Universidade do Minho
Escola de Engenharia

Pedro Miguel Coelho Pereira

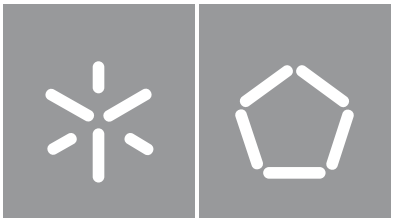
Efficient hardware implementation of 2D convolutions, optimized for point cloud and adjustable to the 3D model requirements for object detection and classification

novembro 2021

Efficient hardware implementation of 2D convolutions, optimized for point cloud and adjustable to the 3D model requirements for object detection and classification

Pedro Pereira

UMinho | 2021



Universidade do Minho
Escola de Engenharia

Pedro Miguel Coelho Pereira

Efficient hardware implementation of 2D convolutions, optimized for point cloud and adjustable to the 3D model requirements for object detection and classification

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Rui Machado
e coorientação do
Investigador Doutor Duarte Fernandes

novembro de 2021

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Acknowledgments

Firstly, I would like to thank my advisor, doctor Rui Machado, as well as my co-advisor, doctor Duarte Fernandes, for all their help and availability both in the work development and in the writing of this dissertation. I finish this dissertation with a much more precise perception of the research concept, thanks to the support given to me from the beginning to the end, despite all the constraints.

Special thanks to my colleague and longtime friend, Duarte Silva, with whom I had the pleasure of working with during this dissertation. I recognize in him an enormous commitment, resilience, and responsibility for the work. With him, this long walk was more instructive, and without a doubt, more fun.

To my family, in particular to my parents, Paula and Adelino, and my sister, Maria, an acknowledgement for all their unconditional support. The constant attention and encouragement were crucial to my personal and academic growth. Additionally, a big thanks to my grandparents, from whom I always affectionately receive extra motivation. I am proud to have them always by my side.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Implementação eficiente em hardware de convoluções 2D, otimizadas para nuvem de pontos e ajustáveis aos requisitos dos modelos 3D de deteção e classificação de objetos

Com o crescente interesse na integração do sensor LiDAR na construção de veículos autónomos, a comunidade académica e indústria têm contribuído para um aumento de algoritmos baseados em dados LiDAR para deteção de objetos. Dado o sucesso das CNNs em tarefas complexas, variantes têm surgido para o processamento de dados 3D, contudo, a natureza esparsa e não estruturada da nuvem de pontos forçou a literatura a aumentar a complexidade dos modelos. O aumento de complexidade e a necessidade de uma computação distribuída, leva à necessidade de plataformas capazes de executar e acelerar os modelos de forma a viabilizar o seu uso em aplicações de tempo real. Considerando os requisitos de desempenho e consumo de potência, várias soluções baseadas em FPGA têm sido apresentadas, porém, geralmente os modelos não são eficientemente desenhados para plataformas com limitação de recursos.

A presente dissertação enquadra-se na necessidade da aplicação de soluções baseadas em CNN otimizadas para nuvem de pontos em dispositivos com recursos reduzidos. O desenho e implementação de um módulo convolucional foi proposto para implementar CNNs em hardware. Ao nível da configurabilidade, é possível ajustar todos os parâmetros típicos e explorar o paralelismo consoante a restrição de recursos, tornando-se uma solução capaz de executar qualquer convolução encontrada na literatura. Usando como caso de estudo o modelo PointPillars, o uso do módulo permitiu diminuir o tempo de processamento até 25% sem comprometer o desempenho nas deteções.

Dada a esparsidade encontrada nos dados LiDAR e a necessidade de um estudo dedicado ao potencial das convoluções esparsas, esta dissertação compara o desempenho de CNNs tradicionais com uma solução otimizada para pontos 3D, chamada esquemas voting. Comparativamente com a convolução densa, a convolução voting provou ser mais rápida para dados com esparsidade superior a 89%. O processamento em pipeline permite consumir apenas duas DSPs e aproveitar a dependência de dados espacialmente próximos para reduzir o tempo de processamento até 30%. Para operações com stride, a convolução de voting é capaz de diminuir a comunicação com a memória e também reduzir o tempo de processamento em 55%. Já a sua integração no modelo PointPillars demonstrou que a convolução voting é capaz de dimiuir o tempo de processamento até 80.44% nas primeiras camadas do Backbone.

palavras-chave: CNN, Convolução, FPGA, LiDAR, Nuvem de Pontos

Abstract

Efficient hardware implementation of 2D convolutions, optimized for point cloud and adjustable to the 3D model requirements for object detection and classification

With the growing interest in LiDAR sensor integration in the construction of autonomous vehicles, the academic community and industry have contributed to an increase in LiDAR-based algorithms for object detection. Given the success of CNNs in complex tasks such as object recognition, variants have emerged for 3D data processing, however, the sparse and unstructured nature of the point cloud has forced the literature to increase the complexity of the models. The increase in complexity and the need for distributed computing leads to the need for platforms capable of executing and accelerating these models to enable their use in real-time applications. Considering the performance and power consumption requirements, several FPGA-based solutions have been presented, however, generally, the models are not efficiently designed for platforms with limited resources.

This dissertation fits into the need to apply CNN-based solutions optimized for point cloud in devices with reduced resources. The design and implementation of a convolutional module were proposed to implement CNNs in hardware. In terms of configurability, it is possible to adjust all typical parameters and explore parallelism depending on the resource constraints, making it a solution capable of performing any convolution found in the literature. Using the PointPillars model as a case study, the use of the module allowed to reduce the processing time up to 25% without compromising the detections performance.

Given the sparseness found in the LiDAR data and the need for a study dedicated to the potential of sparse convolutions, this dissertation compares the performance of traditional CNNs with an optimized solution for 3D points, called voting schemes. Compared to dense convolution, voting convolution proved to be faster for data with sparsity greater than 89%. Pipeline processing allowed to consume only two DSPs and take advantage of spatially close data dependency to reduce the processing time by up to 30%. For stride operations, the voting convolution is able to decrease memory communication and also reduce the processing time by 55%. From the integration with PointPillars, it was possible to reduce the processing time up to 80.44% in the first layers of the Backbone.

keywords: CNN (Convolutional Neural Network), Convolution, FPGA (Field Programmable Gate Array), LiDAR (Light Detection And Ranging), Point cloud

Contents

List of Figures

List of Tables

Code Snippets

List of Abbreviations

BEV	Bird Eye View.
BRAM	Block Random Access Memory.
CNN	Convolutional Neural Network.
DHM	Direct Hardware Mapping.
DSP	Digital Signal Processing.
FPGA	Field Programmable Gate Array.
FV	Front View.
GPU	Graphic Processing Unit.
HLS	High-level synthesis.
IDE	Integrated Development Environment.
ISA	Instruction Set Architecture.
LiDAR	Light Detection And Ranging.
LUT	LookUp Table.
MLP	Multilayer Perceptron.
MPSoC	Multiprocessor System On a Chip.
ReLU	Rectified Linear Unit.
ROI	Region of Interest.
RPN	Region Proposal Network.
RTL	Register Transfer Level.
SIFT	Scale Invariant Feature Transform.
SoC	System On a Chip.
SURF	Speeded Up Robust Features.

Chapter 1: Introduction

This chapter contextualizes the work developed, states the objectives and describes the document structure. The contextualization places the work in the current panorama of autonomous vehicles perception and the needs inherent to the paradigm shift for Edge Computing. Afterwards, the objectives of this dissertation are described, structuring the stages during the development and followed by the description of the adopted methodology. The chapter ends with the organization of the dissertation, to guide the reader throughout the document.

1.1 Contextualization

Autonomous vehicles are increasingly present in the mobility reality and with perspectives of the central focus of innovation in the future [18]. The evolution of sensor technologies has been a support for developments around autonomous vehicles as they heavily rely on perception systems to acquire information about the immediate surroundings [19]. A perception system for autonomous vehicle navigation can be composed of a combination of active and passive sensors, in particular, cameras, radars, and Light Detection And Rangings (LiDARs) [20]. LiDARs are considered active sensors that emit lasers to the surroundings and measure the distances by processing the laser returns received from the reflecting surfaces.

Although there has been a lot of progress regarding camera-based perception, the distance to an object is simply estimated by using image processing methods [21]. As vehicles are constantly exposed to highly dynamic scenarios, a system that relies on distance estimation faces many difficulties which motivate high-level autonomous vehicles using LiDARs as the main component for their perception system [19]. In recent years, companies such as Google [22] and Volvo [23], are developing distinctive LiDAR systems and including them in their setups to ensure the accurate recognition of the 3D space around the vehicle. Figure 1.1 presents an example with a standard station wagon equipped with two high-resolution color and grayscale video cameras. Accurate ground truth is provided by a Velodyne laser scanner and a GPS localization system. This setup was used to capture datasets by driving around the mid-size city of Karlsruhe, providing real-world benchmarks with novel difficulties to the community

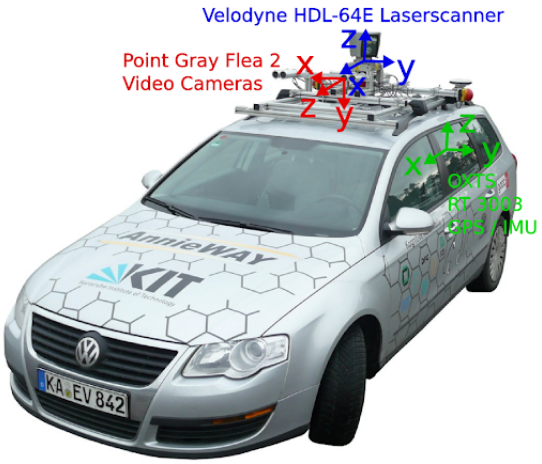


Figure 1.1: Volkswagen Passat B6 equipped with a Velodyne HDL-64E and four cameras [1]

Notable advances in 3D sensors, namely around LiDARs, have drawn the attention of the research community, resulting in the increased development of LiDAR-based algorithms for object detection, classification, tracking, and intention prediction [24]. Despite the LiDAR's superiority in ranging accuracy, camera-based algorithms are good solutions at object recognition, promoting the combination of the two sensors to complement each other [21].

While traditional model-based LiDAR data processing methods are computation friendly and have had years of progress, deep learning-based methods have revealed great potential for perception-type tasks. Recent advances in deep Convolutional Neural Networks (CNNs) have motivated researchers to adapt CNNs to directly model points from point clouds [25]. Regarding the processing of 2D data, the traditional implementations of CNNs have proved to be quite competent, and in recent years, variations have appeared to adapt them for 3D data processing [24].

The characteristics of the data collected from a LiDAR sensor, such as the amount of data (1.3-3 million) [2], sparse and unstructured nature, make dense CNNs (adopted in computer vision for RGB image processing) inappropriate for data processing [26]. As a result, the literature was forced to increase the complexity of object detection models, compared to the ones already developed for computer vision [24]. This increase in complexity leads not only to the adoption of optimized logical mechanisms for sparse data processing but also the use of CNN optimization techniques, such as pruning and quantization [27]. Pruning has proven to be a plausible solution for reducing computational volume and resource consumption in data management, while data quantization reduces the size of the data involved in the arithmetic operations, consequently decreasing the complexity and the memory required.

Given the success of CNNs in complex tasks such as object detection [28], and the need for distributed computing in autonomous vehicles [29] [30], the availability of platforms capable of executing and accel-

erating CNNs, with low power consumption and reduced dimensions, is essential to enable their adoption in real-time applications. For critical applications such as autonomous driving, both real-time performance and power consumption need to be carefully considered. Although Graphic Processing Units (GPUs) are a popular platform for parallel processing, both power and area consumption are usually high [27]. The need for a solution that satisfies the power and performance requirements has directed the focus to more efficient platforms, such as Field Programmable Gate Arrays (FPGAs) [29]. FPGAs can be reprogrammed to the desired application requirements with high level of parallel processing and on-chip data communications, emerging as a real-time low-power embedded system. The computational cost required by a CNN makes one of the main challenges the resources limitations of the target platform. The design of FPGA-based accelerators for CNNs, despite being a topic widely addressed by the research communities over the past few years [30], still presents several technical challenges, namely:

- **Limited FPGA resources**: For CNN models used in real classification tasks, the computing and memory resources of an FPGA may be insufficient [31]. Although some FPGAs boards have higher available resources and bandwidth, this implies a large consumption of both power and area [32].
- **Balance between RTL and HLS approaches**: The high-level abstraction brought by the High-level synthesis (HLS) approaches allows rapid development without the need for a great knowledge about the FPGA architecture. However, the resulting design may not be fully optimized [33]. By comparison, in Register Transfer Level (RTL) approaches, hardware design is done directly and can achieve high efficiency, which in turn requires in-depth knowledge of both the CNN model and the FPGA architecture.
- **Specific design and optimization**: Given the characteristics of a CNN model and the FPGA architecture, a careful design is necessary to optimize throughput and efficiency [34]. Although the reuse of a hardware module to implement multiple layers improves resource efficiency, it can lead to poor performance due to the different computing patterns of each layer [35].

Given the challenges of implementing a CNN in hardware, several types of mechanisms optimized for point cloud processing have been presented in the literature, however, 3D object detection models often resort to less efficient solutions, making their implementation in Edge Devices unfeasible [27]. Considering the FPGAs as a technology with the potential to deploy deep leaning-based models in Edge Devices [36], there is a need for an investigation focused on flexible solutions and their efficient design and implementation in hardware.

1.2 Motivation

Technological advances achieved in recent years have allowed the automation of countless tasks previously performed by human beings. The transformation of a transport vehicle into an autonomous system is already a reality, but it still faces many obstacles given all the complexity involved. The commitment of the research community together with the support of both great car manufacturers and technological companies have facilitated this transformation considering the common goal. Enthusiasts on the theme of autonomous vehicles have also helped to analyze the problem from other points of view, with the emergence of start-ups with completely different approaches to solve autonomous driving.

Despite the diversity found in the different approaches, some components of an autonomous system are essential and integrated into several solutions already developed. One such component is the perception system, which autonomous vehicles heavily rely on to recognize the environment around the vehicle. With the LiDAR sensor increasingly being identified as essential for measuring object distances, deep learning models based on LiDAR data have been developed to detect and classify 3D objects, however, this application has strict requirements. For instance, the inference time should at least be equal to the sampling rate of the LiDAR sensors. As this parameter varies between 10-20 Hz, the maximum acceptable inference time is 100 ms. As will be shown in section 2, the inference time of the majority of the 3D object detection models is superior to this value, which hampers autonomous driving from reaching its full potential.

Despite the success of deep learning models in complex tasks, the combination with the need for a distributed computing in autonomous vehicles implies their deployment on platforms with limited resources. This, together with the high availability of FPGA-based hybrid SoC in the market, brings great opportunity to develop flexible and efficient solutions to implement these models in hardware. Several CNN hardware accelerator architectures and frameworks are available both in the literature and the market, however, usually, some limitations are found, namely: reduced flexibility and efficiency, patented features, and restricted compatibility to certain FPGA boards. Considering the need for research focused on optimized mechanisms for LiDAR data and their efficient design and implementation on resource constraint platforms, this dissertation introduces a configurable hardware architecture to implement CNNs in hardware. While the designed architecture supports the common layers found in CNNs, it also allows different levels of parallelism to improve the performance.

Given the sparse and unstructured nature of the point clouds, sparse convolutions are pointed out as more efficient mechanisms over the traditional ones, to process the data with a high level of sparsity. The

potential of sparse convolutions needs to be analyzed and ideally integrated into real case scenarios to evaluate both the viability and the performance improvements. To the best of our knowledge, there is no work in the state of the art merging the advantages of such configurable CNNs combined with the features of hardware such as an FPGA platform.

1.3 Objectives

The development of this dissertation has the objective of implementing convolution processes in hardware, time and energy efficient and resource-conscious. To this end, it is intended to implement an open and configurable solution to implement CNNs in hardware as well as a solution optimized for point cloud data processing. Summing up, it is intended to answer the following questions:

1. What are the resource costs of building a convolutional module in hardware that is adaptable to the requirements of 3D object detection models?
2. What are the characteristics of the data that influence the performance of sparse convolutions?
3. According to the different possible variations in the data characteristics, what conditions favour the use of sparse convolutions?
4. Are there 3D models in the literature that meet the conditions for the adoption of sparse convolutions?
5. For a real case scenario, what are the practical improvements in adopting a sparse convolution instead of a traditional one?

1.4 Methodology and Methods

The development of a convolutional module to implement CNNs in hardware imposes important requirements in the architecture, given the variation of the characteristics found in CNN layers. To this end, after identifying the operations involved in CNNs, they will be implemented and integrated into a modular hardware architecture. Considering the most used operations in the CNN layers, a hardware architecture will be designed to try to perform these operations as efficiently as possible. Given the enormous flow of data that occurs in the convolutional layers, it is intended to implement a technique to access the input data that does not affect the module's throughput. Furthermore, to increase the module's applicability to situations where there are more resources available, parallelization techniques will be integrated in order to increase the module's performance.

To validate this solution, several tests will be built to analyze the module's performance under different conditions in terms of available resources. Given the extensive applicability of convolutions, RGB images will be used to prove the correct functioning of the solution. Additionally, it is intended to integrate the module with a deep learning-based model for object detection and classification. Through the integration in a real scenario, it will be possible to analyze the feasibility of adopting the module in CNNs, as well as answer research question 1.

Extending the investigation to mechanisms optimized for point cloud processing, it is intended to propose a hardware architecture to execute the voting scheme-based convolution. Besides designing an architecture that supports the use of the voting convolution, it is important to add configurability to the operation, namely to the parameters of stride, padding, and kernel size. Furthermore, an analysis of the data characteristics that affect on the voting convolution performance in hardware will be performed, aiming at implementing possible optimization techniques in data processing to reduce computation cycles.

The voting convolution validation will be performed on small data sets, as well as on images, however, different tests will be constructed to evaluate its performance according to the input data characteristics, answering the research question 2. For each test, the performance of the voting and traditional convolutions will be compared to define which conditions favour the use of each, as indicated in research question 3. To analyze the advantages of using the voting convolution in a real case, a model from the state of the art that satisfies the voting convolution requirements will be selected. This approach will help to answer research questions 4 and 5.

1.5 Publications

This work originated the following articles, submitted for review in the journal: *IEEE Transactions on Circuits and Systems I: Regular Papers*.

- *Customizable FPGA-based Hardware Accelerator For Standard Convolution Processes Empowered with Quantization Applied to LiDAR Data*.
- *Efficient hardware design and implementation of the Voting scheme-based convolution*.

1.6 Dissertation Structure

This dissertation document is divided into a total of six chapters, namely the Introduction, State of the Art, Design, Implementation, Tests and Results, and lastly the Conclusion

The state of the art presents the technologies that supported the development of this dissertation. It starts with a study on Deep Learning-based models for object detection and classification in point cloud data, highlighting the models distinguished in the current state of the art. Given the extensive adoption of convolutions and the sparse characteristic of point clouds, a study is carried out focused on optimized mechanisms for sparse data processing. The chapter ends with the description of a selection of CNN accelerator frameworks, available for FPGA deployment.

In the Design chapter, considerations are taken regarding the desired requirements for the Convolutional Module and the Voting Block and their efficient hardware design is subsequently presented. Following the design specifications, the next chapter describes the hardware for the FPGA implementation of both architectures. In the Tests and Results chapter, the validation of both implementations is carried out, together with a set of performance tests for the case study.

Finally, in the last chapter, some conclusions about the dissertation work are presented with some notes regarding possible future work.

Chapter 2: State of the Art

This chapter begins with a study on LiDAR and cameras sensor's contribution to the perception task in autonomous vehicles over the past few years. In particular, the trend helps to detail the increasing importance and adoption of the data collected by LiDAR sensors for object detection tasks.

The next subsection presents a literature review of 3D object detection models. The characteristics of the models with the major impact on the current state of the art in 3D object detection are highlighted along with the taxonomy that allows distinguishing each one's potentials.

Considering that 3D models usually adopt traditional convolutions for data processing, optimized mechanisms for point cloud data processing are presented next, together with the importance of their adoption in sparse CNNs. Three different types of sparse convolution are described following the advantages and disadvantages of each one.

Finally, CNN accelerator frameworks targeting FPGA platforms are detailed. The study enables an understanding of the tools currently used in the market to migrate object detection models to resource-constrained platforms. The analysis of each framework underlines the particularities that distinguish them, highlighting the limitations and development support.

2.1 Perception system in Autonomous Vehicles

The surrounding environment perception in autonomous vehicles setups mainly relies on two sensors: cameras and 3D scanners [37, 38]. The data collected by both sensors compose the raw information used to detect and classify dynamic objects, such as pedestrians, cyclists, and vehicles. Although there are different camera technologies, RGB cameras are the most adopted in autonomous vehicles according to the current state of the art in perception models [38]. RGB cameras provide a dense representation of the captured scene containing detailed information about object geometry and texture, useful for object detection and classification tasks. While, over the years, object detection models were highly dependent on data collected by 2D sensors, such as cameras, recent models are progressively dependent on the depth information to locate and classify objects in 3D space [25].

LiDAR-based solutions have been considered increasingly viable to meet the perception requirements for autonomous driving and can already be found integrated in several vehicle setups [1, 39, 40, 41].

Compared to other sensing technologies, LiDAR sensors provide both high resolution and precision in 3D measurements of the vehicle's surroundings under several adverse atmospheric conditions [42]. Despite the recognized robustness of LiDAR sensors, several tests have been performed to acknowledge and exploit the vulnerabilities such as: most of the reflection intensity is attenuated or lost in the presence of fog; with strong and non-uniform precipitation false objects can be created; environments with high luminous intensity directed at the sensor create a great difficulty in collecting data from the vehicle's surrounding [43]. Figure 2.1 represents a point cloud example with approximately 12k points, result of a complete scan of a LiDAR. Among several possible attributes associated with each point, the most common ones are the x, y, and z coordinates and the intensity.

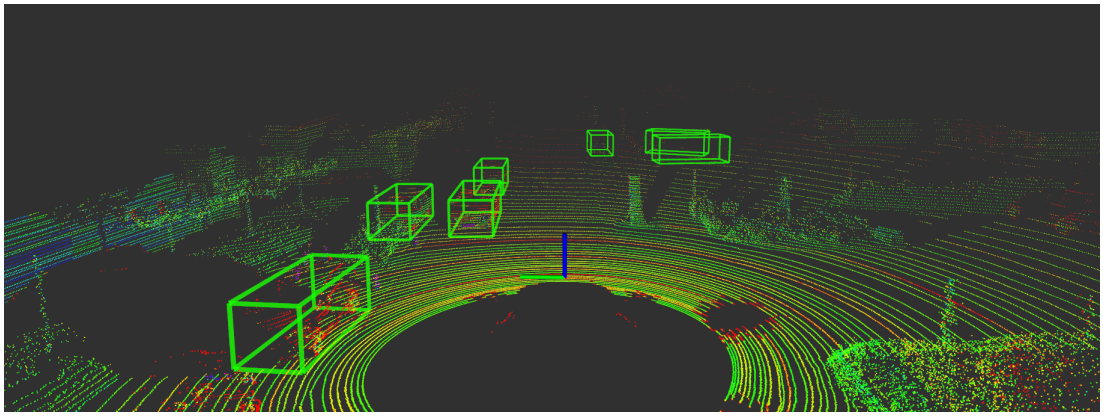


Figure 2.1: Point cloud example from Kitti dataset with pre-defined Bounding Boxes

According to the perception task and the application domain, different key LiDAR performance attributes can be distinguished such as: scan speed, measurement range and accuracy, point density, robustness to unfavorable environmental conditions, and cost. To address such needs, a large number of LiDAR manufacturers have emerged introducing new technologies. Figure 2.2 represents the growth of LiDAR devices patent registrations in the market for autonomous vehicles. The graph illustrates the number of published patents between 1967 and 2018, being identified more than 6480 LiDAR-related patents. The study evidences the companies concern to have a strong position in the LiDAR sensors market. The effect of the growing attention from large companies can be deduced by the strong investigation in the perception area, which translates into a growth of 21% in patent publications between 2007 and 2018 [2].

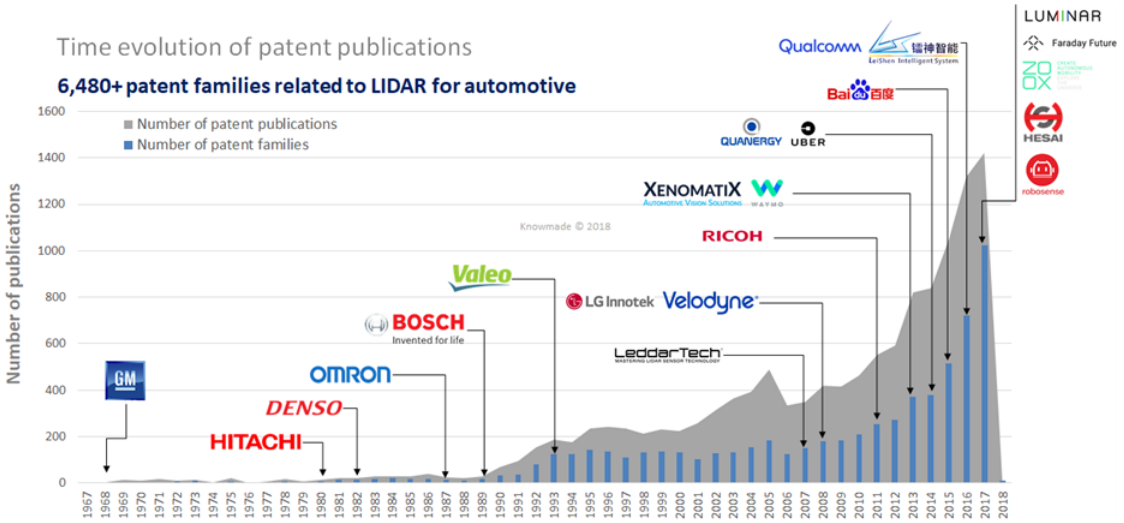


Figure 2.2: LiDAR related patent publications between 1967 and 2018 [2]

2.2 Deep Learning-based 3D object detection models

Object detection and classification tasks take a crucial role as it enables the perception of the surroundings and provide information to the system to perform the best driving decisions. Several approaches were developed based on LiDAR and multi-modal, but LiDAR is one of the main perception sensors to provide real-time object detection and classification. LiDAR sensing technology enabled a new line of research leading to the development of suitable object detector algorithms architectures for self-driving cars, generally referred to as 3D object detectors. Recent research and developments in 3D object detection models have strengthened the state of the art bringing more diversification to the models' architectures together with innovative design choices. As a result, traditional models solo-based on RGB cameras are increasingly integrated or replaced with solutions using LiDAR data [24].

Due to the sparse and unstructured nature of the point clouds as well as the number of generated points, 3D object detection models require high computational costs. Some solutions, such as PointNet [44] and PointNet++ [45] opted by processing the raw point cloud, however, the high computational costs lead to inferences times in the order of a few seconds, which is not acceptable for this type of application. The growing computational need has led researchers to focus on restructuring the point cloud into structured representations, also known as volumetric representations, such as Voxels [3], Pillars [5] or Frustums [7]. After the point cloud restructuring, the feature extracting techniques based on convolutional neural networks are applied. From the set of extracted features, the consecutive stages of the algorithms are responsible for detecting and classifying objects.

3D object detection models can be categorized as single or two-stage detector, according to the number

of stages of the architecture pipeline [26]. A single-stage architecture combines the object location and classification in the same stage, the Multi-task Header. At this stage, feature maps are processed by different networks to generate the object's location and classification through the bounding box coordinates. Examples of single-stage architecture models are PointPillars [5] and SECOND [4]. The algorithms that follow this architecture are distinguished by their reduced execution time and generally present a lower precision when compared with two-stage architecture detectors [24]. The base single-stage architecture is represented in figure 2.3.

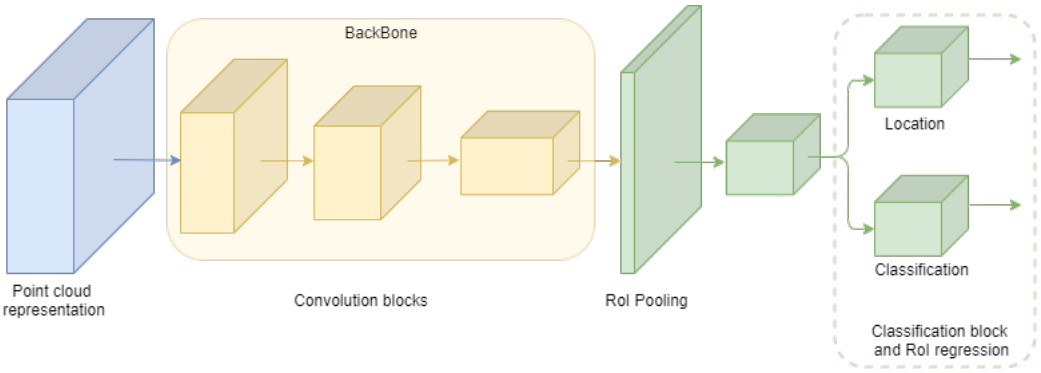


Figure 2.3: Single stage detectors basic architecture

Two-stage detectors models, such as F-PointNet [7] and BirdNet [9], use a Region Proposal Network (RPN) algorithm [46] to generate Region of Interest (ROI) proposals in the first stage. The proposal generated in the first stage is sent to the object classification block and ROI regression, as a second stage. These algorithms achieve greater detection precision than single-stage models as in the classification block a new additional phase of feature extraction is carried out to obtain more detailed information and complement the one extracted in the first stage. However, the additional computation required leads to higher inference times when compared to single-stage detectors [26]. The base architecture of two-stage detectors models is represented in figure 2.4.

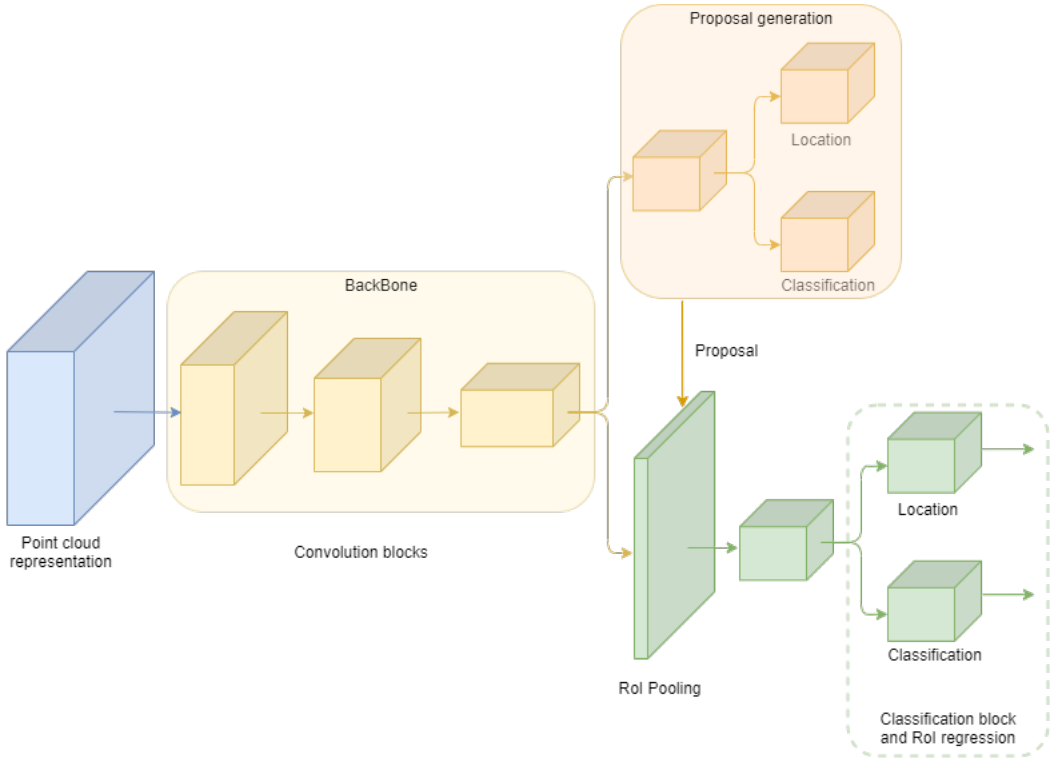


Figure 2.4: Two stage detectors basic architecture

BackBone networks, which consists in CNNs, are used by object detection models to learn and extract the relevant characteristics from the input data and build feature maps. The extracted features are the information used to identify the object's location and class in the captured scene. BackBone networks are commonly constituted by convolutional networks which stand out for their ability to learn the most relevant characteristics for a given task during the training phase [24].

Convolutions are found in different stages of the model's pipeline, both for detection and classification tasks. The model's backbone can vary in architecture, configuration, and between a 2D or 3D CNN, where different types of convolutions can be applied according to the data representation used. Most models choose well-known computer vision backbones as VGG [47] or Resnet [48], nonetheless, recently some CNNs are designed to handle the sparse nature of point clouds. The following subsection analyzes the pipeline of different models found in the literature of 3D object detection.

2.2.1 Single stage models

Single-stage object detectors instead of separating region proposals from classification and bounding box regression pipeline processing, it integrates all those processes as a set of connected layers.

VoxelNet [3] follows a data representation based on voxels and combines a modified version of the RPN architecture with the PointNet [44] algorithm, as illustrated in figure 2.5. PointNet is considered

the pioneer in combining deep learning with LiDAR data and follows an architecture based on Multilayer Perceptron (MLP) to extract features from data. The features extraction in VoxelNet is performed through convolutional layers to build characteristics maps from the point cloud representation as voxels.

The modified RPN architecture allows bounding boxes regression for 3D detection and consequent object's class prediction. The VoxelNet model was the pioneer in introducing voxels where the 3D space is divided into smaller 3D spaces and the processing allows the extraction of features rich in geometrical and location information. The data representation in voxels also enables better results in speed and resource-consuming since irrelevant data is discarded and voxels can be processed simultaneously.

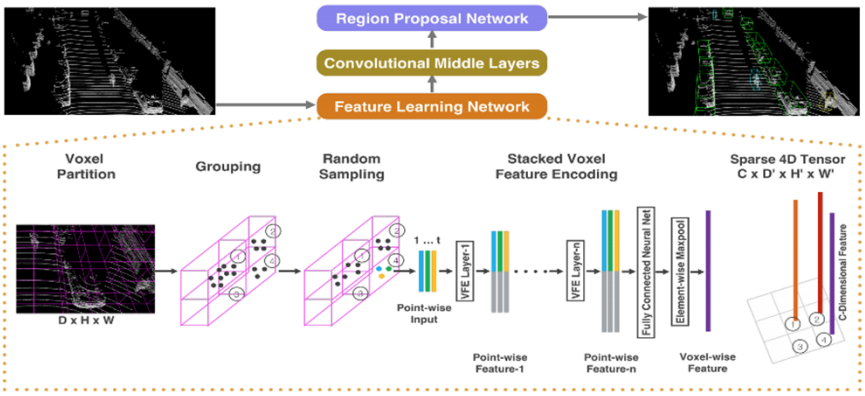


Figure 2.5: VoxelNet architecture [3]

The **SECOND** [4] algorithm also follows a voxel data representation and applies two Voxel Feature Encoding layers followed by a linear layer. Next, a sparse CNN is implemented and the RPN is responsible to generate the detections. The sparse CNN deals with the sparsity nature and spatial dimension of the 3D points through the use of submanifold convolutions, which restricts the processing to the regions of input active sites. Computational cost reduction and better execution times are achieved since irrelevant information is ignored when adopting sparse convolution in the convolutional layers. The algorithm adopts 2D convolutions as it works with a 2D Bird Eye View (BEV) representation of the point cloud after projecting the Z dimension as a simple feature map channel. The comparison between models shows a precision similarity compared to VoxelNet, however, the introduction of a CNN that takes advantage of the point cloud sparse nature allows to drastically reduce the model's inference time, as shown in table 2.1.

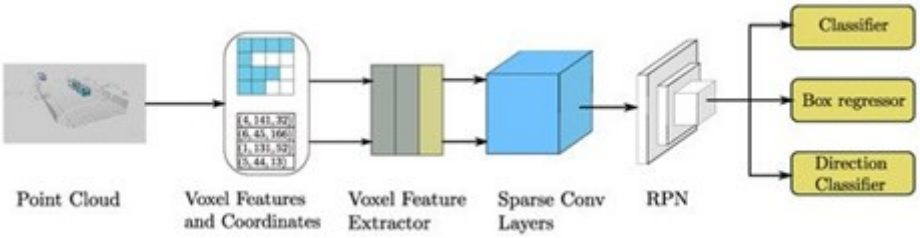


Figure 2.6: SECOND architecture [4]

PointPillars [5] uses a Pillars-based representation of the 3D point cloud, presenting a CNN architecture for extracting high-dimension features and a PointNet-based solution for low-dimension features extraction. In the Pillar Feature Network (PFN) stage only a linear layer is used to extract features from each pillar, but only the most relevant ones are used to create a 2D pseudo-image.

The backbone implemented next to the PFN is a sequence of 2D convolutional layers constituted by three blocks to learn features from the transformed input. The adoption of 2D convolution requires less computational complexity than 3D convolution, resulting in better performances. The Single Shot Detector (SSD) network, composed of 1x1 convolutions, is responsible for generating 2D bounding boxes on the features generated from the backbone layer of the Point Pillars network.

After the bounding box locations regression, the non-Maximum suppression is used to filter out all the noisy predictions. Height and elevation were made additional regression targets in the network to modify the predictions for 3D bounding boxes, as SSD was originally developed for images.

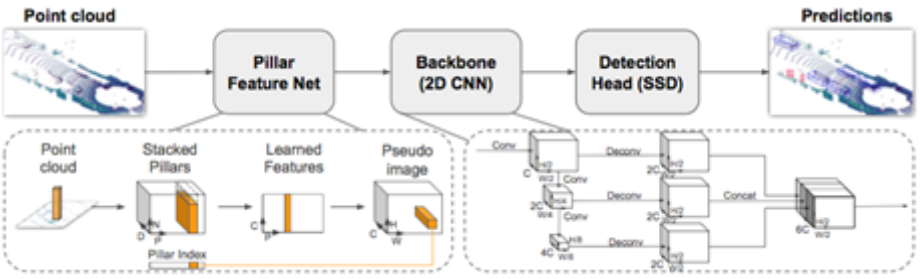


Figure 2.7: PointPillars architecture [5]

The problem identified by the literature about point cloud data compression is related to the potential loss of information, so some studies aim to design mechanisms optimized for sparse data while maintaining the concept of using a sliding window along the three dimensions. In this context, **Vote3D** [49] is presented, to enable the use of sliding windows in 3D data through a strategy to deal with sparse data. In Vote3D, the point cloud starts to be discretized into voxels and each of the grid cells occupied by points is converted into a feature vector. On the other hand, cells that are not occupied by points do not generate any vectors, therefore, no later computation will be dedicated to these cells. For the feature representation,

three shape factors together with the mean and variance of the reflectance values are computed within each occupied cell.

In the detection task, the adoption of mechanisms based on a Voting scheme allows one to explore the sparse nature of the data and take advantage of it through an efficient search for the location and orientation of objects, avoiding unnecessary operations. Voting scheme-based convolution, revealed to be mathematically equivalent to traditional convolutions, which translates into good performances in detection precision, but with efficiency improvements.

Inspired by the Vote3D algorithm, the authors of **Vote3Deep** [50] proposed to explore the mechanism based on the voting scheme to build sparse efficient CNNs, for 3D object detection, without the pre-projection of the point cloud into a smaller representation. The sparsity identified in the 3D point cloud is exploited with a feature-centric voting scheme using the insight that meaningful computation is positioned where the 3D features are non-zero.

An important aspect of this work was the use of a penalty on the filter activations and Rectified Linear Unit (ReLU) blocks which helps to preserve the sparse levels of the intermediate representations in the neural network. In addition, the bias parameter is only added to non-empty output cells since a positive bias would return an output grid with almost all cells occupied with a feature vector, removing the sparsity from the data. The ability to maintain sparse levels allows exploration of sparse mechanisms across all the CNN, reducing the computational cost of the algorithm.

The **MEGVII** [6] algorithm, represented in figure 2.8, although it also follows a data representation based on voxels, implements a feature extractor based on 3D convolutions followed by an RPN. To obtain 3D feature maps with an appropriate format, they are adjusted to apply 2D convolutions transversal to the RPN block and subsequently aggregated to build higher resolution feature maps. While in former RPN-based projects, algorithms adopted RPN to detect and classify as well as for bounding box regression, in [6] RPN is adopted to simply perform regular 2D convolution and deconvolution. The features are aggregated to build feature maps with higher resolution than the feature maps firstly outputted by the Convolution Middle Layers. The tasks of localization, classification and bounding box prediction are carried out by a network called Multi-Group Head.

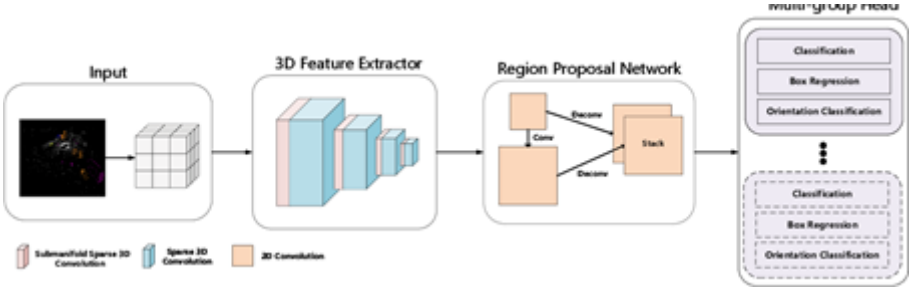


Figure 2.8: MEGVII architecture [6]

2.2.2 Two stage models

Regarding two-stage models, the proposals for the region of interest are generated and submitted to two different neural networks. Those are responsible for verifying the probability level that the region contains an object, classifying the object and determining its location within the region. The F-PointNet [7], MV3D [8] and BirdNet [9] models are examples of works that adopt a two-stage architecture.

F-PointNet [7] starts by segmenting the point cloud into Frustums, using this representation to then extract features for detection. The F-PointNet architecture consists of three modules: Frustum proposal, 3D instance segmentation and 3D amodal bounding box estimation, as illustrated in figure 2.9. As the modality of this algorithm is both point cloud and images, in the first instance, the object detection in the image is carried out using a CNN-based solution. 2D detections performed on the image are projected into the 3D space to extract the 3D bounding frustum corresponding to each 2D bounding box. This mechanism reduces the time and resource cost associated with regions of interest searching in 3D space.

In addition, the F-PointNet segmentation module uses the PointNet architecture to evaluate, within each Frustum, the probability that each point belongs to an object. The last block is an estimation network called Amodal 3D Box Estimation which is implemented to estimate the object amodal oriented bounding box by using a box regression PointNet.

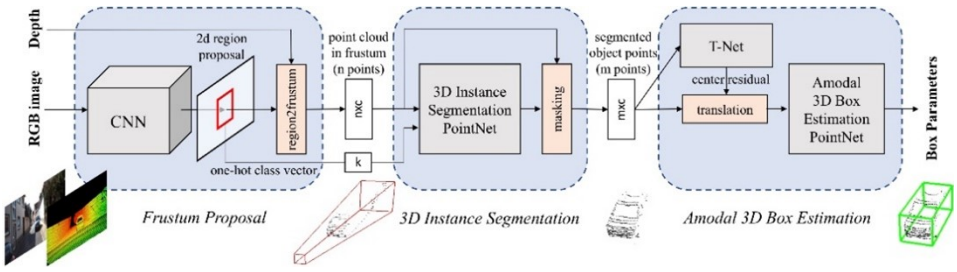


Figure 2.9: F-PointNet architecture [7]

The **MV3D** [8] algorithm, as represented in figure 2.10, has input data from both the LiDAR and RGB sensors, which are combined to extract high-dimensional features and predict the location and orienta-

tion of the bounding boxes. The algorithm uses two representations of the point cloud, BEV, and Front View (FV), where feature maps are extracted and later combined with the features extracted from the images. The MV3D is composed of two networks: the 3D object proposal generation network from the BEV representation of the point cloud; and the multi-view features fusion network.

The main purpose for utilizing multimodal information is to perform region-based feature fusion. Feature maps are designated as region-wise features rather than object-wise as the MV3D first detects the region of interest through the feature map extraction from BEV in detriment of other views due to the advantage of the BEV over a front view/image plan. The multi-view fusion network extracts region-wise features by projecting 3D objects/box proposals (extracted from BEV) to the feature maps from multiple views. The LiDAR point clouds projected to BEV are then used to train a region proposal network for 3D bounding box proposals.

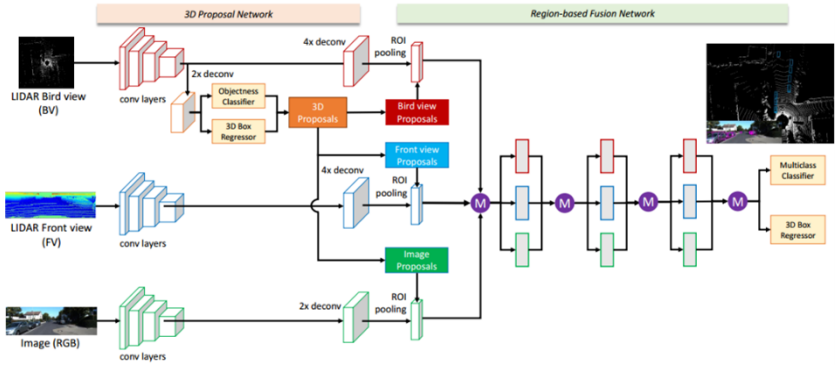


Figure 2.10: MV3D architecture [8]

BirdNet [9], illustrated in 2.11, adopts a Faster R-CNN architecture to perform object detection and orientation. It follows a multi-view 3D object detection network that consists of two parts: a 3D proposal network and a region-based fusion network. The fusion network combines region-wise features from various representations and constructs regions of interest for each one, which are later combined to define the 3D bounding box and object's class prediction. The object's class together with both 2D bounding box coordinates and orientation build up the output of BirdNet.

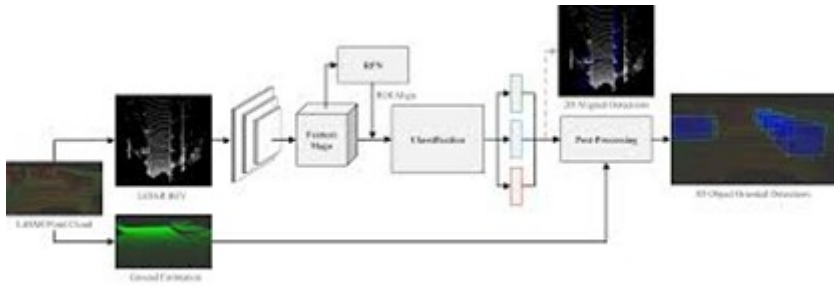


Figure 2.11: BirdNet architecture [9]

2.2.3 Models comparison and taxonomy

The state-of-the-art analysis in 3D object detection models demonstrates the trend over the years in the development of models based on deep learning, progressively abandoning the use of hand-crafted feature extraction methods. Learned features do not rely on fixed encoders and allow algorithms to extract more contextual information from the sensor's data. The increasing ability to interpret complex environments enables a more complete solution to be integrated into a perception system.

The models approached in the state of the art demonstrate the tendency of the two-stage algorithms to handle point clouds and images simultaneously, as an attempt to increase the performance of the detection and classification results. However, despite the accuracy of these algorithms being quite reasonable, they have a longer execution time compared to the single-stage algorithms, mainly due to the high amount of information to process, as shown by the results reported by the MV3D model [8]. In addition, the fact that the use of these models implies an exact synchronization between the scans of both LiDAR and camera [38], requires a system capable of temporally managing all the sensor measurements for a consequent coherent fusion of the data.

The study of single-stage algorithms, as opposed to two-stage algorithms, reveals that they are more computationally efficient, as summarized in table 2.1. It is noted that the algorithms that use 3D representations from the LiDAR data, such as 3D Voxel grids, present a higher computational cost since the depth information of the objects is incorporated in the structure. The superior amount of information makes the processing of a 3D grid very inefficient as it implies a largely redundant and unnecessary computation resulting from the high level of sparsity. The compacted information obtained with 2D projections, on the other hand, results in less computational cost despite introducing loss of information.

The comparison of the two approaches also shows that, although the algorithms that adopt a 2D representation of the data obtain a better performance in terms of execution time, the precision in the results is sacrificed due to the information loss after the initial compression. Given these circumstances, some models, such as SECOND and PointPillars, opt for a 3D representation of the point cloud but use 2D convolutional neural networks to extract features, obtaining better efficiency levels and being able to compete in terms of accuracy with models based on 3D CNNs [24].

Following the characteristics of point clouds, some algorithms explore the sparsity and spatial dimension through the adoption of optimized mechanisms for the point clouds data processing. In this topic, Vote3Deep is presented which, different from SECOND, tries to maintain the sparsity level of the network intermediary representations to reduce the computational cost across the convolutional network.

Architecture	Model	Modality	Data representation	Detection	Accuracy (%)	Speed (fps)
Single stage	Vote3D (2017)	LiDAR	Voxel	3D CNN	39	N/S
Single stage	Vote3Deep (2017)	LiDAR	Voxel	3D CNN	N/S	0.9
Single stage	VoxelNet (2018)	LiDAR	Voxel	RPN	58.25	4.4
Single stage	SECOND (2018)	LiDAR	Voxel	RPN	60.56	26.3
Single stage	PointPillars (2019)	LiDAR	Pillar	SSD	66.19	62
Single stage	MEGVII (2019)	LiDAR	Voxel	Class-balanced multi-head Network	52.8	N/S
Two stages	MV3D (2017)	LiDAR and Image	2D Projection	Region-based Fusion Network	N/S	2.8
Two stages	F-PointNets (2018)	LiDAR and Image	Frustum	Amodal 3D Box Estimation	65.39	5.9
Two stages	BirdNet (2018)	LiDAR	2D Projection	Faster R-CNN	40	9.1

Table 2.1: 3D object detection models comparison

This study seeks to identify models that stand out in the object detection literature, taking into consideration detection's performance and execution time. Models that preserve the high-dimensional representation of the data while implementing an architecture with 2D operations end up registering the best results. Namely, the PointPillars and SECOND models stand out with good levels of precision, being the PointPillars the one that registers a better balance between precision and execution time.

The models previously presented demonstrate that convolutions are used in practically all stages of the pipeline, with the only exception being in the point cloud restructuring stage. Recent models choose to replace the traditional Fully Connected Networks with Fully Convolutional Networks, where the model's output data from the object classification and location tasks is also obtained through convolutions.

2.3 Convolution Neural Networks in the perception task

In the computer vision area, object recognition algorithms depend on the detection and feature extraction to later classify the objects. The past literature focused on improving the points of interest that can be extracted from the data to compose the best possible description. Several solutions have emerged, such as the Scale Invariant Feature Transform (SIFT) [51], known for its robustness against object's rotation and scale variations. However, the robustness provided by SIFT, and other algorithms, implies a great computational cost, leading to the emergence of faster variants, achieved by Speeded Up Robust Features (SURF) [52], serving as viable real-time applications proposals. These mechanisms use a point of interest detector to locate characteristic regions of an image, such as corners, and then describe them using a descriptor capable of distinguishing each point of interest detected.

The paradigm shift came with deep learning [28], which makes it possible to build complex neural networks to solve problems such as object detection and classification. In this type of approach, the solution is based on convolutional neural networks, which behave as extractors of characteristics, as generically represented in figure 2.12. The particularity of this solution is to learn which characteristics best define a set of data in contrast with characteristics previously defined by the author which are extracted manually [53].

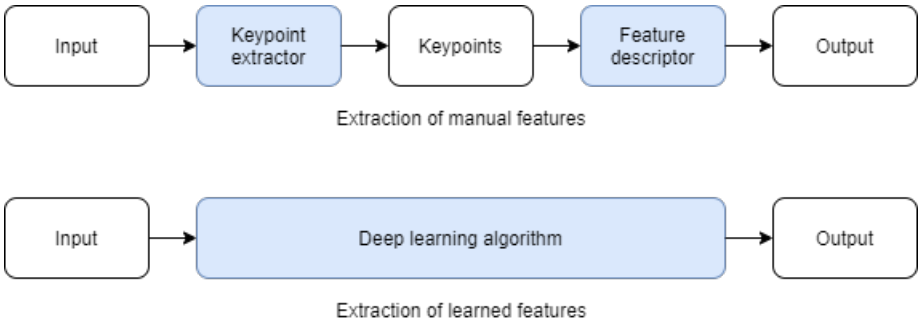


Figure 2.12: Hand crafted features extraction

CNN based solutions for object detection and classification initially had a major impact in the computer vision area, but their effectiveness has extended the scope in recent years to areas such as autonomous driving, namely for LiDAR data processing. Given the popularity of some 2D CNN based models, some were simply reinvented to operate on three-dimensional data. A CNN is similar to the functioning dynamics of the human brain, considering the existence of neurons and connections between them. The constitution of a CNN is mainly composed of convolutions over several layers. Convolutions, applied to the network input, are carried out using filters, composed of a set of values, called weights. The weights values are learned

during the training phase of the convolutional network, and together they assign levels of importance to different types of characteristics found in the data. Figure 2.13 shows the detailed architecture of the already mentioned PointPillars model.

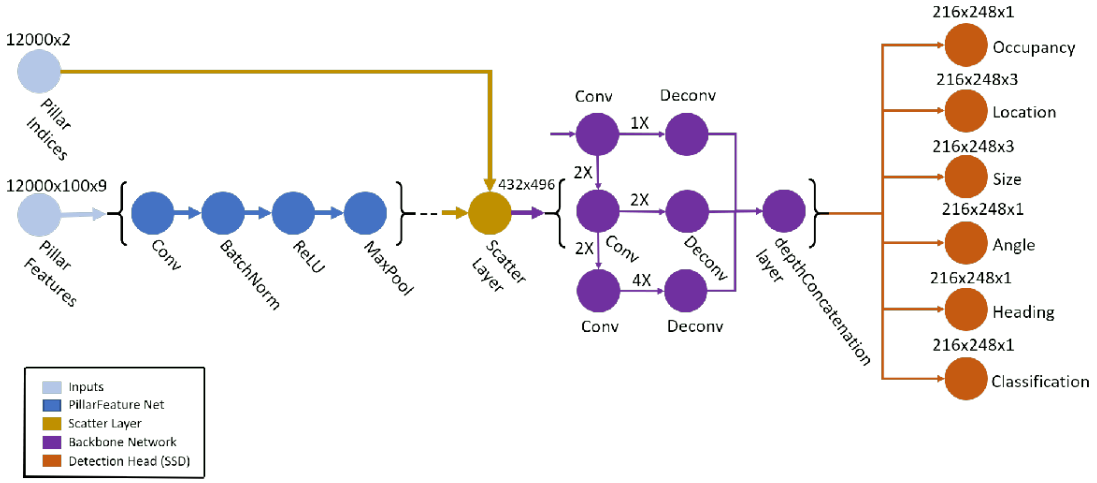


Figure 2.13: PointPillars detailed architecture [10]

The exemplary CNN is composed of several layers where different types of operations can be distinguished. Although both ReLU and Pooling among other operations have an important role on the network processing, the convolution is the main operation and transversal to all the network. As the last stage, the Detection Head, 1x1 convolution are implemented to generate 2D bounding boxes on the features generated from the backbone layer of the Point Pillars network.

CONV: The convolutional layers are the main building block of a CNN, where most of the computation of the network is located. This layer is composed by a set of weight values forming a kernel, which are learned during the training of the network. The number of filters varies from layer to layer, and each of them is applied to the data that comes from the previous layer. A simplistic case is shown in figure 2.14, where the 2x2 kernel is slid along the 3x3 input matrix, resulting in a 2x2 output.

Starting from the example shown in figure 2.14, in a real context of a neural network, the convolutional layers are composed of multiple filters, with several channels, depending on the existing input channels. To synthesize the information, the meaning of the typical parameters of these operations can be summarized as follows.

- Input volume size of W1 x H1 x D1;
- Four hyper-parameters:
  - Number of filters (i.e. number of kernels) **K**;

  - Filter's size **F**;
  - Stride **S**;
  - Padding **P**.

- Output volume size of W2 x H2 x D2, where:
  - W2 = (W1 – F + 2P) / S + 1;
  - H2 = (H1 – F + 2P) / S + 1;
  - D2 = K.

- Each filter has F * F * D1 weights.

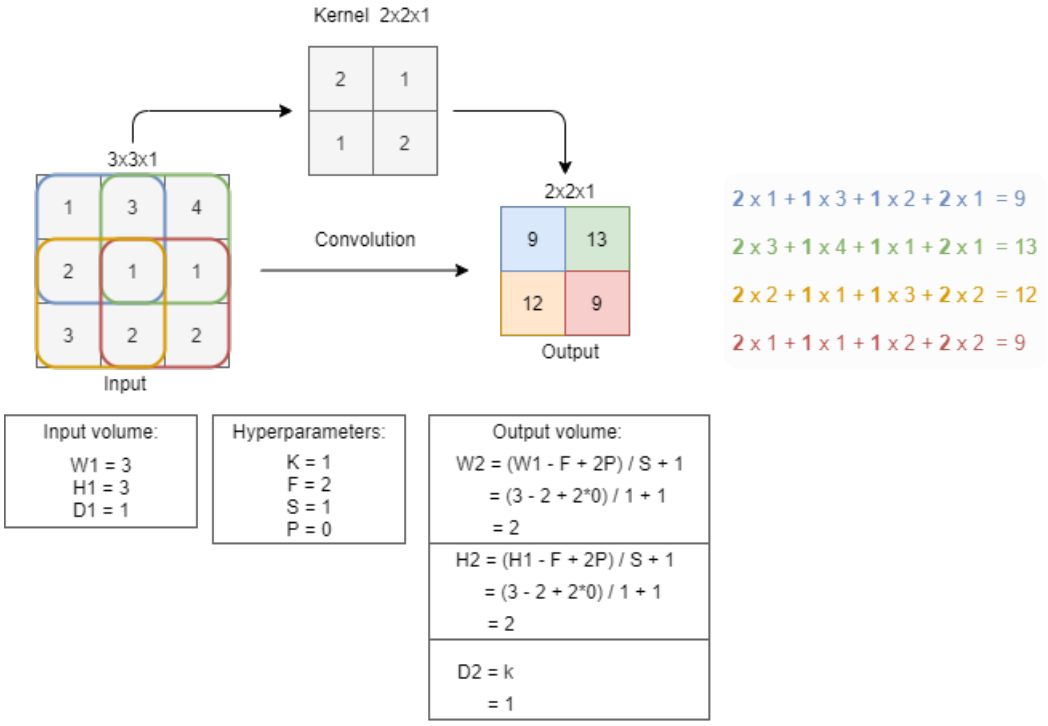


Figure 2.14: Convolution between a 3x3 input and 2x2 kernel

ReLU: ReLU is an activation function used in CNNs and it is defined by the formula max(x, 0), which sets the output to zero when the input is negative. On the other hand, if the input is positive, the output will have that same value, as illustrated in figure 2.15. If a neuron is not relevant, this means that, for certain values of the input, the output would negatively contribute to the output of the neural network. This feature is not desirable as it is preferable to use non-negative activation functions for CNNs, therefore, the ReLU is the most common function among others.

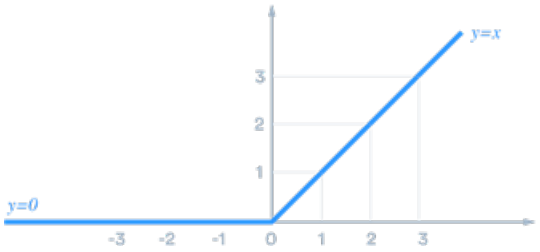


Figure 2.15: ReLU function

POOL: Abbreviation for Pooling, is the layer responsible for the subsampling operation along spatial dimensions. The presence of the Pooling layer along a CNN allows to progressively reduce the size of the data representation, decreasing the number of parameters and consequently computation. The implementation of this layer is done using the MAX operation, normally using a 2x2 filter with a stride of 2 in each operation. In each application of the filter in the data, the highest of the four values is fixed, as shown in figure 2.16.

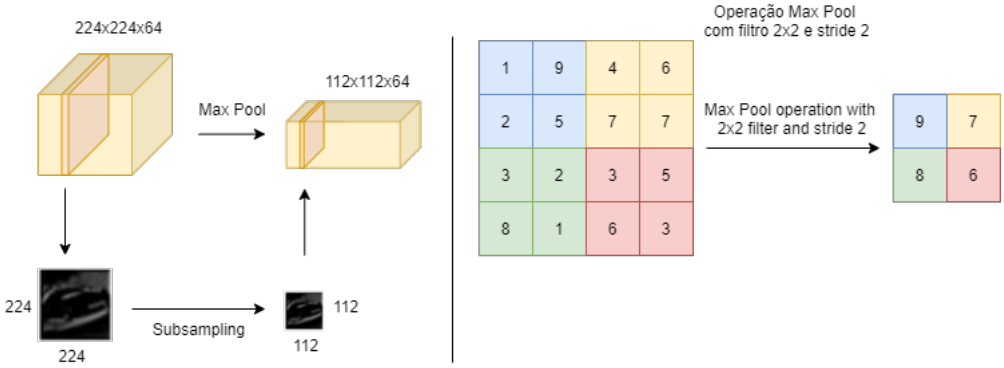


Figure 2.16: Max Pooling operation

FC: In the Fully-connected layer, all neurons have connections with all activations from the previous layer, unlike a convolutional layer, in which neurons are only connected to neurons belonging to a specific region in the previous layer, as shown graphically in figure 2.17. This type of layer can be considered a totally general-purpose connection pattern and makes no assumptions about the features in the data. It's also very expensive in terms of memory (weights) and computation (connections). With this type of layer, the model has the ability to mix signals, since every single neuron has a connection to every single one in the next layer. However, these characteristics make a fully connected layer less efficient and much less specialized when compared with a convolutional layer [54].

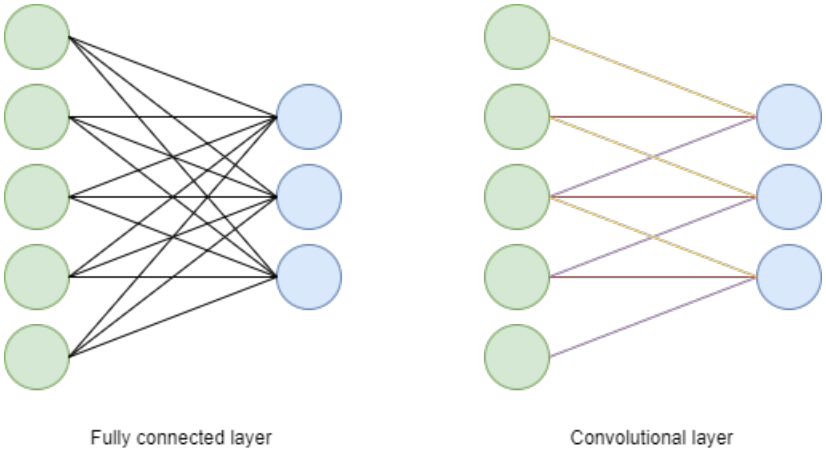


Figure 2.17: Graphical difference between fully connected and convolutional layers

Similar to a convolutional layer, computation in an FC layer is scalar products between matrices, making it possible to convert it into a convolutional layer. In reality, a convolutional layer is represented by an FC layer, with a greater matrix filled mostly with null values, except in certain positions regarding local connections. This approach is useful in practice, being much more efficient than iterating through all locations of the input, thus bringing performance improvements without sacrificing the accuracy of the ratings.

2.4 Point cloud optimized convolutions

The key to achieving efficiency is to denote that there is a fundamental difference in the structure of a 3D point cloud compared to a 2D image. A 3D point cloud is sparse as much of the space is not occupied with information, and so, 3D models work on a large portion of irrelevant data collected by 3D sensors. As an example, in the PointPillars model [5], the pseudo-image from the PFN stage has in each channel a total of 262144 values and only a maximum of 12k values are non-null, which translates into a sparsity level above 95%. Common optimization techniques, such as weight quantization, promote the increase of irrelevant information in the feature maps and further reduce the efficiency of traditional mechanisms, as will be analyzed later 5.12. Faced with the fact that dense implementations of CNNs are inefficient when applied to sparse data, in the last few years several mechanisms (with an emphasis on convolutions) have been presented to operate efficiently on sparse data.

Keeping in mind that the developments in this topic seek to bring improvements in terms of computational cost and resource consumption, it is important to highlight the dynamics adopted by each mechanism and the circumstances in which the integration of each one is appropriate.

2.4.1 Sparse convolution

Following the mechanisms' objective for processing 3D data mentioned, several studies present strategies that direct the processing only to the input data with relevant information [49, 50]. Data without information considered relevant would simply be ignored, saving resources and time previously allocated for the respective processing. Based on this, sparse convolution seeks to take advantage of the sparse nature of the point cloud.

The sparse convolution ignores the input data without relevant information, so the convolution is only carried out if the input data contains any relevant information. With this mechanism, there is an increase of active sites in the output compared to the input, leading to a consecutive expansion of active sites in the feature maps. Figure 2.18 graphically illustrates the dilatation that is verified in the data within a neural network when the sparse convolution is adopted. Through expansion, the active sites gain more and more volume, leading to a consecutive increase in the number of operations in the following layers for processing them.

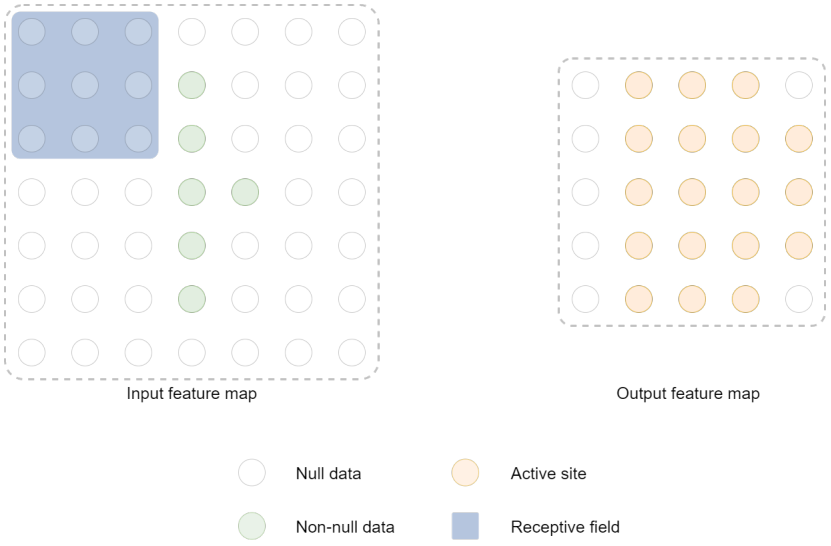


Figure 2.18: Sparse convolution

2.4.2 Submanifold convolution

To find a solution to the dilation problem, the Submanifold Sparse Convolution (VSC) [55] operation restricts the output active sites only to the central input active sites. The difference concerning Sparse Convolution is related to the relevant data since padding is applied over the input data and an output site will be active only if the central input site is also active, as represented in figure 2.19. The advantage of this type of sparse convolution is the computational cost reduction, allowing the construction of deeper

neural networks without high resources consumption due to the successive dilation along with the layers. With this technique, the submanifold convolution helps to maintain the sparsity levels encountered in the input.

To reconstruct popular convolutional neural networks, such as ResNet [48], efficiently with this type of convolutions, the state of the layers is stored in a hash table together with a feature matrix. The cost associated with the construction of each hash table is proportional to the amount of input active sites. Although there is a tendency for this amount to increase in deep convolutional neural networks, preserving the data sparsity level regularizes the creation costs of these tables. Apart from the dilation detail, certain models adopt both types of convolutions since in certain cases it is necessary to preserve the size of feature maps using padding.

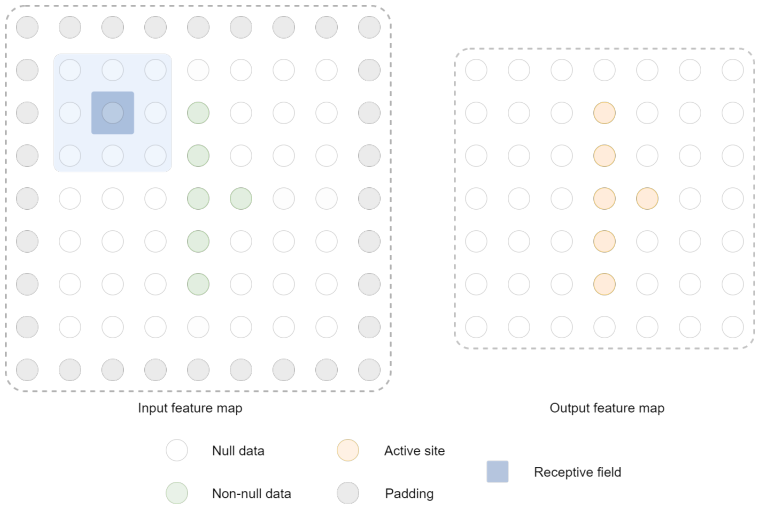


Figure 2.19: Submanifold sparse convolution

2.4.3 Voting scheme-based convolution

Voting scheme-based convolution [50] is also presented as a good alternative since convolutions over 3D input feature maps translate into a large number of unnecessary operations [49]. Therefore, voting scheme-based convolution deals with spatial sparsity to reduce the number of operations compared to traditional convolutions. In the first instance, the three-dimensional space is discretized and represented in the form of a grid. For each cell of the grid occupied with points, a vector of points features is extracted, ignoring and consequently discarding the cells with an insignificant number of points.

The Voting filter is obtained from the inversion of the convolutional filter in each of the two dimensions, as shown in figure 2.20. However, the operation is mathematically proven to be equivalent to a traditional dense convolution in a sparse space since the filter only applies to occupied cells. This type of convolution together with the Submanifold convolution, also preserves the sparsity levels avoiding the dilatation prob-

lem, nonetheless, it also needs information regarding the position of non-null values to make all processing efficient.

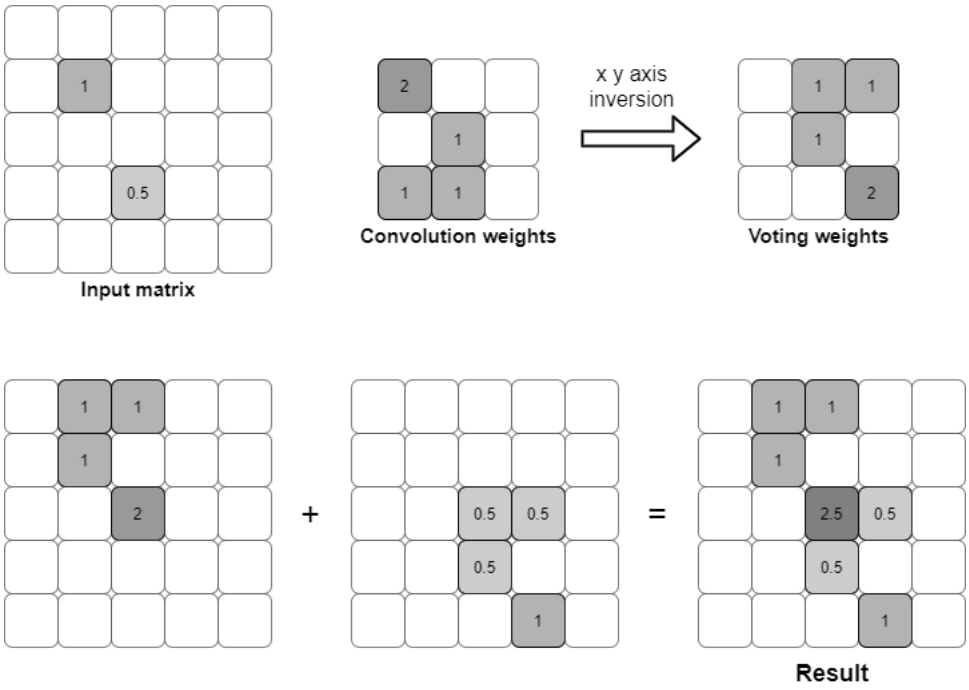


Figure 2.20: Voting scheme based convolution

2.5 Hardware acceleration frameworks

Due to the huge flow of data, certain deep learning models are implemented on servers where time is spent between client and server communication, being impractical in real-time applications. Given these circumstances, in recent years the paradigm has been changing with the migration of computing to platforms known as Edge Devices [34]. Edge Devices have fewer resources when compared to solutions that resort the processing to servers, however, the fact that they are closer to the scene where the action takes place, avoids latency in communication, making vehicles more independent and less susceptible to failures. This trend has opened up space for hardware accelerators as a solution capable of satisfying the requirements of a real-time application [36].

A hardware accelerator is a set of specialized hardware that performs several tasks with great performance and better efficiency, when compared to generic platforms, such as CPUs. Some examples of common accelerators are GPUs, ASICs, and FPGAs. As shown in figure 2.21, CPUs are the most flexible and ASICs are the most efficient in latency and power consumption. FPGAs fall into a category characterized by a good balance between efficiency and flexibility, given the configurability of the hardware. ASICs, although they are also known for efficiency, do not have flexibility. The recurring cost of an ASIC is quite

low, but its non-recurring cost is very high, with no margin for error during the development.

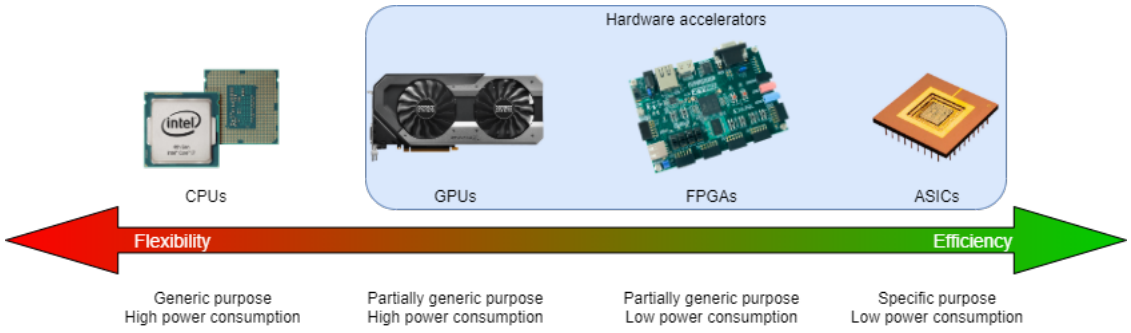


Figure 2.21: Common computing platforms

A lot of work has come up with FPGA-based CNN implementations [56] [57], taking advantage of the hardware flexibility and performance to meet the real-time applications requirements. In this topic, the hardware accelerator frameworks, receive a high-level description of a trained neural network model and generate a synthesizable accelerator to be executed in an FPGA. The high-level description of a model involves not only the deep learning operations/operators and data flow but also the parameter values, normally stored with a specific format according to the framework used to train the network. For this study, some hardware accelerator frameworks were selected and categorized using the metrics: interface as the deep learning framework used, supported network layers, target platforms with support, design specification level and market availability. Table 2.2 summarizes all the information that allows highlighting the frameworks that will be described.

HADDOC2 [11] is a framework that uses Direct Hardware Mapping (DHM) to implement a CNN in hardware. The high-level description of the CNN model is built using the Caffe [58] format and then the VHSIC Hardware Description Language (VHDL) code is generated to instantiate the hardware. CNN can be specified as data flow processing networks, where the nodes represent the actors and the edges the communication channels. Through DHM, it is possible to map in hardware the graphic that represents the processing units, as the nodes, and the connection between them.

Multiplications, transversal to the CNN layers, are implemented with simple logic cells instead of dedicated Digital Signal Processing (DSP) blocks. HADDOC2 assumes that most weights have null value after quantization, being subsequently removed and reducing the logic to be implemented in LookUp Tables (LUTs) [29]. Each CNN layer is implemented in hardware according to a pipeline architecture to increase the data processing rate. The FCLs can be seen as convolutional layers, but without sharing parameters, which translates into higher consumption of resources like multipliers. For these reasons HADDOC2 does not support FCL, besides, with a direct hardware mapping approach, the implementation would imply an increased consumption of resources, making the use of the framework unfeasible.

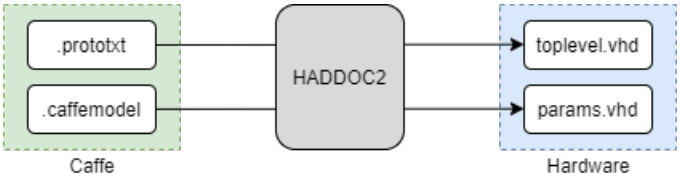


Figure 2.22: HADDOC2 interface [11]

DNNWEAVER [12] is an accelerator based on a single computational unit, using templates written in Verilog. The high-level language interface of the neural network is found in the Caffe format, allowing it to generate instructions and optimize the computational unit for the FPGA platform where the framework is implemented. DNNWEAVER is composed of four software components called: Translator, Design Planner, Design Customizer, and Integrator.

1. Translator: Converts the neural network high-level specification into an Instruction Set Architecture (ISA). These instructions are not executed by the accelerator but used to map the neural network into a data flow graph, with each instruction corresponding to a graph node.

2. Design Planner: Receives instructions generated by the translator. Uses an optimization algorithm to optimize the hardware templates according to the target FPGA platform. This component is responsible for managing operations and transferring data between them.

3. Design Customizer: The input of this component is the resources allocation and the execution schedule, both used to customize the accelerator core as intended.

4. Integrator: Responsible for the interface between the accelerator and the memory. The interface must be defined by the user if the target platform is not supported by the framework.

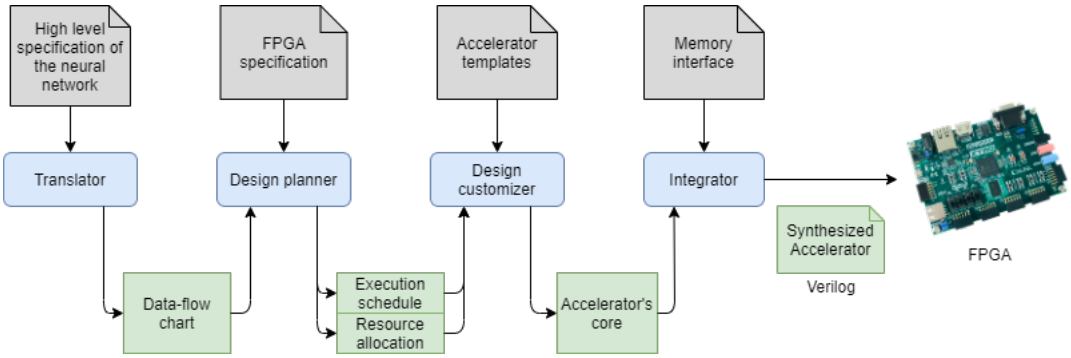


Figure 2.23: DNNWAVER working flow interface [12]

Hls4ml [13] is an open-source package that generates implementations of machine learning algorithms using HLS. More specifically, hls4ml uses Xilinx HLS, consequently targeting support for Xilinx

platforms. This framework uses fixed-point arithmetic and performs weights and bias quantization automatically. The network description and training are performed using Keras / Tensorflow, creating a YAML configuration file, containing the location of the network architecture and the parameter storage directory.

After the HLS code for a specific neuronal network has been generated, the configuration for each individual layer is performed next. During the configuration of each layer, the connection architecture between the multipliers is defined, as well as the precision of the calculations involved.

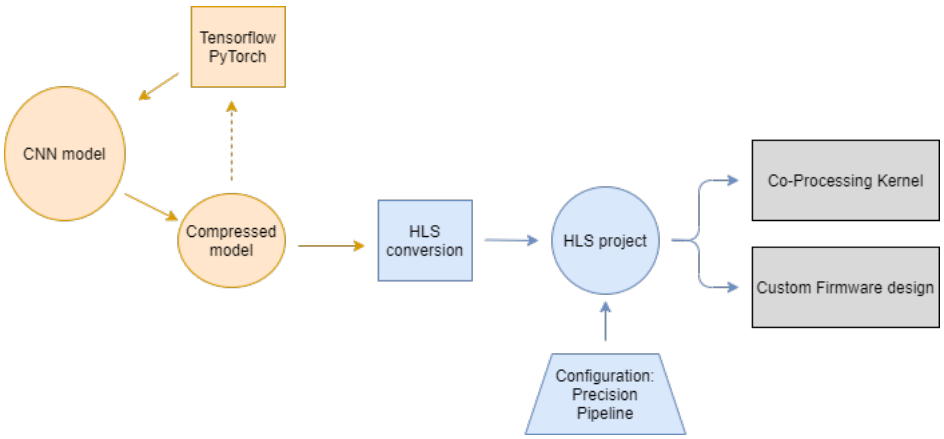


Figure 2.24: HLS4ML working flow interface [13]

CHaiDNN [14] is an open-source accelerator framework developed by Xilinx, for Xilinx System On a Chips (SoC's) themselves. Uses an Integrated Development Environment (IDE) designed to easily perform implementations on processors. In this framework, the user can specify the portion of the code that must be synthesized for hardware or executed in one of the CPU cores. By default, convolutional layers are run on the FPGA and the Fully Connected Layers on the CPU.

This framework supports 6- and 8-bits fixed-point precision for the network parameters, at the discretion of the user. Also, CHaiDNN supports different data sizes across multiple layers, favouring more enhanced accelerator optimization for a specific neural network.

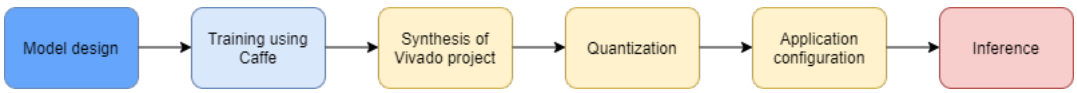


Figure 2.25: CHaiDNN working flow interface [14]

VITIS AI [15] is a state-of-the-art framework for accelerating neural networks, developed by Xilinx for its FPGA SoC and Multiprocessor System On a Chip (MPSoC) platforms. This framework is distinguished by presenting two types of architecture aiming at both Cloud and Edge Devices computing. For Edge applications, the hardware architecture of the DPU is the same as the one implemented in the DNNDK framework, thus, verifying compatibility between VITIS AI and DNNDK applications.

Similar to DNNDK, VITIS AI has an optimizer and quantizer to reduce the number of model parameters and quantize the floating-point data to 8-bits. The code generation is carried out in the next phase using a compiler to build an executable DPU. Finally, the VITIS AI application is created, where the neural network model will be executed, according to the DPU instructions generated in the previous phase.

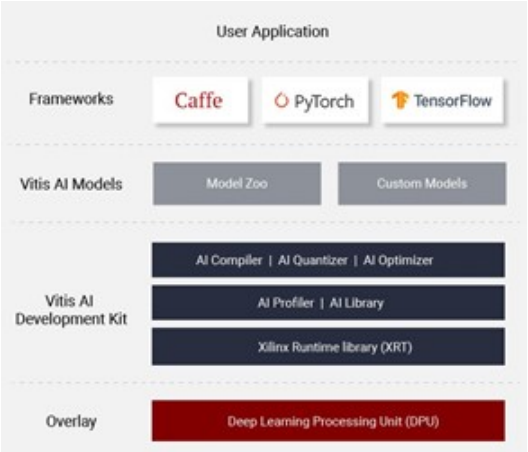


Figure 2.26: Vitis AI architecture [15]

Core Deep Learning (CDL) [16], is a scalable and flexible Convolutional Neural Network solution for FPGAs. The proposed solution it's an FPGA IP core design tool that scales with the target hardware enabling real-time and low power application of deep learning networks. CDL accepts the neural network in TensorFlow or Caffe and takes it through a compression and design space exploration phase to produce the optimal system verilog accelerator core. Using the automated compression flow, deep neural networks are quantized down to 8-bit to save power and memory as an attempt to achive the cheapest best performance possible. After the user specifies the microchip FPGA platform, the CDL's design space exploration algorithm searches through millions of accelerator configurations to find the most suitable design based on the performance and resource specifications.

Besides the scalability offered, CDL can be easily integrated into an FPGA design. The entire network is accelerated in hardware and no host or CPU-based application is required to assist the network execution. The wide-ranging layers support applications using off-the-shelf neural networks such as ResNet for classification and YOLO or SSD for object detection and pose estimation [16].

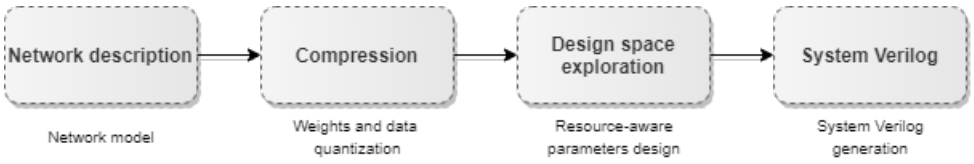


Figure 2.27: Core Deep Learning functionalities [16]

Currently, DNNDK and VITIS AI are the two frameworks most supported by Xilinx, with a DPU-based acceleration together with extensive integration with tools and libraries optimized to implement neural networks models in reprogrammable hardware. In addition to the DNNDK and VITIS AI frameworks supporting the main layers that constitute a CNN, they also have direct compatibility with several popular CNNs such as VGG, GoogleNet, ResNet, YOLO, with ready-to-use solutions. However, although the generic DPU-based acceleration makes the implementation of CNN models on hardware easier and faster, the DPU is not fully customized for all types of networks and may compromise the hardware architecture performance. Both frameworks are not entirely free and open-source as user licenses are required to fully utilize the framework capabilities. Namely in VITIS AI, the integration of the optimizer in the framework requires a separate license, which may be an obstacle given that the optimizer has a fundamental role in the CNN model migration to hardware.

ChaiDNN, also developed by Xilinx, provides support for a wide range of layers and is an open-source framework, however, it lost support from developers as the updates ended in 2018. The HADDOC2 which is open-source as well, generates an accelerator in RTL and the hardware description language used is VHDL, which facilitates the understanding of the framework's operation and how to use it. However, the layers supported in HADDOC2 are limited, not being a viable solution in the hardware implementation of more complex models. CDL is the product of a scalable framework that offers the opportunity to stipulate the desired performance and platform specifications allowing computationally expensive CNNs to be moved to hardware. CDL supports several licensing models from single-use applications to unlimited use subscription for a limited term, and even full technology transfer with full access to source code training.

Framework	**HADDOC2** [11]	**DNN-WEAVER** [12]	**Hls4ml** [13]	**CHaiDNN** [14]	**VITIS AI** [15]	**CDL** [16]
Interface	Caffe	Caffe	Tensorflow and PyTorch	Caffe	Caffe, Tensor-flow and PyTorch	Caffe, Tensor-flow
Supported layers	Conv Pooling Act	Conv Pooling FC Act Norm	Conv Pooling FC Act Norm	Conv Pooling FC Act Norm	Conv Pooling FC Act Norm	Conv Pooling FC Act Norm Concat
Compatibility	Xilinx and Intel	Xilinx and Intel	Xilinx	Xilinx	Xilinx	Xilinx
Design level	RTL	RTL	HLS	HLS	HLS	HLS
Availability	Open Source	Patented	Open Source	Open Source	Patented	Open Source

Table 2.2: Neural Network accelerator frameworks

2.6 Conclusions

The state-of-the-art analysis enabled the perception of the current technologies development state related to the dissertation theme and also serves as a basis for some design and implementation decisions. This study demonstrates that there are no academic studies that aim to present fully flexible solutions to implement CNNs on Edge Devices, such as FPGAs. From the deep learning-based model analysis, it was possible to identify that the convolution is the most common operation, and it can be implemented at any stage of the respective model. As an example, in PointPillars 2.13, despite the different stages of processing, the convolution operation is used across all of them to process the data.

As 3D object detection models deal with different representations of the point cloud, it was verified that 3D models work on a large portion of irrelevant data collected by 3D sensors [49, 50]. The study of sparse convolutions allowed us to realize their potential on sparse data processing. Although dense convolutions continue to be widely adopted for perception-type tasks as proved throughout the deep learning-based model analysis [26], this type of convolutions appears as a good option for the integration with a 3D data processing solution.

Regarding the study of CNN hardware accelerator frameworks, it was possible to understand the advanced development state of frameworks such as Vitis AI. Extensive integration with deep learning tools and a large set of functionalities makes the presented frameworks a complete solution for deploying CNNs in hardware. Along with the advantages, some disadvantages can also be recognized as the form of their configuration-wise limitations and also restrict access to specific functionalities. Although a framework is very useful for the quick hardware implementation of neural networks, it is convenient to develop a completely customizable and open-source convolutional module adaptable to any convolution layer. The hardware implementation of a convolutional module, which is configurable according to the typical parameters of convolution layers, allows building a useful tool to implement any convolutional layer in hardware. In addition, through module replication, there is the possibility of custom building a complete CNN, without implementation or configuration restrictions.

Chapter 3: System Design

This chapter addresses the choices made to design the system and the reasons that led to these decisions are together described. Considering the research goal 1.3, described in the Introduction section, where it is intended to explore the convolution paradigms that leverage the sparse nature of input data and compare the performance with traditional convolutions, the design phase is divided in two different sections. First, the design specification of a convolution model is discussed in section 3.2, in which traditional convolutions are adopted and integrated in an efficient architecture to implement a CNN layer. Then, the second section 3.3, refers to the design of the Voting block, where the hardware architecture to support the Voting scheme-based convolution is analyzed.

3.1 Architecture requirements

As mentioned in chapter Introduction, the main objective of this thesis is to implement efficient 2d convolutions in hardware, optimized for point cloud and adjustable to the 3D model requirements for object detection and classification. To achieve these objectives, some requirements must be considered. The specification of the requirements is important to clarify the intended system functionalities, as well as the restrictions imposed on it. 3D object detection models, as studied in chapter State of the Art, are characterized by using convolutional neural networks as the tool for the perception task. Convolutional neural networks are a set of layers where convolution is the main operation, c.f. subsection 2.3.

Considering the analyzed requirements for the system, the architecture design must enable the implementation of a configurable module. As the module will be instantiated in the Programmable Logic part of the FPGA platform, the data to be processed by the module must be transferred from the external to the internal memory so the module can access the data faster and start performing convolutions. The access to the weights and feature maps is performed through the reading addresses managed by the controller which is also responsible to allocate the Processing Elements used to perform the operations. The number of processing elements instantiated will define the parallelism level which should be managed by the controller according to the suitable resources-consumption.

To design a module that performs 2D convolutions for a hardware implementation, it is relevant to summarize the main functionalities of the CNN convolutional layers typically found in 3D Object detec-

tion models as well as the set of requirements that will be used as guideline to build a suitable system design optimized for a wide range of edge devices and models. Both stride and padding are the most common parameters associated to the convolution, and they are introduced to change the characteristics of the operation, as also described in sub-section 2.3. Besides the variants of the base operation, other convolutional layer parameters can be associated with the input and output data organization. Different convolutional layers have different numbers and sizes of filters, and the input and output, namely their size, is also affected by these parameters.

The analysis and design of the module architecture that performs 2D convolutions in hardware must be aware of the possible parameter values combinations to offer compatibility with the most recent and novel 3D Object detection models found in the literature. With these functional requirements, it is desirable to build an architecture that serves as a base for the configurable module implementation and, consequently, adaptable to any CNN layer. Once the customization level of the module is specified, it is also important to analyze the module's architecture requirements. The hardware deployment of the module is naturally subject to some restrictions, which must be carefully considered during the architecture analysis and design. As already mentioned, the target platform imposes certain restrictions to the design and implementation phases, due to resource limitations. Namely, memory limitation requires efficient hardware design to achieve a good final balance between computational power and resource consumption.

In addition to the convolution as the base operation of a CNN layer, the ReLU and Max Pooling operations are also widely adopted. As the base module architecture, both three operations build up a configurable module able to replace any convolution layer in hardware. From the figure 3.1, assuming that each PE output is ordered, the data passes directly through the ReLU block before performing the Max Pooling operation, when desired. This strategy allows the values to be rectified as they leave the convolution and, as soon as there are enough values, apply the Max Pooling operation. Finally, the module's output data is written back into memory to be accessed by the next module.

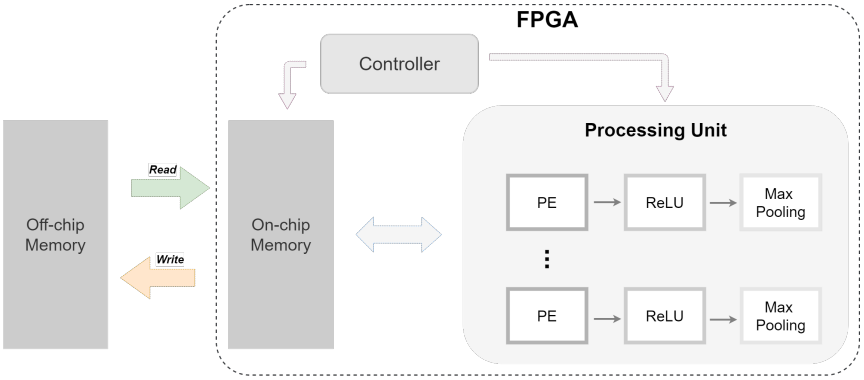


Figure 3.1: Simplified overview of the proposed architecture for a hardware accelerator optimized for convolution operations

3.2 Energy-efficient convolution architecture

CNNs implemented in 3D models process a large amount of data, for instance in the PointPillars model presented in 2.2.1, the backbone stage may receive a total of 262144 values for each of the 64 channels, depending on the model version. The hardware implementation of a CNN layer raises certain questions that must be studied to ensure that the target platform can provide the necessary resources for a specific implementation. Hardware architectures that enable the access or storage of a large volume of data simultaneously have high resources consumption to support data transfer between the memory and the processing units. Thus, it is important to analyze an architecture with an efficient mechanism both in the access and processing of data.

Based on the convolution's properties and the data access costs for processing, an approach focused on efficient data management was considered. This approach is based on [59] and has the main objective of designing a convolutional module architecture that maximizes energy efficiency by reducing redundant access to on-chip memory. The general architecture consists of multiple PEs that work in parallel. Each PE contains a multiplier, an adder, and internal memory to manage data reuse.

In [59], different flow schemes of data transfer are evaluated for both loading input values (so-called load schemes) and storing output values (so-called output store scheme), as well as several connection structures between PEs. The combinations between the data access scheme and the PEs connection structure are a critical factor to energy consumption and processing time since it determines the number of memory accesses and the flexibility to support different parallelism levels. The possible combinations were then analyzed determine the most energy efficient approach. Three types of data load scheme for PEs are identified: broadcast, forwarding, and stay. Regarding the output store scheme, three types are also distinguished: aggregation, migration, and sedimentation.

According to [59], the architecture that presented the best result in execution time and efficiency is characterized by loading the input data in broadcast and the weights in stay scheme, as represented in figure 3.2. The broadcast scheme reads different data every clock cycle and feeds multiple PEs. The stay scheme loads the weights and keeps them in a PE for the entire convolution operation, reducing the number of memory accesses by reusing the loaded data.

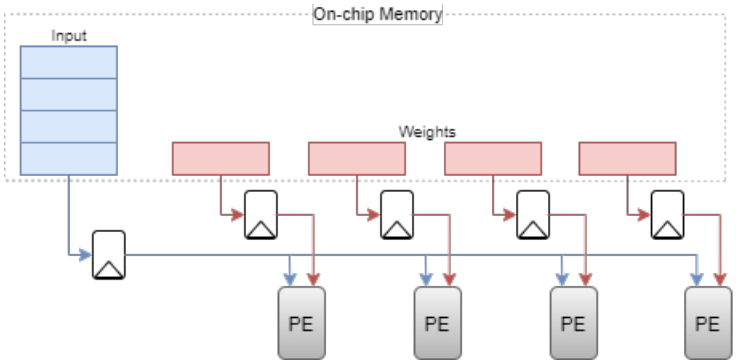


Figure 3.2: Data memory access scheme, namely broadcast for feature map data and stay for fetching weight values

Considering the schemes description and insights from [59], the broadcast and stay schemes were selected for loading the feature map and weight values, respectively. Together with the load schemes, three different output store schemes were evaluated for the accumulation performed between connected PEs. According to the authors research, from the set of possible load and output schemes combinations, the model Broadcast Stay Migration (BSM) was considered the most energy-efficient. The migration scheme gradually accumulates the partial sums over several clock cycles, where each PE passes the aggregated partial sum to a neighbouring PE. As presented in figure 3.4, the long arrows that connect the colored circles extend over two clock cycles, so one shift register is required to hold the partial sum between PEs of different filter lines. In this particular example, a 2x2 filter and 3x3 input are represented, resulting in a shift register with a depth equal to one.

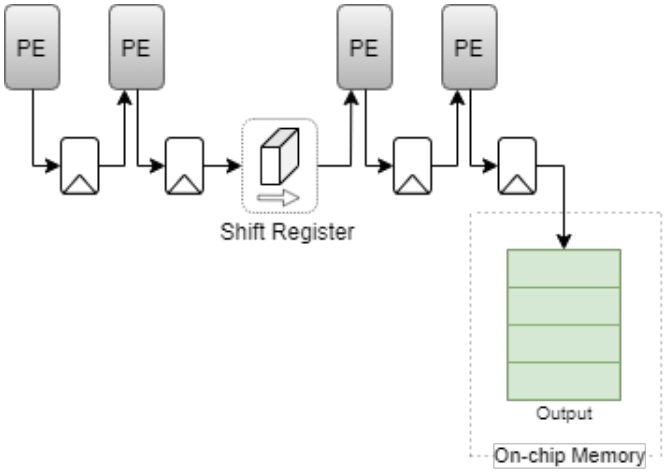


Figure 3.3: Output store scheme

According to the aforementioned load and store types, a convolution dataflow can be derived by projecting the rescheduled dependence graph. For the 2D convolution, in the example with a kernel of size 2 and a feature map with size 3, the proposed convolution dataflow corresponding to the BSM accelerator is represented in figure 3.4. The accelerator reuses the input features map in the parallel PEs to avoid multiple memory accesses. Since the weight parameters stay at fixed positions, one weight register is assigned to each PE to hold the corresponding weight. As shown in figure 3.3, the products computed in independent PEs are migrated to neighbouring PEs resulting in the convolution output values.

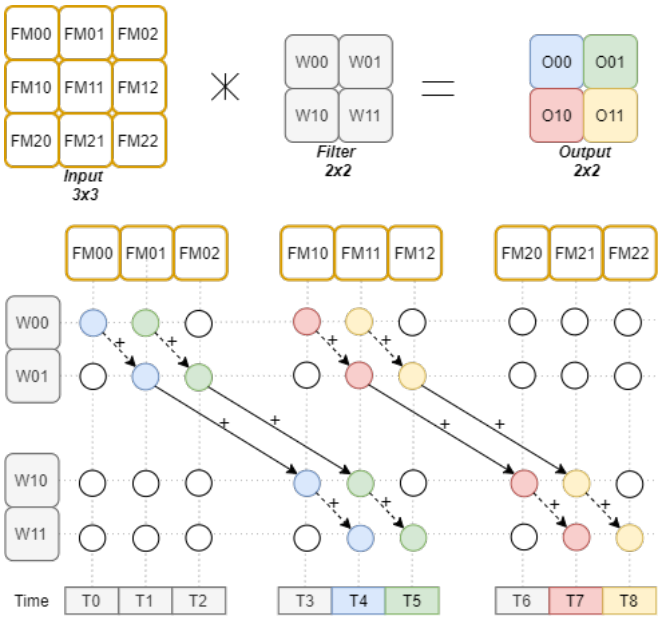


Figure 3.4: BSM dataflow

Considering the processing unit is composed of a cascade of multiple connected PEs, the BSM can be easily scalable to other filter sizes [59]. Besides, it is important to highlight that only one input data

is fetched from memory in each clock, being a great advantage not only due to the complexity of the hardware but mainly due to the energy consumption. Energy consumption was evaluated using models with different input-filter sizes ratio, revealing the superiority of BSM in all cases, proving to be promising in developing an energy-efficient convolutional module.

3.2.1 Processing Element

The PE from the module's architecture depicted in figure 3.1, is responsible for applying the filter over the input data, as it is read from memory. The convolution process is built upon multiplication and addition. The number of operations per convolution vary according to the input and filter size. This mechanism is also known for the Multiply-Accumulate (MAC) operation, which consists of the product between two numbers and addition to an accumulator value.

There are two alternatives when it comes to the use of resources in FPGA to implement hardware multiplications, namely Lookup Tables (LUTs) and DSP slices. The number of DSP blocks and LUTs available has increased significantly on modern FPGA architectures, requiring an assessment of which resource is most appropriate to be allocated for a given implementation. Although it is possible to direct some logic to DSPs or LUTs, the resources themselves are fixed - the FPGA contains a restricted number of LUTs and DSPs. Implementations of multipliers using LUTs are slow and consume significant amounts of resources. DSP slices in FPGAs can be used to substitute LUTs to reduce area and increase the performance of the design.

The embedded DSP blocks in modern FPGAs are highly capable to support a variety of different datapath configurations and have evolved to support a range of applications requiring significant amounts of fast arithmetic. In addition to all the computational capabilities, DSP blocks support runtime reconfigurability, which allows a single DSP block to be used as a different computational block in every clock cycle. Pipeline registers are also embedded in DSP blocks to enhance throughput. Dedicated connections are available for cascading multiple DSP blocks without using the FPGA fabric. This results in better performance and saves resources for other uses.

The DSP slice in the UltraScale architecture depicted in figure 3.5 is defined using the DSP48E2 primitive. The DSP48E2 slice consists of a 27-bit pre-adder, 27 x 18 multiplier, and a flexible 48-bit ALU that serves as a post-adder/subtractor, accumulator, or logic unit. Multiplication is done in two stages:

- The first stage is a multiplication that produces a 45-bit two's complement result as two partial products. These partial products are sign-extended to 48 bits in the X and Y multiplexers and

fed into a four-input adder for final summation. Therefore, when the multiplier is used, the adder becomes a three-input adder;

- The second stage is processed through an adder/subtractor that accepts four 48-bit two's complement operands and produces a 48-bit two's complement result.

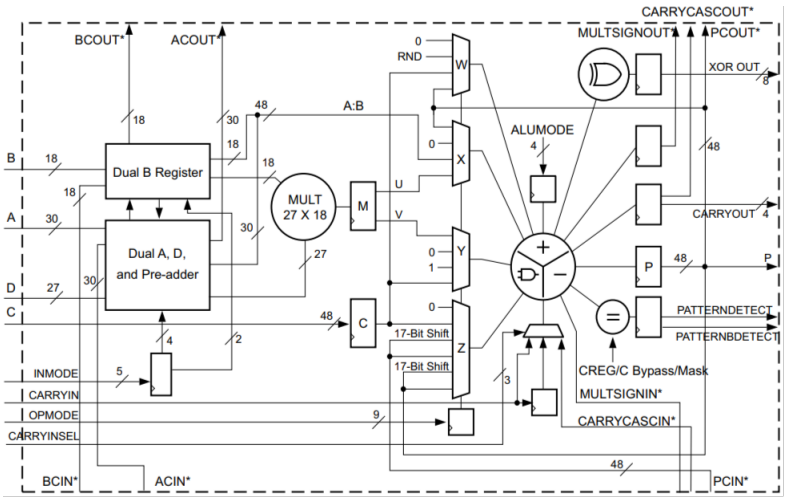


Figure 3.5: DSP48E2 detailed architecture [17]

According to the architecture design described in the previous section, it is necessary to configure the PE to satisfy the desired data flow, so the convolution is performed as intended. After defining the processing unit to be used, it is important to adjust the operating mode through the configuration registers provided, with the main emphasis on INMODE (Input Mode), OPMODE (Operating Mode) and ALUMODE. The INMODE controls the pre-adder functionality and A, B, and D register bus multiplexers that precedes the multiplier. The OPMODE control input contains fields for W, X, Y, and Z multiplexer selects providing a way to dynamically change DSP48E2 functionality from clock cycle to clock cycle. The 4-bit ALUMODE controls the behavior of the second stage add / sub / logic unit.

The DSP48E2 element is capable of operating with different equations considering the available input data ports. Since the intended operation is convolution, it is necessary to configure the DSP to multiply two numbers (the feature map and weight values). Furthermore, since the PEs are connected in a cascade form, the output of one PE connects to the next PE's input. Therefore, in addition to the two inputs already mentioned, the DSPs will have to provide another one that will be added to the multiplication result, so that the consecutive accumulation and consequent construction of the output feature map are carried out.

With that being said, one valid configuration would be considering port A of DSP for the feature map data, port B for the weight value, and port C for the accumulation coming from the previous PE. Thus, with P being the label associated with the output port, the equation implemented in each DSP is:

- **P = A * B + C**

The width of each port is: P – 48 bits; A – 30 bits; B – 18 bits; C – 48 bits. Given this configuration for each DSP, it is possible to build the architecture designed in the previous section. Depending on the size of the filter, a certain amount of DSP blocks will be allocated, proportional to the number of weights in the filter, since each DSP is associated with a single weight. Figure 3.6 shows the final architecture with the integration of DSPs for a 2x2 filter.

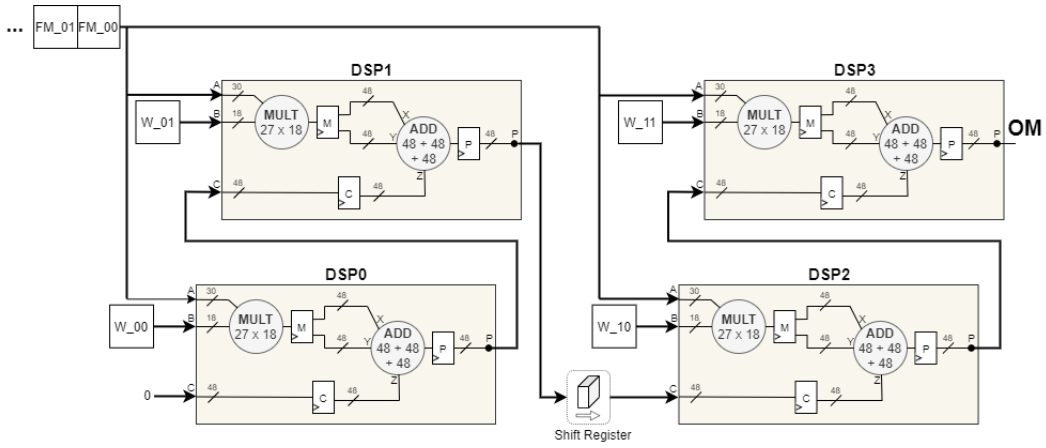


Figure 3.6: DSP48E2 usability in the BSM scheme for a 2x2 filter

3.2.2 Max Pooling

The Max Pooling operation presents four values as input, which are then compared to identify the highest value between them. These values that are compared belong to a specific region to which a max filter is individually applied. According to the architecture described, the convolution between the filter and the feature map is performed and, at each clock cycle, the result of the operation can be read. Keeping in mind that convolution output is ordered by lines, and the max filter is applied to values from different lines the conclusion is that Max Pooling cannot be applied directly. It is necessary to instantiate several modules to carry out the operation orderly or spend resources to store values already read until it is time to apply the max filter to them.

Figure 3.7 describes an example in which the convolution output has a 4x4 size and it is intended to apply Max Pooling. Since the values that come out of the convolution are sent to the ReLU block, the reception order in the Max Pooling module is also line by line. Assuming that the max filter is only applied to a data matrix with even dimensions, then the number of Max Pooling blocks required is equal to half the dimension of the matrix, as shown in figure 3.7. This strategy allows reusing the resources allocated for the instantiation of the Max Pooling blocks to be applied to other consecutive regions of the feature map.

Another important aspect of this approach is that there is no need to save the values that come out from the ReLU block in order to finally apply the max filter. Therefore, as the linear rectification of the values resulting from the convolution is made, the Max Pooling blocks are fed with the corresponding values. As soon as the last value is sent, the result of the operation can be read and saved in the memory, in the next clock cycle.

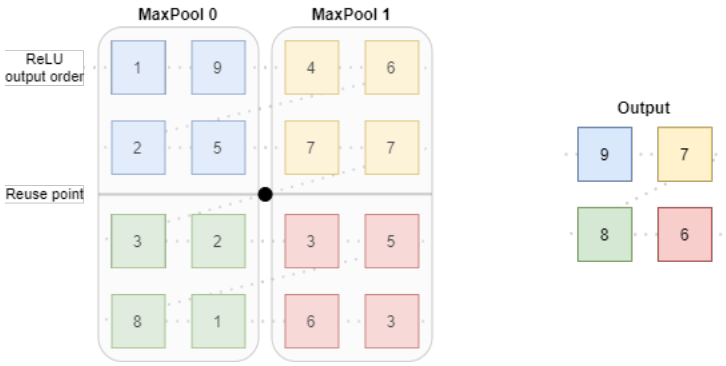


Figure 3.7: Max Pooling mechanism order

3.2.3 BSM architecture parallelism

Energy-efficiency is the focus of the BSM architecture, consequently, it is possible to identify that execution time is sacrificed. From the data flow diagram 3.4, it can be deduced that the time required to perform the complete convolution increases proportionally with the increase of input data size. Since the BSM architecture is able to eliminate redundant data accesses to memory, now it is important to analyze the necessary adaptations to decrease the execution time. The most common way to do this is to parallelize operations throughout convolution.

The BSM Architecture is characterized by a cascade of connected PEs, with accumulations being carried out consecutively over time. This type of mechanism creates a dependency between the data since the output of one PE is used in the next one to perform multiplication and accumulation simultaneously. This evidence leads to the conclusion that the solution will not be changing the PEs connection structure but to take advantage of the architecture modularity to include more PEs. The integration of another set of cascaded PEs allows parallelization to be introduced as the same filter is applied to several locations of the feature map at the same time. Another type of parallelization happens when multiple filters are applied to a given feature map. This type of parallelization is also quite recurrent since CNN layers normally include multiple filters as well as multiple data input channels to convolve with.

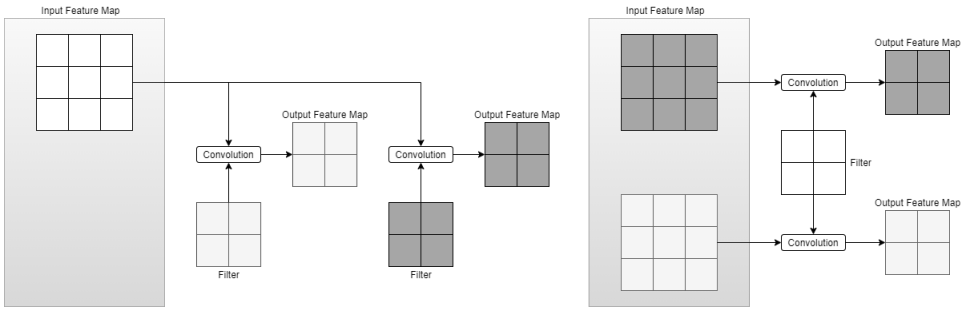


Figure 3.8: Parallelization strategies

The various mechanisms for parallelizing operations must be carefully analyzed to produce a configurable architecture capable of integrating them. It is intended to design a module adaptable to the characteristics of a given CNN layer. The objective is also to use the available resources of the target board to implement the respective layer in hardware and achieve the best execution time results possible. However, the wide variation in the CNN layer's input and output data, namely: input feature map size; input feature map channels; and the number of filters, requires a balanced analysis between the two parallelization variants described above.

The integration of parallelism in the described BSM architecture implies the use of several PEs to apply a filter in several regions simultaneously. Due to the dependence on the input data to produce the output, the input feature map must be carefully distributed by the instantiated PEs and thus eliminate any flaws in the data dependency. As an example, for the reference of the maximum possible parallelism to be applied, figure 3.9 illustrates a case with a 6x6 IFM and a 3x3 filter. For this specific example, since the filter size is 3 and the stride is 1, then each PE processes only 3 lines. Therefore, a total of 4 PEs can be allocated to obtain the maximum parallelism and consequently the shortest possible processing time, in which each PE is responsible for producing one line of the 4x4 OFM. While with a single PE the clock cycles required to complete the convolution would be around 36 (each value fetch at a time), using 4 PEs the cycles are reduced to 18 (each PE consumes only 3 lines).

Figure 3.9: BSM architecture parallelism

Applying maximum parallelism in the convolution operation between a given filter and a IFM also requires the availability of certain resources, namely DSPs, to build each of the PEs. Since the number of DSPs is equal to the number of weights in a filter, for a filter size of three, each of the processing units requires 9 DSPs connected in cascade to be able to apply the filter completely. However, as the size of the matrix increases, the need for processing elements grows proportionally together with the number of DSPs. In situations where there are not enough resources to obtain the maximum parallelism, it is necessary to make careful management regarding the level of parallelism and the consumption of resources that result from it.

Based on the available resources of the target platform, the number of processing units to be allocated will be calculated and considering the filter size, the IFM lines will be distributed to each PE. For convolutions with stride > 1, some adjustments are necessary regarding the distribution of lines by PEs, since some lines are not processed in certain situations. The number of common lines processed by consecutive PEs is equivalent to **Kernel size – Stride**. Figure 3.10 illustrates a case where the distribution of the IFM lines by PEs is carried out to obtain the maximum possible parallelism for a convolution example with stride=3. Although the size of the filter and the IFM is the same as shown in figure 3.9, for a situation where the stride is three, there is no need to allocate 4 PEs, since only two are enough to perform the strided convolution and still obtain the highest index of parallelism. In strided operations with different values, the amount of data to be processed by each processing unit must be reevaluated following the same logic described, so no unnecessary consumption of resources occurs.

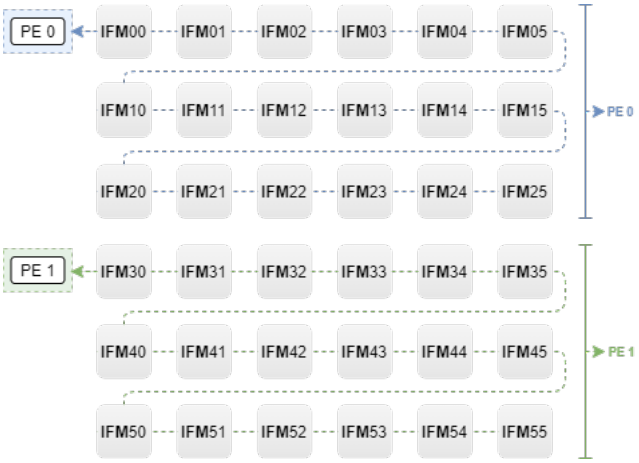


Figure 3.10: Convolution parallelism with stride 3

3.2.4 Memory management

The BSM architecture design flow simplifies data access by fetching from memory one data value each time. With the integration of parallelism in the architecture, data access increases, being necessary to evaluate the control and organization of input and output data. It is important to study the best solution to store the data that will be processed and the data that have already been processed in order to build a solution with low cost of both resources and time. The growing access to memory data is indeed a limiting factor since it may require substantially higher consumption of both logic and memory itself.

The main objective related to this topic is to have a solution that allows fast access to the desired data without compromising its execution time. For this, Block Rams were considered since they are suitable for storing large amounts of data while still allowing quick data access. Another advantage that comes from the use of block rams is the reduction of logical elements such as LUTs, which would not be verified with the use of registers to manage all the data. The use of Block Rams allows instantiating Ram with different types of data transfer ports, which also contributes to the simultaneous data access strategy inherent to the parallelism mechanism described in the previous section. Although the use of Block Ram is beneficial, one of the limitations is the number of ports that each block can provide. In this case, with the use of Block Rams available for Xilinx FPGAs, the maximum number of output ports is two.

With the limitation of simultaneous data access for processing, there is a need to use several Block Rams. With this mechanism the data will be distributed so it will be possible to send input data to all processing units simultaneously. The content and size of each Block Ram that must be allocated to feed the processing units depends on the IFM lines assigned to each PE. The smaller the number of PEs to process a given IFM, the lower the number of Block Rams and consequently the greater the size of each

of them. Figure 3.11 represents the organization of the input memory and the association to each of the three processing units through an example of a filter with size three, stride two and FM of size seven. Since each Block Ram only has capacity for two output ports, four are needed to send data to each of the PEs simultaneously.

Figure 3.11: IFM memory distribution

It is important to note that, for this example, the last Block Ram is smaller than the other three. This is justified by the fact that it is important to maintain the homogeneity concerning the data distribution for all the processing units, in order to decrease the hardware complexity. One of the main aspects is that each PE starts reading the input data from the zero address of the associated Block Ram, that is, the read address of all Block Rams is exactly the same and therefore shared among them. After each of the PEs has processed the first two lines, they must fetch the next data from the consecutive Block Ram, thus ensuring that each memory block is only being accessed by one processing unit. Figure 3.12 illustrates the timing diagram that represents the memory access management for data processing associated with the example shown in figure 3.11. From the diagram 3.12, it is possible to deduce that the strategy used to distribute the data evenly across the memories makes the synchronization simpler to be implemented across the processing units. Considering that the fetch of each input value requires one clock cycle, the time representation in the diagram is translated to IFM width, which represents the fetch of an entire line of values.

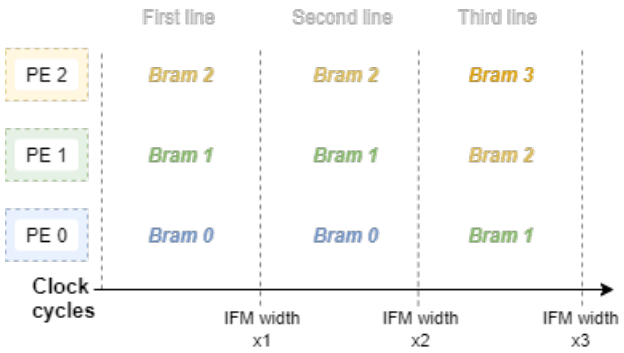


Figure 3.12: Memory access timing diagram

3.2.5 Resources aware adaptable architecture

The layers of a CNN usually involve a large volume of operations via a large amount of data that needs to be processed. The design of an architecture capable of satisfying the requirements of a given layer cannot be carried out without considering the resources available and necessary resources for its implementation. Besides, other questions such as the level of parallelism or energy consumption appear as important aspects to build the best possible solution according to the existing resources. Since the module is configurable based on the characteristics of a particular convolutional layer, it must be not only adaptable to the parameters related to it, but also to the selected board. Given that a large amount of data requires a lot of processing, two of the most critical resources present on the target platform are the DSPs that constitute each PE and the Ram in which the data will be stored. As the module's architecture is build upon the available resources, it must adapt itself to the platform's limitations, for example, by reducing the processing throughput.

The number of DSPs available is related to the number of processing units that can be instantiated within a convolutional module, which consequently translates into the level of parallelism that is possible to achieve. Regarding the memory, it is used to store both the weights and IFM values that are being processed at a given time as well as the result of processing it. For each filter that is applied over an IFM, a certain amount of memory is required to send the output, meaning the amount of available memory will determine how many filters can be applied simultaneously to the input data. An architecture that adapts to the circumstances imposed by the target platform limitations must first be aware of them and later make certain decisions regarding processing based on this information. For this, the available resources for the implementation of a convolutional layer should be specified in the form of the number of DSPs and the amount of memory. From the specification of these parameters, the convolutional module will adjust its base architecture to optimize the operations. Figure 3.13 depicts the resultant architecture for a

scenario where the user specifies the values 54 for the number of DSPs and 4 for the memory available. In addition, the convolutional layer is specified as having four filter of size 3x3.

Although other parameters need to be specified in the module's configuration, the three mentioned above have a direct impact on the parallelism level and the number of filters applied simultaneously. Assuming a filter of size 3 and 54 DSPs available and since each PE requires 9 DSPs, there are sufficient resources to instantiate a total of 6 PEs. Regarding the memory to store the data, of the total 4 Mbits available 2 Mbits are needed to store an IFM together with two filters and another 2 Mbits for two OFM. The input feature map data distribution follows the same logic shown in figure 3.11. Although there are a total of four filters to be applied to the IFM, only two can be applied simultaneously since there is no more additional memory to store another OFM. With a total of six PEs and two filters to be applied at the same time, each convolution block can allocate three PEs that will later be reused in the following iterations with the other two filters.

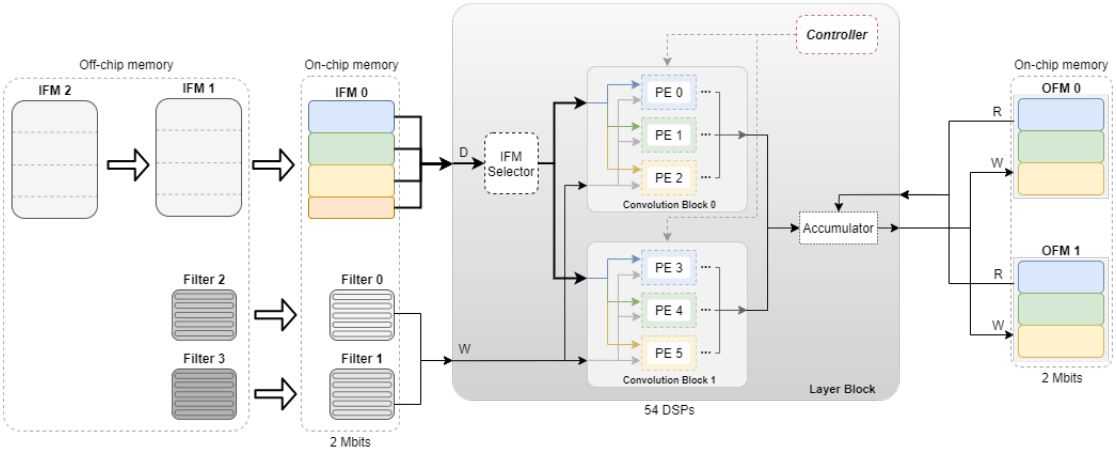


Figure 3.13: Convolutional block memory management

According to the architecture illustrated in figure 3.13, the IFMs are processed one at a time, therefore, some iterations will be carried out to complete all the required processing. The available memory for the convolution results allows only two OFMs to be stored, meaning, initially the first two filters must be applied in all IFMs to completely build the first two OFMs. After iterating through all three IFMs, the process should be repeated with the remaining two filters, reusing the existing PEs in each Convolution Block as well as the output memories. At the end of each iteration, the data relating to the two OFMs must be sent to external memory so the memories can be reused to store new data without the need to allocate more memory. The same technique applies to the memory for the IFM, that is, as soon as the IFM is processed, the memory is reused to store the next IFM and thus save more memory. Figure 3.14 represents the order flow according to the iterations associated with the previous example shown in figure 3.13. Within each

iteration, there are sub-iterations where new IFMs are processed, and so, between sub-iterations, the input memory must be fulfilled with the next IFM. Between the two iterations both OFM must be read from the output memory as it will be overwritten with the new values of the next iteration and also, the first IFM must be sent back to the input memory together with the two following filters.

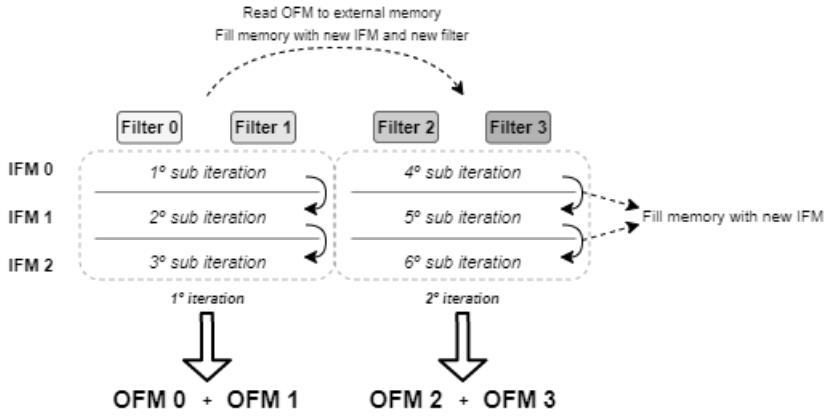


Figure 3.14: Flow of processing iterations

3.3 Voting scheme-based convolution architecture

Traditional convolutions present themselves as a good solution for dense data processing, but data with sparse characteristics make these same convolutions inefficient and inappropriate. On the other hand, the voting-scheme based convolution, introduced, and discussed in sub-section 2.4.3, is a type of sparse convolution useful for processing sparse data. One of the best-known sparse data is the LiDAR data, i.e. point cloud, where its characteristics favor the use of appropriate sparse mechanisms for processing the data efficiently, as discussed in chapter State of the Art. After describing the design of the convolutional module using traditional convolutions, the hardware design of the Voting block follows as a more efficient alternative for processing sparse data with a large spatial dimension.

Although both types of convolutions are mathematically equivalent, the Voting scheme-based convolution presents certain requirements and characteristics that must be reflected in the block architecture. Similar to the convolutional module presented in the previous section, the Voting block has a controller but only one processing unit. The controller is responsible for coordinating the reading of input data and writing of output data in memory, so all the data flow that goes through the Processing Unit is regulated by it. The Processing Unit is the operational unit allocated to carry out all operations involved in the Voting convolution.

Since the Voting block is responsible for performing convolutions, it receives data related to the Input Feature Map and Weights stored in memory. Along with this data, the positions of the values to be processed must also be accessible for reading as it is one of the Voting block's requirements. This information allows the module to operate correctly and efficiently. The reference to the values to process is one of the particularities of the Voting mechanism, crucial to inform the block which input values are relevant to be read from memory and processed later. Without these references, the Voting block would never be appropriate for processing any type of data as it would require an intensive previous search of which values should be processed or discarded. This process of searching relevant values over a large amount of data would remove all the effectiveness of Voting, making it more efficient to adopt traditional convolutions in these situations. Moreover, the Voting mechanism is also inappropriate to point clouds with low level of sparsity as will be further analyzed in chapter Tests and results.

Figure 3.15 illustrates the base architecture of the Voting block, highlighting the communication with memory outside the Voting block. Within the block two functional units are distinguished, the Control Unit and the Processing Unit, which are responsible to manage all the data flow and the operations to be made. As will be further detailed, the Processing Unit was designed to perform two operations simultaneously.

This operation mode is supported by a selector, which allows the use of the OFM memory's second port either for enabling the double write operation or for reading data. When the Control Unit sets the second port for the read operation, the selector works as a direct wire to feed the multiplexer. The second input port of the multiplexer is connected to the Control Unit, which is in charge of evaluating for each iteration if the data required for processing is saved in the double FIFO. As will also be described later, the FIFO helps to optimize the processing of input values with data dependency by reducing the communication with the output memory.

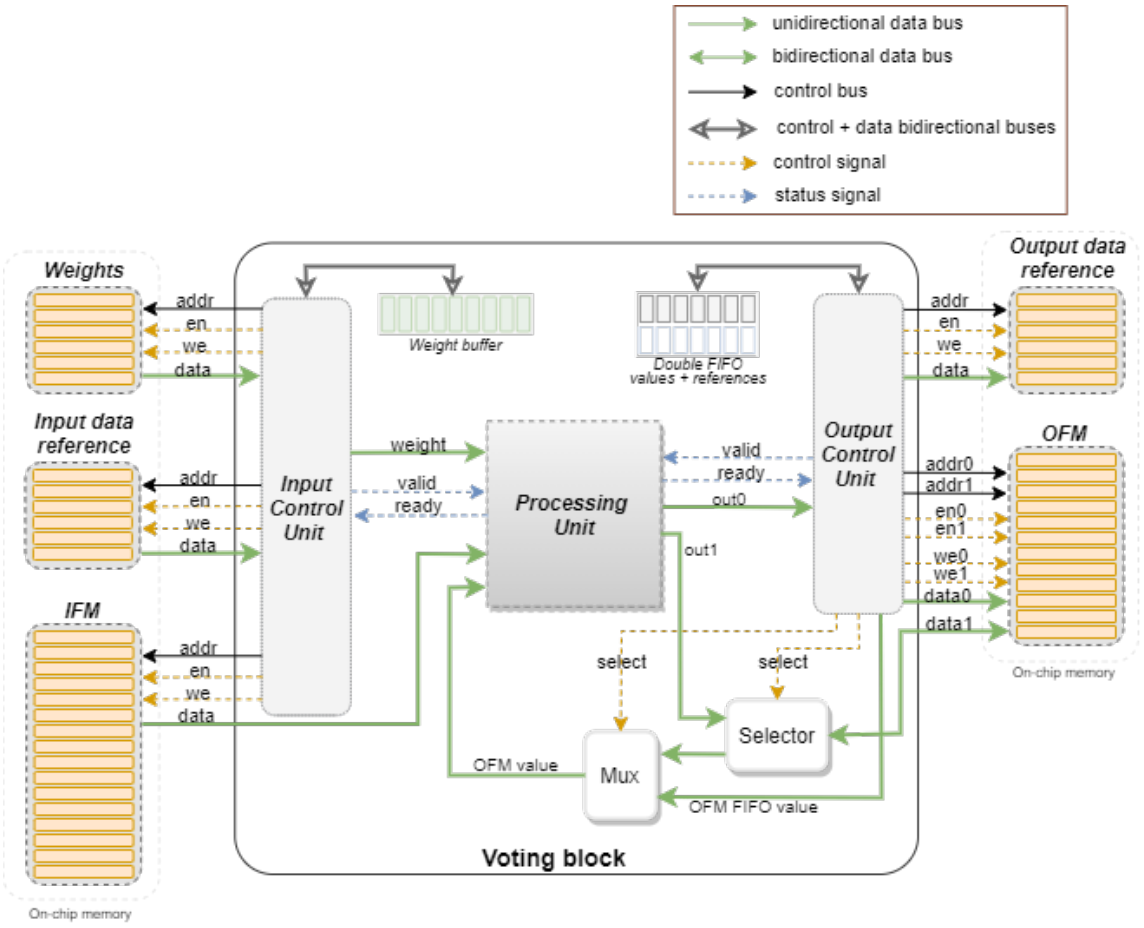


Figure 3.15: Voting block architecture

3.3.1 Four-stage sequential convolution

From the example shown in figure 2.20 with a size three filter, for each input value, nine multiplications must be performed between the input and the Voting weights. The result of each of the multiplications must be added to the value already stored in the corresponding output position. The mechanism is detailed in figure 3.16, which presents a simple example of how voting scheme-based convolution works. Voting convolution receives as inputs the IFM value to be processed, the filter and the values that are stored in the

output memory before the convolution is performed. After all the data read, operations can be performed, and the result must be written back into the output memory. According to the illustrated mechanism, the Voting block's Processing Unit can then be represented as a set of DSPs, proportional to the filter size used, where multiplications and additions are performed simultaneously. For the example illustrated in figure 3.16, the input is a 6x6 FM with only one non-null value and the convolution operation is performed with stride 1. Assuming a filter size of 3x3, nine DSPs would be responsible for multiplying the input value by one of the Voting weights and also perform a sum with the value coming from the output memory. Thereby, each one of the DSPs will have its weight and partial sum value specifically associated, however, they all share the same input value where the convolution will be applied.

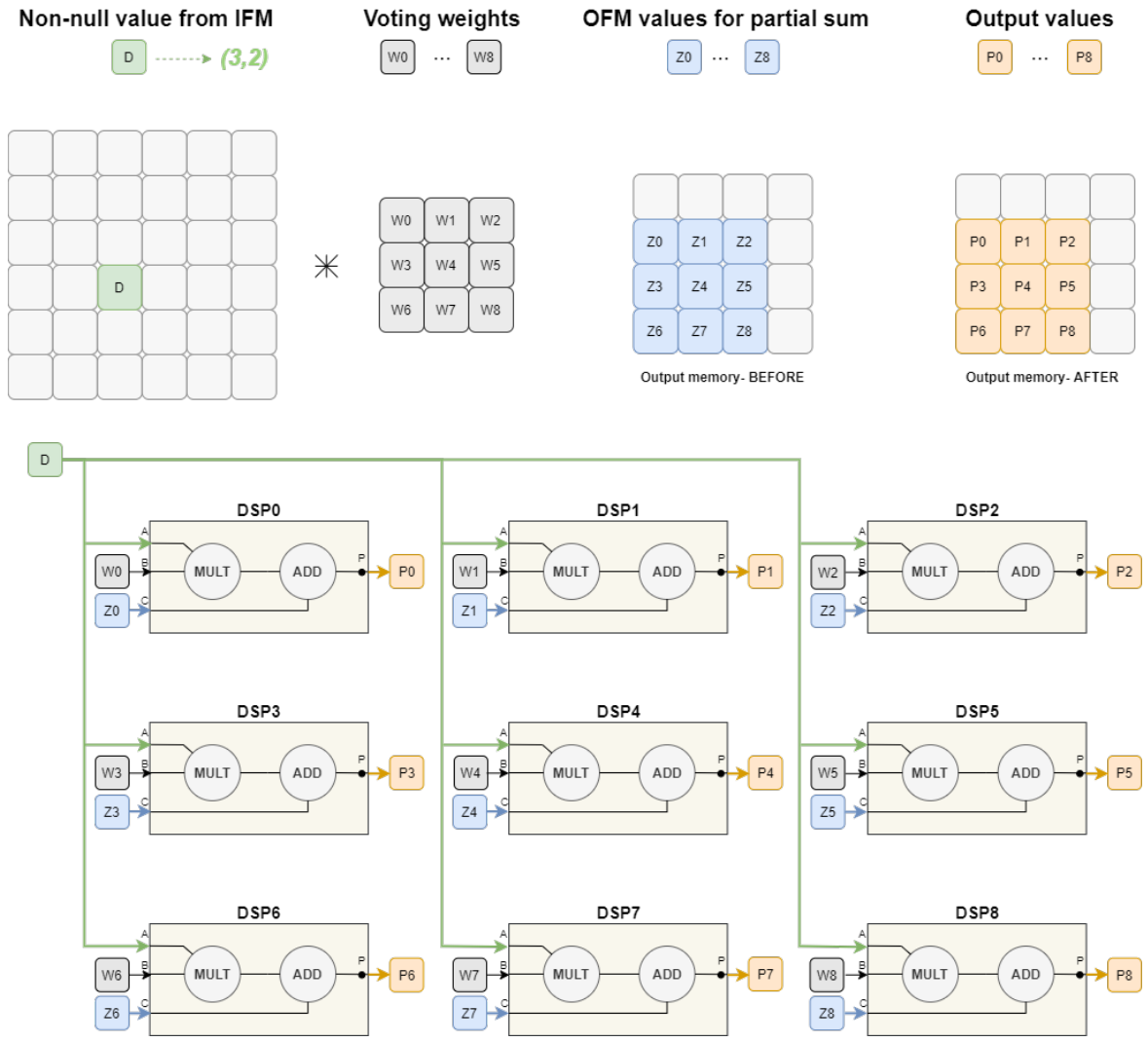


Figure 3.16: Voting convolution mechanism

Considering the architecture presented in figure 3.16, the convolution can be described as a four-stage methodology. The process starts with the fetch of the input non-null value using the values references to be processed as represented in 3.15. From the value reference, it is possible to read the exact position in

the input memory. This is followed by the fetch of the partial output values used in the sum operation after the result of the multiplications are calculated. In the next stage, the multiplication and addition operations are carried out inside the block's Processing Unit using all the data previously collected from the memory. As the last stage, the output from each DSP is stored in the output memory.

The process can be distinguished into the four phases described previously, both represented in figure 3.17. The multiplication and addition operations are performed simultaneously as all necessary data has already been read from memory during the two initial stages. Regarding the memory read and write operations, these are carried out sequentially and individually due to the limitation of the memory data access ports.

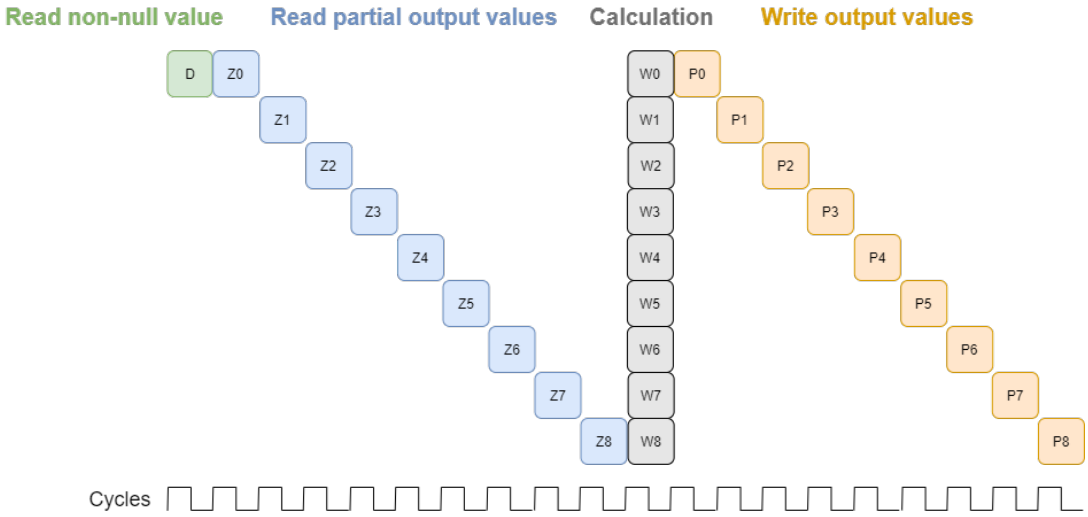


Figure 3.17: Four stage sequential voting convolution operation

From the analysis of the stages involved in the convolution, an important aspect to be noticed is the time consumption of the second and fourth stages, corresponding to the read and write operations of data from/to memory. Using the cycles shown at the bottom of figure 3.17 as a reference, the stages together consume 18 of the 20 total cycles, corresponding to 90% of the time.

The two stages mentioned are distinguished by the type of operation, read, and write, nonetheless both operate on the same memory. This suggests the integration of efficient solutions in terms of memory type to store the data involved in the convolution. With the design of the Voting block for a hardware implementation, Block Random Access Memory (BRAM) emerge as a viable solution for storing data. BRAMs offer different types of fast-access and low-power consumption memory which is fundamental to mechanisms that depend on intensive memory access to fully operate.

3.3.2 Pipeline-based single computing

According to Voting requirements, read and write operations to the same memory are required for each input value to be processed. One of the solutions that allow simultaneous read and write in the same memory is the Dual Port BRAM. This memory block is composed of two ports responsible for reading and writing, respectively. This feature allows evolving from the 4-stage sequential processing, illustrated in 3.17, to pipeline processing, shown in figure 3.18.

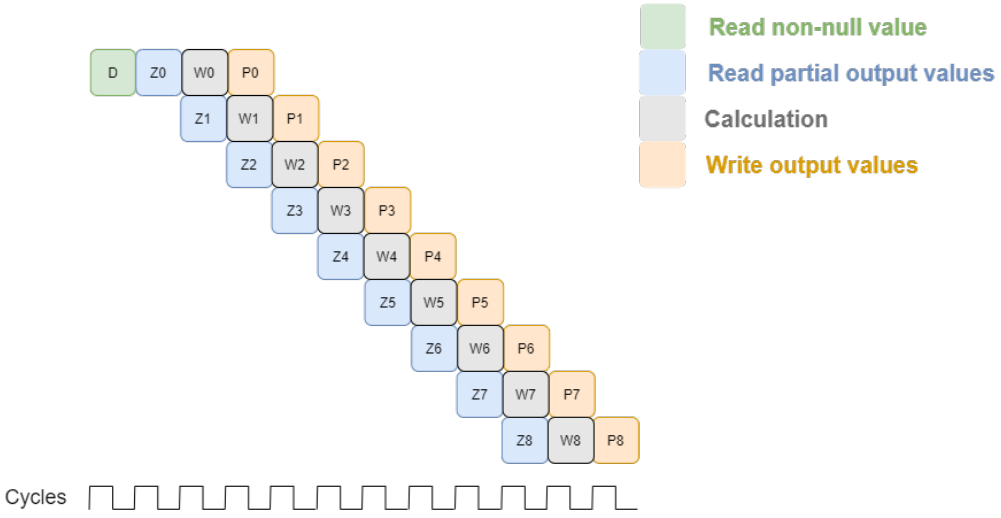


Figure 3.18: Pipeline-based voting convolution

Comparing the two strategies reveals a notable advantage in the pipelined processing in terms of the convolution's execution time. With this mechanism, the input value to be processed and the first partial output value are read, followed by a pipeline execution until the last filter value is applied to the input data. With the pipeline mechanism, the cycles needed to complete a convolution are reduced to 12, which corresponds to a decrease of 60% compared to the sequential mechanism.

Pipeline-based voting convolution reveals yet another important aspect, as only one calculation is performed per cycle. According to the architecture presented in figure 3.18, the number of possible multiplications to be simultaneously performed corresponds to the kernel size used in the convolution. Thus, for the illustrated example 3.16, nine DSPs would be allocated to complete all calculations in just one cycle, translated by the third stage of diagram 3.17. With the pipelined approach, only one calculation is performed each cycle, thereby there is no need to allocate resources to execute more than one multiplication and sum.

As represented in figure 3.19, the processing unit only has a logic element that implements the generic equation for any input value, weight, and partial output value. The Voting block processing unit is then capable of performing all the multiplication and addition operations required in a convolution without

additional resources. In each cycle during the pipelined processing, the Processing Unit receives new input values and new results are computed.

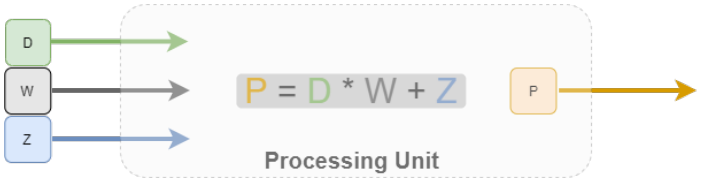


Figure 3.19: Voting block Process Unit

All data flow that enters and leaves the logic unit and the necessary synchronism between reading and writing data in memory is regulated by the Control Unit in the Voting block. The controller is also responsible for managing the operation's order within each convolution and consequently the order of all convolutions under the input data. This entire process can be identified from a loop, as shown in the figure 3.20. The beginning of each convolution starts with reading the IFM position where the value to be processed is. After reading the position, the respective value is read from memory and then sent to the Processing Unit.

In each of the operations performed within the processing unit, a weight value is required to multiply by the input, as represented in 3.19. To prevent kernel values from being repeatedly read from memory whenever a new convolution starts, they are initially read from memory and stored in an internal buffer. This is a viable solution because it is a reduced amount of data and also allows quick access without the need to communicate with the memory outside the block in each iteration.

As already illustrated by figure 3.18, for each input value the corresponding values in the output memory are also read. After all values read and the convolution complete, the position of the next input value must be read and the process repeats. After applying the convolution over the last input value, the process ends, as the filter was applied over the entire input data. The diagram shown in figure 3.20 describes the entire process detailed above. It should be noted that the Processing Unit is limited to carry out the operations that follow the equation presented in 3.19, while the Control Unit manages the communication with the memories.

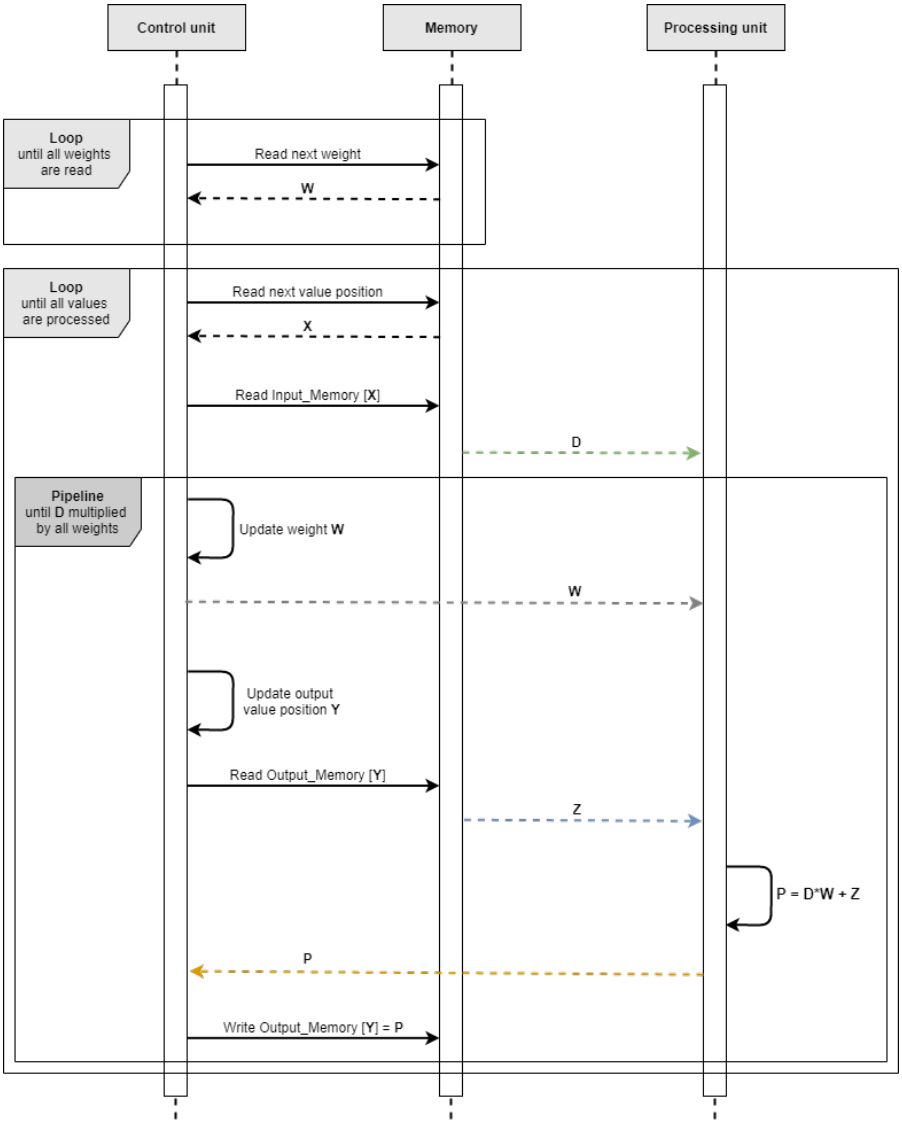


Figure 3.20: Voting convolution iteration loop

The constant update of the write and read addresses in the output memory is another of the Control Unit tasks. In the example shown in figure 3.16 for a size three filter, the Voting convolution needs the nine values present in the output memory to perform the convolution. Each of these memory addresses must be calculated individually on each iteration. The output memory address calculation follows a specific logic and depends on the memory address of the input value being processed at the moment. Following the example shown in 3.16, it can be deduced that the coordinates of the output values follow the dependency logic represented in figure 3.21. Whenever a new input value is processed, new positions of the output values must be calculated in each iteration based on the position of the input value. Naturally, for different kernel sizes, more or fewer values must be read from the output memory and therefore new positions need to be calculated to access them.

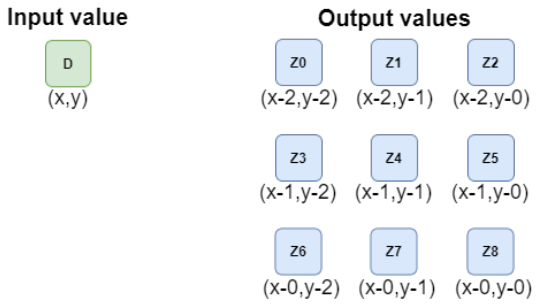


Figure 3.21: Coordinate's dependency

3.3.3 Pipeline-based double computing

With the hardware design of the Voting block, one of the main focuses is efficient implementation taking the maximum advantage of the sparse characteristic of the data to achieve the best possible performance. Sparse data is regularly associated with point clouds or their projections for 2D representations. One example is pseudo-images, used in 3D object detection models, such as PointPillars [5]. When objects are detected in the scene, clusters of relevant points can be found after the LiDAR sensor scan. This feature can then be analyzed and exploited to process spatially close points more efficiently.

The processing of two consecutively positioned non-null values in the input memory share certain positions in the output memory to complete the convolution. An example of it is shown in figure 3.22. For the first convolution, the Z0-Z8 positions are read from the output memory and after completing the convolution, the P0-P8 values are written in the same positions. With the processing of D1, some output memory positions are shared with those read for D0 processing. Namely, the positions filled with the values P1, P2, P4, P5, P7 and P8. These positions correspond simultaneously to the output of D0 processing and to the input of D1 processing.

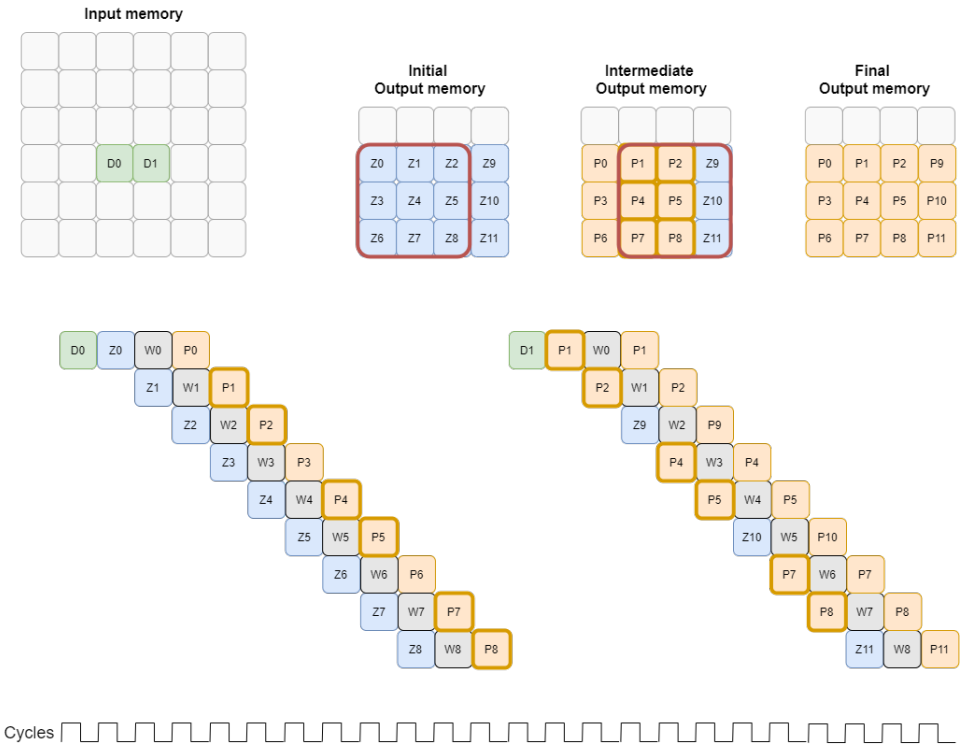


Figure 3.22: Data dependency on spatially close values

Although the Voting block processes each input value individually, some techniques can be adopted to decrease the processing time and improve the block performance. In figure 3.22 is an example of two spatially close values that are processed consecutively. In this scenario, part of the D0 processing output matches some of the D1 processing input. Given this scenario, there is a possibility of leverage the data dependency and share data directly between different iterations. This sharing can be done through temporary storage inside the block of the D0 processing output until the D1 processing starts. This mechanism gives the possibility of evaluating, in the current iteration, if there is any data dependency. In the case of data dependency, part of the processing data is already inside the block so there is no need to communicate with the memory outside the block to read that data.

The advantage of storing the processing output of each input value can be analyzed from figure 3.23. Unlike what happens in scheme 3.22, data dependency is leveraged to accelerate the current pipeline. With the necessary weights and partial output values already stored within the block to perform the calculations, they can be parallelized in the Processing Unit. In the example shown in 3.23, although six values are reused from the last convolution and the weight values are all available, only two calculations are performed simultaneously. The number of operations performed at the same time is regulated by the memory ports limitation, which only allows two memory operations at once. For True Dual Port BRAMs, the usability of the ports can vary between: write + write; read + read or write + read. In practice, the six

calculations could be parallelized inside the Processing Unit, however, only two values could be stored in the output memory at a time, which makes it useless to parallelize more than two calculations.

With the parallelization of two calculations, the pipeline format is modified, and the optimization is verified as the two available output memory ports are both used to write already processed values. This mechanism is not only logically correct but also improves pipelined processing performance. In the detailed example, it is possible to obtain a reduction of three cycles compared to the one shown in 3.22. Output values stored temporarily at the end of each pipeline are replaced each time a new input value is completely processed. This mechanism does not consume large amounts of memory, given the constant reuse of the buffer within the block. On the other hand, for a large sparse dataset with many clusters, this technique can lead to notable improvements in performance.

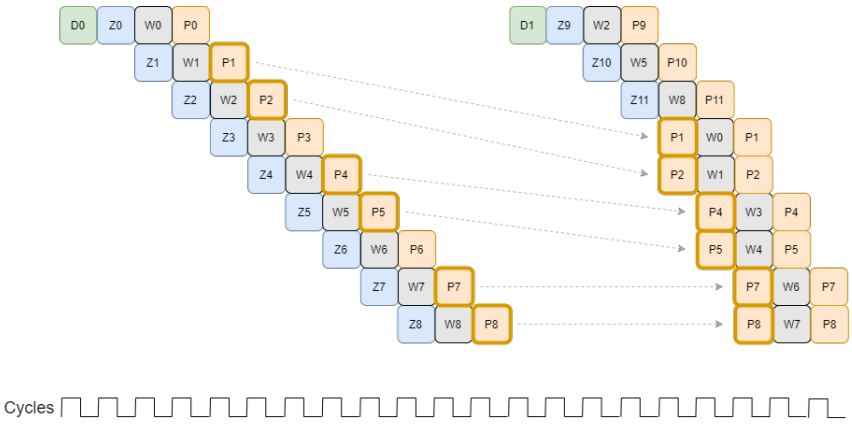


Figure 3.23: Convolution shift

Given the possibility of increasing the throughput of the processing pipeline, there is a need to adapt the Processing Unit internally. According to the design shown in 3.19, a single processing element would be used within the unit. This element would be responsible for implementing the highlighted equation to fulfil the block computation rate represented in figure 3.20.

With the integration of the technique illustrated in figure 3.23, in certain pipeline iterations, two calculations should be performed at the same time. Then, the Processing Unit's functionalities must be extended to be able to perform twice the operations presented in 3.23. Following the same approach demonstrated in 3.6 and 3.16, the Voting Block Processing Unit is now composed of two DSPs. Figure 3.24 represents the internal organization of the block's Processing Unit, consisting of two DSP's. Each one is designed to perform a multiplication and addition operation, doubling the execution capacity of the block and fulfilling the requirements to integrate the technique show in figure 3.23.

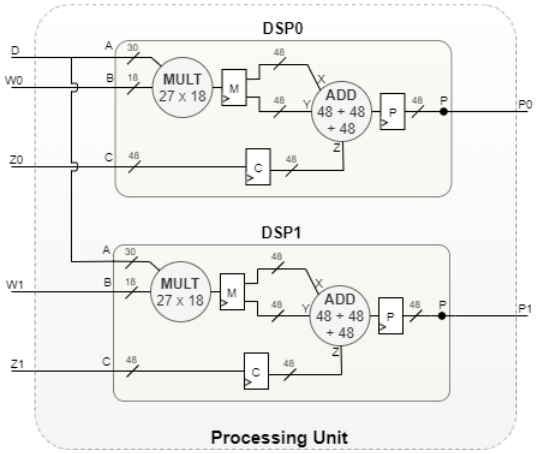


Figure 3.24: Processing Unit design architecture

With the dual computation unit, it is possible to take advantage of the data dependency that occurs in situations as exemplified in 3.23. The second DSP instantiated in the Processing Unit serves as a computation assistant whenever parallelism is viable, as described in figure 3.23. As soon as the data stored in the temporary output buffer can be reused in the current pipeline, the Control Unit must assign the data over the two DSP's and get all the processing power simultaneously. In addition, the Control Unit must disable the read + write operation and enable the double write operation on the output BRAM. Thus, the results from the Processing Unit can be immediately stored in the output memory without negatively affecting the DSP's computation rate.

The scheme presented in 3.25 refers to the processing of the D1 value, relative to diagram 3.23. Through the computation iterations of the Processing Unit, it is possible to observe that in the first three iterations only DSP0 is used to perform the calculations. As mentioned before, in the first iterations there is no advantage in using the two DSPs to perform the calculations because there is only one port available for writing values into the output memory as the second is being used to read the Z9, Z10 and Z11 values.

During the last three iterations, no value needs to be read from memory since the values are already stored in the auxiliary buffer within the Voting block as the result of the D0 processing. After the Control Unit detects the data dependency using the input values coordinates, activates the two computation elements, with the base and auxiliary DSP both operating in parallel as illustrated in 3.25.

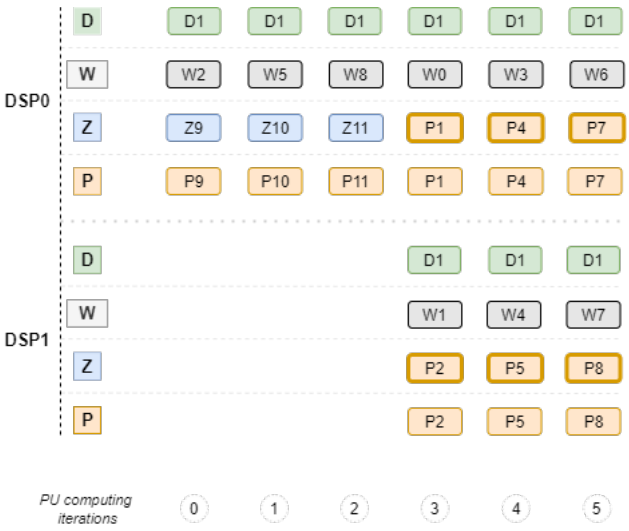


Figure 3.25: Processing Unit iterations example

3.3.4 Weight-based optimization

According to the specificity of Voting scheme-based convolution, some optimizations can be performed when the data to be processed assumes certain characteristics. One of the optimizations already detailed is related to the data dependency existing in the processing of spatially close data in the IFM. This situation favours, even more, the use of the Voting mechanism since with the technique described in 3.23 it becomes possible to reduce the processing time of the input values by sharing data between different iterations.

In addition to the characteristics of the input data, certain particularities of the filters can also bring opportunities to optimize the pipeline processing. The values assumed by the filter weights depend on the training phase and the model itself. However, weights with null values are a possible situation and even quite recurrent when integrated with optimization techniques such as pruning and quantization, as will be further shown in chapter Tests and results.

Regarding the Voting mechanism, there is the possibility of taking advantage of such scenarios when null weights are present in the filters to increase the performance of the pipeline. According to the base equation presented in figure 3.19, when the weight value is null, then it is verified: **P = Z**. Z is the value read from the output memory at position *(x, y)*, and P the value that will be written in the output memory at the same *(x, y)* position. This means that the value stored in that position before processing a given input value will be the same after the process finishes. Given these circumstances, the multiplication of D by 0 can be ignored and the pipeline jumps to the next calculation. The Control Unit of the block should identify these occurrences and remove from the pipeline the calculation execution associated with the zero-weight value.

Figure 3.26 shows the procedure that should be assumed when there are null weights in the kernel. Note that by ignoring the two unnecessary calculations, in this specific situation the pipeline can be finished two cycles earlier. This technique is especially advantageous when a filter containing null weights is applied over a large amount of input data since many unnecessary calculations are discarded and the execution time is proportionally reduced.

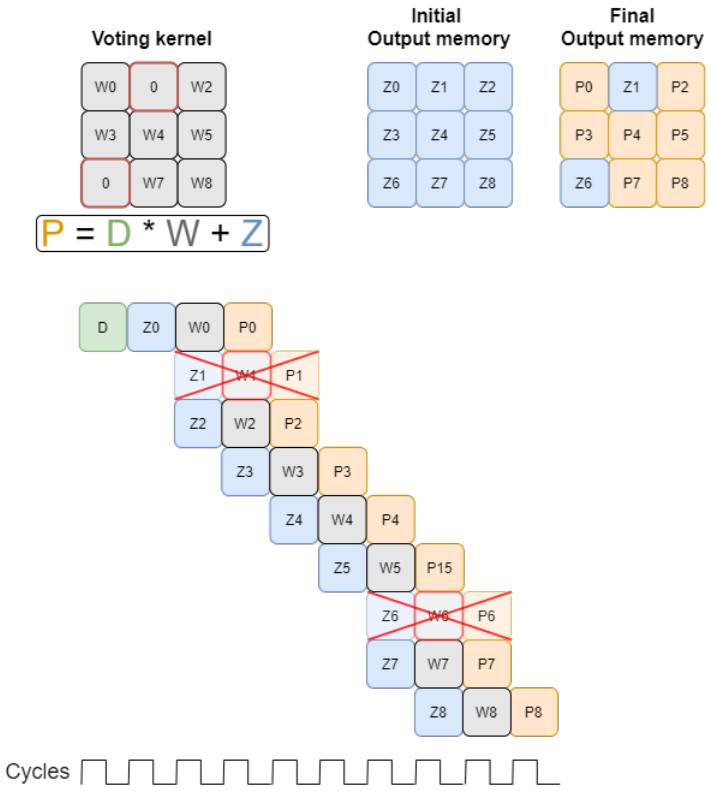


Figure 3.26: Null weights optimization

3.3.5 Output references generation

The relevant information can be associated with the values from the feature map that are not null, as a null value is considered meaningless information for perception-type tasks. The number of non-null values in the feature maps tend to increase as more processing layers are applied. Figure 3.16 shows a simple example where the input only has one value to be processed, but the output result has nine non-null values after the filter has been applied. In the same way, CNNs decrease the sparsity as data flows to deeper layers in the network. However, in certain situations, sparsity levels can be high across multiple layers favouring the use of sparse convolutions in more than just one layer. In practical cases, the Voting block would be replicated to use the voting scheme-based convolution in the desired layers. The network architecture would then be presented with several linked Voting blocks as the output of one block would correspond to the input of the next block. While the instantiation of several Voting blocks can bring great

improvements in the efficiency of sparse data processing, it is important to note that the output of one block must satisfy the input requirements of the next one.

One of the most critical requirements is the position where the non-null values are located. To build consecutive convolution layers using the Voting block, each one must register the non-null value positions at the output. It is essential to extend the block's functionalities to register each output value together with the corresponding position stored in memory. Furthermore, the number of values stored in the output memory must be counted to inform the next block of how many iterations are needed to complete the convolution. With this information, each block will have all the necessary data to perform voting scheme-based convolutions.

Figure 3.27 presents an example case of processing two values, D0 and D1, accessed from the positions stored in the input reference memory. The scheme helps to describe the order of values that are written to output memory along with their corresponding positions. Simultaneously to the writing of the values in the output memory, their positions are written in the output reference memory. However, the two input values are spatially close, meaning that some output positions will be shared as described in the previous example 3.22. With the sharing of positions, some will be repeated and therefore should not be rewritten in memory. References to the repeated positions in memory cannot happen as it compromises the processing of the next Voting block. As a result, the same position will be read more than once which leads to wasted time. For the example, in figure 3.27, positions 5, 6, 9 and 10 are written twice to the output memory, however, the second time, the reference to those positions do not need to be recorded.

The management of the position references written in memory uses a FIFO. Each time a new value is written to the output memory, it is checked if its position is already present in the FIFO. The verification helps to identify if that position has recently been written to memory. If the position is already stored in the FIFO then it will not be written in memory, however, if it is not stored in the FIFO it will be written in memory and also in the FIFO itself. Although this technique is viable to prevent references from being repeated in memory, there must be an efficient management of the FIFO size for each specific situation. To ensure that no values are repeated in FIFO, the size should match the filter size times the OFM size. Despite the allocation of more memory for the FIFO is not logically incorrect, it increases the resources and time costs to search and store the values.

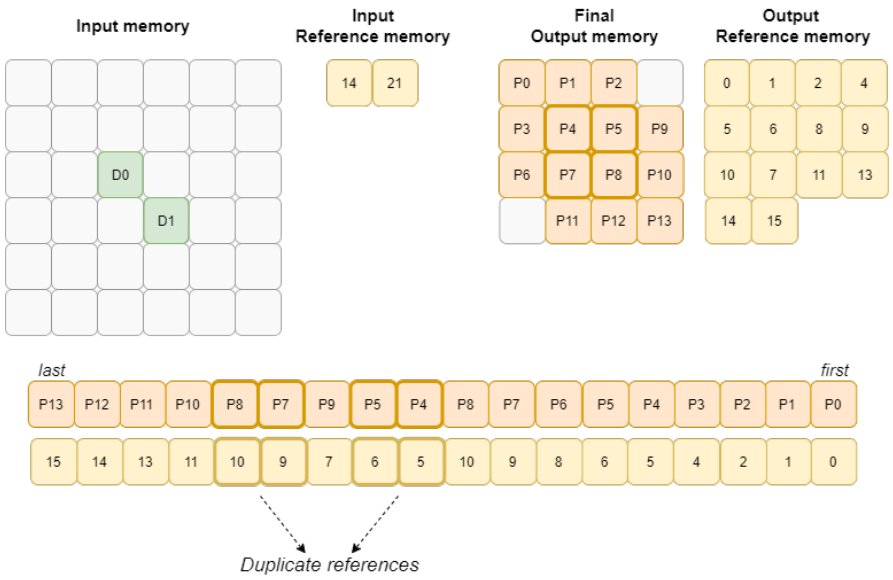


Figure 3.27: Voting Block output references generation

3.3.6 Data optimization

Convolutional neural networks for sparse data processing usually have as input a representation of the data captured by a LiDAR sensor. This data can later be used by models for perception tasks in 3D space. In some applications, the 3D models assigned to these tasks are executed on platforms with great computational power, such as GPUs. Using GPUs, the models have access to memory and processing cells capable of delivering a high rate of operations between data with high resolution and still achieving good results in inference performance. However, in resource-limited, high-performance, and low-latency scenarios, data quantization is required to compress the model according to the memory available in the target board and to enhance power efficiency and performance in terms of inference time.

The integration of the Voting block in a CNN for sparse data processing will be confronted with the resolution of the data in the network. In situations where the network is running on a platform with low resource limitations, the data is usually a 32-bit floating point. To enable the hardware to process the data and reduce hardware design complexity, data quantization is adopted. Voting block input data needs to be quantized before participating in the Processing Unit internal operations. Through quantization, the values are transformed to fixed-point and assume a certain number of bits for the integer and fractional parts. The quantization level can be customized individually for the weights and the feature map values and it is also up to the user to decide how many bits will be allocated to each part of the values in fixed-point format.

According to the configuration made by the user, the Voting block controller must configure the pro-

cessing unit, so the DSPs operate correctly according to the input values format. For the calculations, the values must first be converted from floating-point to fixed-point. The Intellectual Property (IP) adopted for the conversion is the Xilinx Floating-Point Operator IP core which enables floating-point arithmetic on hardware and can be customized for operation, word length, latency, and interface. After the DSPs perform the calculations, the result also comes out in the fixed-point format. However, if the values written to memory are in floating-point format, then before the output value is sent to the output memory it must be previously converted to floating-point. Given the extensive functionalities of the Xilinx Floating-Point Operator IP, it is also used for dequantization.

Figure 3.28 illustrates where the Xilinx Floating-Point Operator IP is connected within the Voting block to perform the quantization of feature map values, weights, and output partial values. As they are three distinct values, three IPs are instantiated and connected to the inputs of the two DSPs. Counting with the dequantization of the output values from the DSPs, a total of seven Floating-Point Operator IPs are used inside the processing unit as both DSPs share the same input value D.

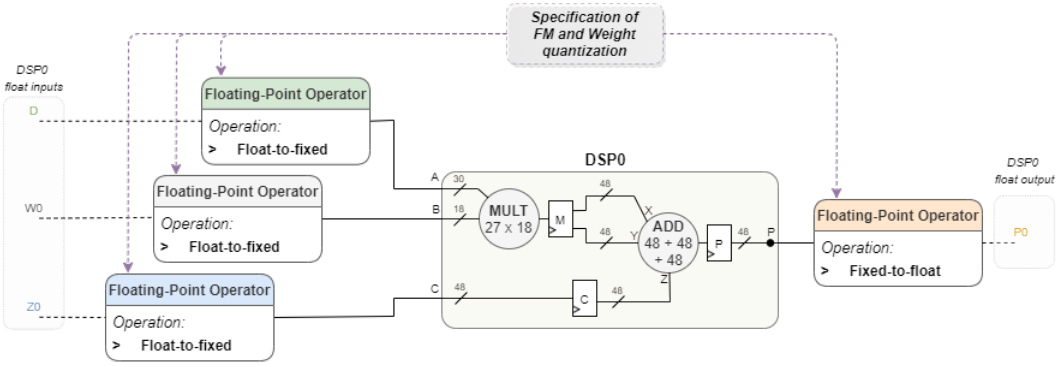


Figure 3.28: Processing Unit I/O quantization

For each of the DSPs within the processing unit, the inputs have different widths. For the three inputs used, A, B and C, the corresponding ports' widths are 30, 18 and 48 respectively. From each input port, a fixed-point format value is used for the multiplication and addition operations. However, for the operations to be done correctly, first, it is necessary to align the data in the input ports. Alignment is achieved by positioning the decimal point at the same position in both the feature map and weight values. This mechanism is important as a multiplication between two values with decimal points in different positions may generate incorrect results. From figure 3.29, it is noticeable the same fraction width for A and B ports.

As a second operation, the multiplication result is added to the C port value to satisfy the equation 3.19. The multiplication result between two fixed-point values is a number twice as wide in the fractional part. Again, to align the position of the decimal point, the fractional part of port C value must have twice

the width of A and B ports, so it can be added with the multiplication result correctly. Figure 3.29 illustrates the rules that should be followed regarding the decimal point alignment, for a generic situation. Depending on the quantization level specified by the user, one must take care to correctly configure the quantization and dequantization IPs, so the values are converted to the desired formats.

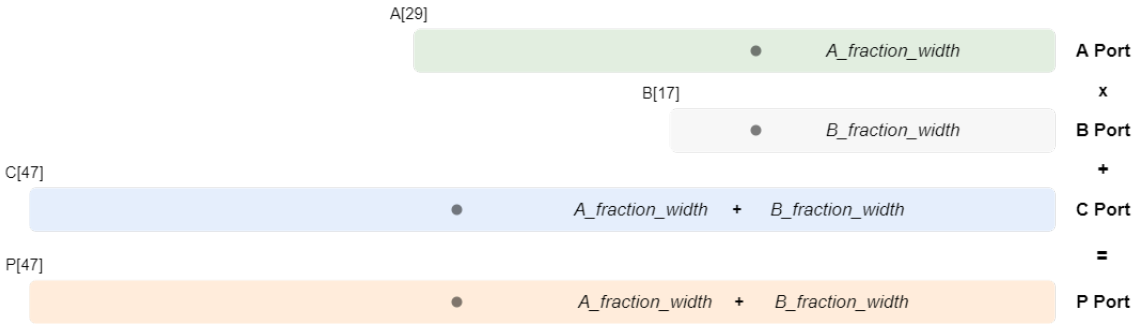


Figure 3.29: DSP input data alignment

3.3.7 User configuration

The hardware design of the Voting block allows the construction of a tool to perform voting scheme-based convolution over sparse data. The convolution mechanism requires the specification of certain parameters that help define the characteristics and type of convolution to be applied. Considering the configuration requirements, the Voting block presents itself as being customizable according to the typical parameters of convolutions. As such, a set of variables are available to the user which offers a certain customization level for the data processing. Although the level of customization of the block allows only a few modifications to be implemented on top of the voting scheme-based convolution mechanism, the parameter values are important. The non-specification of a single parameter compromises the correct functioning of the block, and the entire processing can become logically incorrect, although default values are set for each one.

Initially, the user should specify both the filter and the Input Feature Map sizes used in the convolution. According to the Voting requirements, the filter must have an odd size so that a central position is used as a reference to apply the filter in a certain region. Along with these parameters, the user must also specify whether the convolution has padding plus the stride value. Padding and stride are two common parameters in convolutions. Nonetheless, the values typically used are shown in figure 3.30, covering the vast majority of the desired requirements for a convolution.

One of the most important requirements of Voting is the position of non-null values in the input feature map. For all positions to be read and their values processed consequently, it is convenient to inform the Voting block how many values there are in the input to be processed. The user must then specify

the number of values that need to be read from the input memory so the block can recognize when the convolution is finished.

The Voting block hardware development proposed will be validated by replacing one of the layers of a 3D object detection model, which in this case is PointPillars. In software, the model operates with a 32-bit floating-point resolution, both for the feature map data and weights. To perform hardware operations, these values need to be quantized. With the feature map and weights quantization, these values are converted to a fixed-point with a given resolution. There are different levels of quantization, and the user is able to choose which one is intended by specifying how many bits must be allocated for the integer and fractional parts individually.

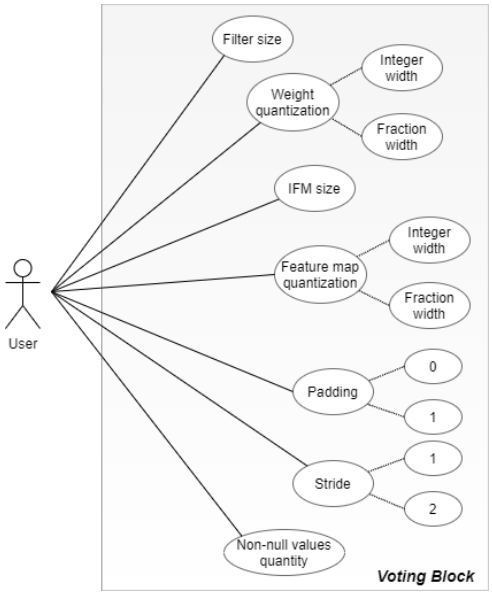


Figure 3.30: Voting Block configuration Use Cases

Chapter 4: System implementation

This chapter presents the hardware implementations of the dense/traditional convolution module 4.1 and the voting block 4.2. The implementation details are addressed according to the design architecture discussed and proposed in sections 3.2 and 3.3, respectively.

4.1 Convolution-based hardware accelerator

During the task of implementing an hardware accelerator optimized to perform convolution operations, all the decisions raised in section 3.2 were carefully considered. Therefore, this implementation has as baseline the architecture represented in figure 3.13, which comprises several sub-modules with very specific tasks. To enable all the functionalities to be integrated into the system, each module needs to interface with other sub-modules. The interface can be represented as a communication channel where a given system component can receive and transmit relevant information. In order to provide the necessary interoperability between these sub-modules for successful execution of convolution operations, these sub-modules were integrated into a module called *top module*. The *top module* has the responsibility to communicate with resources external to the module in order to access all data needed to operate correctly, as well as distribute this data across the sub-modules according to the data flow discussed in section 3.2.

The interface of the top module is detailed in code snippet 4.1. The parameters seen on the module definition reflect the configuration level provided to the user together with the information needed to inform the module about the resources' limitations. From the values specified for each parameter, the module is able to calculate how many filters will be simultaneously applied to the input feature map as well as the appropriate level of parallelism, as discussed in sub-section 3.2.5. The calculations help to define the interface of the top module as the ports' width depend both on the number of filters used for each iteration and the number of processing elements allocated for each filter. For example, the *i_ifm* port's width depends on the *pe_filter* parameter since, for each processing element used to perform a convolution, a value must be fetched from the input feature map memory each clock cycle. As a consequence of the input feature map values fetched at the same time, the *o_ifm_r_addr* port's width is also adapted to support a read address for each processing element. The same logic applies to the *i_ofm* port, since for each PE allocated to perform a convolution an output feature map value is needed for the addition operation.

Following the same criteria, both *o_ofm_r_addr* and *o_ofm_w_addr* ports' width are proportional to the number of PEs allocated, and each read and write address is compacted in the same port.

Besides the ports used to transfer the feature map data and the addresses for the memory access, signals are used to synchronize the convolutional module's controller and the data management outside the module. For a resource-constrained platform, on-chip memory space may not be sufficient to store all the input data. The implemented strategy consists of continuously manage the input/output data fetching process by transferring the amount of data possible for the top module's block rams. The controller uses the signal *o_load_ifm* to inform when the next input feature map should be transferred to the block rams, and the *i_ready* indicates when a new iteration can start.

```
1  `include "global.v"
2
3  module convolutional_module#(
4      parameter  KERNEL_SIZE      = `KERNEL_SIZE,
5      parameter  IFM_SIZE         = `IFM_SIZE,
6      parameter  PADDING          = `PADDING,
7      parameter  STRIDE           = `STRIDE,
8      parameter  MAXPOOL          = `MAXPOOL,
9      parameter  IFM_CH           = `IN_FM_CH,
10     parameter  OFM_CH           = `OUT_FM_CH,
11     parameter  DSP_AVAILABLE    = `DSP_AVAILABLE,
12     parameter  MEM_AVAILABLE    = `MEM_AVAILABLE,
13
14     //  output feature map size
15     localparam ofm_size = (((IFM_SIZE - KERNEL_SIZE + 2 * PADDING) / STRIDE) + 1),
16     //  memory consumed by each filter
17     localparam mem_filter = ofm_size**2 * `D_WIDTH,
18     //  number of parallel filters possible
19     localparam parallel_filters = ((MEM_AVAILABLE / mem_filter) > OFM_CH) ? OFM_CH
20                                   : MEM_AVAILABLE / mem_filter,
21     //  number of PEs available
22     localparam pe_available = DSP_AVAILABLE / (KERNEL_SIZE**2),
23     //  number of PEs for each filter
24     localparam pe_filter = pe_available / parallel_filters,
25     //  number of filters' iterations
26     localparam num_iterations = OFM_CH / parallel_filters,
27 )(
28     input wire i_clk,
29     input wire i_rst,
30     input wire i_ready,
31     input wire signed [`DSP_PORT_A_WiDTH * pe_filter -1:0] i_ifm,
32     input wire signed [`DSP_PORT_B_WiDTH * parallel_filters -1:0] i_weight,
33     input wire signed [`DSP_PORT_C_WiDTH * pe_available -1:0] i_ofm,
34
```

```
35      output reg signed [DSP_PORT_P_WiDTH * pe_available -1:0] o_conv_result,
36      output reg [$clog2(IFM_SIZE**2) * pe_filter :0] o_ifm_r_addr,
37      output reg [$clog2(KERNEL_SIZE**2) * parallel_filters:0] o_weight_r_addr,
38      output reg [$clog2(ofm_size**2) * pe_available:0] o_ofm_r_addr,
39      output reg [$clog2(ofm_size**2) * pe_available:0] o_ofm_w_addr,
40      output reg o_ofm_w_en,
41      output reg o_load_ifm,
42      output reg o_done
43  );
```

Code Snippet 4.1: Top module interface

4.1.1 Controller

The convolutional module's development aims to build a configurable tool to implement CNN networks in hardware. In a CNN layer, several filters are applied over an input set of data to generate the output feature map. The input data volume in 2D CNNs are characterized by a certain width and height and also by a number of channels. When implementing a CNN layer in hardware using the convolutional module, efficiency and performance are the two most important metrics. Efficient hardware implementation is convenient to enable the module deployment on platforms with constrained resource and power consumption, however, performance is also relevant to achieve good execution time results, since CNN's target applications are often demanding with regard to inference times. The hardware implementation of the convolutional module must try to find a good balance between the two metrics since an efficient solution with little performance or vice versa compromises its use in a real-time application.

With the focus on the functionalities intended for the hardware module, in an ideal scenario, the filters of a certain layer would be applied simultaneously over all input data. This strategy would allow the best execution time performance, however, it would imply an excessive consumption of both memory resources and processing units. The limitations of the target platform are a crucial factor in determining which performance level is most appropriate and does not compromise the desired level of efficiency. The module must be aware of the resources that the platform has available and together with the configuration specified by the user, find the best possible balance.

Keeping in mind that the complete parallelization of all processing within the module is a difficult scenario, it is important to develop a mechanism that splits the operations into several iterations, as discussed in sub-section 3.2.5. The division of processing into iterations makes it possible to sequence the application of filters over the input data, as exemplified in the diagram 3.14. Since memory and processing units are the most critical resources, they should be used as the main criteria to define how

many iterations are needed. The processing units used within the convolutional module are the DSPs, and the amount available restricts how many PEs can be instantiated to perform the convolutions, as referenced in code snippet 4.1. On the other hand, the amount of memory available affects how many filters can be applied at the same time to the input data, since for each filter used, memory is needed to store the convolution result.

The management of the resources consumed by the convolutional module and consequently the number of iterations in which the processing will be distributed is managed by the controller. Figure 4.1 presents the convolutional module's state machine, composed of four states, each one representing a specific processing stage.

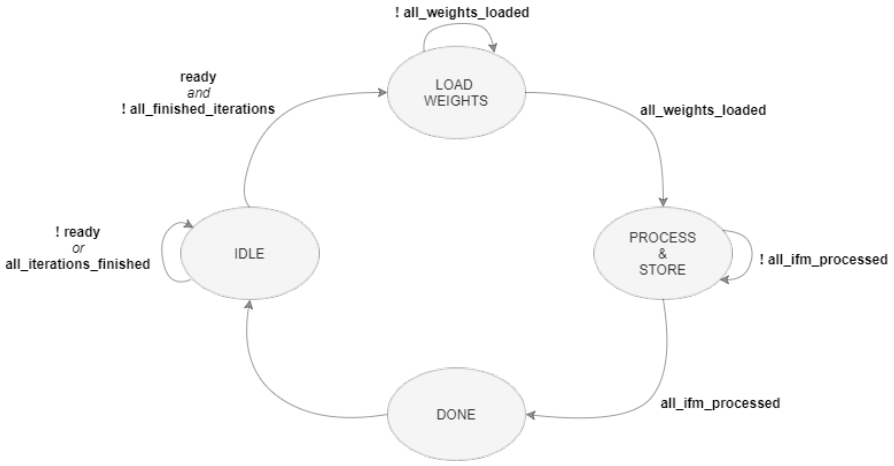


Figure 4.1: Convolutional module Finite State Machine

As mentioned before, the resource-limitation makes it impractical to perform all the process at once, so the state machine is adaptable to execute the same states as often as needed. The process starts in the *IDLE* state, where the convolutional block waits for a signal, called *ready*, to start processing using the interface ports presented in 4.1. The signal is used not only at the beginning of all the processing but also between each iteration, as the signal informs that a new input feature map has been loaded to the on-chip memory. With *ready* signal equal to one and the iterations not finished, the state machine jumps to state *LOAD WEIGHTS*, where the filters' weights are loaded to the module's internal registers. The *LOAD WEIGHTS* state is the one responsible to recognize which filters will be applied in the current iteration and for reading all the corresponding weights from the memory.

After all weights are loaded, the *PROCESS & STORE* state is executed. This state can be highlighted from the other ones since it is responsible to process all the input data and store the results in the output memory simultaneously. Depending on the current iteration, several filters can be applied concurrently, and for each one multiple PEs can be used to perform the convolutions, so it is important to note that, in

each clock cycle, several output values are processed and need to be stored in the memory at once. It is without a doubt the state that consumes the most time and it is only left when the convolutions of the current iteration are completed. As the last value from the input feature map is processed in the PE and stored in the output memory, the state machine jumps to the last state named *DONE*. In the last state, no processing is carried out, however, it is in charge of notifying that the current iteration is completed, and a new input feature map should be loaded to the on-chip memory for the next iteration. Besides that, at the end of each iteration, the data stored in the output block rams should be sent to an external memory, as they will be re-utilised to save the next iteration's results.

Code snippet 4.2 represents the iterations' management performed by the controller in the *DONE* state. Two registers are used for monitoring the end of each iteration and the corresponding sub-iterations. The variable *r_filter_it* represents the filters' iterations and *r_ifm_it* the iterations through the input feature map channels. Each time a set of filters is applied to the input data, one needs to be convolved to all the input channels, specified with the *IFM_CH* parameter. When the last IFM channel participates in the convolution, all the sub-iterations are completed, and the next iteration begins with the first input channel again. As represented in the example diagram 3.14, the processing ends when the last set of filters are convolved with the last IFM channel.

```
1  reg [$clog2(IFM_CH):0]          r_ifm_it;
2  reg [$clog2(num_iterations):0]  r_filter_it;
3
4  always @(posedge i_clk) begin
5      if(i_rst || r_state == s_idle) begin
6          r_ifm_it    <= 0;
7          r_filter_it <= 0;
8          o_load_ifm  <= 0;
9          o_done      <= 0;
10     end
11     else if(r_state == s_done) begin
12         o_load_ifm <= 1;                    //  load new IFM
13
14         if(r_ifm_it < IFM_CH - 1) begin
15             r_ifm_it <= r_ifm_it + 1;       //  process next IFM
16         end
17         else begin
18             r_filter_it <= r_filter_it + 1; //  apply next set of filters
19             r_ifm_it    <= 0;               //  start from IFM0
20             o_done      <= 1;               //  current iteration finished
21         end
22     end
```

```
23 end
```

Code Snippet 4.2: Iterations management

During the processing inside the convolutional module, input and output data are recurrently read and written in the memory, respectively. To manage all the communication with the memory, the controller manipulates the *write-enable* signal together with the addresses to access a specific memory position. Following the BSM scheme design presented in 3.4, for each cycle, a new input value should be fetched from the memory to continue the convolution. To control the sequence of read operations, each clock cycle the *Process & Store* state is being executed, the *o_ifm_r_addr* output register is continuously incremented so the next input memory position is read next. Similar to the read address, a different register is connected to the output memory's write address port to select the position where the current convolution result value should be written on. Each time the processing element indicates that a new value has been processed, the controller should increment the output memory's write address and also enable the write operation by assigning the value one to the *o_ofm_w_en* register.

The convolution between the input data and a filter extends to all the input channels, as the filter is applied to all the input feature maps. The result of the convolution along each channel is added to generate an output feature map. The controller should not only manage the write address to the output memory but also read the output values to perform the partial sum during the sub-iterations. The architecture design presented in the figure 3.13 indicates the instantiation of an accumulator and a double connection to the output memory where the values can be read and written at the same time. Following the same logic as the write address, each clock cycle the processing element outputs a new value, the *o_ofm_r_addr* register is incremented by one, to fetch the value stored in the next output memory position. Code snippet 4.3 refers to the memories' addresses management performed on the convolutional module's controller during the *Process & Store* state.

```
1  always@(posedge i_clk) begin
2      if(i_rst || r_state == s_idle) begin
3          o_ifm_r_addr    <= 0;
4          o_ofm_r_addr    <= 0;
5          o_ofm_w_addr    <= 0;
6          o_ofm_w_en      <= 0;
7      end
8      else if(r_state == s_process_store) begin
9          o_ifm_r_addr    <= o_ifm_r_addr + 1;
10         o_ofm_w_en      <= 0;
11
12         //  new output value from PE
13         if(r_pe_en) begin
```

```
14              //  increment write addr and read next OFM value
15              o_ofm_r_addr <= o_ofm_r_addr + 1;
16              o_ofm_w_addr <= o_ofm_w_addr + 1;
17
18              //  process not finished, enable write operation
19              if(row_num < last_row_num) begin
20                  o_ofm_w_en <= 1;
21              end
22          end
23      end
24  end
```

Code Snippet 4.3: Read and write addresses management

4.1.2 Inter output and intra kernel parallelisms

Two different types of parallelism techniques are adopted in the convolutional module to increase processing performance. The inter output parallelism is related to the output data generation and consists of building several output features maps at the same time. This mechanism is possible to implement since different output feature maps are independent of each other, enabling convolutions with several filters to be parallelized. The intra kernel parallelism is also used as a strategy to parallelize the operations associated with a convolution. Different regions of the input feature map are also independent of each other so they can be convolved simultaneously without interfering with the processing of another region. This type of parallelism was very detailed throughout the analysis and design chapter and is performed using several processing elements for the same filter.

Although both parallelism techniques are useful to achieve better computational power, they are implemented considering the resource restrictions imposed by the target platform specifications. With the application of more than one filter at once, each clock cycle a new set of output values need to be stored in separated memories. The amount of memory required to store each output feature map is proportional to the corresponding width and height, however, the level of the inter output parallelism is also conditioned by the memory available. On the other hand, the number of processing elements allocated for the filters that are being applied at the same time is dependent on the number of DSPs available. Plus, since the processing elements available are equally distributed, the more filters applied in parallel, the fewer processing elements are assigned to each one.

According to the parallelization mechanisms described, code snippet 4.4 represents the management of the output values that come out of all the processing elements instantiated. When both types of parallelisms are integrated with the convolutional module, for each filter, every related processing element

will output a new value each clock cycle. The access to each output value is done using two consecutives *for cycles* to iterate through those values and aggregate them into the convolutional module's output port named *o_conv_result*. The output port is prepared to concatenate all processing elements' output values, since each part of the register is mapped to a different output memory.

During *Process and Store* state, each iteration is characterized by the set of filters applied in parallel, and the sub-iterations are distinguished by the input feature map channel used for the convolution. As mentioned before, when the convolutional module is composed of a set of input channels, each one needs to be read and convolved with the filter. Code snippet 4.4 shows that when the channel being processed is not the first one, the convolution's result values are added to those already stored in the output memory. For memory-saving purposes, after the last input feature map channel is processed, the current iteration ends and the values from the output memory are transferred to an external memory, so the same output memory can be re-utilized in the next iteration. Between iterations, no time is wasted cleaning the memory as the controller is configured to ignore the output values when the first input feature map channel is being processed. After the convolution with the first channel, the values stored in the previous iteration are totally overwritten by the current ones, so in the following sub-iterations the output memory values can already be read and added with the ongoing convolutions' results.

```
1  //  convolutions' result from PE
2  reg [(`D_WIDTH * parallel_filters * pe_filter)-1:0]  r_conv_result;
3
4  always @(posedge i_clk) begin
5      if(i_rst || r_state == s_idle) begin
6          o_conv_result <= 0;
7      end
8      else if(r_state == s_process_store) begin
9          //  current iteration's filters
10         for(i = 0; i < parallel_filters; i = i + 1) begin
11             //  for each filter's PE
12             for(j = 0; j < pe_filter; j = j + 1) begin
13                 //  ignore stored results from last iteration
14                 if(r_ifm_it == 0) begin
15                     o_conv_result[(i * pe_filter + j) * `D_WIDTH +: `D_WIDTH]
16                       <= r_conv_result[(i * pe_filter + j) * `D_WIDTH +: `D_WIDTH];
17                 end
18                 //  include partial output sum after convolution
19                 else begin
20                     o_conv_result[(i * pe_filter + j) * `D_WIDTH +: `D_WIDTH]
21                       <= r_conv_result[(i * pe_filter + j) * `D_WIDTH +: `D_WIDTH]
22                         + i_ofm[(i * pe_filter + j) * `D_WIDTH +: `D_WIDTH];
23                 end
24             end
```

```
25          end
26      end
27  end
```

Code Snippet 4.4: Parallelism output data management

4.1.3 Processing Element

The processing unit of the convolutional module is composed of a set of DSPs connected in cascade, as shown in figure 3.6. The number of DSPs used is proportional to filter size and their connection respects the *Broadcast Stay Migration* model, characterized by schemes 3.2 and 3.3, where each PE represents a single DSP. All the operations required to complete the convolutions inside the module are assigned to DSPs, so the Processing Element is the component responsible for both instantiating and connecting the DSPs and also managing the cascade's input and output data.

Code snippet 4.5 demonstrates the instantiation and configuration of the DSPs required to build the cascade scheme. The presented configuration can be divided into Data Path, Data Ports and Control Bits. Starting with the datapath, each DSP is configured to read the input values directly from both A and B ports, discard the use of D port and enable the intern multiplier to generate a 48-bit result. Regarding the control bits, these are useful to configure the internal operating mode of the DSP. As described in the convolutional module design, each element of the cascade is responsible to implement the equation **P = A * B + C**, so the 9-bit operation mode configures the values that are selected in the internal multiplexers and participate in the ALU. *ALUMODE* refers to the operation mode performed in the ALU and is defined to add the multiplication result between A and B with the C port value.

```
1  generate
2  genvar i;
3      for(i = 0; i < KERNEL_SIZE**2; i = i + 1) begin
4
5          DSP48E1 #(
6              // Data Path Selection
7              .A_INPUT("DIRECT"),      // A input source
8              .B_INPUT("DIRECT"),      // B input source
9              .USE_DPORT("FALSE"),     // D port usage
10             .USE_MULT("MULTIPLY"),   // Multiplier usage
11             .USE_SIMD("ONE48")       // SIMD selection
12         )
13         DSP48E1_inst (
14             // Data Port
15             .P(w_outDSP[(`DSP_PORT_P_WiDTH * i) + (`DSP_PORT_P_WiDTH - 1) :
16                         `DSP_PORT_P_WiDTH * i]), // Primary data output
```

```
17
18              // Control Inputs/Status Bits
19              .ALUMODE(4'd0),          // ALU control input
20              .CARRYINSEL(3'd0),       // Carry select input
21              .CLK(i_clk),             // Clock input
22              .INMODE(5'd0),           // INMODE control input
23              .OPMODE(9'b000110101),   // Operation mode input
24
25              // Data Ports
26              .A(i_DataFM),            // A input
27
28              .B(i_Weight[(`DSP_PORT_B_WiDTH * i) + (`DSP_PORT_B_WiDTH - 1) :
29                          `DSP_PORT_B_WiDTH * i]), // B input
30
31              .C(i == 0 ? 0
32                  : (i % KERNEL_SIZE == 0) ?
33                   w_outRAM[`DSP_PORT_P_WiDTH * (i / KERNEL_SIZE) - 1 :
34                            `DSP_PORT_P_WiDTH * (i / KERNEL_SIZE) - `DSP_PORT_P_WiDTH]
35                   : w_outDSP[(`DSP_PORT_P_WiDTH * i) - 1 :
36                              (`DSP_PORT_P_WiDTH * i) - `DSP_PORT_P_WiDTH]) // C input
37          );
38      end
39  endgenerate
```

Code Snippet 4.5: DSPs instantiation

Operations within each DSP are performed using the input values at ports A B and C. From code snippet 4.5 it is possible to noted that the input feature map values are shared by all units as described in the data-flow diagram 3.4. As for the filter weights, through the same diagram it is perceptible that they are distributed across all DSPs using the *i_Weight* register where input weight data is previously mapped. The C port input data source is dependent on the weight position. According to figure 3.6, a shift register is allocated between the set of DSPs that represents a row in the filter, therefore, the C port of the first DSP is connected to a shift register rather than the previous DSP's output port. Following the same strategy, all the output data ports are concatenated into a wire which will have certain positions connected to DSPs and others to shift registers to form the desired cascade scheme. Figure 4.2 presents the generated RTL schematic for a 3x3 filter. For each filter row, three DSPs are allocated together with a shift ram, as highlighted in the extension.

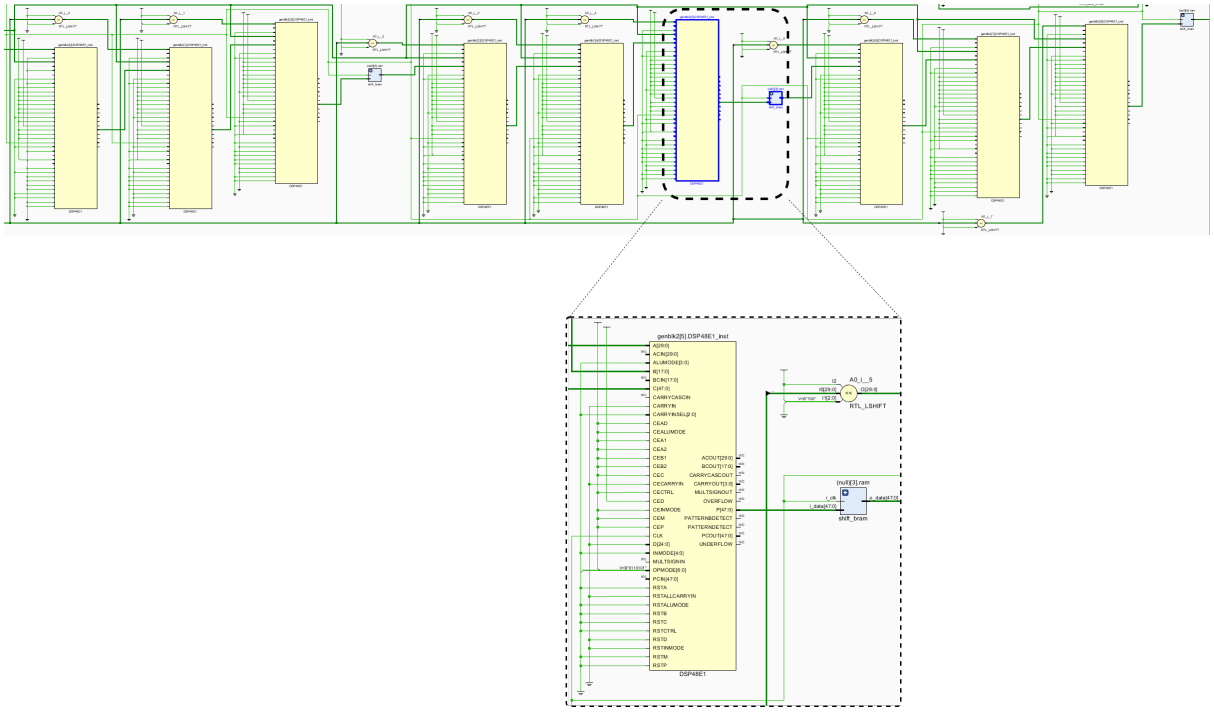


Figure 4.2: DSP schematic

4.1.4 ReLU and MaxPooling integration

In addition to convolution, there are other operations usually implemented in a CNN layer. The convolutional module implements, together with the convolution, the ReLU and Max Pooling operations. Although the ReLU operation is much more common, the Max Pooling is also available since it can be easily integrated bringing more functionalities to the module. The architecture design of the convolutional module presents the convolution and ReLU operations as the base for processing, while the use of Max Pooling is optional and should be specified by the user through the parameters described in the interface 4.1. The fact that the controller must be flexible for situations where the Max Pooling can or cannot be used demands for a mechanism to manage the module's output source.

The ReLU operation is always performed after a convolution is completed, so when the user specifies not to use the Max Pooling, the module's output source is directly connected to the output port of the ReLU block. On the contrary, if the user specifies the use of Max Pooling, the ReLU block's output is connected to the Max Pooling block's input port while the convolutional module's output is connected to the output port of the Max Pooling block. Code snippet 4.6 represents the data management described where the *w_o_data_ReLU* and *r_mp_result* are the ReLU and Max Pooling blocks' output, respectively, and the *o_conv_result* the module's output register. Regardless of whether Max Pooling is used or not, every clock cycle a new output value is processed, the *o_en* signal is set and the write operation in the

output memory is enabled.

Following the design architecture for the Max Pooling block presented in section 3.2.2, the number of instantiated blocks is not only dependent on the size of the input feature map, but also the number of allocated Processing Elements for each filter. The example shown in figure 3.7 illustrates the logic for one PE, however, when several PEs are used to perform a convolution, several sets of Max Pooling blocks are also allocated to operate in parallel. With the *for* cycle in code snippet 4.6 the output is built with one output value from each array of Max Pooling blocks. As each PE processes a different region of the feature map, the Max Pooling operation is also applied on more than one region at the same time, so when the operation is completed, one result from every region is written in the output memory at each clock cycle.

The controller manages the output data from each array of Max Pooling blocks, using the *r_mp_cnt* as a counter to monitor the module's sequence output. For each PE, an array of *OUT_SIZE/2* Max Pooling blocks are used, so when the operation is completed, the counter is reset. After the controller detects the counter reset, the values are stored in the output memory. The iteration ends when the *r_mp_cnt* counter reaches the *OUT_SIZE/2* value. Again, the controller waits for the *r_mp_out_rdy* signal to reset the counter and send new values to the memory.

```
always @(posedge i_clk) begin
    o_en <= 0;
    if(i_rst) begin
        o_conv_result   <= 0;
        r_mp_cnt        <= OUT_SIZE / 2;   // number of output values from MaxP
    end
    else if(w_o_ReLU && (MAXPOOL == 0)) begin   // MaxP disable, output from ReLU
        o_en            <= 1;
        o_conv_result   <= w_o_data_ReLU;
    end
    else if(r_mp_cnt < (OUT_SIZE / 2)) begin    // remaining values from MaxP
        o_en        <= 1;
        r_mp_cnt    <= r_mp_cnt + 1;

        for(i = 0; i < pe_filter; i = i + 1) begin  //  one MaxP block for each PE
            o_conv_result[i * `D_WIDTH +: `D_WIDTH]
                <= r_mp_result[i * (OUT_SIZE / 2) + r_mp_cnt];
        end
    end
    else if(r_mp_out_rdy) begin // MaxP ready, reset output counter
        r_mp_cnt <= 0;
    end
end
```

Code Snippet 4.6: ReLU and MaxP output management

4.2 Voting Block

The module was designed to perform voting scheme-based convolutions, however, even a sparse convolution should be modifiable according to the typical parameters of a convolution. The possibility for the user to apply different kernel sizes, or change the padding and stride, extends the scenarios where the voting scheme-based convolution can be adopted to process sparse data, consequently bringing more flexibility to the block. From the parameter specification made by the user, the module automatically infers the output feature map size that will be generated after the convolution is completed.

The interface specified through code snippet 4.7 has an extensive list of ports, which are divided into categories, according to their function in the interface. Considering the Voting block base design architecture presented in figure 3.15, the module is connected to several block memories that provide the access to the data that needs to be processed and a place where the output values can be stored. A total of five block memories are connected to the Voting block. Regarding the *IFM*, *Weights* and *Input data reference*, both three store relevant input data for processing, so for each one, a data port is needed to read the data itself, and another one to specify the memory address. The opposite is verified for the *Output data reference* memory, which is only used to write the non-null values' position on the output feature map. The *o_ref_w_en* is the signal used to enable the write operation mode, the write address is specified with *o_ref_w_addr* and the actual non-null value position data is transferred through the *o_ref_data* port.

In section 3.3, the voting scheme-based convolution was detailed. As illustrated in diagram 3.22, the output data needs to be fetched in order to complete each iteration. This procedure requires the data to be continuously read and written to the output memory at the same time. Following the design, a True Dual Port BRAM is the most suitable memory to support the combination of memory access operations needed in the Voting block, so the communication with the output memory is established using a total of six ports presented in the interface 4.7.

```
1  module VotingBlk #(
2      parameter   KERNEL_SIZE = `KERNEL_SIZE,
3      parameter   IFM_SIZE    = `IFM_SIZE,
4      parameter   PADDING     = `PADDING,
5      parameter   STRIDE      = `STRIDE,
6      parameter   IFMVALUES   = `IFMVALUES,
7      localparam  OFM_SIZE    = ((IFM_SIZE - KERNEL_SIZE + 2 * PADDING) / STRIDE) + 1
8  )(
9      input wire i_clk,
10     input wire i_rst,
11     input wire i_start,
12     // non-null input values addrs
```

```
13      input wire          [$clog2(IFM_SIZE**2)-1:0] i_data_ref_addr,
14      // convolution's input data
15      input wire signed [`A_DSP_WIDTH-1:0]          i_data,
16      input wire signed [`B_DSP_WIDTH-1:0]          i_weight,
17      input wire signed [`OUTPUT_DSP_WIDTH-1:0]     i_partoutvalue,
18      input wire signed [`OUTPUT_DSP_WIDTH-1:0]     i_partoutvalue_b,
19      // output addrs ports
20      output reg  [$clog2(IFM_SIZE**2):0]       o_nnvr_r_addr,
21      output reg  [$clog2(IFM_SIZE**2)-1:0]     o_ifm_r_addr,
22      output reg  [$clog2(OFM_SIZE**2)-1:0]     o_ofm_w_addr,
23      output wire [$clog2(OFM_SIZE**2)-1:0]     o_ofm_r_addr,
24      output reg  [$clog2(KERNEL_SIZE**2):0]    o_wght_r_addr,
25      output reg  [$clog2(OFM_SIZE**2)-1:0]     o_ref_w_addr,
26      // reference memory interface ports
27      output reg  [$clog2(OFM_SIZE**2)-1:0]     o_ref_data,
28      output reg  o_ref_w_en,
29      // non-null output values
30      output reg  [$clog2(OFM_SIZE**2)-1:0]     o_values,
31      // output memory data ports
32      output wire o_en,
33      output wire o_en_b,
34      output wire signed [`OUTPUT_DSP_WIDTH-1:0] o_data,
35      output wire signed [`OUTPUT_DSP_WIDTH-1:0] o_data_b,
36      output reg o_done
37  );
```

Code Snippet 4.7: Voting block interface

The output memory has two ports, and the Voting block uses both for read and write operations. Although the address for each of the ports is specified with the *o_ofm_r_addr* and *o_ofm_w_addr* registers, both can be used to define the address for the double write operation, as described in 3.23. As both ports are prepared to write two values at the same time in the memory, two data ports *o_data* and *o_data_b* are also allocated to support the double write operation, not compromising the block's output throughput.

The RTL schematic of the Voting block together with connections to the Block Rams is presented in figure 4.3. Signal *o_done* is set when all the processing is completed, and *o_values* indicates the number of non-null output values. For this particular case, a 4x4 IFM and a 3x3 filter was considered, resulting in a 2x2 OFM. Since the maximum number of non-null output values is four, a total of three bits were allocated for the *o_values* port.

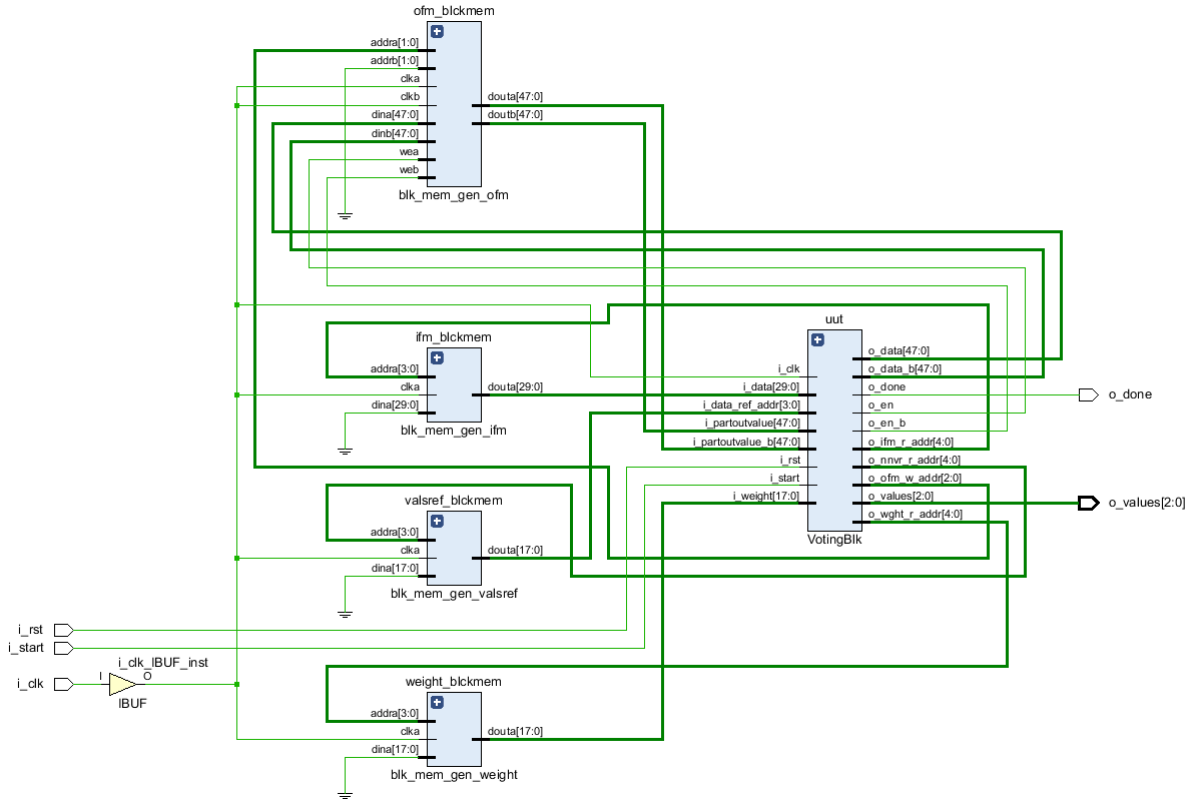


Figure 4.3: RTL schematic of Voting block connected to all memories

4.2.1 Kernel construction

The voting scheme-based convolution is a type of sparse convolution that processes only the input feature map regions with meaningful information. The characteristics of the voting convolution make it unique and can be distinguished from both traditional and other types of sparse convolutions. Although the submanifold convolution, presented in sub-section 2.4.2, is also only applied to regions of interest, the voting scheme-based convolution uses a different mechanism. The result of each multiplication is first added to the values stored in the corresponding output feature map region, before the results are written in the output memory.

The state machine implemented in the Voting block, described in figure 3.20, represents the different stages of data fetch and processing. As part of the data fetch stage, the filter weights are initially read from the input memory in the *Filter load* stage. Once a weight is read the controller loads it into an internal buffer where the filter will be stored and accessed from. The filter's weights are read from the memory only once and stored until the end of the processing, so no time is wasted reading the same filter each time a matrix multiplication is performed. As the number of weights that compose a filter is relatively small, the additional memory consumption within the Voting block will not compromise implementation

on a platform with limited memory resources.

According to the design, one of the voting scheme-based convolution's particularities is the use of the filter inverted. More specifically, the filter has to be inverted along the *x* and *y* axis to later be used in the convolution operation in the processing stage, as visually demonstrated in figure 2.20. Code snippet 4.8 presents the control over the weight memory read address and inversion of the filter weights in the internal buffer *r_filter*. Despite the memory positions being accessed in order the counter *r_weight_cnt* is decremented by one each time a new weight is read, and as a result the buffer is back-filled and the filter is reversely stored.

```
/* Weight memory read management */
always@(posedge i_clk) begin
    r_weight_en     <= 0;
    if(r_state == s_idle) begin
        o_wght_r_addr   <= 0;
    end
    else if(r_state == s_filter_load) begin
        if(o_wght_r_addr < KERNEL_SIZE**2)begin
            o_wght_r_addr   <= o_wght_r_addr + 1;
            r_weight_en     <= 1;
        end
    end
end

/* Store weights in reverse */
always@(posedge i_clk) begin
    if(r_state == s_idle) begin
        r_weight_cnt    <= KERNEL_SIZE**2 - 1;
    end
    else if(r_weight_en) begin
        r_filter[r_weight_cnt]  <= i_weight;
        r_weight_cnt            <= r_weight_cnt - 1;
    end
end
```

Code Snippet 4.8: Voting weights organization

4.2.2 Input references management

As discussed in sub-section 2.4, a dataset with a high level of sparsity can be characterized by having a small percentage of relevant data to process, since most of the data are null values. While in traditional mechanisms, input data processing is done sequentially for all the feature map values, sparse mechanisms iterate only through the non-null values, requiring the information about where these values are located

in the input feature map. As described in the design chapter, one of the requirements for the voting convolution is the reference to non-null values, as the lack of this information would require an intensive previous search for these values, resulting in efficiency loss.

From the non-null values' references, the Voting block's control unit is able to recognize the relevant data location and fetch all the non-null values from the input feature map. Code snippet 4.9 presents the section of the controller that is responsible for fetching the input data. The *o_nnvr_r_addr* register is connected to the input reference memory's read address port and is manipulated by the controller to sequentially read the next non-null value address in the input feature map. Since the values positions are stored consecutively in the reference memory, the register is incremented by one each time the current iteration ends so the next non-null values address is available to the controller in the next iteration.

Following the diagram presented in 3.23, the current iteration finishes after the output values are written into memory, and before the next iteration begins all read addresses should be updated to access the next input values. This process is performed between iterations, and it is intended to be as fast as possible, so no clock cycles are wasted before a new iteration starts. The read operation performed on the input reference memory returns the result to the *i_data_ref_addr* port and, at the same time, the current non-null value address is accessed through the *o_ifm_r_addr* port. Considering that each read operation requires two clock cycles to get the result, the *o_nnvr_r_addr* is always pointing to the next input value address so between iterations the address is ready and immediately loaded to the input data memory read port.

```
always@(posedge i_clk) begin
    if(i_rst || r_state == s_idle) begin
        o_nnvr_r_addr   <= 0;
        o_ifm_r_addr    <= i_data_ref_addr;   // load first non-null value
    end
    else if((r_state == s_filter_load) && (o_wght_r_addr >= KERNEL_SIZE**2)) begin
        o_nnvr_r_addr   <= o_nnvr_r_addr + 1; // read second non-null value addr
    end
    else if((r_state == s_process_data) &&
            (r_ref_index_it_shift == r_ref_shift_index) &&
            (r_ref_index_it == r_ref_addrs_index)) begin // current process complete

        o_nnvr_r_addr   <= o_nnvr_r_addr + 1; // read next non-null value addr
        o_ifm_r_addr    <= i_data_ref_addr;   // load value from previous addr
    end
end
```

Code Snippet 4.9: Input non-null values' reference management

4.2.3 Optimization engine

The pipeline processing described during the design chapter is an efficient mechanism well adapted to the voting scheme-based convolution's requirements. Whenever a new non-null input value is fetched from the data memory, a new iteration begins as the pipeline stages are again executed following the procedure described in diagram 3.20. According to the diagram, as the pipeline processing goes, the necessary data is fetched from the memories, the calculations are performed by the Processing Unit, and the results are written in the output memory. Despite the processing of all non-null input values follow the same processing stages, for each new value, the data that is sent to the Processing Unit needs to be prepared. For instance, both filter weights and output partial values addresses should be calculated to define which values should be read from the memory or which calculations can be ignored.

Following the sub-section 3.3.2 together with figure 3.21, the memory addresses of the output values **Z** and **P**, are calculated based on the input non-null value address from the input feature map, represented as *o_ifm_r_addr*. Code snippet 4.11 refers to the logic implementation where the input value address is used to determine the addresses of the *KERNEL_SIZE**2* values that are read from the output memory as well as the weights for the individual calculations. Each position of the kernel has a corresponding output value and the circumstance for every value needs to be evaluated by the controller to efficiently adjust the processing when possible. As commented in code snippet 4.10, strided convolutions are considered as a scenario where an additional verification is important, since the filter is only applied to some regions of the input feature map depending on the *STRIDE* specification made by the user.

```
always@(r_detect_pos) begin
    r_ref_shift_index   = 0;
    r_ref_addrs_index   = 0;

    for(i = 0; i <= KERNEL_SIZE**2 - 1; i = i + 1) begin // kernel positions
        r_last_d_addrs[i]   = 0; // clean and disable latches
        r_last_d_val[i]     = 0;
        r_last_w_addrs[i]   = 0;
        r_ref_d_addrs[i]    = 0;
        r_ref_w_addrs[i]    = 0;

        r_x = (o_ifm_r_addr / FM_SIZE + (KERNEL_SIZE/2))
                - ((KERNEL_SIZE**2 - i - 1) / KERNEL_SIZE); // window x coordinate
        r_y = (o_ifm_r_addr % FM_SIZE + (KERNEL_SIZE/2))
                - ((KERNEL_SIZE**2 - i - 1) % KERNEL_SIZE); // window y coordinate

        r_shift_val = 0;

```

```
19          if(((r_x >= (KERNEL_SIZE/2) - PADDING) && // output boundaries
20              (r_x <= FM_SIZE - (KERNEL_SIZE/2 +1) + PADDING) &&
21              (r_y >= (KERNEL_SIZE/2) - PADDING) &&
22              (r_y <= FM_SIZE - (KERNEL_SIZE/2 +1) + PADDING)) &&
23              ((r_x - (KERNEL_SIZE/2) + PADDING) % STRIDE == 0) && // strided conv
24              ((r_y - (KERNEL_SIZE/2) + PADDING) % STRIDE == 0) &&
25              (r_filter[i] != 0)) begin // null weights
26
27              /* optimization logic here */
28
29          end
30      end
31  end
```

Code Snippet 4.10: Output values address calculation

The controller is responsible for evaluating the conditions of each convolution and posteriorly adapt itself, so the processing is as efficient and appropriate as possible. The implementation of a controller which is aware of the optimization possibilities that may appear in the data, although it requires additional logic, can bring significant performance improvements to the Voting block processing. For instance, a filter with most null weights reduces the communication with the output memory and as a result, the processing time can also be significantly decreased. In addition, non-null values that are spatially close in the input feature map are widely exploited using the technique designed in 3.3.3, increasing the performance of the system without compromising the correct functioning of the block.

Figure 4.4 shows that, although two different input feature maps have the same level of sparsity, that does not mean that the non-null values are equally concentrated. Although the sparsity levels is criteria chosen to determine whether sparse convolutions are worth using, the impact of the optimization technique through data reuse between iterations is conditioned only by the concentration level of the non-null values in the feature map. For instance, according to figure 4.4, the processing of *IFM 0* is more likely to utilize the data reuse technique since the non-null values of *IFM 1* do not belong to the same regions on the feature map.

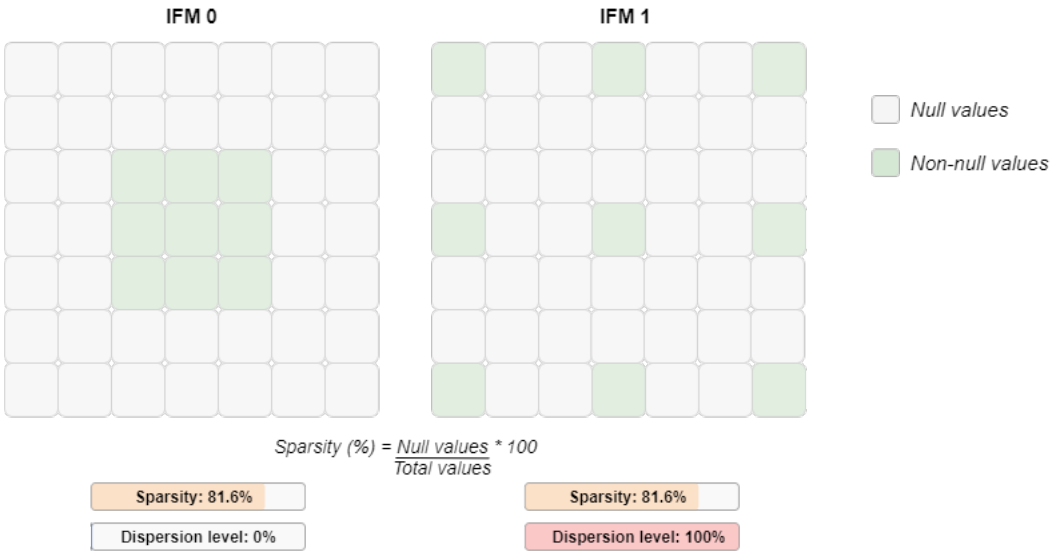


Figure 4.4: Sparsity vs Dispersion

Following the previous code snippet, the optimization logic presented in 4.11 is used to evaluate if the data needed for the current iteration matches the output of the previous one. Considering that during the pipeline processing each result is written in the output memory and also in the internal buffer, between iterations the controller has the information to verify if any data can be shared. For situations where the output values address from the last iterations are equal to the output partial values addresses of the current iteration, the controller should detect and divide the pipeline data into two different parts. For the data that will be reused in the current iteration, the actual values are stored in the *r_last_d_val* register while their addresses of the output memory are saved in the *r_last_d_addrs* register. The counter *r_shift_idx* is incremented each time a new value can be reused from the last iteration and is later used to monitor the double PE computing of the pipeline.

```
1  for(j=1; j <= KERNEL_SIZE**2; j = j + 1) begin // values needed for next iteration
2
3      if((r_last_addr_out[j-1] ==
4          ((r_x - (KERNEL_SIZE/2) + PADDING)/STRIDE) * OUT_SIZE +
5          ((r_y - (KERNEL_SIZE/2) + PADDING)/STRIDE)) && // search in temp buffer
6         (r_shift_val == 0)) begin // requested output is already in register
7
8          r_last_d_addrs[r_shift_idx]   = r_last_addr_out[j-1]; //  out value addr
9          r_last_d_val[r_shift_idx]     = r_last_d_out[j-1];    //  out value
10         r_last_w_addrs[r_shift_idx]   = i;                    //  weights
11         r_shift_idx                   = r_shift_idx + 1;      //  cnt reuse values
12         r_shift_val                   = 1;                    //  reuse value
13     end
14 end
15 if(r_shift_val == 0) begin // fetch the value from the output memory
16     r_ref_d_addrs[r_addrs_idx] =
```

```
17                          ((r_x - (KERNEL_SIZE/2) + PADDING)/STRIDE) * OUT_SIZE +
18                          ((r_y - (KERNEL_SIZE/2) + PADDING)/STRIDE); // out mem addrs
19
20      r_ref_w_addrs[r_addrs_idx] = i; // filter weights buffer addr
21      r_addrs_idx = r_addrs_idx + 1;  // how many values need to be read from outmem
22  end
```

Code Snippet 4.11: Data reuse mechanism

4.2.4 Double computing

The double computing technique, discussed in sub-section 3.3.3, was designed for scenarios when the temporary buffer contains part of the data needed for the current pipeline processing. When the data is already available inside the Voting block to perform the calculations, the controller takes the opportunity to activate the two processing elements and increase the Voting block's throughput. By activating the double computing in the Processing Unit, two values can be outputted at the same time and written in the output memory simultaneously. As mentioned in 3.3.3, the double write operation on the memory is possible due to the functionalities of the True Dual Port BRAM, which allows the two ports to work both with writing and reading operations. Although the ability to switch the operation of each port is important to support the double PE computation mechanism, it can still be considered a limitation imposed on the Voting block architecture, as more processing elements could be allocated if the memory interface had more available ports to store the output results.

The management required to distribute the data across the Processing Unit's elements is done by the controller which must verify how many calculations are needed to complete the pipeline processing. After the data organization and the shift technique integration, described previously with code snippets 4.10 and 4.11, the controller mechanism has to define whether the Process Unit will activate both DSPs or only one. Each instruction launched during the pipeline processing is under the responsibility of the block's control unit and along with the data, it must active either the single or double computing mode. The example described in 3.23 shows that in the first three pipeline instructions the Processing Unit works as a single computing mode, however, the last six iterations are grouped in pairs and the controller enables the double computing mode to increase the throughput.

Code snippet 4.12 refers to the decision block implemented in the controller, not only to manage the Processing Unit's operation mode, but also the partial output values fed into the DSPs from the internal buffer. Signals *r_PE_en1* and *r_PE_en2* are both connected to each processing element and setting these signals indicates that the corresponding element is being activated. In-parallel, the output values from the

last iterations are stored in the *r_last_d_val* register and loaded to the PEs' input ports *i_partoutvalue1* and *i_partoutvalue2*.

```
always@(posedge i_clk) begin
    r_idx_it    <= 0; // index to monitor the PU data
    r_PE_en1    <= 0;
    r_PE_en2    <= 0;

    if((r_state == s_process_data) && (r_idx_it < r_shift_idx)) begin
        if (((r_shift_idx - r_idx_it) / 2) > 0) begin // double computing

            r_idx_it    <= r_idx_it + 2; // two values at once

            r_PE_en1    <= 1; // enable both PEs
            r_PE_en2    <= 1;

            i_partoutvalue1 <= r_last_d_val[r_idx_it];  // output partial values
            i_partoutvalue2 <= r_last_d_val[r_idx_it+1];
        end
        else begin  // single computing
            r_idx_it        <= r_idx_it + 1;
            r_PE_en1        <= 1;
            i_partoutvalue1 <= r_last_d_val[r_idx_it];
        end
    end
end
```

Code Snippet 4.12: Double PE computing

The RTL schematic resulting from the instantiation of two processing elements is shown in figure 4.5. Since for each pipeline iteration only one non-null input value is being processed, the same value is shared between the two PEs. As for the weight and partial output values, each PE is fed by separated registers. Although the double computing mode is only activated when data reuse is possible, the output data port of both PEs needs to be connected to the output memory in order to support the double write operation, as illustrated in figure 3.15.

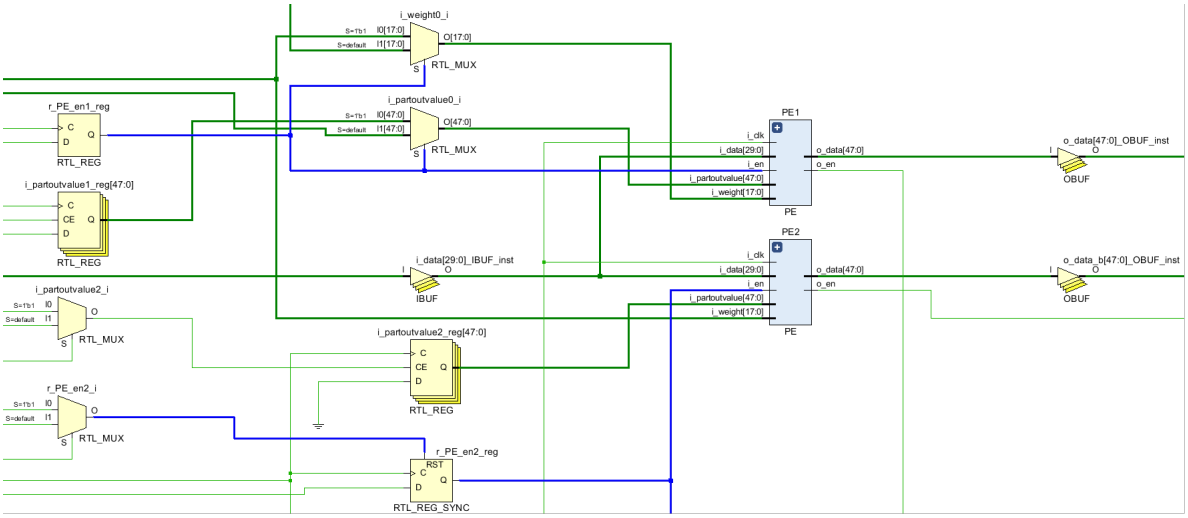


Figure 4.5: Double PE RTL schematic

4.2.5 Data reuse

During the pipeline processing, the output values from the calculation stage are written in the output memory and also stored in the internal buffer managed by the control unit. Together with the output values, the corresponding addresses are also registered on another buffer so the controller can verify if any data can be reused for the next iteration. The internal buffer designed to temporarily save the output data of each iteration is used as a FIFO with a limited storage capacity. The data reuse technique is only performed when the values are horizontally close on the input feature map. As suggested in figure 3.22, values that are spatially close in the x-axis can be exploited by integrating the reuse mechanism, nonetheless, the spatial proximity on the y axis can also be leveraged depending on the processing order of the non-null values.

Assuming that the processing of the input values is performed line by line, the values horizontally close are processed consecutively, and therefore, using the output of one iteration for the input of the next one is simpler. On the other hand, if the processing is performed column by column, sharing data between values vertically close in the feature map would be the most appropriate technique. In order for the technique to be applied to both vertically and horizontally close values, it is required that the buffer has a higher capacity so that values belonging to consecutive rows or columns can have data shared through the buffer. Although using the output data from previous pipeline processing iterations is beneficial to avoid spent processing time in the communication with the memory outside the block, it is important to evaluate that the memory consumed by the internal buffers (actual values plus related addresses) do not compromise the Voting block implementation on a platform with memory constraints.

Code snippet 4.13 refers to the controller implementation responsible to register all the output values

that result from the calculation stage of the pipeline processing. The internal buffers *r_last_d_out* and *r_last_addr_out* work as a FIFO, where the output values and their memory addresses are stored, respectively. When the double computation is activated in the Processing Unit, the signal *o_en_b* indicates that two new results need to be sent to the output memory and registered in the buffer. Regardless of whether one or two values are outputted from the Processing Unit at the same time, it is always necessary to shift the buffer, so the oldest values are replaced by the newest ones.

```
always@(posedge i_clk) begin
    if(o_en_b) begin // double computation output

        for(i = (KERNEL_SIZE**2)-1; i > 1; i=i-1) begin // fifo shift
            r_last_d_out[i]      <= r_last_d_out[i-2];
            r_last_addr_out[i]   <= r_last_addr_out[i-2];
        end

        r_last_d_out[0]      <= o_data;       // current output values
        r_last_d_out[1]      <= o_data_b;
        r_last_addr_out[0]   <= o_ofm_w_addr; // current values addrs
        r_last_addr_out[1]   <= o_ofm_r_addr;
        r_out_cnt            <= r_out_cnt + 2;

    end
    else if(o_en) begin // single computation output

        for(i = (KERNEL_SIZE**2)-1; i > 0; i=i-1) begin
            r_last_d_out[i]      <= r_last_d_out[i-1];
            r_last_addr_out[i]   <= r_last_addr_out[i-1];
        end

        r_last_d_out[0]      <= o_data;
        r_last_addr_out[0]   <= o_ofm_w_addr;
        r_out_cnt            <= r_out_cnt + 1;
    end
    else begin
        r_out_cnt <= 0;
    end
end
```

Code Snippet 4.13: Processing Unit's output record

4.2.6 Output references generation

To implement several convolutional layers with the Voting block, each one must provide all the needed information for the next one to be able to operate without restrictions. Considering that the position where

the non-null values are located is a critical requirement of the Voting block, each instantiated block must record the non-null values positions in the output memory. Code snippet 4.14 shows how the output values' addresses registration is performed by the Control Unit, each time a new value is outputted from the processing pipeline. Signal *o_en* indicates that a new result from the calculation stage is ready to be written in the output memory, but also its address needs to be sent to the reference output memory, where all the output values' positions in the feature map are located. Considering the duplication problem that may occur during the address registration, as described in 3.27, a FIFO is adopted to prevent that the same output value address is written in the output memory more than once.

The *r_out_addrs* register is used to log the most recent addresses as a mechanism to evaluate if in close regions data can shared. The register size is specified with a capacity to store *KERNEL_SIZE* output values, since it is the minimum size required to avoid any repetition of addresses. Whereas the processing of values can be done either row-by-row or column-by-column, the number of rows or columns that are involved in each matrix multiplication corresponds to the kernel size. The loop implemented is responsible to check if the current output value address is already stored in the FIFO, and if not, the signal *r_bit1S* indicates that the address has to be written in memory.

```
reg [$clog2(OUT_SIZE**2)*(KERNEL_SIZE*OUT_SIZE)-1:0] r_out_addrs; // addrs FIFO
reg [$clog2(KERNEL_SIZE*OUT_SIZE)-1:0]               r_indx;
reg [$clog2(OUT_SIZE**2)-1:0]                        r_ofm_addr_cpy; // aux addr
reg r_bit1S;

always@(posedge i_clk) begin
    r_bit1S <= 0;

    if(i_rst) begin
        r_ofm_addr_cpy    <= 0;
    end
    else if(o_en) begin // new value from the Processing Unit
        r_ofm_addr_cpy <= o_ofm_w_addr; // address backup
        r_bit1S        <= 1;

        // check if current output value address is already registered
        for (r_indx=0; r_indx<=KERNEL_SIZE*OUT_SIZE -1; r_indx=r_indx+1) begin
            if(r_out_addrs[r_indx * $clog2(OUT_SIZE**2) +: $clog2(OUT_SIZE**2)]
                    == o_ofm_w_addr) begin
                r_bit1S <= 0; // already in the out ref memory
            end
        end
    end
```

```
24 end
```

Code Snippet 4.14: Duplicate references management

Together with the non-null output values positions, each Voting block should also output the number of non-null values stored in the memory, through the port *o_values* presented in the interface 4.7. The number of output values is considered another requirement of the Voting block, since it is important to inform the next block of how many iterations are needed to complete the convolution. Following the verification performed in the previous code snippet 4.14, when the signal *r_bit1S* indicates that a new output value is not yet registered in the output reference memory, the counter is incremented, and the value address is written both in the FIFO and the memory.

Although the management of the output address registration is a simple process assumed by the Control Unit, additional logic is required. Depending on the output feature map size and data width, the amount of logic that needs to be allocated to control the FIFO may be unfeasible for an efficient implementation. Keeping in mind that the output references generation is only a functionality useful if another Voting block is used next, if this is not the case, all the additional logic could be disabled without any interference in the functioning of the block.

```
1  always@(posedge i_clk) begin
2      o_ref_w_en          <= 0;
3
4      if(i_rst) begin
5          r_out_addrs     <= 0;
6          o_ref_w_addr    <= 0;
7          o_values        <= 0;
8      end
9      else if(r_bit1S) begin // new value addr to register
10         r_out_addrs  <= {r_out_addrs[$clog2(OUT_SIZE**2)*(KERNEL_SIZE*OUT_SIZE)-2:0]
11                                 , r_ofm_addr_cpy}; // FIFO shift
12         o_ref_data   <= r_ofm_addr_cpy;           // addr backup
13         o_ref_w_addr <= o_ref_w_addr + 1;         // out ref mem write addr
14         o_values     <= o_values +1;              // count non-null values
15         o_ref_w_en   <= 1;                        // write op
16     end
17 end
```

Code Snippet 4.15: Output references registration

Chapter 5: Tests and results

This chapter presents the validation of the hardware implementation of both convolution module 4.1 and voting block 4.2, presented and discussed in the previous chapter. This section aims at validating the design of these solutions by testing the implementations of convolution block and voting block with real data.

5.1 Convolutional Module

The evaluation of the correct functioning of the several features of these models follows a validation process containing several steps. To start, the module was implemented in hardware platform using Vivado, where it was possible to assess the resources consumption according to the user's configuration after the generation of the design bitstream. Using the implementation detailed in section 4.1, several images are adopted to test the module and then determine whether the convolutions are correctly applied throughout the input data. To determine the novel model feature regarding their customization capability, according to the user's specifications and resources restrictions imposed, several tests were suggested. Finally, the module was integrated with the PointPillars model, as a case study, where a detailed analysis regarding the effect on the detection precision was carried out.

5.1.1 Functional validation

From the code snippets presented throughout section 4.1, Vivado was used to generate the bitstream for the target FPGA platform, the Xilinx Zynq Z-7010 [60]. Despite the limitations of this board, it has enough resources to validate the work developed. Furthermore, it is the board that is most available to the author of this thesis, who has purchased one in the past. Figure 5.1 represents the Convolutional Module IP created, with the parameters' fields open to the user configuration. After the user specifies the module's parameters, a new bitstream is generated together with the resource's utilization description.

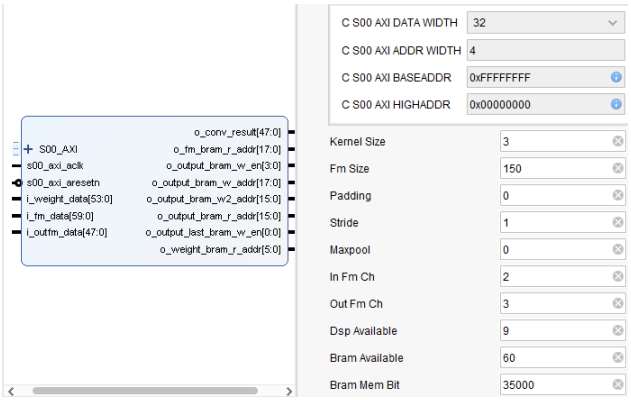


Figure 5.1: Convolutional Module IP

From the parameters specified in figure 5.1, the corresponding resources usage is presented in table 5.1. According to the resources available in the Zynq Z-7010 board, 80 DSPs are provided, however, only 9 were specified for the processing in order to later evaluate the management made by the controller. As the board only has a total of 2.1 Mb of Block Rams, 60 blocks, each with 35Kb, were also defined in the interface. From the utilization table, it can be noticed that only 9 DSPs are actually used by the module, while the utilization of Block Rams is 93.33%. Considering a 150x150 FM and a 3x3 filter, both LUTs and Flip Flops consumption percentage is reasonably low, despite the limited resources of the board used, which also proves the efficient hardware implementation of the convolutional module. As for the on-chip power, a total of 1.8W was reported.

Resource	Utilization	Available	Utilization (%)
LUT	6621	17600	37.62
LUTRAM	1269	6000	21.15
FF	8190	35200	23.27
BRAM	56	60	93.33
DSP	9	80	11.25

Table 5.1: Convolutional Module consumption results obtained from the Vivado Report Utilization tool

For the functional validation, a block design was created to provide the input data for the module and to store the processing results. From the design presented in figure 5.2, three Block Rams are distinguished corresponding to the memory allocated for the weights, the input feature map, and also the output feature map. As a first iteration, the input data was initially stored into an SD card and sent to the DDR memory. Using the Central Direct Memory Access (CDMA), the data from the DDR is transferred to the Weight and Input Block Rams, where the input feature map and the weights is stored, respectively. After that, a signal is sent by the Processing System to the Convolutional Module informing that all the input data is ready to be accessed so the processing can start. Depending on the number of iterations required to complete

the processing and the memory limitations, the Block Ram used to store the input feature map may need to be filled with new data between iterations. As referenced in the module's interface 4.1, every time an iteration ends and a new input feature map is required to continue the processing, *o_load_ifm* signal is used to request a new feature map to be stored in the Block Ram.

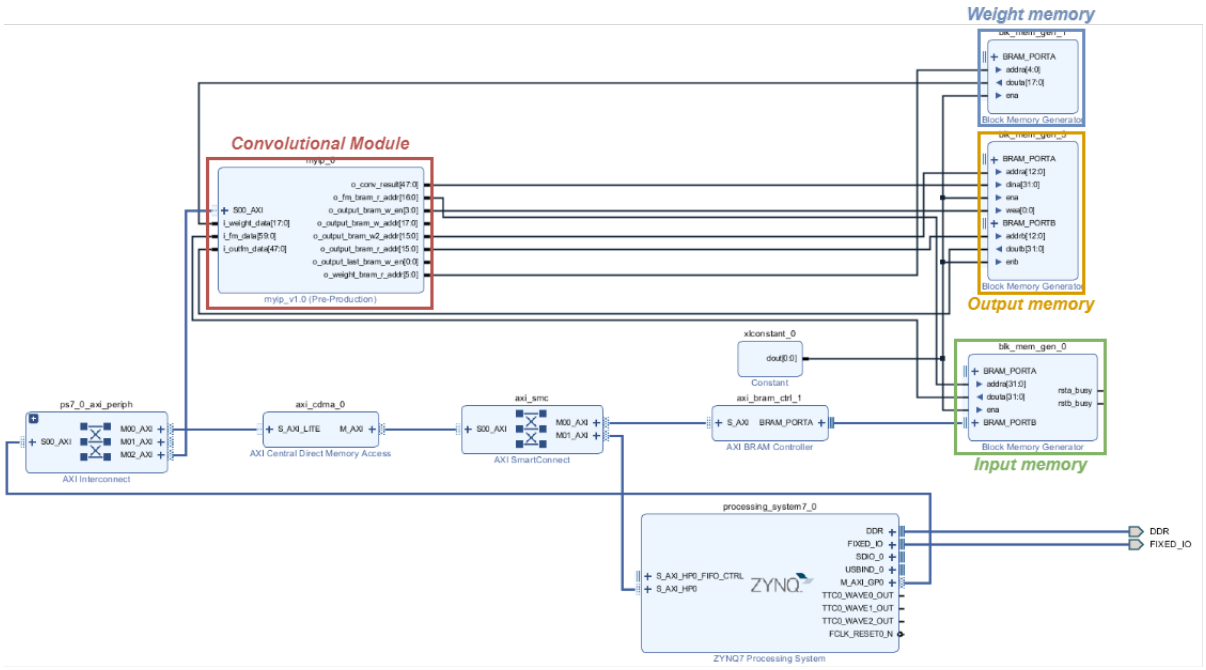


Figure 5.2: Block design used for the Convolutional Module validation

Since the Convolution Module was developed mainly to perform dense convolutions, the first tests were executed on RGB images. Among many types of filters, a few ones were selected to validate the module. Figure 5.3 is related to one of the tests, with the application of a 3x3 sharpen filter to an image of a car. As an RBG image is composed of three channels, the filter was applied in all the channels and concatenated in the end. From the image result, it is possible to note that the edges were sharpened throughout the entire image, which indicates that the convolution was correctly performed by the module. Besides the visual aspect of the image pixels, the output image dimensions also suggest that the filter was applied as intended. Although it is not visually noticeable from the figure, with the *padding* and *stride* specified with values zero and one, respectively, the output dimensions for a 3x3 filter is reduced to: *150 - 3 + 1 = 148*.

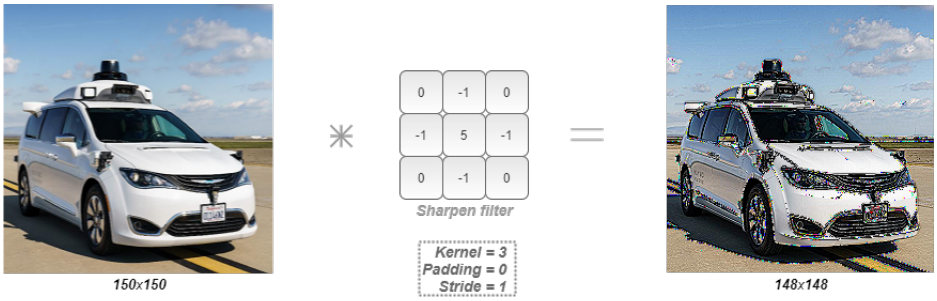


Figure 5.3: Convolution result using a sharpen filter

As in the previous test the padding and stride were specified with the default values, the next test was created to validate the execution of a strided convolution together with the use of padding. Excessive use of padding was chosen not only to test if the Convolution Module still operates correctly but also to be visually perceptible in the output image. Figure 5.4 presents the output image together with the parameter values specified for the convolution, namely: the kernel size, padding, and stride. This test was performed using the same input image from the previous test, however, the output images are quite different from each other. A 3x3 identity filter was randomly selected for this convolution, and the pixels of the output image prove that the image has only changed its dimensions compared to the input image. Once again, following equation presented in 2.14, the output image size can be calculated as: *((150 - 3 + 2 * 10) / 2) + 1 = 84*. Considering that a padding of ten was used in this convolution, the effect on the output image is perfectly recognizable. It is important to note that when the specified value for the padding parameter is different from zero, then zero values are introduced around the input image, which justifies the black boarder in the output.

Figure 5.4: Convolution result using an identity filter

5.1.2 Resource-aware performance

In addition to the module operation validation tests, there is another category of tests executed to determine the module flexibility and scalability. The Convolutional Module performance in this context

was measured considering the ability to adjust according to user specifications and resource constraints that may exist. A test was performed to evaluate the controller behavior when the memory available did not allow more than one filter to be applied at the same time. Figure 5.5 shows the parameter values, specifying a layer with three filters and two input feature maps. Meanwhile, the module's available memory was limited so that only one filter could be applied to an input feature map at a time.

The simplified simulation window 5.5 presents the iterations assumed by the controller as each filter is being convolved with one input feature map at a time. Despite the restrictions imposed on the module's operation, the controller was able to adapt and modify the processing flow into several iterations and sub-iterations, identified with the orange and blue circles, respectively. Since for each filter being applied, a block memory is required to store the output, each iteration refers to one filter and the sub-iteration is associated with the input feature map. Through the simulation window, it is possible to analyze that after two sub-iterations are completed (filter was applied to all input channels), a new iteration starts, meaning that the next filter will also be applied on both input feature maps.

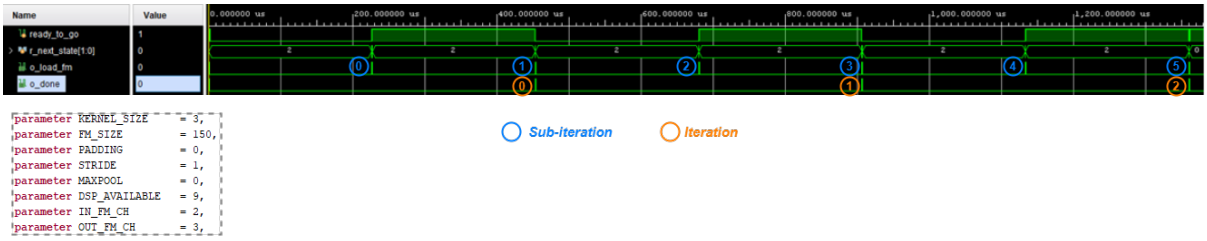


Figure 5.5: Processing iterations validation

In addition to the ability of splitting the processing into several iterations whenever is required, it is also desirable that, when no constraint is applied, the best performance solution is obtained from the developed module. As the management of the number of filters applied simultaneously was analyzed before, another way to enhance its performance is to increase the level of parallelism in each sub-iteration. The processing parallelism inside the Convolutional Module, as detailed in sections 3.2 and 4.1, is achieved through the allocation of several PEs to process different input regions at the same time. The fact that several regions are being processed simultaneously means that the time needed to complete a convolution is reduced.

The graph presented in figure 5.6 indicates the relation between the execution time and the number of instantiated PEs, for the processing of a 512 x 512 feature map with a 3 x 3 filter. Although it is possible to deduce that with more PEs used the execution time is reduced, the greater reduction percentage is observed in the first set of PEs. With a filter of size three, the addition of one PE requires a total of nine DSPs, which makes the allocation of more PEs to exponentially increase the number of DSPs consumed.

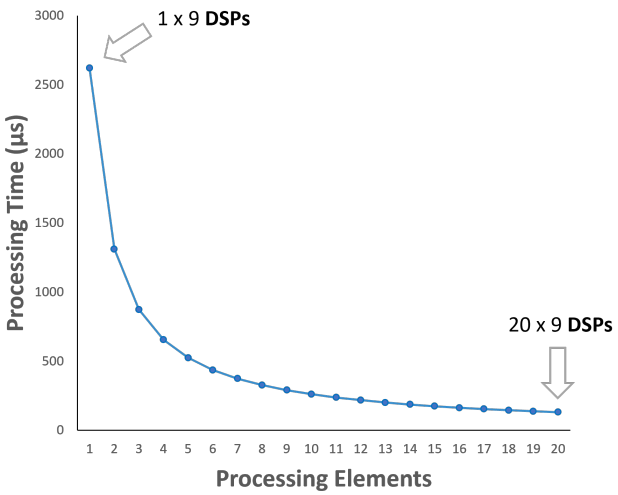


Figure 5.6: Effect of parallelism in the processing time consumption

Combining the two parallelism mechanisms implemented in the Convolutional Module, a performance test can also be executed to analyze the gains. While the increasing number of parallel filters reduces the number of iterations required to complete the processing, the amount of PEs is also important to complete a convolution faster. Following the same 512 x 512 input feature map and the 3 x 3 filter, the time consumed by the module was evaluated, while also increasing the available memory to enable more filters to be applied at the same time. On the other hand, the number of available PEs was fixed at twenty to validate the distribution of PEs by the controller.

Figure 5.7 presents the graph with the processing time results obtained from each test, with the indication of how many PEs were allocated. As the number of filters applied at the same time reduces the number of iterations, the gray line indicates the evolution of processing time while the memory available increases. Considering that in this particular test, a total of ten filters needs to be applied on the input data, the lowest time is achieved when 120Mb of memory is available, enabling all filters to be applied at once. The orange line specifies the real results obtained during the tests due to the management done by the controller regarding the number of PEs instantiated for each filter.

Since the processing of a 512 x 512 feature map with only one PE takes around 2621μs, an approximation to the time results from the graph's orange line can be made using the formula: *(2621 / num_PEs * num_iterations)*μs. Although the amount of memory is important for the parallel application of filters, the reality is that the amount of memory and the number of PEs complement each other. As 12Mb of memory only allows one output feature map to be stored, only one filter is used each time, meaning that all PEs can be allocated for that filter. On the other hand, 120Mb of memory is enough to store all the output data, so all the filters can be applied at once and two PEs are assigned to each one. Besides that, since

the controller was developed to make the iterations homogeneous, different amounts of available memory are associated with the same number of filters and PEs allocated for the processing. This mechanism was adopted to reduce the hardware complexity since for each iteration the amount of filters actually used is always the same, as discussed in sub-section 3.2.5.

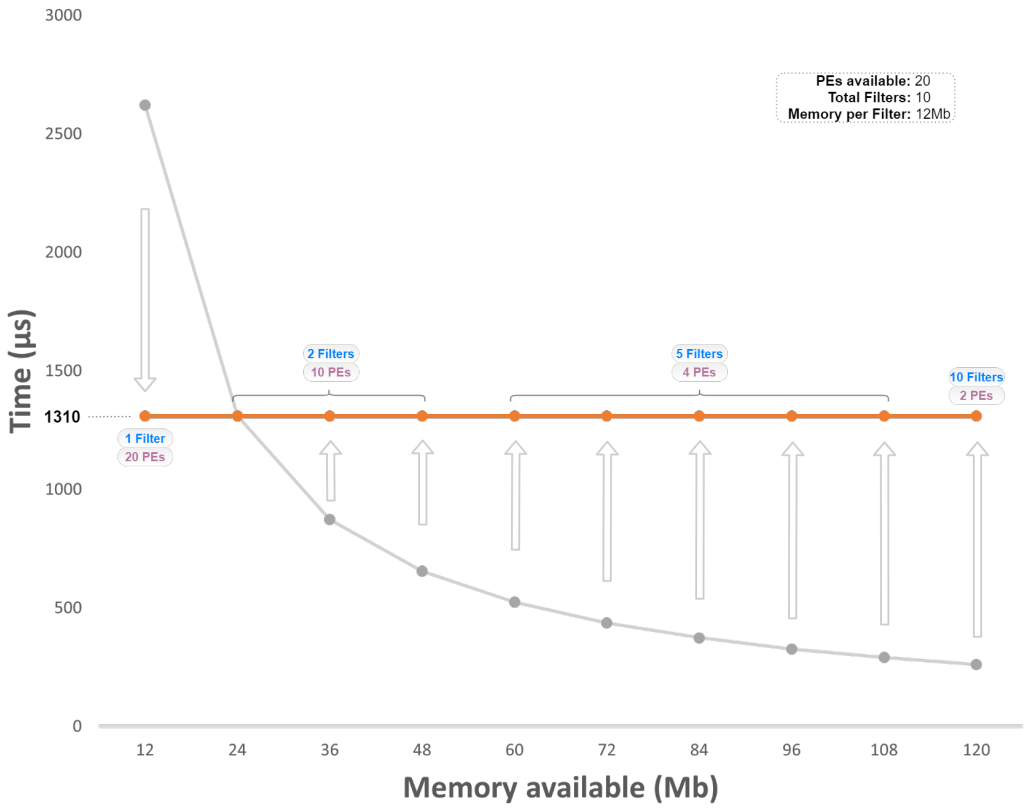


Figure 5.7: Time consumption for different combinations of filters and number of PEs

5.1.3 PointPillars

As a case study for the validation of the developed Convolutional Module, the 3D model for object detection in point clouds selected was the PointPillars. According to sub-section 2.2.1, the model is composed of three stages: the PFN, BackBone and SSD. As the PointPillars is a Deep Learning-based model, different convolutional layers on both stages can be used for the module integration and validation. In this particular validation test, the Convolutional Module's performance is evaluated by implementing the two backbone layers of the PointPillars, using the developed module. After the integration, the model's performance in the hybrid version is analyzed by comparing the precision of the results from the software version.

The last two convolutional layers of the PointPillars backbone stage were chosen. Although the backbone is composed of three blocks and a total of sixteen layers could be used to validate the module, the

last two were selected to simplify module integration. The backbone is identified as the stage where the largest computational percentage of the model is located, and as described in 2.2.1, it is composed of a convolutional neural network together with three deconvolution blocks and one block of concatenation in the end. As shown in figure 5.8, the integration can be represented with the output of *Conv3* layer connected to the first module instance, which is the one connected to the input of the second instance. Finally, the output of the second instance is connected to the *deconv3* block, as the software implementation of both *Conv4* and *Conv5* were cut during the validation test.

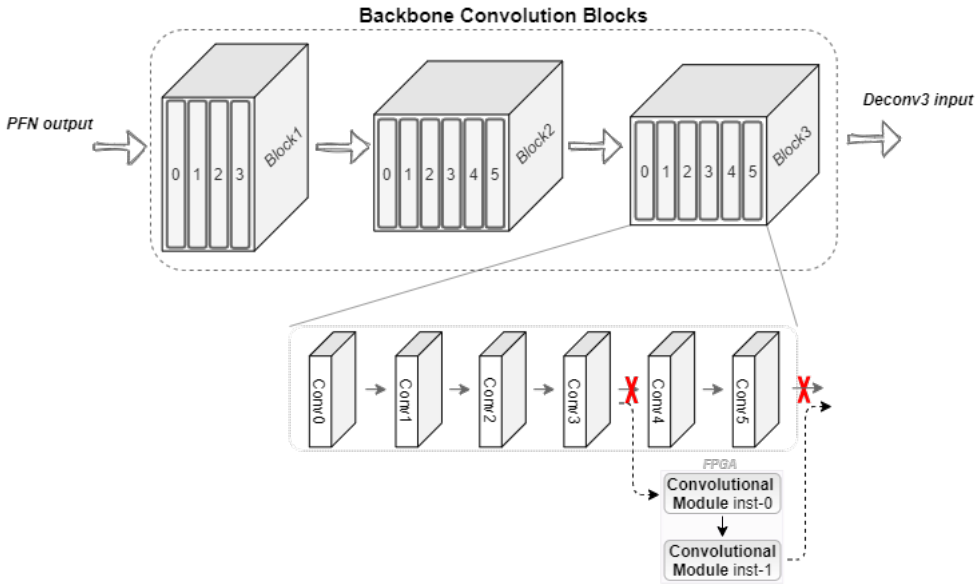
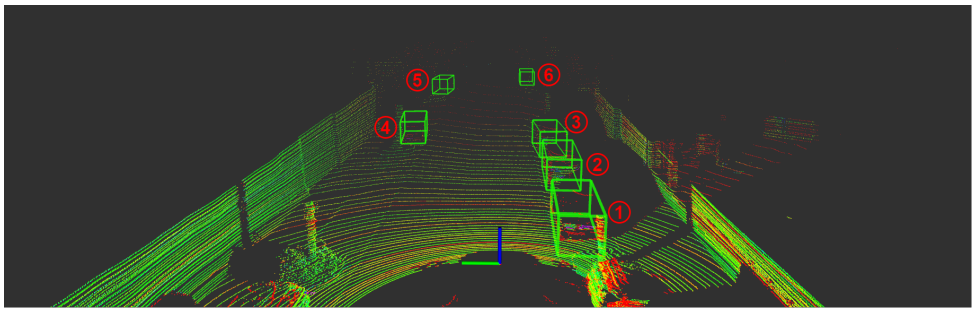


Figure 5.8: Convolutional Module integration in the PointPillars model

To create the same conditions for both the software and hybrid versions, the same frame was chosen from the Kitti dataset. Since before the validation, the software version of PointPillars was running on a GPU, the detection results were initially registered for later comparison. Figure 5.9 presents the randomly selected frame from the Kitti archives [61], and the corresponding point cloud with the bounding boxes displayed for each object detected in the scene. As can be noted, a total of six objects were detected, which in this particular case, all refer to parked cars.

(a) Frame from Kitti dataset [61]



(b) Pointcloud resulting from the LiDAR scan

Figure 5.9: Detections from the software version of PointPillars

Once again, this analysis is important to later compare with the results obtained from the integration with the Convolutional Module and evaluate the scores obtained. The scores of each detection, from the output of the PointPillars model in the software-only version were registered. Table 5.2 presents the association between the bounding box from figure 5.9b and the corresponding detection score. Through the analysis of both, it is noticeable that for this case, the farther the object is from the LiDAR sensor, the lower the score registered by the model. This is further evidenced through the distance specification presented in the middle column of the table. While the bounding box identified with number one is only 4.42 meters away from the sensor, the bounding box number six is at a distance of 48.08 meters.

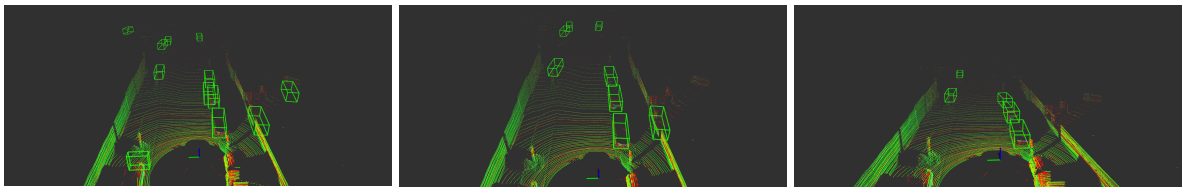
The difference in the distances between the closest and the furthest object is about 43 meters, resulting in scores of 0.954 and 0.609, respectively. Although the object identified with bounding box six presents a score of only 0.609, it is still detected by the model. Since in the software version, the model was configured with a score threshold of 0.5, all detections with a score value above that threshold appear in the output.

Bounding Box ID	Distance from the LiDAR sensor (meters)	Score
1	4.42	0.954
2	11.24	0.906
3	17.36	0.847
4	21.85	0.839
5	44.78	0.692
6	48.08	0.609

Table 5.2: Server's detections score for the bounding boxes identified in sub-figure 5.9b

Using the same input frame from the software version analysis 5.9, the hybrid version was executed, and the results are presented in figure 5.10. Although both 5.10b and 5.10c figures refer to the processing of the same frame, the number of bounding boxes displayed are different due to the score threshold specified for the model. Compared to the detections obtained in the software version, in the hybrid version the scores are a little higher in certain situations, resulting in false positive detections for the same threshold value. In the sequence, the threshold values were increased in order to remove the additional detections that are incorrect, resulting in similar detections for both versions.

Although the two screenshots were taken from different angles, it is possible to notice that the detections present in image 5.10c are similar to the results of the software version 5.9, with the exception that the bounding box relative to the furthest car does not appear, thus being counted as a false negative. In this test case, the threshold was purposely increased to exclude the object with the lowest score value, as it is also discriminated in table 5.3.



(a) Detections with a score threshold of 0.5 (b) Detections with a score threshold of 0.6 (c) Detections with a score threshold of 0.75

Figure 5.10: Detection results with the implementation of the last two layers of Block 3 using the Convolutional Module

The comparison of results obtained from the software-only version and the hybrid version can be analyzed in table 5.3. Focusing once again on the six detections specified in the software version 5.2, with the integration of two instances of the developed Convolutional Module, the scores for each of the bounding boxes are also detailed. Regarding the SW + HW version, two different tests were executed, being the *Conv5* layer the first one replaced by the module. After noting the results, both *Conv4* and

Conv5 were substituted by two instances, as presented in figure 5.8. The results from the SW version show the tendency for more distant objects to have a lower score value, however, with the hybrid version it is possible to observe a score increase pattern of these same objects as more layers are implemented with the module. On the other hand, detections closer to the sensor tend to lower the score value.

The detections from the software 5.9b and hybrid 5.10 versions are visually similar, and a certain difference is verified when comparing the score values themselves. The deviation of results can be explained both by the loss of information during the quantization of both feature map and weights values, and the lower resolution of operations performed inside the DSPs. Since the data width used in software was 32-bit floating-point, when the weight values are converted to an 8-bit and the feature map values to 16-bit fixed-point formats, the representation of the values has a reduced resolution. In addition, the operations within the DSPs, together with the final dequantization process, causes even more information loss and the difference in scores, relative to the software version, becomes accentuated as more layers are implemented in hardware.

Bounding Box ID	**SW version scores (threshold 0.5)**	**Hybrid version scores (threshold 0.75)**		
		Conv5	*Conv4 + Conv5*	*Score decrease vs SW version (%)*
1	0.954	0.917	0.841	11.3
2	0.906	0.857	0.821	8.5
3	0.847	0.842	0.817	3.0
4	0.839	0.806	0.815	2.4
5	0.692	0.788	0.763	-7.1
6	0.609	0.661	0.744	-13.5

Table 5.3: Scores comparison between software and hybrid versions

5.2 Voting Block

In order to validate the Voting block implemented, a set of tests were built, which were further divided in three major categories. The first category corresponds to the group of tests that was intended to verify if the block's operation was logically correct, using small feature maps. This initial test case is useful not only to quickly evaluate whether the block is working correctly but also to rectify possible implementation flaws easily. The second category had the purpose of evaluating the block's performance. Following an initially evaluation, the tests were extended to measure the performance of the Voting block under several circumstances. From the block's behavior along the tests' conditions created, it was possible to differentiate the potential of the voting scheme-based convolution compared to traditional solutions on real case scenarios. Finally, the Voting block was integrated with the PointPillars model as a case study, where the integration in certain layers for sparse data processing was analyzed.

5.2.1 Functional validation

The validation performed on the Voting block at an early stage has simple characteristics and is aimed at testing whether the block hardware implementation is operating as intended. Once again, using Vivado, a bitstream was generated in order to check if no errors were made while describing the hardware, as shown in section 4.2. Figure 5.4 presents the table with the resource-consumption obtained for the Zynq Z-7010 board. As the block was developed only to perform one voting scheme-based convolution, the resources consumed are quite low when comparing with the consumption level of the Convolutional Module, presented in table 5.1. Figure 5.11 refers to the first validation test, with the specification of a convolution between a 4x4 IFM and a 3x3 filter. As a result of this simple test case, the utilization of both LUTs and FFs is 13.43% and 12.55%, respectively. Regarding the energy consumption, the total on-chip power reported was approximately 0.2W.

Resource	Utilization	Available	Utilization (%)
LUT	2364	17600	13.43
FF	4416	35200	12.55
BRAM	1	60	1.67
DSP	2	80	2.50
IO	19	100	19.00

Table 5.4: Voting block consumption results obtained from the Vivado Report Utilization tool

After the bitstream was generated and loaded into the FPGA, the first validation test was started. Following the voting requirements, together with the input feature map values, the non-null input values

positions are also loaded into the memory. It is possible to note that the filter is read into the internal array *r_filter* in a reverse format, which is in accordance with the implementation described in code snippet 4.8. From the input feature map values, the sparsity level obtained is only 0.3125, since the ratio between the number of zero values and the number of non-zero values in the matrix is around 11:16. Although in this particular case the sparsity level does not favour the adoption of sparse data processing mechanisms, the Voting block can still be validated in such circumstances and should also reproduce the same output result as a traditional convolution would have. Besides, although the Voting block has been developed to process LiDAR data, simple 2D matrices composed only of integer values are also appropriate to validate the block's implementation at an early stage.

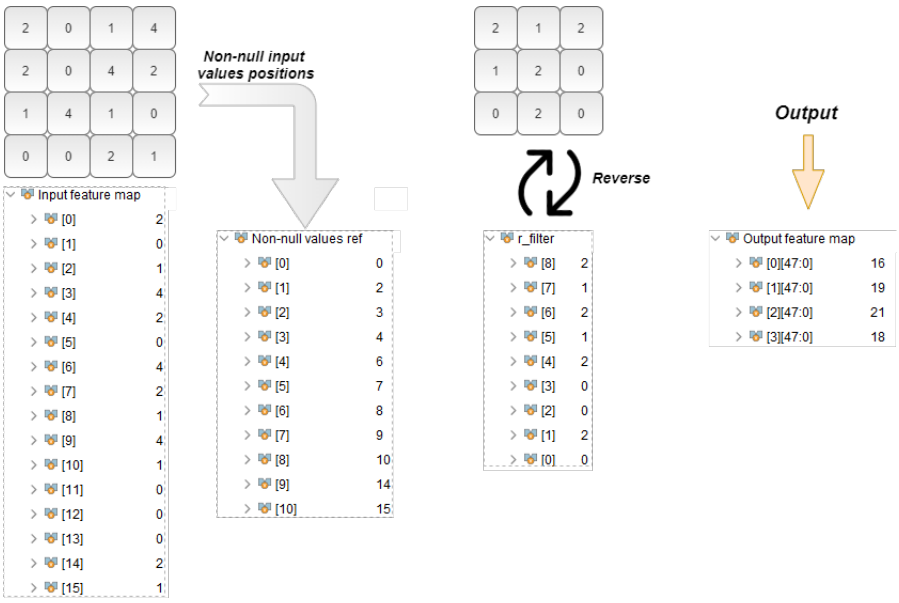


Figure 5.11: Voting block simulation data preparation

The simulation screenshot 5.12 from Vivado, refers to the convolution specified before, and it is indicative of the Voting block proper functioning. Each signal presented in the simulation window can be distinguished with a specific color and is associated with a particular role within the block. For instance, the orange color is assigned to the output data represented through the registers *o_data_PE0* and *o_data_PE1*, as well as the signals *o_en0* and *o_en1*. According to the dimension of both the input feature map and filter, the output feature map size is 2x2, and the values are identified in the simulation with red circles. Although the output enable signal from each processing element has a value of one several times during the simulation, the last ones are associated with the last values that are written in memory, and therefore, correspond to the convolution result.

Following the logic described in sub-section 4.14, each Voting block is responsible for recording the non-null output values position in the output feature map. From the simulation window 5.12, it is possible

to note that the output has four values, with the red arrows indicating the four different positions where the values are stored in the memory. The *o_ref_val* port, also presented in the interface of the block 4.7, identifies the coordinate in the output feature map, while the signal *o_re_w_en* activates the write operation mode on the output reference memory.

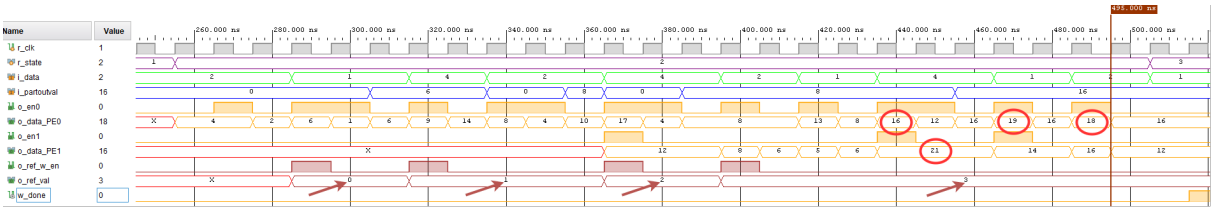


Figure 5.12: Voting block simulation window

Moving on to the next validation tests, figure 5.13 presents the convolution result between an image of a car and the Gaussian blur filter. Once again, despite the voting scheme-based convolution being only suitable for sparse data processing, the final result of any convolution should always be the same as using a traditional convolution mechanism. Since an image is a dense data representation, the time consumed by the voting block to complete the convolution is very high and even longer than the time that a traditional convolution would take. Nonetheless, the output image indicates that the convolution was performed correctly. The output image size is also correct since the padding and stride were both specified with values zero and one, respectively.

Figure 5.13: Voting convolution with a Gaussian Blur filter

5.2.2 Sparsity effect

This test aims to demonstrate and evaluate the applicability of the voting convolution by analyzing for which sparsity levels it is faster than the traditional one. The number of non-null values in the input feature map affects the data sparsity level, which can vary from zero to one, according to the math formula.

$$sparsity = 1 - \frac{count\_nonzero(A)}{total\_elements\_of\_(A)}$$

While zero sparsity indicates that all the input values are non-null, a sparsity with value one means

that all the values are null. To evaluate the Voting block performance in processing sparse data, several conditions should be created for the block to be subject to different levels of sparsity.

Table 5.5 discriminates the different sparsity levels selected for the set of performance tests executed on the Voting block. Although performance tests with lower sparsity levels are also possible, the number of values that need to be processed increases and compromises the adoption of the voting scheme-based convolution over the traditional mechanisms. Eleven levels of sparsity were selected to evaluate the Voting block performance for the processing of an 512x512 input feature map. The feature map size was not chosen randomly but based on the case study detailed in sub-section 5.2.6, to enable more performance comparisons.

Total values *IFM = 512x512*	**Sparsity** *(%)*	**Non-null values**	**Null values**
262,144	80	52,429	209,715
	82	47,186	214,958
	84	41,944	220,200
	86	36,702	225,442
	88	31,458	230,686
	90	26,215	235,929
	92	20,972	241,172
	94	15,729	246,415
	96	10,486	251,658
	98	5,243	256,901
	100	0	262,144

Table 5.5: Levels of sparsity selected for testing

For each sparsity level presented in table 5.5, a performance test was carried out to evaluate the processing time evolution according to the sparsity level variation. Figure 5.14 presents a graph where the different processing times registered with Voting block are identified with the blue line while the orange line indicates the time consumed by the Convolutional Module. As expected, the Voting block consumes less time when the sparsity level is higher, since less input values need to be processed. In the tests performed, a clock of 100 MHz was defined, which is equivalent to 10 nanoseconds per clock cycle. The Y-axis represents the processing time related to each measurement, specified in microseconds.

From graph 5.14, the higher processing time registered for the Voting block was around 4600 microseconds while the lower one was 390 (when ignoring the test with sparsity level of one, since a 100% of sparsity means that all values are null, and no time is consumed to process the data). On the other hand, the orange line indicates that the Convolutional Module consumes the same processing time regardless of the input sparsity level. Since a traditional convolution was implemented in the Convolutional Module, all

the input feature map regions are convolved with the filter, even if there is no relevant information located there.

The intersection point between the two lines is positioned around the 89% sparsity, corresponding to a processing time of around 2600 microseconds. In fact, the time consumed by the Convolutional Module to process a 512x512 input feature map is always about 2600 microseconds, while the Voting block only consumes the same time specifically with 89% of sparsity. For higher sparsity levels than 89%, the Voting block is faster than the Convolutional block and for lower sparsity levels the Voting block is slower, as graph 5.14 suggests. The performance tests performed to evaluate the processing time evolution with the sparsity variation were carried out considering an input feature map with a total of 262144 values, and the non-null values for each sparsity level were randomly distributed by the feature map. This detail is important to mention since the processing time will also vary depending on the dispersion of the non-null values in the feature map. Therefore, a separate performance test was built to exploit this variant.

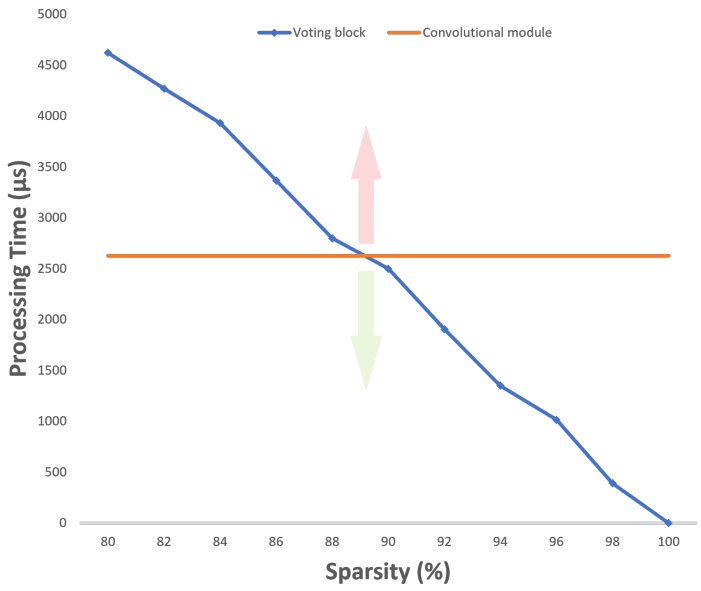


Figure 5.14: Processing time variation with increasing sparsity

5.2.3 Concentration metric

The dispersion level of non-null values in the input feature map together with the sparsity level have an impact on the processing time obtained by the Voting block. While sparsity refers to the number of values that need to be processed, the dispersion level is related to the proximity of these values in the feature map. The two extreme and opposite levels of dispersion were highlighted in figure 4.4, and although the sparsity level is equal for both cases, the processing time obtained can be quite different. The optimization

technique described in sub-section 4.11, was designed to reuse data and reduce the time consumed in the communication with the memory. Since values spatially close in the feature map belong to the same region, data can be shared during the convolution. On the other hand, spatially distant values are located in different regions so no data can be shared.

The impact of the implemented data reuse technique is presented in table 5.6, and it is noted that the improvement in processing time can be more than 30% when all the non-null values are spatially close in the input feature map. The registered times for 0% concentration are presented in the second column and correspond to the blue line on graph 5.14. The processing times for maximum spatial proximity of the non-null values in the input feature map are presented in the third column. The improvement in processing time presented in the table reflects the effect of the data reuse technique when the concentration of values is maximum. Another aspect is the improvement percentage increase with the increase of sparsity level, since more non-null values improve the performance of the data reuse technique, thus less time is consumed for higher sparsity levels.

Sparsity (%)	**Processing time (μs)**		**Improvement (%)**
	Concentration (0%)	Concentration (100%)	
80	4,622	3,507	24.1
82	4,270	3,227	24.4
84	3,932	2,946	25.1
86	3,367	2,506	25.6
88	2,800	2,079	25.8
90	2,500	1,845	26.2
92	1,905	1,393	26.9
94	1,412	1,026	27.3
96	1,085	773	28.8
98	392	272	30.6
100	0	0	0.0

Table 5.6: Values concentration effect on processing time

After concluding that the concentration of non-null values in the input feature map contributes to better execution time, sparsity level maintenance is also an important topic. Following the same performance tests executed on table 5.6, for each sparsity level, the number of non-null output values was counted for both stipulated concentration levels. The results are presented in table 5.7, regarding the number of non-null output values, where a huge difference can be noted between the two cases. When the non-null values are randomly distributed in the feature map, the number of non-null output values greatly increases compared to the input. On the other hand, with the non-null input values fully concentrated, the number of non-null output values increase registered is quite inferior.

In summary, the non-null input values concentration is a relevant metric, since with higher concentration levels, not only the processing times obtained are lower but also the number of generated non-null output values is much smaller. The substantial increase in the number of non-null values compromises the use of the Voting block in the next layer, since lower sparsity levels do not favour the adoption of sparse convolutions to process the data. Together with the number of non-null input values, the level of concentration should also be considered in order to verify if the use of the voting convolution will be advantageous over the tradition mechanisms.

Sparsity (%)	**Concentration (0%)**		**Concentration (100%)**	
	Output values	Increase (%)	Output values	Increase (%)
80	232,103	342.7	53,372	1.8
82	219,792	365.8	48,082	1.9
84	206,322	391.9	42,782	2.0
86	194,557	430.1	37,472	2.1
88	182,550	480.3	32,181	2.3
90	160,121	510.8	26,870	2.5
92	137,995	558.0	21,580	2.9
94	110,307	601.3	16,232	3.2
96	83,416	695.5	10,905	4.0
98	43,480	729.3	5,515	5.2
100	0	0.0	0	0.0

Table 5.7: Output values increase ratio

5.2.4 Null-weights processing optimization

In sub-section 3.3.4, it was described that the pipeline processing mechanism can be optimized by removing the instructions associated with the filter's null weights. According to the formula presented in 3.19, the calculation can be ignored if the weight is equal to zero, meaning that the value stored in the output memory does not need to be read, neither the write operation needs to be performed posteriorly. From that, one more set of performance tests was carried out to evaluate how the processing time is optimized in the presence of zero weights in the filter. For each test, a 3x3 filter was used, and the values of the nine weights were manipulated to verify the influence of the number of zero weights in the execution time.

Figure 5.15 shows a graph with a total of eleven lines and the description of each one below in the legend. As detailed in sub-section 4.2.3 through the code 4.10, a zero weight optimizes the processing as the corresponding instruction can be removed from the pipeline. With the increase in the number of zero

weights in the filter, the processing time evolution for each sparsity level can be analyzed from the graph. Once again, to separate each performance test performed on the Voting block, the non-null values were randomly distributed in the input feature map according to the specified sparsity level, and the processing times were measured for each situation. The results indicate that null weights have also a good impact on the processing time whatever the sparsity level of the input data. For instance, although the gray line registers higher processing times than the Convolutional Module (black line), it becomes faster when the number of zero weights is greater than five.

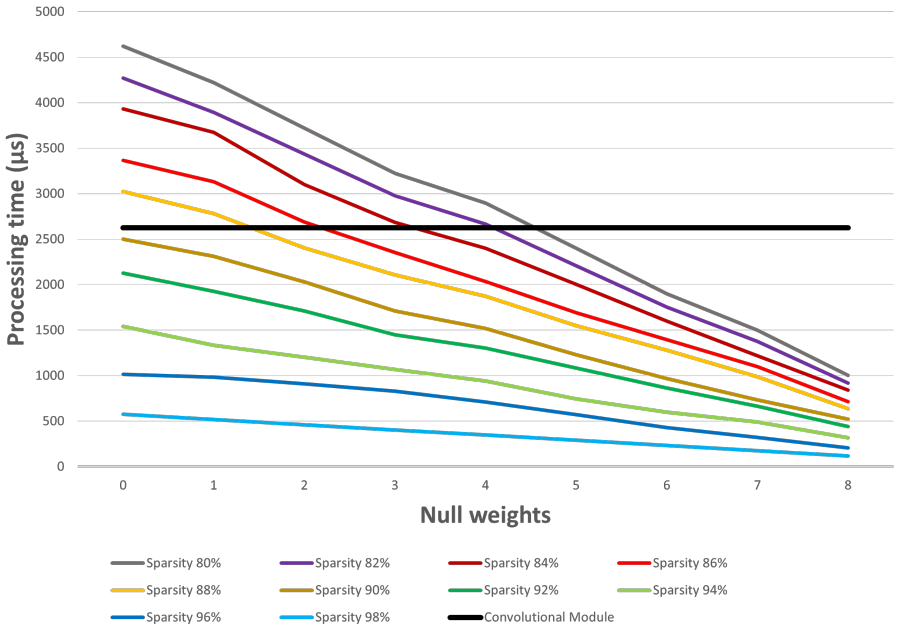


Figure 5.15: Null weights effect on processing time

5.2.5 Strided operation boost

The level of customization provided to the user enables both the specification of the padding and stride, according to the use case presented in figure 3.30. While introducing padding into the voting scheme-based convolution is a simple operation, the integration of stride to a sparse convolution is subject of study and promotes another set of performance tests on the Voting block. The two value options (1 and 2) that the user is able to specify for the stride parameter covers most of the scenarios and interesting performance differences can be identified from the tests executed. Table 5.8 presents, for each stride value and sparsity level, the measurements of both processing time and number of non-null output values generated. When a stride of two is defined, although the output size is half of the input, the sparsity levels are about the same as the ones obtained with a stride of one, since either null and non-null values are ”discarded” while performing a strided convolution.

Another important aspect is the difference in processing time between the operation with a stride of one and a stride of two. The results from table 5.8 indicate that, when the convolution is performed using a stride of two, the processing time consumed by the Voting block is reduced. This difference can be justified by the fact that, while with a stride of one all the input feature map regions need to be convolved with the filter, with a stride of two, some rows and columns are "discarded". As a result, the communication with the output memory is reduced since fewer values need to be read and written. Contrary to the Convolutional Module, the Voting block leverages from the strided operations as each pipeline iteration is optimized and less time is allocated for each non-null input value processing. The Convolutional Module, in turn, needs to read all the input values and equally perform the cascade operations, so no time reduction is achieved when the user specifies a stride of two.

Input sparsity (%)	**Stride = 1**			**Stride = 2**		
	Time (μs)	Output values	Output sparsity (%)	Time (μs)	Output values	Output sparsity (%)
80	3,507	53,372	79.6	1,378	12,845	80.4
82	3,227	48,082	81.6	1,266	11,665	82.2
84	2,946	42,782	83.7	1,156	10,420	84.1
86	2,506	37,472	85.7	989	9,043	86.2
88	2,079	32,181	87.7	881	7,667	88.3
90	1,845	26,870	89.8	714	6,488	90.1
92	1,393	21,580	91.8	585	5,046	92.3
94	1,026	16,232	93.8	441	3,932	94.0
96	773	10,905	95.8	285	2,490	96.2
98	272	5,515	97.9	121	1,245	98.1
100	0	0	100	0	0	100

Table 5.8: Strided voting convolution performance test

5.2.6 PointPillars

To extend the validation of the Voting block hardware implementation, the PoinPillars model was chosen as a case study to verify the advantages of the voting scheme-based convolution. In addition of being a state-of-the-art model in 3D object detection, as described in section 2.2.1, the model meets the requirements for the Voting block integration. As one of the critical requirements is the position of the non-null values in the feature map, that information can be accessed through the data structure that composes the Pillar Index from the PFN stage. Furthermore, the point cloud representation in a pseudo-image ensures high levels of sparsity for 2D data processing, which is ideal for evaluating the performance of a sparse convolution.

Table 5.9 shows the convolutional blocks architecture in the PointPillars' backbone stage, specifying

the number of filters and channels, both the feature maps and filters dimensions, and the stride and padding parameters for each convolution. Although from the backbone architecture, three deconvolution blocks together with a concatenation layer can be also identified, the table only presents the existing convolution layers, since the Voting block just performs convolutions. While all convolutions are carried out with a kernel size of three and padding of one, only the first convolution layer of each block is performed with a stride of two.

Stage	Layer	Type	Channels	Filters	IFM	OFM	Kernel	S	P
Backbone	Block1	Conv0	64	32	512	256	3	2	1
		Conv1	32	32	256	256		1	
		Conv2	32	32	256	256			
		Conv3	32	32	256	256			
	Block2	Conv0	32	32	256	128		2	
		Conv1	32	32	128	128		1	
		Conv2	32	32	128	128			
		Conv3	32	32	128	128			
		Conv4	32	32	128	128			
		Conv5	32	32	128	128			
	Block3	Conv0	32	64	128	64		2	
		Conv1	64	64	64	64		1	
		Conv2	64	64	64	64			
		Conv3	64	64	64	64			
		Conv4	64	64	64	64			
		Conv5	64	64	64	64			

Table 5.9: PointPillars' backbone convolutional blocks

Considering that the PFN output data has high levels of sparsity, it is also relevant to analyze the sparsity evolution across the convolutional layers of each backbone block. From this analysis, it is possible to determine the layers where the Voting block integration is appropriate and later evaluate its performance. Since the sparsity level is the main criterion that defines whether the Voting scheme-based convolution should be applied or not, in the PointPillars software version, the ratios between null and non-null values of the feature maps from each layer were identified. Considering that the number of null and non-null values presented in each feature map depends on the frame being processed at the moment, a frame with the highest possible number of non-null values was purposely chosen as a reference. According to the PointPillars model specification, for each one of the 64 output channels from the PFN stage, the number of non-null values can vary between 2k and 12k. Assuming the worst case, 12k non-null values in each channel added together result in the 768,000 number shown in table 5.10.

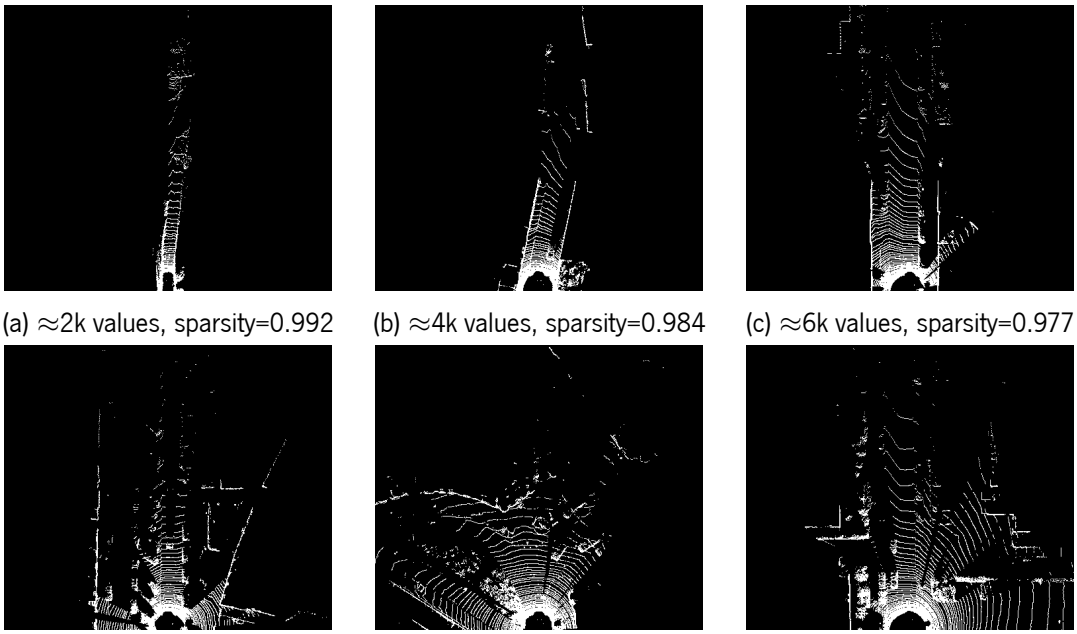
The sparsity levels decrease across the layers, meaning that the number of non-null values grows as more convolutions are applied to the feature map, due to the dilation phenomenon in convolutions. In

addition to each convolution negatively affecting the sparsity level, the use of positive bias has also a great impact, as all feature map values need to be added to the bias at the end of each convolution, which may possibly turn all values into non-nulls. Following the performance tests presented in figure 5.14, sparsity levels below 89% do not favour the use of the voting convolution, therefore only the first layer of block 1 is a good candidate for voting-based convolutions.

		Non-null values	**Null values**	**Sparsity**
PFN		768,000	16,009,216	0.95
Block1	*Conv0*	293,601	1,803,551	0.86
	Conv1	440,402	1,656,750	0.79
	Conv2	734,003	1,363,149	0.65
	Conv3	775,946	1,321,206	0.63
Block2		256,901	267,387	0.51
Block3		138,936	123,208	0.47

Table 5.10: Backbone layers' sparsity for the worst case, i.e. each channel with 12k non-null values

To prepare the data for the different sparsity level scenarios, several frames were selected from the Kitti dataset [61]. Figure 5.16 presents the frames' representation in a black-and-white format to better distinguish the number of null values in each one. The non-null values are represented with white pixels while the null values are represented with black ones. From each sub-figure, it is possible to recognize the captured scene and also the substantial difference in the number of values that need to be processed between the feature map with the highest 5.16a and lowest 5.16f sparsity level.



(a) ≈2k values, sparsity=0.992 (b) ≈4k values, sparsity=0.984 (c) ≈6k values, sparsity=0.977

(d) ≈8k values, sparsity=0.969 (e) ≈10k values, sparsity=0.961 (f) ≈12k values, sparsity=0.954

Figure 5.16: Test frames with different sparsity levels

Considering the range of non-null values for each of the 64 feature maps from the PFN, several tests were carried out to evaluate the Voting block performance when processing only one of the feature maps. The blue line on graph 5.17 presents the time consumption results of the voting block processing. From the graph, there is a natural increase in time consumption with the increase of the number of values to be processed by the block. As the range of non-null output values from the PFN stage for each feature map is between 2k and 12k, the extreme points of the blue line are identified as best and worst-case scenarios, respectively.

Taking into account that the size of the feature maps from the PFN has a dimension of 512x512, the Convolutional Module always consumes the same time regardless of the feature maps sparsity level. Similar to graph 5.14, the orange line has no inclination as the result of each performance test with the Convolution Module was 2621μs. Given the possibility of integrating parallelism in the Convolutional Module through the use of several Processing Elements, the gray arrows show the level of parallelism required for the processing to be as fast as the one in the Voting block. Assuming a clock of 100MHz, in the best-case scenario, the Voting block only consumes 69μs, meaning that the Convolutional Module needs to allocate at least 38 PEs for the processing times to be approximately equal. Since for the worst case, the Voting block consumes more time, only 8 PEs are required to decrease the processing time from 2621μs to 341μs.

Although the use of several PEs to introduce parallelism is possible, it requires extra resource consumption, with the majority being DSPs. Some FPGAs provide a large number of DSPs, however, to build a single PE with a 3x3 filter, nine DSPs need to be allocated. In addition, given the on-chip memory data access ports limitations, namely the Block Rams, the processing of different feature map regions simultaneously implies the data to be previously distributed into several block memories.

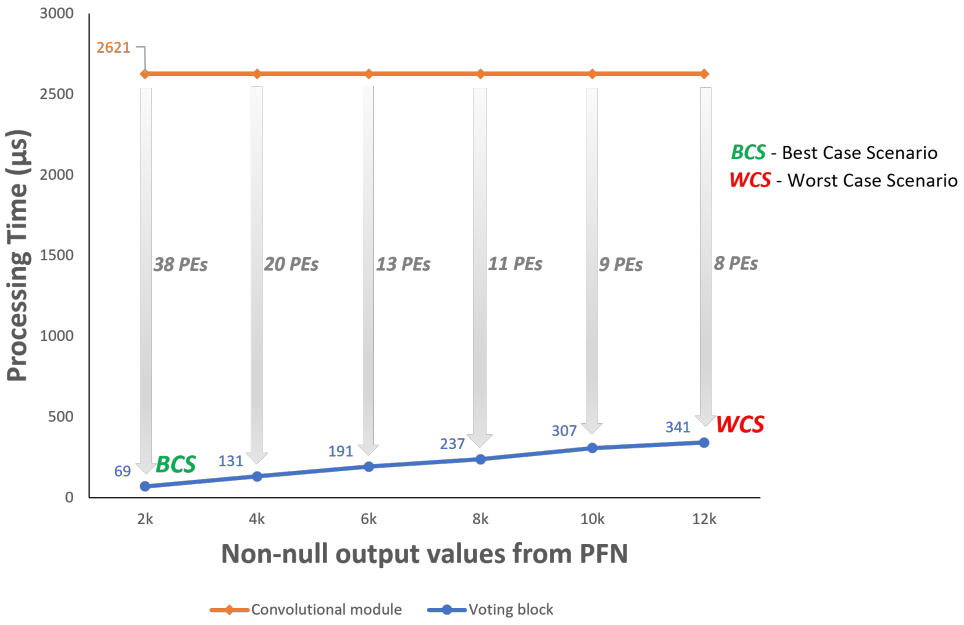


Figure 5.17: Processing time comparison between dense and voting convolutions, for the selected frames 5.16

Considering the layers where the sparsity level favours the use of the voting scheme-based convolution, and the concentration level of the non-null values in the frames 5.16, the integration in the first three layers of block 1 was carried out, assuming a 200MHz clock on the board. In each test, the time consumed, by the Convolutional Module and the Voting block, processing the feature maps associated with these layers were measured. Regarding the Convolutional Module, the parallelism was set to maximum per filter to improve performance, using all available DSPs. The results are presented in table 5.11, where it was assumed the worst-case scenario, with each feature map from the PFN stage containing a total of 12k non-null values to process.

The processing time improvements for the Voting Block, located in the last column, are only positive for the first two layers, since the sparsity level decreasing across the layers increases the time consumed. As a result, comparing with the software version, in the third layer of block 1 no improvement is verified since the time to process is 10.9% longer. On the other hand, a big improvement can be seen in the first layer of block 1, with the Voting block being 80.44% faster. This great improvement is achieved due to the level of sparsity being higher in the first layer, and also because it is a strided operation, which helps the Voting to achieve substantially lower execution times, as referenced in 5.8.

	Software	Convolutional Module		Voting Block	
	Time (μs)	Time (μs)	Improvement (%)	Time (us)	Improvement (%)
B1-Conv0	874	654	25.18	171	80.44
B1-Conv1	321	262	18.32	247	23.05
B1-Conv2	321	262	18.32	356	-10.90

Table 5.11: Processing time comparasion betwen SW version, Convolutional Module, and Voting Block for the first three layers of Block 1

As detailed throughout the voting block design 3.3 and implementation 4.2 sections, null weights have a positive impact on the pipeline processing time during convolutions. With PointPillars being the case study for the validation of the block, the value of parameters, in particular the weights, was also an object of study. The hardware implementation of the convolution layers to execute the performance tests was carried out assuming an 8-bit quantization for the parameters. For both floating-point and fixed-point representation, the number of null weights was counted, and the results are presented in table 5.12.

The analysis shows that with the parameters' quantization, the weights that are represented in floating-point with the smallest values are converted to zero in the 8-bit fixed-point format. For the first layer of block 1, which is the one with better sparsity conditions to integrate the voting scheme-based convolution, a total of 2200 weights are transformed to zero value. Considering that the first convolutional layer applies 32 filters each with 64 channels, an average of one null weight per channel is verified, which benefits even more the use of a Voting block to process the data from the PFN. In summary, if no weights were converted to zero after quantization, the processing time results presented in table 5.11 would be slightly higher.

As for feature map values, although the quantization process also converts some values to zero in the 16-bit fixed-point format, the position of the non-null values received by the Voting block does not predict this conversion. As a result, all the non-null values positions stored in the pillar index structure are read and the corresponding values are processed, so the quantization performed on the feature map values had no effect on the processing time.

	Before quantization		**After 8-bit quantization**	
	Non-null weights	Null weights	Non-null weights	Null weights
Block1	46,080	0	41,697	4,383
Conv0	18,432	0	16,232	2,200
Conv1	9,216	0	8,492	724
Conv2	9,216	0	8,521	695
Conv3	9,216	0	8,452	764
Block2	55,296	0	50,543	4,753
Block3	202,752	0	165,712	37,040

Table 5.12: Null weight count after quantization

5.3 Results summary

During this chapter, the tests performed on the Convolutional Module and the Voting block, as well as the results obtained, were presented in sections 5.1 and 5.2. Although in each sub-section the results of the validation and performance tests have been separately discussed, it is important to synthesize them to concretely define the ideal applicability of both. With the presentation of Voting convolution as an alternative to the traditional one, normally adopted in 3D object detection models, this section also refers the guideline on the ideal conditions for the use of voting convolution.

The performance tests showed the flexibility achieved with the proposed architecture. Depending on various restrictions imposed in terms of available resources, e.g., Block Rams and the number of DSPs, the module was able to adapt the resource consumption and improve the performance up to 25% compared to the software version. This performance was also evaluated through different levels of parallelism, both in the application of several filters simultaneously, as well as with the use of more than one processing element to perform a convolution. The time consumed was analyzed in detail, and it is possible to verify that it is reduced up to 49% when the module has more resources at its disposal to speed up processing.

As a case study and final validation, the module was integrated into the PointPillars model. The tests performed involved the replacement of two convolutional layers on the Backbone stage, and the detection results reveal similarities between the software-only and hybrid versions. The data quantization in hardware showed some influence on the detection scores, however, for the selected frames, the existing objects were equally detected. Thus, the results obtained give good prospects for the integration in deep learning-based models, while providing enough flexibility to build custom CNNs.

While the dense convolution is the base operation of the convolutional module, the voting scheme-based convolution was implemented in the Voting block with the purpose of evaluating the potential of a sparse convolution. This evaluation was performed by executing a set of tests on the block, varying the data characteristics, i.e., sparsity level, concentration of non-null values, number of null weights, and also changing the value of the stride parameter. Regarding the validation tests, the results showed the correct functioning of the block, even when used to process dense data, i.e., images. The performance tests were useful to understand which conditions favour the use of the Voting convolution and which conditions are more suitable for the traditional one.

Using the dense convolution implemented in the convolutional module as a reference, the data sparsity level was naturally the characteristic that had the most influence on the time consumed by the Voting block, thus, being the most important factor determining when the use of the Voting convolution is more

advantageous. The results presented showed that data with sparsity greater than 89% benefit from the use of voting convolution, however, the remaining factors must be analyzed for each situation to conclude which type of convolution is best suited. Starting with the influence of null weights on the processing time, there was an almost linear reduction with the increase of null weights. Based on the results obtained, it can be roughly deduced that each null weight reduces the processing time by approximately 10%, for 3x3 filters. The impact of a null weight is related to the ratio that one weight represents in the total number of weights in a filter. For example, for a 5x5 filter, a null weight would reduce the processing time by approximately 4% (1/25*100). On the other hand, for a 2x2 filter, the verified reduction would be around 25% (1/4*100).

Another factor that has an impact on the Voting block processing time is the concentration of non-null values in the feature map. Although the tests performed represent the two extreme cases (0% or 100% concentration), this is a difficult characteristic to assess. Through the case study, some frames were visually analyzed in figure 5.16, however, it is not easy to define exactly the concentration level of the non-null values. Using as a reference the tests with the non-null values completely dispersed, presented in table 5.6, the processing time of the pseudo-images was on average 15% lower, for the same sparsity levels. This result shows that, in pseudo-images, the non-null values are somewhat concentrated in certain regions, and a reduction in the processing time of about 15% can be expected when using the Voting convolution.

Last but not least, the value of the stride parameter also has a substantial impact on processing time. Through the tests, for the same input data, a stride value greater than one helps to reduce the processing time by about 55%. Together with the other factors mentioned before, all these conditions must be considered for each scenario before deciding which type of convolution will bring the best performance. For example, in a given convolutional layer, the input data may have a sparsity level that does not favour the use of the Voting convolution (less than 89%), however, the operation may specify a stride of two, which favours the voting convolution. If the convolution is with a stride of one, maybe the values are concentrated in the feature map or some filter weights are null. All these factors have to be put into the equation, that will tell if the Voting block will improve processing time, or the traditional convolution is still the best choice.

Chapter 6: Conclusion

In this last chapter, reflections on the work performed throughout the dissertation are presented in section 6.1, followed by the suggestions for future work in section 6.2.

6.1 Conclusions

With the recognized importance of the LiDAR sensor in autonomous vehicles setups, a lot of LiDAR-based algorithms for object detection have been published in the literature. Over the last few years, deep learning-based models have revealed their potential in perception-type tasks, however, the need for distributed computing requires an efficient design of these models targeting resource-constraints platforms. Given the need for a study focused on both efficient design and implementation of these models in hardware, a convolutional module was developed as a customizable IP to implement CNNs in hardware.

The study conducted throughout this dissertation enabled answering the following research questions, stated in the objectives.

1. What are the resource costs of building a convolutional module in hardware that is adaptable to the requirements of 3D object detection models?

The convolutional module development shows that it is possible to improve the processing time and energy consumption without significantly sacrificing the accuracy of the models. By resorting to parallelism, the module was able to accelerate the execution of these operations, however, more resources need to be allocated. Starting with the minimum conditions to properly execute the convolutions, the minimum number of DSPs required is set by the filter size, which means the allocation of only one processing element. The parallelism integration requires the allocation of more processing elements, each requiring the same number of DSPs.

Regarding the memory, the amount available on the target platform conditions the number of filters applied simultaneously (the other form of parallelism). To enable the execution of a single convolution, the memory required is defined by the amount of both input (IFM) and output data (OFM). The total memory can be calculated considering the number of values from the feature maps and the size that each one occupies in memory.

2. What are the characteristics of the data that influence the performance of sparse convolutions?

The performance of sparse convolutions depends on the type of convolution and its implementation. Although the sparsity level influences the performance of all types of sparse convolution, the proposed hardware architecture for the voting convolution integrates processing optimization techniques targeting the presence of null weights, spatially close non-null values, and operations with stride. Thus, all four of these metrics influence the voting convolution performance.

3. According to the different possible variations in the data characteristics, what conditions favour the use of sparse convolutions?

According to the investigation carried out during the tests, the four metrics mentioned above had a significant impact on the voting convolution performance, as opposed to the traditional convolution implemented in the convolutional module, which maintained its performance with the variation of these metrics. Starting with sparsity, for levels above 89%, the voting convolution is faster than the traditional one. Second, two-stride operations strongly favour the use of voting convolution, which is able to reduce communication with the output data memory and reduce the processing time by 55%. Additionally, the presence of null weights in the filters is also a point in favour of the voting convolution, as fewer operations are performed for each non-null input value, leading to an almost linear reduction in processing time for each null weight. Finally, the concentration of non-null values in the feature maps also benefits the adoption of the voting convolution, which is able to reduce the computation cycles up to 30.6%.

4. Are there 3D models in the literature that meet the conditions for the adoption of sparse convolutions?

The study carried out throughout this dissertation has shown that the main criteria to evaluate whether the sparse convolutions are applicable is the availability of the location of non-null values in a point cloud. After studying the current state of the art in 3D object detection and classification models, it was possible to identify that most of the models opted by restructuring the point cloud into 3D volumetric representations, where information about the location of non-null values is stored in specific data structures. This information is critical for further stages of the pipeline, as it retains information either about the location of the volumetric parts or the individual non-null values as seen in PointPillars, VoxelNet, and Second models.

5. For a real case scenario, what are the practical improvements in adopting a sparse convolution instead of a traditional one?

The adoption of sparse convolutions, in particular the voting convolution, is mainly intended to reduce processing time. For data with a high level of sparsity, the voting convolution is able to discard useless operations related to null values from the feature map, and direct the processing to non-null values, where

the relevant information is located. As a case study, the voting block was integrated with PointPillars for validation of the hardware implementation and analysis. Due to the high level of sparsity, the performance of the voting convolution in the first two backbone layers was superior to the traditional one, implemented in the convolutional module. The use of voting convolution allowed to reduce the processing time by 80.44% and 23.05% in the first and second layers, respectively.

6.2 Future Work

Several improvements can be implemented, and suggestions can be indicated to continue the work developed. Regarding the convolutional module, the integration of parallelism through the allocation of several PEs is conditioned by the limitation of data access ports of each memory block. As one Block Ram only provides two ports, when several PEs are used simultaneously to process the same feature map, the data has to be distributed across several block memories beforehand. During the module development, this process was not automated, requiring the manual data distribution to take advantage of the parallelism. Thus, one of the suggestions for future work would be to analyze a solution to this limitation, which could involve modifying the implemented parallelism by using several PEs to process the same region of the feature map.

Although the functionalities implemented in the convolutional module cover the requirements of most convolution layers, a few more can be integrated to build an even more complete solution. As seen in several 3D object detection models' architecture, the batch normalization technique is present in some layers. This feature was not included in the developed module, however, its integration would increase the applicability in more cases. Moreover, the deconvolution operation was also identified. Although the purpose of the convolution module was to perform convolutions, the implementation of the deconvolution should be considered given the similarities of both operations. Additionally, the implementation of the entire PointPillars model in hardware using the developed module would be an advantage for a broader analysis of convolutions in hardware.

Regarding the Voting block, the potential of voting convolution was clearly demonstrated through the performance tests carried out and from the integration into the PointPillars model. Given the usefulness of sparse convolutions in CNNs, another idea for future work would be the integration of the Voting convolution in the convolutional module. Combining the two architectures would bring even more flexibility to the module, as the user would be able to choose which type of convolution to use for each instantiation.

References

[1] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3354–3361, 2012, DOI: 10.1109/CVPR.2012.6248074.

[2] D. P. Leclaire and D. N. Baron, "LIDAR for Automotive – Patent Landscape Analysis – April 2018." `https://www.knowmade.com/wp-content/uploads/2018/05/LIDAR-for-Automotive-Patent-Landscape-2018-FLYER.pdf`, 2018. Accessed on 2021-01-20.

[3] Y. Zhou and O. Tuzel, "Voxelnet: End-to-end learning for point cloud based 3d object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4490–4499, 2018.

[4] Y. Yan, Y. Mao, and B. Li, "Second: Sparsely embedded convolutional detection," *Sensors*, vol. 18, no. 10, p. 3337, 2018.

[5] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom, "Pointpillars: Fast encoders for object detection from point clouds," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12697–12705, 2019.

[6] B. Zhu, Z. Jiang, X. Zhou, Z. Li, and G. Yu, "Class-balanced grouping and sampling for point cloud 3d object detection," *arXiv preprint arXiv:1908.09492*, 2019.

[7] C. R. Qi, W. Liu, C. Wu, H. Su, and L. J. Guibas, "Frustum pointnets for 3d object detection from rgb-d data," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 918–927, 2018.

[8] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia, "Multi-view 3d object detection network for autonomous driving," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 1907–1915, 2017.

[9] J. Beltrán, C. Guindel, F. M. Moreno, D. Cruzado, F. Garcia, and A. De La Escalera, "Birdnet: a 3d object detection framework from lidar information," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 3517–3523, IEEE, 2018.

[10] MathWorks, "Lidar 3-D Object Detection Using PointPillars Deep Learning." `https://www.mathworks.com/help/lidar/ug/object-detection-using-pointpillars-network.html`, 2019. Accessed on 2019-11-18.

[11] K. Abdelouahab, M. Pelcat, J. Sérot, C. Bourrasset, J.-C. Quinton, and F. Berry, "Hardware automated dataflow deployment of cnns," *arXiv preprint arXiv:1705.04543*, 2017.

[12] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016, DOI: 10.1109/MICRO.2016.7783720.

[13] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and et al., "Fast inference of deep neural networks in fpgas for particle physics," *Journal of Instrumentation*, vol. 13, p. P07027–P07027, Jul 2018.

[14] Xilinx, "Deep Neural Network Accelerator library for Xilinx Ultrascale+ MPSoC devices." `https://github.com/Xilinx/CHaiDNN`, 2018. Accessed on 2021-01-20.

[15] Xilinx, "Adaptable and Real-Time AI Inference Acceleration." `https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html`, 2020. Accessed on 2021-01-20.

[16] CDL, "A deep learning platform optimized for implementation of FPGAs." `https://coredeeplearning.ai/`, 2020. Core Deep Learning (CDL). Accessed on 2021-05-02.

[17] xilinx, "UltraScale Architecture DSP Slice." `https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf`, 2020. Accessed on 2021-06-22.

[18] Y. Li and J. Ibanez-Guzman, "Lidar for autonomous driving: The principles, challenges, and trends for automotive lidar and perception systems," *IEEE Signal Processing Magazine*, vol. 37, no. 4, pp. 50–61, 2020, DOI: 10.1109/MSP.2020.2973615.

[19] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun, "Deep learning for 3d point clouds: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2020, DOI: 10.1109/TPAMI.2020.3005434.

[20] J. Leonard, J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang, S. Karaman, *et al.*, "A perception-driven autonomous urban vehicle," *Journal of Field Robotics*, vol. 25, no. 10, pp. 727–774, 2008.

[21] Y. Li, *Stereo vision and Lidar based dynamic occupancy grid mapping: Application to scenes analysis for intelligent vehicles*. PhD thesis, Université de Technologie de Belfort-Montbeliard, 2013.

[22] S. L. Poczter and L. M. Jankovic, "The google car: driving toward a better future?," *Journal of Business Case Studies (JBCS)*, vol. 10, no. 1, pp. 7–14, 2014.

[23] D. M. West, "Moving forward: self-driving vehicles in china, europe, japan, korea, and the united states," *Center for Technology Innovation at Brookings: Washington, DC, USA*, 2016.

[24] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun, "Deep learning for 3d point clouds: A survey," *IEEE transactions on pattern analysis and machine intelligence*, vol. PP, 2020, DOI: 10.1109/TPAMI.2020.3005434.

[25] Z.-Q. Zhao, P. Zheng, S.-t. Xu, and X. Wu, "Object detection with deep learning: A review," *IEEE transactions on neural networks and learning systems*, vol. 30, no. 11, pp. 3212–3232, 2019.

[26] D. Fernandes, A. Silva, R. Névoa, C. Simões, D. Gonzalez, M. Guevara, P. Novais, J. Monteiro, and P. Melo-Pinto, "Point-cloud based 3d object detection and classification methods for self-driving applications: A survey and taxonomy," *Information Fusion*, vol. 68, pp. 161–191, 2021, DOI: 10.1016/j.inffus.2020.11.002.

[27] F. Libano, B. Wilson, M. Wirthlin, P. Rech, and J. Brunhaver, "Understanding the impact of quantization, accuracy, and radiation on the reliability of convolutional neural networks on fpgas," *IEEE Transactions on Nuclear Science*, vol. 67, no. 7, pp. 1478–1484, 2020.

[28] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[29] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on fpgas: A survey," *arXiv preprint arXiv:1806.01683*, 2018.

[30] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016, DOI: 10.1109/JIOT.2016.2579198.

[31] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2018.

[32] S. Mittal, "A survey of techniques for managing and leveraging caches in gpus," *Journal of Circuits, Systems, and Computers*, vol. 23, no. 08, p. 1430002, 2014.

[33] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 16–25, 2016.

[34] A. Rahman, S. Oh, J. Lee, and K. Choi, "Design space exploration of fpga accelerators for convolutional neural networks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 1147–1152, IEEE, 2017.

[35] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pp. 161–170, 2015.

[36] S. Mittal, "A survey of fpga-based accelerators for convolutional neural networks," *Neural computing and applications*, vol. 32, no. 4, pp. 1109–1139, 2020.

[37] H. Gao, B. Cheng, J. Wang, K. Li, J. Zhao, and D. Li, "Object classification using cnn-based fusion of vision and lidar in autonomous vehicle environment," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 9, pp. 4224–4231, 2018.

[38] M. Liang, B. Yang, S. Wang, and R. Urtasun, "Deep continuous fusion for multi-sensor 3d object detection," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 641–656, 2018.

[39] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, "nuscenes: A multimodal dataset for autonomous driving," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 11621–11631, 2020.

[40] P. Sun, H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik, P. Tsui, J. Guo, Y. Zhou, Y. Chai, B. Caine, *et al.*, "Scalability in perception for autonomous driving: Waymo open dataset," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2446–2454, 2020.

[41] J. Geyer, Y. Kassahun, M. Mahmudi, X. Ricou, R. Durgesh, A. S. Chung, L. Hauswald, V. H. Pham, M. Mühlegg, S. Dorn, *et al.*, "A2d2: Audi autonomous driving dataset," *arXiv preprint arXiv:2004.06320*, 2020.

[42] C. Papachristos, S. Khattak, and K. Alexis, "Autonomous exploration of visually-degraded environments using aerial robots," in *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, pp. 775–780, IEEE, 2017.

[43] A. Carballo, J. Lambert, A. Monrroy, D. Wong, P. Narksri, Y. Kitsukawa, E. Takeuchi, S. Kato, and K. Takeda, "Libre: The multiple 3d lidar dataset," in *2020 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1094–1101, IEEE, 2020.

[44] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 652–660, 2017.

[45] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," *arXiv preprint arXiv:1706.02413*, 2017.

[46] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, pp. 91–99, 2015.

[47] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[48] B. Graham, M. Engelcke, and L. Van Der Maaten, "3d semantic segmentation with submanifold sparse convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 9224–9232, 2018.

[49] D. Z. Wang and I. Posner, "Voting for voting in online point cloud object detection.," in *Robotics: Science and Systems*, vol. 1, pp. 10–15607, Rome, Italy, 2015.

[50] M. Engelcke, D. Rao, D. Z. Wang, C. H. Tong, and I. Posner, "Vote3deep: Fast object detection in 3d point clouds using efficient convolutional neural networks," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1355–1361, IEEE, 2017.

[51] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[52] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *European conference on computer vision*, pp. 404–417, Springer, 2006.

[53] K. Bora, M. Chowdhury, L. B. Mahanta, M. K. Kundu, and A. K. Das, "Pap smear image classification using convolutional neural network," in *Proceedings of the Tenth Indian Conference on Computer Vision, Graphics and Image Processing*, pp. 1–8, 2016.

[54] W. Ma and J. Lu, "An equivalence of fully connected layer and convolutional layer," *arXiv preprint arXiv:1712.01252*, 2017.

[55] B. Graham and L. van der Maaten, "Submanifold sparse convolutional networks," *arXiv preprint arXiv:1706.01307*, 2017.

[56] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[57] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 26–35, 2016.

[58] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," 1408.5093, 675–678, 2014.

[59] J. Jo, S. Kim, and I. Park, "Energy-efficient convolution architecture based on rescheduled dataflow," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4196–4207, 2018, DOI: 10.1109/TCSI.2018.2840092.

[60] xilinx, "https://digilent.com/reference/programmable-logic/zybo/reference-manual." `https://digilent.com/reference/programmable-logic/zybo/reference-manual`, 2020. Accessed on 2021-06-22.

[61] versatran01, "kittiarchives." `https://gist.github.com/versatran01/19bbb78c42e0cafb1807625bbb99bd85`, 2021. GitHub repository. Accessed on 2021-05-02.