp_file.read(),source_extension=".cpp",
  extra_compile_args=['-Wno-unused-result', '-Wsign-compare', '-DNDEBUG', '-g', '-fwrapv', '-O2', '-Wall', '-Wstrict-prototypes',
  '-g', '-fdebug-prefix-map=/build/python3.5.2=.', '-specs=/usr/share/dpkg/no-pie-compile.specs', '-fstack-protector-strong',
  '-Wformat', '-Werror=format-security','-I/usr/local/include','-L/usr/local/lib'],
  extra_link_args=['-Wl,-Bsymbolic-functions','-specs=/usr/share/dpkg/no-pie-link.specs',
  '-Wl,-z,relro','-specs=/usr/share/dpkg/no-pie-compile.specs','-D_FORTIFY_SOURCE=2','-fPIC']  ,
  libraries=['pthread','expat','z','dl','util','m','tensorflow'])
#if you want to simply access a global variable you just use its name.
# However to change its value you need to use the global keyword.
python_file=open("./DTPU_delegate.py","r")
ffibuilder.embedding_init_code(python_file.read())

ffibuilder.compile(target="DTPU_delegate.*", verbose=True)

cpp_file.close()
python_file.close()
```

../../dtpu/software/create_library.py

C++ code of the library:

../../dtpu/software/DTPU_delegate.cpp

```
/// release dependent libraries tensorflow r2.1
#include <tensorflow/lite/c/builtin_op_data.h>
#include <tensorflow/lite/c/c_api_internal.h>
#include <tensorflow/lite/builtin_ops.h>
#include <tensorflow/lite/context_util.h>
#include <tensorflow/c/c_api.h>
#include <vector>
#include <time.h>
#define DEBUG 1

static bool destroy_p(void );
static bool CopyFromBufferHandle_p(void);
static bool CopyToBufferHandle_p(void);
static void FreeBufferHandle_p(void);
static bool SelectDataTypeComputation_p(int);
static bool Init_p(int,int,int );
static bool Prepare_p(int );
static bool Invoke_p(bool,int);
static void load_overlay(void);
static bool ResetHardware_p(void);
static void push_weight_to_heap( void  *,int *,int);
static void push_input_tensor_to_heap( void  *,int *,int);
static void push_output_tensor_to_heap( void  *,int *,int);
static bool print_power_consumption_p(void);
static bool start_power_consumption(void);
static void activate_time_probe_p (bool);
static bool print_python_time_probes(void);


/*
possible operations
  kTfLiteBuiltinAdd = 0,
  kTfLiteBuiltinConcatenation = 2,
  kTfLiteBuiltinConv2d = 3,
  kTfLiteBuiltinDepthwiseConv2d = 4,
  kTfLiteBuiltinDepthToSpace = 5,
  kTfLiteBuiltinFullyConnected = 9,
  kTfLiteBuiltinMul = 18,
  kTfLiteBuiltinSub = 41,
  kTfLiteBuiltinDelegate = 51,
  kTfLiteBuiltinAddN = 106,struct timespec ts_start,ts_end;
*/


int bit_width_computation;
int NO_FP=-1;
```

```
bool signed_computation=false;
bool only_con2d=false;

// time probes
bool time_probe=false;
int n_execution=0;
double avg_time_delegate;
double avg_time_data_exchange;


using namespace tflite;

#ifdef __cplusplus
extern "C" {
#endif
// This is where the execution of the operations or whole graph
    happens.
// The class below has an empty implementation just as a guideline
// on the structure.
class DTPU_delegate {
 public:
  // Returns true if my delegate can handle this type of op.
  static bool SupportedOp(const TfLiteRegistration* registration) {
  // from builtin_ops.h
  #ifdef DEBUG
  printf("[DEBUG - C]--- Supported Operation of DTPU delegate class
    --- \n");
  #endif
    switch (registration->builtin_code) {
      /*case kTfLiteBuiltinConv2d:
        only_con2d=true;
        #ifdef DEBUG
        printf("[DEBUG-C]-- Supported operations only 2d convolution
    ----\n");
        #endif
        */
      case kTfLiteBuiltinDepthwiseConv2d:
      #ifdef DEBUG
        printf("[DEBUG - C]--Hello world! I can make 2D convolution
    and depth wise 2D convolution---\n");
        #endif
        return true;
      default:
        return false;
    }
  }

  // Any initialization code needed
 bool Init(TfLiteContext* context,const TfLiteDelegateParams*
  delegate_params) {
#ifdef DEBUG
   printf("[DEBUG - C]--- Init of DTPU delegate class --- \n");
   #endif

    #ifdef DEBUG
      printf("[DEBUG - C]--- Init of DTPU delegate class check if
  tensors indexes are equal to the ones in the Invoke --- \n");
   for (int input_index: TfLiteIntArrayView(delegate_params->
  input_tensors)){

      printf("[DEBUG - C]--- Init of DTPU delegate class getting
  tensors %d--- \n",input_index);

   }
     #endif

   if(time_probe){
     avg_time_delegate=0.00;
     avg_time_data_exchange=0.00;
     n_execution=0;
   }

   // instantiate buffcfers and soft reset of accelerator
   return Init_p(context->tensors_size,delegate_params->
  input_tensors->size ,delegate_params->output_tensors->size);

 }
 // Any preparation work needed (e.g. allocate buffers)
 TfLiteStatus Prepare(TfLiteContext* context, TfLiteNode* node) {
#ifdef DEBUG
 printf("[DEBUG - C]--- Prepare of DTPU delegate class --- \n");
#endif
     // initialize , link the buffers accordint to the size of node
  data
   // kTfLiteMmapRo  aka weights
   int num_weight_tensor=0;
   // set precison check
   if(NO_FP==-1){
          printf("ERROR! Need to execute SelectDataTypeComputation
  function before calling the Tensorflow Interpreter\n");
           return kTfLiteError ;
     }


   for (int input_index : TfLiteIntArrayView(node->inputs)){
     // one of this should be the weight tensor
     auto&  in_t= context->tensors[input_index];
```

```
      if(in_t.allocation_type==kTfLiteMmapRo){
              num_weight_tensor++;
              #ifdef DEBUG
              printf("[DEBUG -C]---found a tensor weight %d----\n", input_index);
              #endif
      // get dimesion of tensors
      // push to python sublayer
       if(!NO_FP){
      switch(bit_width_computation){
      default:
      case 8:
          #ifdef DEBUG
                    if(signed_computation){
                      printf("[DEBUG-C]---- kTfLiteInt8 -------\n");
                    }else{
                      printf("[DEBUG-C]---- kTfLiteUInt8 -------\n");
                    }
          #endif
          if(signed_computation){
          push_weight_to_heap(in_t.data.int8, in_t.dims->data, in_t.dims->size);
          }else {
          push_weight_to_heap( in_t.data.uint8, in_t.dims->data, in_t.dims->size);
          }

        break;
      case 16:
            #ifdef DEBUG
                    printf("[DEBUG-C]---- kTfLiteInt16 -------\n");
            #endif
                  push_weight_to_heap( in_t.data.i16, in_t.dims->data, in_t.dims->size);
                  break;
      case 32:
      #ifdef DEBUG
                  printf("[DEBUG-C]---- kTfLiteInt32 -------\n");
          #endif
            push_weight_to_heap( in_t.data.i32, in_t.dims->data, in_t.dims->size);
            break;
      case 64:
          #ifdef DEBUG
                printf("[DEBUG-C]---- kTfLiteInt64-------\n");
          #endif
                push_weight_to_heap( in_t.data.i64, in_t.dims->data, in_t.dims->size);
                break;
      }
      }
      else { // use fp units
        switch (bit_width_computation){
        case 16:
               if(context->allow_fp32_relax_to_fp16 && NO_FP==3 ){ //
 NO_FP==3 -> fp active and bfp active
                  #ifdef DEBUG
            printf("[DEBUG-C]---- kTfLitefloat32 relaxed aka bfp16
 ------\n");
            #endif
              // remembedr f16 is TfLiteFloat16*

            /*typedef struct {
                 uint16_t data;
               } TfLiteFloat16;
               */
            push_weight_to_heap( in_t.data.f16, in_t.dims->data, in_t
 .dims->size);
              }
              break;
        case 32:
            #ifdef DEBUG
            printf("[DEBUG-C]---- kTfLitefloat32 ------\n");
            #endif
              push_weight_to_heap( in_t.data.f, in_t.dims->data, in_t
 .dims->size);
              break;
        default:
            printf("[DEBUG-C]---- ERROR! no fp precision defined
 ------\n");
            break;
      }

      }
      }
    }
    #ifdef DEBUG
    printf("[DEBUG-C]--- number of weights found= %d \n",
   num_weight_tensor);
    #endif
      if(Prepare_p(num_weight_tensor)){
      return kTfLiteOk;
      }
     return kTfLiteError ;

  }
  // Actual running of the delegate subgraph.
  TfLiteStatus Invoke(TfLiteContext* context, TfLiteNode* node) {
    struct timespec ts_start,ts_end;
    int curr_input=0;
   #ifdef DEBUG
    printf("[DEBUG - C]--- Invoke of DTPU delegate class --- \n");
    printf("[DEBUG - C]--- Invoke of DTPU delegate class getting tensors --- \n");
   #endif


    if(time_probe){
          if(!timespec_get(&ts_start,TIME_UTC)){
          fprintf(stderr,"error during the acquisition of start time!\n");
          exit(-1);
              }
    }

    // run inference on the delegate  and data transfer to/from memory/accelerator
    for (int input_index : TfLiteIntArrayView(node->inputs)){
      // one of this should be the weight tensor
      #ifdef DEBUG
      printf("[DEBUG - C]--- Invoke of DTPU delegate class getting tensors %d--- \n",input_index);
      #endif
      TfLiteTensor  in_t= context->tensors[input_index];
      if(!(in_t.allocation_type==kTfLiteMmapRo)){ //cause the weights have been transferred into the Prepare method
          if(curr_input!=0){
            curr_input=input_index;
          }
      // get dimesion of tensors
      // push to python sublayer
        if(!NO_FP){
      switch(bit_width_computation){
      default:
      case 8:
          #ifdef DEBUG
                    if(signed_computation){
                      printf("[DEBUG-C]---- kTfLiteInt8 ------\n");
                    }else{
                      printf("[DEBUG-C]---- kTfLiteUInt8 ------\n");
                    }
          #endif
          if(signed_computation){
        push_input_tensor_to_heap(in_t.data.int8,in_t.dims->data,in_t.dims->size);
          }else {
```

```
          push_input_tensor_to_heap(in_t.data.uint8,in_t.dims->data,in_t.dims->size);
          }

        break;
      case 16:
            #ifdef DEBUG
                    printf("[DEBUG-C]---- kTfLiteInt16 ------\n");
            #endif
              push_input_tensor_to_heap(in_t.data.i16,in_t.dims->data,in_t.dims->size);
                  break;
      case 32:
      #ifdef DEBUG
                  printf("[DEBUG-C]---- kTfLiteInt32 ------\n");
          #endif
            push_input_tensor_to_heap(in_t.data.i32,in_t.dims->data,in_t.dims->size);
            break;
      case 64:
          #ifdef DEBUG
                printf("[DEBUG-C]---- kTfLiteInt64------\n");
          #endif
            push_input_tensor_to_heap(in_t.data.i64,in_t.dims->data,in_t.dims->size);
                break;
      }
      }
      else { // use fp units
        switch (bit_width_computation){
        case 16:
              if(context->allow_fp32_relax_to_fp16 && NO_FP==3 ){ // NO_FP==3 -> fp active and bfp active
                #ifdef DEBUG
            printf("[DEBUG-C]---- kTfLitefloat32 relaxed aka bfp16 ------\n");
            #endif
              push_input_tensor_to_heap(in_t.data.f16,in_t.dims->data,in_t.dims->size);
              }
              break;
        case 32:
            #ifdef DEBUG
            printf("[DEBUG-C]---- kTfLitefloat32 ------\n");
            #endif
              push_input_tensor_to_heap(in_t.data.f,in_t.dims->data,in_t.dims->size);
              break;
        default:
            printf("[DEBUG-C]---- ERROR! no fp precision defined
------\n");
            break;
      }

      }

      }
    }


    for (int output_index : TfLiteIntArrayView(node->outputs)){
      auto&  out_t= context->tensors[output_index];
      // get dimesion of tensors
      // push to python sublayer

      #ifdef DEBUG
      printf("[DEBUG - C]--- Invoke of DTPU delegate class getting
output tensors %d--- \n",output_index);
      #endif

        if(!NO_FP){
      switch(bit_width_computation){
      default:
      case 8:
          #ifdef DEBUG
                      if(signed_computation){
                        printf("[DEBUG-C]---- kTfLiteInt8 ------\n");
                      }else{
                        printf("[DEBUG-C]---- kTfLiteUInt8 ------\n");
                      }
          #endif
          if(signed_computation){
          push_output_tensor_to_heap(out_t.data.int8,out_t.dims->data
,out_t.dims->size);
          }else {
          push_output_tensor_to_heap(out_t.data.uint8,out_t.dims->
data,out_t.dims->size);
          }

        break;
      case 16:
            #ifdef DEBUG
                      printf("[DEBUG-C]---- kTfLiteInt16 ------\n");
            #endif
              push_output_tensor_to_heap(out_t.data.i16,out_t.dims->
data,out_t.dims->size);
                        break;
      case 32:
```

```
    #ifdef DEBUG
                printf("[DEBUG-C]---- kTfLiteInt32 -------\n");
        #endif
          push_output_tensor_to_heap(out_t.data.i32,out_t.dims->
data,out_t.dims->size);
          break;
    case 64:
        #ifdef DEBUG
              printf("[DEBUG-C]---- kTfLiteInt64-------\n");
        #endif
          push_output_tensor_to_heap(out_t.data.i64,out_t.dims->
data,out_t.dims->size);
              break;
    }
    }
    else { // use fp units
      switch (bit_width_computation){
      case 16:
            if(context->allow_fp32_relax_to_fp16 && NO_FP==3 ){ //
NO_FP==3 -> fp active and bfp active
              #ifdef DEBUG
          printf("[DEBUG-C]---- kTfLitefloat32 relaxed aka bfp16
-------\n");
          #endif
            push_output_tensor_to_heap(out_t.data.f16,out_t.dims->
data,out_t.dims->size); // a uint16 pointer
            }
            break;
      case 32:
          #ifdef DEBUG
          printf("[DEBUG-C]---- kTfLitefloat32 -------\n");
          #endif
            push_output_tensor_to_heap(out_t.data.f,out_t.dims->
data,out_t.dims->size);
            break;
      default:
          printf("[DEBUG-C]---- ERROR! no fp precision defined
-------\n");
          break;
    }

    }

    }
    if(time_probe){
    if (!timespec_get(&ts_end, TIME_UTC)){
      fprintf(stderr,"erorr during the acquisition of end time!\n")
;
      exit(-1);
      }
      // update average and execution time
      avg_time_data_exchange+=ts_end.tv_sec*1000 + ((double)ts_end.
   tv_nsec)/1000000 - ts_start.tv_sec*1000 - ((double)ts_start.
   tv_nsec)/1000000;

      n_execution++;
      }

      if(time_probe){
      if(!timespec_get(&ts_start, TIME_UTC)){
        fprintf(stderr,"erorr during the acquisition of end time!\n")
   ;
        exit(-1);
      }
      }
     if(Invoke_p(only_con2d,curr_input)){
      if(time_probe){
      if(!timespec_get(&ts_end, TIME_UTC)){
        fprintf(stderr,"erorr during the acquisition of end time!\n")
   ;
        exit(-1);
      }
      avg_time_delegate+=ts_end.tv_sec*1000 + ((double)ts_end.tv_nsec
   )/1000000 - ts_start.tv_sec*1000 - ((double)ts_start.tv_nsec)
   /1000000;
      }
      return kTfLiteOk;
      }
     return kTfLiteError ;
    }

};


  TfLiteStatus  SelectDataTypeComputation(int data_type ){
  #ifdef DEBUG
  printf("[DEBUG - C]--- SelectDataTypeComputation of DTPU delegate
    class --- \n");
  #endif
  int precision= data_type & 0x000f;
  signed_computation= ((data_type & 0x00100)>>8)==1 ? true : false;

  NO_FP= (data_type & 0x060)>>5;
  switch(precision){
    default:
    case 1: //INT8
    bit_width_computation=8;
    break;
    case 3: //INT16
    bit_width_computation=16;
    break;
    case 7: //INT32
    bit_width_computation=32;
    break;
    case 15: //INT64
      bit_width_computation=64;
      break;
  }
  // check compatibilyt of signed and unsigned
  if(signed_computation && bit_width_computation!=8){
    printf("ERROR-> signed/unsigned distinction is only compatible
    with 8 bit computation");
    return kTfLiteError;
  }
  if(SelectDataTypeComputation_p(data_type) ){
      return kTfLiteOk;
      }
     return kTfLiteError ;
  }

  TfLiteStatus  ResetHardware( ){
  #ifdef DEBUG
  printf("[DEBUG - C]---  Reset underlaying hardware --- \n");
  #endif
  if(ResetHardware_p()){
      return kTfLiteOk;
      }
     return kTfLiteError ;

  }

// Create the TfLiteRegistration for the Kernel node which will
    replace
// the subgraph in the main TfLite graph.
TfLiteRegistration GetMyDelegateNodeRegistration() {
  // This is the registration for the Delegate Node that gets added
    to
  // the TFLite graph instead of the subGraph it replaces.
  // It is treated as a an OP node. But in our case
  // Init will initialize the delegate
  // Invoke will run the delegate graph.
  // Prepare for preparing the delegate.
  // Free for any cleaning needed by the delegate.
  #ifdef DEBUG
  printf("[DEBUG - C] --- get delegate node registration function
    ---\n");
  #endif
  TfLiteRegistration kernel_registration;
  kernel_registration.builtin_code = kTfLiteBuiltinDelegate;
  kernel_registration.custom_name = "DTPU_delegate";
  kernel_registration.free = [](TfLiteContext* context, void* buffer)
    -> void {
    delete reinterpret_cast<DTPU_delegate*>(buffer);
  };
  kernel_registration.init = [](TfLiteContext* context, const char*
   buffer,
                                   size_t) -> void* {
    // In the node init phase, initialize MyDelegate instance
    const TfLiteDelegateParams* delegate_params =
        reinterpret_cast<const TfLiteDelegateParams*>(buffer);
    DTPU_delegate* my_delegate = new DTPU_delegate;
    if (!my_delegate->Init(context, delegate_params)) {
      return nullptr;
    }
    return my_delegate;
  };
  kernel_registration.invoke = [](TfLiteContext* context,
                                   TfLiteNode* node) -> TfLiteStatus
   {
    DTPU_delegate* kernel = reinterpret_cast<DTPU_delegate*>(node->
   user_data);
    return kernel->Invoke(context, node);
  };
  kernel_registration.prepare = [](TfLiteContext* context,
                                    TfLiteNode* node) -> TfLiteStatus
    {
    DTPU_delegate* kernel = reinterpret_cast<DTPU_delegate*>(node->
   user_data);
    return kernel->Prepare(context, node);
  };

  return kernel_registration;
}

// TfLiteDelegate methods
// interface to tensorflow runtime
TfLiteStatus DelegatePrepare(TfLiteContext* context, TfLiteDelegate*
    delegate) {
  // Claim all nodes that can be evaluated by the delegate and ask
   the
  // framework to update the graph with delegate kernel instead.
  // Reserve 1 element, since we need first element to be size.
  #ifdef DEBUG
  printf("[DEBUG - C] ---- preparing the delegate ----\n");
  #endif
  std::vector<int> supported_nodes(1);
  TfLiteIntArray* plan;
  TF_LITE_ENSURE_STATUS(context->GetExecutionPlan(context, &plan));
  TfLiteNode* node;
  TfLiteRegistration* registration;
  for (int node_index : tflite::TfLiteIntArrayView(plan) ) {
    TF_LITE_ENSURE_STATUS(context->GetNodeAndRegistration(
        context, node_index, &node, &registration));
    if (DTPU_delegate::SupportedOp(registration)) {
      supported_nodes.push_back(node_index);
    }
  }
  // Set first element to the number of nodes to replace.
  supported_nodes[0] = supported_nodes.size() - 1;
  TfLiteRegistration my_delegate_kernel_registration =
      GetMyDelegateNodeRegistration();

  // This call split the graphs into subgraphs, for subgraphs that
    can be
  // handled by the delegate, it will replace it with a
  // 'my_delegate_kernel_registration'
  return context->ReplaceNodeSubsetsWithDelegateKernels(
      context, my_delegate_kernel_registration,
      reinterpret_cast<TfLiteIntArray*>(supported_nodes.data()),
    delegate);
}

void FreeBufferHandle(TfLiteContext* context, TfLiteDelegate*
    delegate,
                       TfLiteBufferHandle* handle) {
  #ifdef DEBUG
  printf("[DEBUG - C]--- Do any cleanups---\n");
  #endif
  FreeBufferHandle_p();
}


TfLiteStatus CopyToBufferHandle(TfLiteContext* context,
                                TfLiteDelegate* delegate,
                                TfLiteBufferHandle buffer_handle,
                                TfLiteTensor* tensor) {
  #ifdef DEBUG
  printf("[DEBUG - C]--- Copies data from tensor to delegate buffer
    if needed.----\n");
  #endif
  if(CopyToBufferHandle_p()){
  return kTfLiteOk;
  }
  return kTfLiteError;
}
```

```

TfLiteStatus CopyFromBufferHandle(TfLiteContext* context,
                                  TfLiteDelegate* delegate,
                                  TfLiteBufferHandle buffer_handle,
                                  TfLiteTensor* tensor) {
  #ifdef DEBUG
   printf("[DEBUG - C]---Copies the data from delegate buffer into the
      tensor raw memory----\n");
  #endif
   if(CopyFromBufferHandle_p()){
   return kTfLiteOk;
   }
   return kTfLiteError;
}

TfLiteStatus activate_time_probe(bool activate){
  #ifdef DEBUG
   printf("[DEBUG-C]---- activating time probes ----\n");
  #endif
   if (!time_probe && activate){
     time_probe=true;
       #ifdef DEBUG
            printf("[DEBUG-C]---- activated time probes ----\n");
         #endif
     activate_time_probe_p(activate);
   } else{
     printf("ATTENTION! Time probes are not active\n");

   }
    return kTfLiteOk;

}


TfLiteStatus print_execution_stats(){
  #ifdef DEBUG
   printf("[DEBUG - C]----- printing time probes of the library -----\
    n");
  #endif
     printf("If you are seeing too many zeros you probably did not set
      the time probes variable to true!\n");

   // print c time probes
     printf("Overall time of delegate invoke: %3f [ms]\n",
    avg_time_delegate/n_execution);
     printf("Data exchange between interfaces (C->Python->C): %3f [ms
    ]\n",avg_time_data_exchange/n_execution);

   // print python time probes
  if(print_python_time_probes()){
    return kTfLiteOk;
  }
  return kTfLiteError;

}

TfLiteStatus measure_power_consumption(){
#ifdef DEBUG
  printf("[DEBUG - C]---Measuring power consumption of the
    accelerator during invoke ----\n");
  #endif
  if(start_power_consumption()){
    return kTfLiteOk;
  }
  return kTfLiteError;

}

TfLiteStatus print_power_consumption(){
  #ifdef DEBUG
  printf("[DEBUG - C]---Printing power consumption of the accelerator
     during invoke ----\n");
  #endif
  if(print_power_consumption_p()){
    return kTfLiteOk;
  }
  return kTfLiteError;
}

// instantiate the delegate, it returns null if there is an error
TfLiteDelegate * tflite_plugin_create_delegate()
//char** argv , char** argv2, size_t argc, void (*report_error)(const
     char *) )
  {
  TfLiteDelegate* delegate = new TfLiteDelegate;

  delegate->data_ = nullptr;
  delegate->flags = kTfLiteDelegateFlagsNone;
  delegate->Prepare = &DelegatePrepare;
  // This cannot be null.
  delegate->CopyFromBufferHandle = &CopyFromBufferHandle;
  // This can be null.
  delegate->CopyToBufferHandle = &CopyToBufferHandle;
  // This can be null.
  delegate->FreeBufferHandle = &FreeBufferHandle;
  // load overlay
  load_overlay();
  #ifdef DEBUG
  printf("[DEBUG - C] ---the delegate method of DTPU is born for
    TensorFlow %s---\n",TF_Version());
 #endif
  return delegate;
}


void tflite_plugin_destroy_delegate(void * delegate_op , void *
    argtypes) {
// destroy the delegate
TfLiteDelegate * delegate= (TfLiteDelegate *) delegate_op;
#ifdef DEBUG
printf("[DEBUG - C]-----cleaning memory -> callback of python
    function---\n");
#endif
if(!destroy_p()) {
  printf("ERROR IN FREEING BUFFERS!");
}
//free(argtypes);
free(delegate);
}
#ifdef __cplusplus
} // extern "C"
#endif
```

Frozen python code in the accelerator library:

../../dtpu/software/DTPU_delegate.py

```
from dtpu_lib import ffi
from pynq import Overlay
from pynq import allocate
from pynq import MMIO
from pynq import Xlnk
from pynq.lib import dma
import numpy as np
import math
import _thread
import sys
import time
import struct # see https://docs.python.org/3/library/struct.html#
    struct-examples
_DEBUG_PRINT=True
_TIME_PROBES=False
###############################
##### memory map of xadc #####
################################
C_BASEADDRESS=0x43C10000 #
SRR=  0x0 # w software reset register
SR=  0x04 # r status register
AOSR= 0x08 # r allarm output status register
CONVSTR= 0x0C # w Bit[0] = ADC convert start register (3) Bit[1] =
    Enable temperature update logic Bit[17:2] = Wait cycle for
    temperature update
SYSMONRR=0x10 # w xadc hard macro reset register
GIER=0x5C # rw global interrupt enable register
IPISR=0x60 # r toggle on write ip interrupt status register
IPIER=0x68 # rw ip interrupt enable register
TEMPERATURE=0x200 # The 12-bit Most Significant Bit (MSB)  justified
    result of on-device temperature measurement is stored in this
    register
VCC_INT=0x204 # The 12-bit MSB justified result of on-device V CCINT
    supply monitor measurement is storedin this register.
VCC_AUX=0x208 # The 12-bit MSB justified result of on-device V CCAUX
    Data supply monitor measurement is stored in this register
VP_VN=0x20C # rw When read: The 12-bit MSB justified result of A/D
    conversion on the dedicated analog input channel (Vp/Vn) is stored
     in this register.When written: Write to this regiter resets
    theXADC hard macro
VREF_P=0x210 # r The 12-bit MSB justified result of A/D conversion on
     the reference input V REFP is stored in this register.
VREF_N= 0x214 #r The 12-bit MSB justified result of A/D conversion on
     the reference input V REFN is stored in this register.
VCC_BRAM= 0x218 # r The 12-bit MSB justified result of A/D conversion
     on the reference input V BRAM is stored in this register
```

```
SUPPLY_A_OFFSET=0x220 # r The calibration coefficient for the supply
     sensor offset of ADC A is stored in this register
ADC_A_OFFSET=  0x224 # r The calibration coefficient for the ADC A
    offset calibration is stored in this register.
ADC_A_GAIN_ERR=0x228 # r The calibration coefficient for the gain
    error of ADC A is stored in this register.
DEV_CORE_SUPPLY=0x234 #r  The  VCCINT of PSS core supply. Present
    only  on Zynq-7000 devices.
DEV_AUX_CORE_SUPPLY=0x238 # r The VCCAUX of PSS core supply. Present
    only on Zynq-7000 devices.
DEV_CORE_MEM_SUPPLY=0x23C # r The VCCMEM of PSS core supply. Present
    only on Zynq-7000 devices
# v axux p/n
V_AUX_0=  0x240 #r The 12-bit MSB justified result of A/D conversion
    on the auxiliary analog input 0 is  stored in this register.
V_AUX_1=  0x244 #r
V_AUX_2=  0x248 #r
V_AUX_3=  0x24C #r
V_AUX_4=  0x250 #r
V_AUX_5=  0x254 #r
V_AUX_6=  0x258 #r
V_AUX_7=  0x25C #r
V_AUX_8=  0x260 #r
V_AUX_9=  0x264 #r
V_AUX_10= 0x268 #r
V_AUX_11= 0x26C #r
V_AUX_12= 0x270 #r
V_AUX_13= 0x274 #r
V_AUX_14= 0x278 #r
V_AUX_15= 0x27C #r
## value of 12 bit msb
MAX_TMP= 0x280    # r
MAX_VCC_INT= 0x284    # r
MAX_VCC_AUX= 0x288    # r
MAV_V_BRAM= 0x28C    # r
MIN_TMP= 0x290    # r
MIN_VCC_INT= 0x294    # r
MIN_VCC_AUX= 0x298    # r
MIN_V_BRAM=0x29C    # r
MAX_VCC_PINT= 0x2A0 # r
MAX_VCC_PAUX= 0x2A4 # r
MAX_VCC_DDRO= 0x2A8 # r
MIN_VCC_PINT= 0x2b0 # r
MIN_VCC_PAUX= 0x2b4 # r
MIN_VCC_DDRO= 0x2b8 # r
SUPPLY_B_OFFSET= 0x2C0 # r The calibration coefficient for the supply
      sensor offset of ADC A is stored in this register
ADC_B_OFFSET=  0x2C4 # r The calibration coefficient for the ADC A
    offset calibration is stored in this register.
```

```
ADC_B_GAIN_ERR= 0x2C8 # r The calibration coefficient for the gain
    error of ADC A is stored in this register.
FLAGS=0x2FC # The 16-bit register gives general status information of
     ALARM, Over Temperature (OT), Disable XADC information. Whether
    the XADC is using the internal reference voltage or external
    reference voltage is also p
CONF_REG_0=0x300 # rw
CONF_REG_1=0x304 # rw
CONF_REG_2=0x308 # rw
SEQ_REG_0= 0x320 # r/w adc channel selection
SEQ_REG_1= 0x324 # r/w adc channel selection
SEQ_REG_2= 0x328 # r/w adc channel average enable
SEQ_REG_3= 0x32C # r/w adc channel average enable
SEQ_REG_4= 0x330 # r/w adc channel analog input mode
SEQ_REG_5= 0x334 # r/w adc channel analog input mode
SEQ_REG_6= 0x338 # r/w adc channel acquistion
SEQ_REG_7= 0x33C # r/w adc channel acquistion
ALLARM_THRESHOLD_0= 0x340 #rw The 12bit MSB justified alarm threshold
     register 0 Temperature Upper
ALLARM_THRESHOLD_1= 0x344 #rw he 12bit MSB justified alarm threshold
    register 1 V CCINT Upper
ALLARM_THRESHOLD_2= 0x348 #rw The 12bit MSB justified alarm threshold
     register 2 V CCAUX Upper
ALLARM_THRESHOLD_3= 0x34C #rw the 12bit MSB justified alarm threshold
     register 3 T Upper
ALLARM_THRESHOLD_4= 0x350 #rw the 12bit MSB justified alarm threshold
     register 4 Temperature Lower
ALLARM_THRESHOLD_5= 0x354 #rw  the 12bit MSB justified alarm
    threshold register 5 V CCINT Lower
ALLARM_THRESHOLD_6= 0x358 #rw The 12bit MSB justified alarm threshold
     register 6 V CCAUX Lower
ALLARM_THRESHOLD_7= 0x35C # rw The 12bit MSB justified alarm
    threshold register 7 OT Lower
ALLARM_THRESHOLD_8= 0x360 # rw The 12bit MSB justified alarm
    threshold register 8 VBRAM Upper
ALLARM_THRESHOLD_9= 0x364 # rw The 12bit MSB justified alarm
    threshold  register 9 V CCPint Upper This register is only on Zynq
    -7000 devices.
ALLARM_THRESHOLD_10= 0x368 # rw The 12bit MSB justified alarm
    threshold  register 10 V CCPaux Upper This register is only on
    Zynq-7000 devices
ALLARM_THRESHOLD_11= 0x36C # rw The 12bit MSB justified alarm
    threshold register 11  CCDDRO Upper This register is  only on Zynq
    -7000 devic
ALLARM_THRESHOLD_12= 0x370 # rw he 12bit MSB justified alarm
    threshold register 12 VBRAM Lower
ALLARM_THRESHOLD_13= 0x374 # rw The 12Bit MSB justified alarm
    threshold register 13 V CCPint Lower This register is only on Zynq
    -7000 devices
```

```
ALLARM_THRESHOLD_14= 0x378 # rw The 12bit MSB justified alarm
    threshold register 14 V CCPaux Lower This register is only on Zynq
    -7000 devices
ALLARM_THRESHOLD_15= 0x37C # rw he 12bit MSB justified alarm
    threshold register 15 v CCDDRO Lower This register is only on Zynq
    -7000 devices
##########################################################
############ MEMORY MAP of acceleratro #################
##########################################################
BASE_ADDRESS_ACCELERATOR=0x43C00000
ADDRESS_RANGE_ACCELERATOR=0x10000
# address reg offset
CTRL =0x0000
STATUS =0x0004
IARG_RQT_EN =0x0010
OARG_RQT_EN =0x0014
CMD =0x0028
OARG_LENGTH_MODE =0x003C
ISCALAR_FIFO_RST =0x0040
OSCALAR_FIFO_RST =0x0044
ISCALAR_RQT_EN =0x0048
OSCALAR_RQT_EN =0x004C
ISCALAR0_DATA =0x0080
ISCALAR1_DATA =0x0084
ISCALAR2_DATA =0x0088
ISCALAR3_DATA =0x008C
ISCALAR4_DATA =0x0090
ISCALAR5_DATA =0x0094
ISCALAR6_DATA =0x0098
ISCALAR7_DATA =0x009C
ISCALAR8_DATA =0x00A0
ISCALAR9_DATA =0x00A4
ISCALAR10_DATA=0x00A8
ISCALAR11_DATA =0x00AC
ISCALAR12_DATA =0x00B0
ISCALAR13_DATA =0x00B4
ISCALAR14_DATA =0x00B8
ISCALAR15_DATA =0x00BC
OSCALAR0_DATA =0x00C0
OSCALAR1_DATA =0x00C4
OSCALAR2_DATA =0x00C8
OSCALAR3_DATA =0x00CC
OSCALAR4_DATA =0x00D0
OSCALAR5_DATA =0x00D4
OSCALAR6_DATA =0x00D8
OSCALAR7_DATA =0x00DC
IARG0_STATUS =0x0100
IARG1_STATUS =0x0104
IARG2_STATUS =0x0108
```

```
IARG3_STATUS =0x010C
IARG4_STATUS =0x0110
IARG5_STATUS =0x0114
IARG6_STATUS =0x0118
IARG7_STATUS =0x011C
OARG0_STATUS =0x0140
OARG1_STATUS =0x0144
OARG2_STATUS =0x0148
OARG3_STATUS =0x014C
OARG4_STATUS =0x0150
OARG5_STATUS =0x0154
OARG6_STATUS =0x0158
OARG7_STATUS =0x015C
ISCALAR0_STATUS =0x0180
ISCALAR1_STATUS =0x0184
ISCALAR2_STATUS =0x0188
ISCALAR3_STATUS =0x018C
ISCALAR4_STATUS =0x0190
ISCALAR5_STATUS =0x0194
ISCALAR6_STATUS =0x0198
ISCALAR7_STATUS =0x019C
ISCALAR8_STATUS =0x01A0
ISCALAR9_STATUS =0x01A4
ISCALAR10_STATUS =0x01A8
ISCALAR11_STATUS =0x01AC
ISCALAR12_STATUS =0x01B0
ISCALAR13_STATUS =0x01B4
ISCALAR14_STATUS =0x01B8
ISCALAR15_STATUS =0x01BC
OSCALAR0_STATUS =0x01C0
OSCALAR1_STATUS =0x01C4
OSCALAR2_STATUS =0x01C8
OSCALAR3_STATUS =0x01CC
OSCALAR4_STATUS =0x01D0
OSCALAR5_STATUS =0x01D4
OSCALAR6_STATUS =0x01D8
OSCALAR7_STATUS =0x01DC
OSCALAR8_STATUS =0x01E0
OSCALAR9_STATUS =0x01E4
OSCALAR10_STATUS =0x01E8
OSCALAR11_STATUS =0x01EC
OSCALAR12_STATUS =0x01F0
OSCALAR13_STATUS =0x01F4
OSCALAR14_STATUS =0x01F8
OSCALAR15_STATUS =0x01FC
OARG0_LENGTH =0x0200
OARG1_LENGTH =0x0204
OARG2_LENGTH =0x0208
OARG3_LENGTH =0x020C
```

```
OARG4_LENGTH =0x0210
OARG5_LENGTH =0x0214
OARG6_LENGTH =0x0218
OARG7_LENGTH =0x021C
OARG0_TDEST =0x0240
OARG1_TDEST =0x0244
OARG2_TDEST =0x0248
OARG3_TDEST =0x024C
OARG4_TDEST =0x0250
OARG5_TDEST =0x0254
OARG6_TDEST =0x0258
OARG7_TDEST =0x025C
##########################################################
##########          CSR DEFINITIONS           ##########
##########             MEMORY MAP             ##########
##########             bitwidth 8             ##########
##########             see csr_definition.vh  ##########
##########################################################
ARITHMETIC_PRECISION=0
FP_MODE=1
BATCH_SIZE=2 # aka active rows
TRANSPARENT_DELAY_REGISTER=3
DEBUG=4
TEST_OPTIONS=5
ACTIVATE_CHAIN=0x1
INT8=0x1
INT16=0X3
INT32=0x7
INT64=0xF
# precision of fp computation is tuned using the
# integer precision
ACTIVE_FP=1
ACTIVE_BFP=0x03
ROUNDING=0x00
NO_FP=0x00
SIGNED=0x1
NO_SIGNED=0x0
WMEM_STARTING_ADDRESS=0 #32 MSB
##################################################
#### accelerator adapter command #############
##################################################
CMD_UPDATE_IN_ARG=0x0
CMD_UPDATE_OUT_ARG=0x1
CMD_EXECUTE_STEP=0x2
CMD_EXECUTE_CONTINOUS=0x4
CMD_STOP_EXECUTE_CONTINOUS=0x5
##################################################
BASE_ADDRESS_INTC=0x40800000
ADDRESS_RANGE_INTC=0x10000
```

```
BASE_ADDRESS_DMA_INFIFO=0x40400000
ADDRESS_RANGE_DMA_INFIFO=0x10000
BASE_ADDRESS_DMA_WM=0x40410000
ADDRESS_RANGE_DMA_WM=0x10000
accelerator=None
infifo_buffer_transfer=None
output_fifo_buffer=None
weight_buffer=None
csr_buffer=None
overlay=None
driver_csr=None
driver_wm=None
driver_fifo_in=None
driver_fifo_out=None
########################################
### DESIGN DEPENDENT DEFINITION #####
########################################
WMEM_SIZE=16384 # 1Mbytes
CSRMEM_SIZE=1024
INFIFO_SIZE=2048  #1Kbytes
OUTFIFO_SIZE=2048 #1Kbytes
ROWS=0
COLUMNS=0
DATAWIDTH=64
BUFFER_DEPTH=2
output_size=0
input_size=0
tot_size_weight=0
tot_size_input=0
tot_size_output=0
curr_data_precision=INT8
curr_bitwidth_data_computation=8
PACK_TYPE="b" # default is 1 byte signed for integer lower case ->
    signed upper case-> unsigned
DTYPE_NP=np.uint8
FP=False
BFP=False
size_tot=0
num_weight=0
global_iteration=1 ## at least one execution of the tensor
    accelerator
global_iteration_shift_wm=[]
weight_tensors=[]
input_tensors=[]
output_tensors=[]
output_tensors_p=[]
weight_buffer_multiple=[]
index_wm=0
class Tensor:
    def __init__(self,data, tot_dim,size_l):
        self.tot_dim=tot_dim
        self.data=data
        self.size_l=size_l
filter_height=0
filter_width=0
########################
##### time probes #####
#######################
avg_hw_execution=0.0
n_execution=0
avg_hw_execution_internal=0.00
n_execution_internal=0
#####################
###### XADC #########
#####################
xadc_mon=None
##################################################
##### Retrieve and display power consumption  ####
##### Supply sensor: Vccint,Vccaux,Vccbram    ####
#####       Vccpint,  Vccpaux,Vcc0ddr        #####
##### Nominal values of resistances and Vcc ######
##################################################
# from vivado report power
# [V]
vcc_pl_int_nom=1.00
vcc_pl_aux_nom=1.80
vcc_pl_bram_nom= 1.00
vcc_ps_int_nom=1.80
vcc_ps_aux_nom=1.80
vcc_ddr_nom=1.50
# equivalent series resitstances of capacitor -> worst case
# [omh]
r_pl_int=225
r_pl_aux=300
r_pl_bram=225
r_ps_int=225
r_ps_aux=400
r_ddr= 0.005
n_sample=1
ps_power=0
pl_power=0
mem_power=0
ps_power_max=sys.float_info.min
pl_power_max=sys.float_info.min
mem_power_max=sys.float_info.min
ps_power_min=sys.float_info.max
pl_power_min=sys.float_info.max
mem_power_min=sys.float_info.max
```

```
tmp_max=sys.float_info.min
tmp_min=sys.float_info.max
tmp_avg=0.00

def sample_power( threadName, delay):
  global ps_power
  global pl_power
  global mem_power
  global n_sample
  global xadc_mon
  global vcc_ps_aux_nom
  global ps_power_max
  global pl_power_max
  global mem_power_max
  global ps_power_min
  global pl_power_min
  global mem_power_min
  global tmp_max
  global tmp_min
  global tmp_avg
  while True:
    time.sleep(0.8/1000)
    vcc_pl_int=( xadc_mon.read(VCC_INT) & 0x0000FFF0) >> 4
    vcc_pl_int= (vcc_pl_int* vcc_ps_aux_nom) / 4096
    vcc_pl_aux=( xadc_mon.read(VCC_AUX) & 0x0000FFF0) >> 4
    vcc_pl_aux= (vcc_pl_aux* vcc_ps_aux_nom) / 4096
    vcc_pl_bram= ( xadc_mon.read(VCC_BRAM) & 0x0000FFF0) >> 4
    vcc_pl_bram= (vcc_pl_bram* vcc_ps_aux_nom) / 4096
    vcc_ps_int= ( xadc_mon.read(DEV_CORE_SUPPLY) & 0x0000FFF0) >> 4
    vcc_ps_int= (vcc_ps_int* vcc_ps_aux_nom) / 4096
    vcc_ps_aux=( xadc_mon.read(DEV_AUX_CORE_SUPPLY) & 0x0000FFF0) >>
   4
    vcc_ps_aux= (vcc_ps_aux* vcc_ps_aux_nom) / 4096
    vcc_ddr= ( xadc_mon.read(DEV_CORE_MEM_SUPPLY) & 0x0000FFF0) >> 4
    vcc_ddr= (vcc_ddr* 3) / 4096
    n_sample+=1
    ps_power_i=((vcc_ps_int_nom-vcc_ps_int)/r_ps_int)*vcc_ps_int_nom
   + ((vcc_ps_aux_nom-vcc_ps_aux)/r_ps_aux)*vcc_ps_aux_nom
    pl_power_i= ((vcc_pl_int_nom-vcc_pl_int)/r_pl_int)*vcc_pl_int_nom
    + ((vcc_pl_aux_nom-vcc_pl_aux)/r_pl_aux)*vcc_pl_aux_nom + ((
   vcc_pl_bram_nom-vcc_pl_bram)/r_pl_bram)*vcc_pl_bram_nom
    mem_power_i=((vcc_ddr-vcc_ddr_nom)/r_ddr)*vcc_ddr
    ## update max
    if pl_power_i > pl_power_max:
      pl_power_max=pl_power_i
    if ps_power_i > ps_power_max:
      ps_power_max=ps_power_i
    if mem_power_i > mem_power_max:
      mem_power_max=mem_power_i
    #update min
    if pl_power_i < pl_power_min:
      pl_power_min=pl_power_i
    if ps_power_i < ps_power_min:
      ps_power_min=ps_power_i
    if mem_power_i < mem_power_min:
      mem_power_min=mem_power_i
    ## update values for the averages
    ps_power+=ps_power_i
    pl_power+=pl_power_i
    mem_power+=mem_power_i
    # temperature
    tmp=( xadc_mon.read(TEMPERATURE) & 0x0000FFF0) >> 4
    tmp=(tmp* 503.975)/4096 - 273.15
    ## update max
    if tmp > tmp_max:
      tmp_max=tmp
    ## update min
    if tmp < tmp_min:
      tmp_min=tmp
    tmp_avg+=tmp

########################################
############ LOAD DESIGN ##############
########################################
@ffi.def_extern()
def load_overlay():
  global accelerator
  global overlay
  global xadc_mon
  global ROWS
  global COLUMNS
  global   ps_power
  global pl_power
  global mem_power
  global ps_power_max
  global pl_power_max
  global mem_power_max
  global ps_power_min
  global pl_power_min
  global mem_power_min
  global tmp_max
  global tmp_min
  global tmp_avg
  ## modify this part for choosing a different overlay and recompile
    the library
  f_clk="30mhz"
  datawidth="only_integer8"
  mxu_size="mxu_8x8"
  ROWS=8
  COLUMNS=8
  print("Hardware design space points",f_clk," ", " ", mxu_size, " ",
     datawidth)
  overlay = Overlay("/home/xilinx/dtpu_configurations/"+datawidth+"/"
    +f_clk+"/" + mxu_size+"/pynqz2.bit") # tcl is also parsed
  overlay.download() # Explicitly download bitstream to PL
  if overlay.is_loaded():
   # Checks if a bitstream is loaded
   if _DEBUG_PRINT: print("[DEBUG- PYTHON] ----overlay is loaded ----
   ")
  else :
    if _DEBUG_PRINT: print("[DEBUG- PYTHON] ---- overlay is not
   loaded----")
    exit(-1)
  if overlay.monitor is not None:
    xadc_mon=overlay.monitor.xadc_wiz_0_0
    xadc_mon.write(SRR,0x0000000A) # reset
  else:
    print("ERROR NO XADC")
  if overlay.dtpu is not None:
    accelerator=overlay.dtpu.axis_accelerator_ada
  else:
    print("ERROR NO ACCELERATOR")
    exit(-1)
  overlay.reset()
  # clean power variable
  n_sample=1
  ps_power=0
  pl_power=0
  mem_power=0
  ps_power_max=sys.float_info.min
  pl_power_max=sys.float_info.min
  mem_power_max=sys.float_info.min
  ps_power_min=sys.float_info.max
  pl_power_min=sys.float_info.max
  mem_power_min=sys.float_info.max
  tmp_max=sys.float_info.min
  tmp_min=sys.float_info.max
  tmp_avg=0.00


@ffi.def_extern()
def Init_p(tot_tensors,input_tens_size,output_tens_size):
  global accelerator
  global overlay
  global size_tot
  global input_size
  global output_size
  global avg_hw_execution
  global n_execution
  global avg_hw_execution_internal
  global n_execution_internal
  global tmp_avg
  if _DEBUG_PRINT: print("[DEBUG - PYTHON] --- Init p function ---")
  ## soft reset and accelerator configuration
  accelerator.write(CTRL,0x0000001)
  accelerator.write(CTRL,0x0000000)
  accelerator.write(IARG_RQT_EN,0x000000007) ## all data avialable
    csr, weights and data
  accelerator.write(OARG_LENGTH_MODE,0x00000001) # software mode
  accelerator.write(OARG0_LENGTH,OUTFIFO_SIZE) # size outfifo
  accelerator.write(ISCALAR_RQT_EN,0) # NO input SCALAR
  accelerator.write(OSCALAR_RQT_EN,0) # no output scalar
  accelerator.write(OARG0_TDEST,0) # only one output
  size_tot=tot_tensors
  if _DEBUG_PRINT: print("[DEBUG-PYTHON]--- total tensors",size_tot,"
   ---")
  input_size=input_tens_size
  if _DEBUG_PRINT: print("[DEBUG-PYTHON]--- int tensors",input_size,"
   ---")
  output_size=output_tens_size
  if _DEBUG_PRINT: print("[DEBUG-PYTHON]--- out tensors",
    output_tens_size,"---")
  n_execution=0
  avg_hw_execution=0.00
  avg_hw_execution_internal=0.00
  n_execution_internal=0
  tmp_avg=0.00
  return True


@ffi.def_extern()
def SelectDataTypeComputation_p(data_type):
  global curr_data_precision
  global curr_bitwidth_data_computation
  global PACK_TYPE
  global FP
  global BFP
  global DTYPE_NP
  if _DEBUG_PRINT: print("[DEBUG - PYTHON ] ---
    SelectDataTypeComputation DTPU class ---")
  if data_type!=0:
    #case switch
    if ((data_type)&0x00000f)==INT8:
      curr_data_precision=INT8
      curr_bitwidth_data_computation=8
      if (data_type&0x00100)==SIGNED:
        PACK_TYPE="b"
        DTYPE_NP=np.int8
      else:
        PACK_TYPE="B"
        DTYPE_NP=np.uint8
    elif ((data_type)&0x00000f)==INT16:
      curr_data_precision=INT16
      curr_bitwidth_data_computation=16
      if (data_type&0x00100)==SIGNED:
        PACK_TYPE="h"
        DTYPE_NP=np.int16
      else:
        PACK_TYPE="H"
        DTYPE_NP=np.uint16
    elif ((data_type)&0x00000f)==INT32:
      curr_data_precision=INT32
      curr_bitwidth_data_computation=32
      if (data_type&0x00100)==SIGNED:
        PACK_TYPE="i"
        DTYPE_NP=np.int32
      else:
        PACK_TYPE="I"
        DTYPE_NP=np.uint32
    elif ((data_type)&0x00000f)==INT64:
      curr_data_precision=INT64
      curr_bitwidth_data_computation=64
      if (data_type&0x00100)==SIGNED:
        PACK_TYPE="q"
        DTYPE_NP=np.int64
      else:
        PACK_TYPE="Q"
        DTYPE_NP=np.uint64
    else:
      print("ERROR PYTHON! Setting the Data type of computation")
    # floating point check
    if ((data_type & 0x000060)>> 5)== ACTIVE_FP:
      FP=True
      BFP=False
      PACK_TYPE="f"
      DTYPE_NP=np.float32
    elif ((data_type & 0x000060)>> 5)== ACTIVE_BFP:
      FP=True
      BFP=True
      PACK_TYPE="e"
      DTYPE_NP=np.uint16 ## accroding to tensorflow bfp16
    representation
    else:
      FP=False
      BFP=False
```

```python
  else:
    curr_data_precision=INT8
    curr_bitwidth_data_computation=8
    FP=False
    BFP=False
  if _DEBUG_PRINT:
    print("[DEBUG-PYTHON]----precision default 8 bit signed----")
    print("[DEBUG-PYTHON]---- Signed :",PACK_TYPE.islower(),"type: ", curr_data_precision, "  ->", curr_bitwidth_data_computation," -------")
  return True

@ffi.def_extern()
def push_input_tensor_to_heap( tensor,size,dim_size):
  global input_tensors
  global tot_size_input
  #push the tensor to the heap for handling their transfefr in the Prepare_p
  tot_size=1
  if not(FP) or not(BPF):
    if PACK_TYPE.islower(): # signed
      if curr_data_precision==INT8:
        tensor_i=ffi.cast("int8_t *",tensor)
      elif curr_data_precision==INT16:
        tensor_i=ffi.cast("int16_t *",tensor)
      elif curr_data_precision==INT32:
        tensor_i=ffi.cast("int32_t *",tensor)
      else: # int64
        tensor_i=ffi.cast("int64_t *",tensor)
    else: #unsigned
      if curr_data_precision==INT8:
        tensor_i=ffi.cast("uint8_t *",tensor)
      elif curr_data_precision==INT16:
        tensor_i=ffi.cast("uint16_t *",tensor)
      elif curr_data_precision==INT32:
        tensor_i=ffi.cast("uint32_t *",tensor)
      else: # int64
        tensor_i=ffi.cast("uint64_t *",tensor)
  else:
    if BFP:
      tensor_i=ffi.cast("uint16_t *",tensor)
    else:
      tensor_i=ffi.cast("float *",tensor)
  size_i=ffi.cast("int *",size)
  tot_size=1
  size_l=4*[1]
  data_p=[]
  for i in range(dim_size):
    size_l[i]=size[i]
    tot_size*=size[i]
  tot_size_input+=tot_size
  if _DEBUG_PRINT: print("[DEBUG-PYTHON]------ size of tensor input ",
    tot_size_input,"------")
  for i in range (tot_size):
    data_p.append(tensor_i[i])
  input_tensors.append(Tensor(data_p,tot_size,size_l))

@ffi.def_extern()
def push_output_tensor_to_heap(tensor, size,dim_size):
  global output_tensors
  global tot_size_output
  global output_tensors_p
  #push the tensor to the heap for handling their transfefr in the
    Prepare_p
  tot_size=1
  output_tensors_p.append(tensor)
  if not(FP) or not(BPF):
    if PACK_TYPE.islower(): # signed
      if curr_data_precision==INT8:
        tensor_i=ffi.cast("int8_t *",tensor)
      elif curr_data_precision==INT16:
        tensor_i=ffi.cast("int16_t *",tensor)
      elif curr_data_precision==INT32:
        tensor_i=ffi.cast("int32_t *",tensor)
      else: # int64
        tensor_i=ffi.cast("int64_t *",tensor)
    else: #unsigned
      if curr_data_precision==INT8:
        tensor_i=ffi.cast("uint8_t *",tensor)
      elif curr_data_precision==INT16:
        tensor_i=ffi.cast("uint16_t *",tensor)
      elif curr_data_precision==INT32:
        tensor_i=ffi.cast("uint32_t *",tensor)
      else: # int64
        tensor_i=ffi.cast("uint64_t *",tensor)
  else:
    if BFP:
      tensor_i=ffi.cast("uint16_t *",tensor)
    else:
      tensor_i=ffi.cast("float *",tensor)
  size_i=ffi.cast("int *",size)
  tot_size=1
  size_l=4*[1]
  data_p=[]
  for i in range(dim_size):
    size_l[i]=size[i]
    tot_size*=size[i]
  tot_size_output+=tot_size
  if _DEBUG_PRINT: print("[DEBUG-PYTHON]------ size of tensor output ",tot_size,"------")
  for i in range (tot_size):
    data_p.append(0)
  output_tensors.append(Tensor(data_p,tot_size,size_l))

@ffi.def_extern()
def push_weight_to_heap(tensor,size,dim_size):
  global weight_tensors
  global tot_size_weight
  #push the tensor to the heap for handling their transfefr in the Prepare_p
  tot_size=1
  if not(FP) or not(BPF):
    if PACK_TYPE.islower(): # signed
      if curr_data_precision==INT8:
        tensor_i=ffi.cast("int8_t *",tensor)
      elif curr_data_precision==INT16:
        tensor_i=ffi.cast("int16_t *",tensor)
      elif curr_data_precision==INT32:
        tensor_i=ffi.cast("int32_t *",tensor)
      else: # int64
        tensor_i=ffi.cast("int64_t *",tensor)
    else: #unsigned
      if curr_data_precision==INT8:
        tensor_i=ffi.cast("uint8_t *",tensor)
      elif curr_data_precision==INT16:
        tensor_i=ffi.cast("uint16_t *",tensor)
      elif curr_data_precision==INT32:
        tensor_i=ffi.cast("uint32_t *",tensor)
      else: # int64
        tensor_i=ffi.cast("uint64_t *",tensor)
  else:
    if BFP:
      tensor_i=ffi.cast("uint16_t *",tensor)
    else:
      tensor_i=ffi.cast("float *",tensor)
  size_i=ffi.cast("int *",size)
  tot_size=1
  size_l=4*[1]
  data_p=[]
  for i in range(dim_size):
    size_l[i]=size[i]
    tot_size*=size[i]
  tot_size_weight+=tot_size
  if _DEBUG_PRINT: print("[DEBUG-PYTHON]------ size of tensor weight ",tot_size_weight,"------")
  for i in range (tot_size):
    data_p.append(tensor_i[i])
  weight_tensors.append(Tensor(data_p,tot_size,size_l))

@ffi.def_extern()
def Prepare_p(weight_num):
  global output_fifo_buffer
  global infifo_buffer_transfer
  global weight_buffer
  global csr_buffer
  global overlay
  global driver_wm
  global driver_csr
  global driver_fifo_in
  global driver_fifo_out
  global num_weight
  global global_iteration
  global global_iteration_shift_wm
  global curr_data_precision
  global weight_tensors
  global filter_height
  global filter_width
  global weight_buffer_multiple
  global index_wm
  if _DEBUG_PRINT: print("[DEBUG - PYTHON ] ---- Prepare p of DTPU
    class ----")
  if _DEBUG_PRINT: print("[DEBUG - PYTHON ] ---- in size",input_size,"
    output size",output_size," ----")
  if _DEBUG_PRINT: print("[DEBUG - PYTHON ] ---- weigth size",
    weight_num," ----")
  #allocate buffers for data transfer
  num_weight=weight_num
  filter_height=num_weight*[0]
  filter_width=num_weight*[0]
  ## symmetric input/output fifo
  output_fifo_buffer=allocate(shape=(INFIFO_SIZE,),dtype='u8')
  weight_buffer=allocate(shape=(WMEM_SIZE,),dtype='u8')
  csr_buffer=allocate(shape=(CSRMEM_SIZE,),dtype='u8')
  infifo_buffer_transfer=allocate(shape=(INFIFO_SIZE,),dtype='u8')
  driver_wm=overlay.axi_dma_weight_mem
  driver_csr=overlay.axi_dma_csr_mem
  driver_fifo_in=overlay.axi_dma_infifo
  driver_fifo_out=overlay.axi_dma_outfifo
  #
    ##########################################################################

  ########### populate buffers pack depending on the precision
    #########
  #
    ##########################################################################
  if _DEBUG_PRINT:
    print("[DEBUG - PYTHON ] --- Prepare p of DTPU class ",num_weight
    ,"weight to transfer  ---")
    for i in range(num_weight):
      tmp=weight_tensors[i]
      print("[DEBUG-PYTHON] ---- weight ",i,"------")
      print("[DEBUG-PYTHON] ---- size ",*tmp.size_l,"------")
      for j in range(tmp.tot_dim):
        print(tmp.data[j],end=" ")
    print("",end="\n")
  index_wm=0# it eats the first data ?
  shift=int(64/curr_bitwidth_data_computation)
  iter=int(tot_size_weight/(WMEM_SIZE*(64/
  curr_bitwidth_data_computation))) # if it fits in th eaccelerator
  memory
  # always 4D tensors
  # assumptio is that the filter sizes always fit the accelerator
  if False:
    weight_buffer_multiple=1
    for w_ind in range(1): # pack only the weight for deep wise
    convolution
      tmp=np.array(weight_tensors[w_ind].data, dtype=DTYPE_NP)
      tmp=tmp.reshape(*weight_tensors[w_ind].size_l)
      filter_height[w_ind],filter_width[w_ind]=tmp.shape[1:3]
      for i in range(len(tmp)):
        for l in range(weight_tensors[w_ind].size_l[3]):
          global_iteration_shift_wm.append(index_wm)
          for j in range(len(tmp[i])):
              # boundary check
              shift=int(64/curr_bitwidth_data_computation)
              if shift > len(tmp[i]):
                shift=len(tmp[i])
              weight_buffer[index_wm]=np.uint64(int.from_bytes( tmp[i
    ,j,0:shift,l],byteorder="little",signed=False))
              index_wm+=1
          for j in range(ROWS-len(tmp[i])):
            weight_buffer[index_wm]=0
            index_wm+=1 # padding with zeros
  else:
    #print("it requires multiple iterations for the weight matrix") #
    multiple iteration on total weight 1MB should be enou-gh
    weight_buffer_multiple=[]*np.uint64(0)
    for w_ind in range(1): # pack only the weight for deep wise
    convolution
      tmp=np.array(weight_tensors[w_ind].data, dtype=DTYPE_NP)
      tmp=tmp.reshape(*weight_tensors[w_ind].size_l)
      filter_height[w_ind],filter_width[w_ind]=tmp.shape[1:3]
      for i in range(len(tmp)):
        for l in range(weight_tensors[w_ind].size_l[3]):
          global_iteration_shift_wm.append(index_wm)
          for j in range(len(tmp[i])):
              # boundary check
              shift=int(64/curr_bitwidth_data_computation)
              if shift > len(tmp[i]):
                shift=len(tmp[i])
              weight_buffer_multiple.append(np.uint64(int.from_bytes(tmp[i,j,0:shift,l],byteorder="little",signed=False)))
              index_wm+=1
          for j in range(ROWS-len(tmp[i])):
            weight_buffer[index_wm]=0
            index_wm+=1 # padding with zeros
  if _DEBUG_PRINT:
    for i in range(10):
      print(hex(weight_buffer[i]))
  #################################
  ###### transferring data #######
  #################################
  weight_buffer.flush()
  return True

@ffi.def_extern()
def Invoke_p(only_conv2d,input_shift):
  global infifo_buffer_transfer
  global driver_csr
  global driver_wm
  global driver_fifo_in
  global driver_fifo_out
  global csr_buffer
  global weight_buffer
  global output_fifo_buffer
  global accelerator
  global global_iteration
  global global_iteration_shift_wm
  global curr_data_precision
  global input_tensors
  global output_tensors
  global filter_width
  global filter_height
  global tot_size_output
  global tot_size_input
  global output_tensors_p
  global avg_hw_execution
  global n_execution
  global avg_hw_execution_internal
  global n_execution_internal
  global weight_buffer_multiple
```

```
#
 ##########################################################################
############ populate buffers pack depending on the precision
 #########
#
 ##########################################################################
tmp=[]
if _DEBUG_PRINT:
    print("[DEBUG - PYTHON ] --- Invoke p of DTPU class",input_size-
  num_weight,"input tensors to transfer ---")
    for i in range( input_size-num_weight):
      tmp=input_tensors[i]
      print("[DEBUG-PYTHON] ---- input tensor ",i,"------")
      print("[DEBUG-PYTHON] ---- size ",*tmp.size_l,"------")
      for j in range(tmp.tot_dim):
        print(tmp.data[j],end=" ")
index=0
shift=int(64/curr_bitwidth_data_computation)
# check if it fits the inputs
# always 4D tensors
# assumptio is that the filter sizes always fit the accelerator
#then compact
## split the input shape into submatrices equalt to filter sizes
applyed_weight=0
#over allocate input_fifo_buffer
input_fifo_buffer = []*np.uint64(0)
for w_ind in range(len(input_tensors)):
   tmp=np.array(input_tensors[w_ind].data, dtype=DTYPE_NP)
   tmp=tmp.reshape(*input_tensors[w_ind].size_l)
   for batch in range(len(tmp)):
      for channel in range(tmp.shape[-1]):
         tmp_s=tmp[batch,:,:,channel]
         #iteration for the whole matrix
         for i in range(len(tmp_s)-filter_height[applyed_weight]):
            for j in range(len(tmp_s[i])-filter_width[applyed_weight]):
               tmp_ss=tmp_s[i:i+filter_height[applyed_weight],j:j+
  filter_width[applyed_weight]]
               for row in range(len(tmp_ss)):
                  shift=int(64/curr_bitwidth_data_computation)
                  if shift> len(tmp_ss):
                     shift=len(tmp_ss)
                  input_fifo_buffer.append(np.uint64(int.from_bytes(
  tmp_ss[row,0:shift],byteorder="little",signed=False)))
                  index+=1
input_fifo_buffer=np.array(input_fifo_buffer,dtype='u8')
input_fifo_buffer=np.reshape(input_fifo_buffer,newshape=(index,))
if _DEBUG_PRINT:
    for i in range(10):
        print(hex(input_fifo_buffer[i]))
#iterate on the output matrix with also multiple weight iteration and inputs
## assumption is that the output tensor is always one!
## getting the output matrix structure
#accelerator.write(CMD, (0x0000000 |(CMD_EXECUTE_CONTINOUS<<16)))
output_matrix=np.array(output_tensors[0].data, dtype=DTYPE_NP)
output_matrix=output_matrix.reshape(*output_tensors[0].size_l)
point_wise=np.array(weight_tensors[1].data,dtype=DTYPE_NP)
#####################################
####### deepwise convolution #########
#####################################
if _DEBUG_PRINT:
    print("[DEBUG-PYTHON] ---------- deepwise convolution --------")
if _TIME_PROBES:
    start_time=time.time()
for shift_w in range(math.ceil(len(weight_buffer_multiple)/WMEM_SIZE)):
    ###############################################
    ###### program the dma for the weight ##########
    ###############################################
    if _DEBUG_PRINT: print("[DEBUG-PYTHON]--- transfering weight buffer ----")
    weight_buffer[0:len(weight_buffer_multiple[WMEM_SIZE*(shift_w):WMEM_SIZE*(shift_w+1)])]=weight_buffer_multiple[WMEM_SIZE*(shift_w):WMEM_SIZE*(shift_w+1)]
    driver_wm.sendchannel.transfer(weight_buffer)
    driver_wm.sendchannel.wait()
    for batch_i in range(input_tensors[0].size_l[0]):
        for channel_i in range(input_tensors[0].size_l[-1]):
            ###############################################
            ###### program the dma for the csr reg #########
            ###############################################
            if _DEBUG_PRINT: print("[DEBUG-PYTHON]--- transfering csr buffer for weight----")
            csr_buffer[ARITHMETIC_PRECISION]=(global_iteration_shift_wm[channel_i]<<32) | ((NO_FP<<8)) | (ACTIVATE_CHAIN<<4)| (curr_data_precision)
            #csr_buffer.flush()
            driver_csr.sendchannel.transfer(csr_buffer)
            #driver_csr.sendchannel.wait()
            for infifo_shift in range(math.ceil(input_fifo_buffer.size/INFIFO_SIZE)):
                #####################################################
                ###### program the dma for the in/out fifos ##########
                #####################################################
                if _TIME_PROBES:
                    start_time_i=time.time()
                if _DEBUG_PRINT: print("[DEBUG-PYTHON]--- transfering input buffer",infifo_shift," ----")
                infifo_buffer_transfer[0:input_fifo_buffer[INFIFO_SIZE*(infifo_shift):INFIFO_SIZE*(infifo_shift+1)].size]=input_fifo_buffer[INFIFO_SIZE*(infifo_shift):INFIFO_SIZE*(infifo_shift+1)]
                driver_fifo_in.sendchannel.transfer(infifo_buffer_transfer)
                #driver_fifo_in.sendchannel.wait()
                accelerator.write(OARG0_LENGTH,OUTFIFO_SIZE) # size outfifo
                accelerator.write(CMD, (0x0000000 |(CMD_EXECUTE_STEP<<16)))
                accelerator.write(CMD,((CMD_UPDATE_OUT_ARG<<16)|(1)))
                driver_fifo_out.recvchannel.transfer(output_fifo_buffer)
                if _DEBUG_PRINT: print("[DEBUG-PYTHON]----- getting output data -----")
                driver_fifo_out.recvchannel.wait()
                if _TIME_PROBES:
                    end_time_i=time.time()
                    avg_hw_execution_internal+=end_time_i-start_time_i
                    n_execution_internal+=1
                if _DEBUG_PRINT: print(output_fifo_buffer)
                accelerator.write(CMD,((CMD_UPDATE_IN_ARG<<16)|(4))) # update input fifo
                #################################################################################
                ####### unpack the output buffer depending on the precision ########
                #################################################################################
                ## get values from output fifo buffer and put them into an array in order to sum all the data
                for i in range(output_matrix.shape[1]-1):
                    for j in range(output_matrix.shape[2]-1):
                      tmp_sum=np.zeros(shape=(ROWS,int(64/curr_bitwidth_data_computation)),dtype=DTYPE_NP)
                      tmp_data=output_fifo_buffer[channel_i*(ROWS*COLUMNS)+i*ROWS+j*COLUMNS:channel_i*(ROWS*COLUMNS)+(i+1)*ROWS+(j+1)*COLUMNS]
                      tmp_sum=np.frombuffer(tmp_data.tobytes(),dtype=DTYPE_NP)
                      #reshuffle and check if it is worth it
                      #if tmp_data.size >0:
                      #  for row in range(len(tmp_data)):
                      #     if row in tmp_data:
                      #       tmp_sum[row]=np.frombuffer(tmp_data[row].tobytes(),dtype=DTYPE_NP)#convert(tmp_data[row])
                      output_matrix[batch_i,i,j,channel_i]=np.multiply(tmp_sum.sum(dtype=DTYPE_NP),point_wise[channel_i],dtype=DTYPE_NP)
                      accelerator.write(CMD,((CMD_UPDATE_IN_ARG<<16)|(1))) # update csr
                    accelerator.write(CMD,((CMD_UPDATE_OUT_ARG<<16)|(1)))
                accelerator.write(CMD,((CMD_UPDATE_IN_ARG<<16)|(2))) # update w memory
          #if _DEBUG_PRINT:
          # print("[DEBUG-PYTHON]-------- point wise convolution ----------")
          if _TIME_PROBES:
            end_time=time.time()
            avg_hw_execution+=end_time-start_time
            n_execution+=1
          accelerator.write(STATUS,0x00000003)##clear status
          #accelerator.write(CMD,((CMD_UPDATE_IN_ARG<<16)|(1))) # update csr
          #accelerator.write(CMD, (0x0000000 |(CMD_STOP_EXECUTE_CONTINOUS<<16))) # stop accelerator
          ##############################################
          ######## point wise convolution ########### moved inside previous loop
          ##############################################
          #for batch_i in range(len(output_matrix)):
          #  for i in range(len(output_matrix[batch_i])):
          #    for j in range(len(output_matrix[batch_i,i])):
          #      for channel_i in range(len(output_matrix[batch_i,i,j])):
          #        output_matrix[batch_i,i,j,channel_i]=output_matrix[batch_i,i,j,channel_i]*weight_tensors[1].data[channel_i]
          if _DEBUG_PRINT: print("[DEBUG -PYTHON] ---- accelerator done ----")
          if _DEBUG_PRINT:
            print("[DEBUG-PYTHON] ------ final output data to tensorflow ------")
            print(output_matrix)
          # copy the output matrix to tensorflow environment ffi.memmove(dest,src,nbytets)
          ffi.memmove(ffi.buffer(output_tensors_p[0],output_matrix.nbytes),output_matrix,output_matrix.nbytes)
          # save the pointer to the output and then substitute the values into the point wise convolution
          #clean up input/output
          input_tensors=[]
          output_tensors=[]
          tot_size_input=0
          tot_size_output=0
          del input_fifo_buffer
          return True

@ffi.def_extern()
def ResetHardware_p():
  global accelerator
  global overlay
```

```
    if _DEBUG_PRINT: print("[DEBUG - PYTHON ] ---- Reset hardware p
      function ----")
    overlay.reset()
    accelerator.write(CTRL,0x0000001)
    accelerator.write(CTRL,0x0000000)
    return True

@ffi.def_extern()
def destroy_p():
    global infifo_buffer_transfer
    global output_fifo_buffer
    global csr_buffer
    global weight_buffer
    global accelerator
    global overlay
    global global_iteration_shift_wm
    global weight_tensors
    global input_tensors
    global output_tensors
    if _DEBUG_PRINT: print("[DEBUG - PYTHON ] ---- destroying the
      buffers ----")
    infifo_buffer_transfer.freebuffer()
    output_fifo_buffer.freebuffer()
    csr_buffer.freebuffer()
    weight_buffer.freebuffer()
    del accelerator
    del overlay
    del global_iteration_shift_wm
    del weight_tensors
    del input_tensors
    del output_tensors
    return True

@ffi.def_extern()
def CopyFromBufferHandle_p():
    if _DEBUG_PRINT: print("[DEBUG - PYTHON ] ----  the  from  delegate
      and buffers ----")
    return True

@ffi.def_extern()
def CopyToBufferHandle_p():
    if _DEBUG_PRINT: print("[DEBUG - PYTHON ] ---- copying  to  the
      delegate and buffers ----")
    return True
@ffi.def_extern()
def FreeBufferHandle_p():
    global output_fifo_buffer
    global csr_buffer
    global weight_buffer
```

```
  global driver_csr
  global driver_wm
  global driver_fifo_in
  global driver_fifo_out
  global accelerator
  if _DEBUG_PRINT: print("[DEBUG - PYTHON ] --- freeing buffers ---")
  output_fifo_buffer.freebuffer()
  csr_buffer.freebuffer()
  weight_buffer.freebuffer()
  del accelerator
  del driver_csr
  del driver_wm
  del driver_fifo_in
  del driver_fifo_out

@ffi.def_extern()
def start_power_consumption():
  global xadc_mon
  if _DEBUG_PRINT: print("[DEBUG-PYTHON] ---- start measurement of power consumption ----")
  if xadc_mon is not None:
    try:
      _thread.start_new_thread( sample_power, ("Sampling power", 0.5 ) ) # every 1ms
    except:
      print("Error: unable to start thread")
  return True

@ffi.def_extern()
def print_power_consumption_p():
  global xadc_mon
  global ps_power
  global pl_power
  global mem_power
  if _DEBUG_PRINT: print("[DEBUG-PYTHON] ---- printing power consumption from xadc readings ----")
  ###############################################
  ### Retrieve and display current temperature ###
  ###############################################
  tmp=( xadc_mon.read(TEMPERATURE) & 0x0000FFF0) >> 4
  tmp=(tmp* 503.975)/4096 - 273.15
  print("Current temperature:",round(tmp,3)," C")
  print("Average execution temperature:", round(tmp_avg/n_sample,3)," C")
  print("Max temperature:", round(tmp_max,3) ," C")
  print("Min temperature:", round(tmp_min,3) ," C")
  # printing power consumption
  tot_power=ps_power+pl_power+mem_power
  print("Average power consumption=", round(tot_power*1000/n_sample,5)," mWatt")
  print("---> Processing System:",round(ps_power*1000/n_sample,5)," mWatt")
  print("---> Programmable Logic:",round(pl_power*1000/n_sample,5)," mWatt")
  print("---> Memory:",round(mem_power*1000/n_sample,3)," mWatt")
  print("Maximum power consumption")
  print("---> Processing System:",round(ps_power_max*1000,5)," mWatt")
  print("---> Programmable Logic:",round(pl_power_max*1000,5)," mWatt")
  print("---> Memory:",round(mem_power_max*1000,3)," mWatt")
  print("Minimum power consumption")
  print("---> Processing System:",round(ps_power_min*1000,5)," mWatt")
  print("---> Programmable Logic:",round(pl_power_min*1000,5)," mWatt")
  print("---> Memory:",round(mem_power_min*1000,5)," mWatt")
  return True

@ffi.def_extern()
def activate_time_probe_p(activate):
  global _TIME_PROBES
  if _DEBUG_PRINT: print("[DEBUG-PYTHON]--- activating time probe in python -----")
  if not(_TIME_PROBES) and activate:
    print("Time probes activated")
    _TIME_PROBES=True

@ffi.def_extern()
def print_python_time_probes():
  if _DEBUG_PRINT: print("[DEBUG-PYTHON]------ printing python time probes -----")
  print("Hardware execution time and rebuilding output matrix:", avg_hw_execution/n_execution," [s]")
  print("Hardware execution time:", avg_hw_execution_internal/n_execution_internal," [s]")
  #print("Hardware calls:",n_execution_internal)
  return True
```

Appendix B

Top level entity of DTPU core

../../dtpu/files/dtpu_core.v

```verilog
// ====================================================================
//  Filename      : dtpu_core.v
//  Created On    : 2020-04-22 17:05:56
//  Last Modified : 2020-05-20 15:03:03
//  Revision      :
//  Author        : Angione Francesco
//  Company       : Chalmers University of Technology,Sweden - Politecnico di Torino, Italy
//  Email         : francescoangione8@gmail.com
//
//  Description   : Cogitantium, the dumb tensor processor unit,top level enity of the accelerator
//
//
// ====================================================================

`timescale 1ns / 1ps
`include "precision_def.vh"

//`define DUMMY

module dtpu_core
```

```verilog
#(parameter DATA_WIDTH_MAC=64,
    ROWS=3 ,
    COLUMNS=3,
    SIZE_WMEMORY=8196,
    ADDRESS_SIZE_WMEMORY=32,
    ADDRESS_SIZE_CSR=32,
    SIZE_CSR=1024,
    DATA_WIDTH_CSR=8,
    DATA_WIDTH_WMEMORY=64,
    DATA_WIDTH_FIFO_IN=64,
    DATA_WIDTH_FIFO_OUT=64,
    MAX_BOARD_DSP=220
    )
(
    input wire clk,
    (* X_INTERFACE_INFO = "xilinx.com:signal:reset:1.0 aresetn RST" *)
    (* X_INTERFACE_PARAMETER = "POLARITY ACTIVE_LOW" *)
    input wire aresetn,
    input wire test_mode,
    input wire enable,

    ///////////////////////////
    ////// CSR INTERFACE ///////
    ///////////////////////////
    (* X_INTERFACE_PARAMETER = "MASTER_TYPE BRAM_CTRL,MEM_ECC no,MEM_WIDTH 8,MEM_SIZE 1024 " *)
    (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl:1.0 csr_mem_interface EN" *)
    output wire         csr_ce,
    (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl:1.0 csr_mem_interface DOUT" *)
    input wire [DATA_WIDTH_CSR-1:0]     csr_dout,
    (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl:1.0 csr_mem_interface DIN" *)
    output wire [DATA_WIDTH_CSR-1:0]    csr_din,
    (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl:1.0 csr_mem_interface WE" *)
    output wire         csr_we,
    (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl:1.0 csr_mem_interface ADDR" *)
    output wire [ADDRESS_SIZE_CSR-1:0]     csr_address,
    (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl:1.0 csr_mem_interface CLK" *)
    output wire           csr_clk,
 (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl:1.0
 csr_mem_interface RST" *)
 output wire         csr_reset,

   ////////////////////////////
   ////// WEIGHT MEMORY ///////
   ///////////////////////////
   (* X_INTERFACE_PARAMETER = "MASTER_TYPE BRAM_CTRL,
MEM_ECC no,MEM_WIDTH 64,MEM_SIZE 8192 " *)
   (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl
:1.0 weight_mem_interface EN" *)
   output wire  wm_ce,
   (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl
:1.0 weight_mem_interface DOUT" *)
   input wire [DATA_WIDTH_WMEMORY-1:0]       wm_dout,
   (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl
:1.0 weight_mem_interface DIN" *)
   output wire [DATA_WIDTH_WMEMORY-1:0]       wm_din,
   (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl
:1.0 weight_mem_interface WE" *)
   output wire              wm_we,
   (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl
:1.0 weight_mem_interface ADDR" *)
   output wire [ADDRESS_SIZE_WMEMORY-1:0]  wm_address,
   (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl
:1.0 weight_mem_interface CLK" *)
   output wire    wm_clk,
   (* X_INTERFACE_INFO = "xilinx.com:interface:bram_rtl
:1.0 weight_mem_interface RST" *)
   output wire           wm_reset,

   ////////////////////////////////////////////
   /////////// INPUT DATA FIFO ////////////////
   ////////////////////////////////////////////
   /////////// using stream axi
   (* X_INTERFACE_INFO = "xilinx.com:interface:
acc_fifo_read:1.0 input_fifo RD_DATA" *)
   input wire [DATA_WIDTH_FIFO_IN-1:0] infifo_dout,
     (* X_INTERFACE_INFO = "xilinx.com:interface:
acc_fifo_read:1.0 input_fifo RD_EN" *)
   output wire infifo_read,
     (* X_INTERFACE_INFO = "xilinx.com:interface:
acc_fifo_read:1.0 input_fifo EMPTY_N" *)
   input wire infifo_is_empty,


   ///////////////////////////////////////////
   /////////// OUTPUT DATA FIFO ///////////////
   ///////////////////////////////////////////
   /////////// using stream axi
   (* X_INTERFACE_INFO = "xilinx.com:interface:acc_fifo_write:1.0 output_fifo WR_DATA" *)
   output wire [DATA_WIDTH_FIFO_OUT-1:0] outfifo_din,
     (* X_INTERFACE_INFO = "xilinx.com:interface:acc_fifo_write:1.0 output_fifo WR_EN" *)
   output wire outfifo_write,
     (* X_INTERFACE_INFO = "xilinx.com:interface:acc_fifo_write:1.0 output_fifo FULL_N" *)
   input wire outfifo_is_full,

   ///////////////////////////////////////////
   /////////// CONTROL FROM/TO PS /////////////
   ///////////////////////////////////////////
    (* X_INTERFACE_INFO = "xilinx.com:interface:acc_handshake_rtl:1.0 control_interface ap_start" *)
   input wire cs_start,
    (* X_INTERFACE_INFO = "xilinx.com:interface:acc_handshake_rtl:1.0 control_interface ap_ready" *)
   output wire cs_ready,
    (* X_INTERFACE_INFO = "xilinx.com:interface:acc_handshake_rtl:1.0 control_interface ap_done" *)
   output wire cs_done,
    (* X_INTERFACE_INFO = "xilinx.com:interface:acc_handshake_rtl:1.0 control_interface ap_continue" *)
   input wire cs_continue,
    (* X_INTERFACE_INFO = "xilinx.com:interface:acc_handshake_rtl:1.0 control_interface ap_idle" *)
   output wire cs_idle,

    // debug state
   output wire[3:0]state,
   output wire[3:0]d_out
     );
 //////////////////////////////////////////
 ///*********************************///
 ///// --------- Cogitantium -------- /////
 ///// the dumb tensor processing unit ////
 ///*********************************///
 //////////////////////////////////////////

```

```
   wire [COLUMNS*ROWS*DATA_WIDTH_FIFO_OUT-1:0] weight_to_mxu;
   wire [COLUMNS*DATA_WIDTH_FIFO_IN-1:0] input_data_to_mxu;
   wire [ROWS*DATA_WIDTH_FIFO_OUT-1:0] output_data_from_mxu;
   wire enable_deskew_ff_i,enable_enskew_ff_i;
   wire [`LOG_ALLOWED_PRECISIONS-1:0] data_precision;
   wire enable_i;
   wire enable_load_array;
   wire [ROWS*COLUMNS-1:0]read_weight_memory;
   wire [COLUMNS:0]enable_load_activation_data;
   wire [COLUMNS:0]enable_store_activation_data;
   wire enable_cnt;
   wire ld_max_cnt;
   wire enable_cnt_weight;
   wire ld_max_cnt_weight;
   wire enable_chain;
   wire ld_weight_page_cnt;
   wire [1:0]enable_fp_unit;

   wire [ADDRESS_SIZE_WMEMORY-1:0]start_value_wm;
   wire [$clog2(COLUMNS):0]max_cnt_from_cu;
   wire [$clog2(ROWS*COLUMNS):0]max_cnt_weight_from_cu;
   wire reset_i;

   assign d_out=data_precision;

   assign reset_i=~aresetn;
  ////////////////////////////////////////////
 ////// MATRIX MULTIPLICATION UNIT //////////
 ////////////////////////////////////////////
mxu_wrapper
 #(.M(ROWS), // matrix row -> weights
     .K(COLUMNS), // matrix columsn -> input data
     .max_data_width(DATA_WIDTH_MAC),// it must be a divisor of 64
     .MAX_BOARD_DSP(MAX_BOARD_DSP)
     ) engine  (
         .data_type(data_precision),
         .reset(reset_i),
         .clk(clk),
         .enable(enable_i),
         .enable_chain(enable_chain),
         .enable_fp_unit(enable_fp_unit),
         .enable_in_ff(enable_enskew_ff_i),
         .enable_out_ff(enable_deskew_ff_i),
         .test_mode(test_mode),
         .input_data(input_data_to_mxu),
         .weight(weight_to_mxu),
         .y(output_data_from_mxu)
     );

 //////////////////////////////////////////////
 ///////////// CONTROL UNIT ///////////////////
 //////////////////////////////////////////////
 control_unit #( .DATA_WIDTH_FIFO_IN(DATA_WIDTH_FIFO_IN),
             .DATA_WIDTH_FIFO_OUT(DATA_WIDTH_FIFO_OUT),
             .DATA_WIDTH_WMEMORY(DATA_WIDTH_WMEMORY),
             .DATA_WIDTH_CSR(DATA_WIDTH_CSR),
             .ROWS(ROWS),
             .COLUMNS(COLUMNS),
             .ADDRESS_SIZE_CSR(ADDRESS_SIZE_CSR),
             .ADDRESS_SIZE_WMEMORY(ADDRESS_SIZE_WMEMORY))
 cu(
     .clk(clk),
     .reset(reset_i),
     .test_mode(test_mode),
     .glb_enable(enable),
     .enable_mxu(enable_i),
     .csr_address(csr_address),
     .csr_dout(csr_dout),
     .csr_ce(csr_ce),
     .csr_reset(csr_reset),
     .csr_we(csr_we),
     .wm_ce(wm_ce),
     .wm_reset(wm_reset),
     .wm_we(wm_we),
     .infifo_is_empty(infifo_is_empty),
     .infifo_read(infifo_read),
     .outfifo_is_full(outfifo_is_full),
     .outfifo_write(outfifo_write),
     .cs_continue(cs_continue),
     .cs_done(cs_done),
     .cs_idle(cs_idle),
     .cs_ready(cs_ready),
     .cs_start(cs_start),
     .state_out(state),
     .enable_deskew_ff(enable_deskew_ff_i),
     .enable_enskew_ff(enable_enskew_ff_i),
     .enable_fp_unit(enable_fp_unit),
     .enable_chain(enable_chain),
     .enable_load_array(enable_load_array),
     .data_precision(data_precision),
     .read_weight_memory(read_weight_memory),
     .enable_load_activation_data(
enable_load_activation_data),
     .enable_store_activation_data(
enable_store_activation_data),
     .enable_cnt(enable_cnt),
     .ld_max_cnt(ld_max_cnt),
     .enable_cnt_weight(enable_cnt_weight),
     .ld_max_cnt_weight(ld_max_cnt_weight),
     .ld_weight_page_cnt(ld_weight_page_cnt),
     .start_value_wm(start_value_wm),
     .max_cnt_from_cu(max_cnt_from_cu), // it depends on
the current bitwidt [$clog2(COLUMNS):0]
     .max_cnt_weight_from_cu(max_cnt_weight_from_cu) //[
$clog2(ROWS):0]

        );

//////////////////////////////////////////////
///////// LOAD AND STORE ARRAY     /////////
//////////////////////////////////////////////
`ifndef DUMMY


ls_array
#(   .ROWS(ROWS),
  .COLUMNS(COLUMNS),
  .data_in_width(DATA_WIDTH_FIFO_IN),
  .data_in_mem(DATA_WIDTH_WMEMORY),
  .address_leng_wm(ADDRESS_SIZE_WMEMORY),
  .size_wmemory(SIZE_WMEMORY)) ls_array_inst
(
.clk(clk),
.reset(reset_i),
.enable_load_array(enable_load_array),
.data_precision(data_precision),
.read_weight_memory(read_weight_memory),
.infifo_read(infifo_read),
.outfifo_write(outfifo_write),
.input_data_from_fifo(infifo_dout), //[data_in_width-1:0]
.data_to_fifo_out(outfifo_din), //[data_in_width-1:0]
```

```
.data_from_weight_memory(wm_dout), //[data_in_mem-1:0]
.data_from_mxu(output_data_from_mxu), //[data_in_width*
 ROWS-1:0]
.data_to_mxu(input_data_to_mxu), //[data_in_width*COLUMNS
 -1:0]
.weight_to_mxu(weight_to_mxu), //[data_in_width*ROWS-1:0]
.wm_address(wm_address), //[address_leng_wm-1:0]
.enable_load_activation_data(enable_load_activation_data),
.enable_store_activation_data(enable_store_activation_data
 ),
.enable_cnt(enable_cnt),
.ld_max_cnt(ld_max_cnt),
.enable_cnt_weight(enable_cnt_weight),
.ld_max_cnt_weight(ld_max_cnt_weight),
.ld_weight_page_cnt(ld_weight_page_cnt),
.start_value_wm(start_value_wm),
.max_cnt_from_cu(max_cnt_from_cu), // it depends on the
 current bitwidt [$clog2(COLUMNS):0]
.max_cnt_weight_from_cu(max_cnt_weight_from_cu) //[$clog2(
 ROWS):0]
);


`endif



`ifdef DUMMY
always @(posedge(clk)) begin
if(reset_i) begin
input_data_from_fifo<=0;
weight_from_memory<=0;
end else begin
          if (enable_load_array && infifo_read ) begin
          input_data_from_fifo<=infifo_dout;
          weight_from_memory<= wm_dout;
          end

end
end
// dummy assignment for 3 columns and rows
assign outfifo_din=( outfifo_write ? input_data_to_fifo:
 64'b0);


```

```
  `endif

  // same clock for bram interface
  assign csr_clk=clk;
  assign wm_clk=clk;

endmodule
```