FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FPGA-Based Real-time Inference for Semantic Segmentation

André Machado Campanhã

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Canas Ferreira

July 12, 2023

Abstract

Advanced Driver Assistance Systems (ADAS) have become integral to developing new automotive vehicles due to increasing safety and comfort requirements in the driving experience. An essential task of an ADAS pipeline is semantic segmentation, which must meet real-time requirements while operating in a power-constrained environment. Using the 3D point cloud generated with light detection and ranging (LiDAR) or images, convolutional neural networks (CNN) such as SalsaNext and SemanticFPN can perform semantic segmentation. Although it is possible to run these computationally expensive and highly parallel algorithms on Graphics Processing Units (GPU), Field Programmable Gate Arrays (FPGA) can be better performing and more power-efficient alternatives. The CNNs' performance can be further improved by applying channel pruning at the cost of accuracy.

This dissertation implements accurate and automatic channel pruning (AACP), an evolutionary genetic algorithm that automatically performs channel pruning to any CNN using dependency graphs. Then, it applies the Vitis AI workflow to quantize and generate deployable models of multiple pruned and unpruned versions of SalsaNext and SemanticFPN to run on the Deep-Learning Processing Unit (DPU) in a Xilinx Versal ACAP VCK190 FPGA board. The resulting DPU implementations match the performance on SalsaNext and achieve on average 2.25× better performance on SemanticFPN while being 10× more energy efficient when compared to a desktop GPU.

Resumo

Os Advanced Driver Assistance Systems (ADAS) tornaram-se parte indispensável do desenvolvimento de novos veículos automóveis devido ao aumento dos requisitos de segurança e de conforto na experiência de condução. Uma tarefa essencial de um ADAS é a segmentação semântica, que deve cumprir requisitos em tempo real enquanto opera num ambiente com restrições de energia. Utilizando a nuvem de pontos 3D gerada com light detection and ranging (LiDAR) ou imagens, convolutional neural networks (CNN) como SalsaNext e SemanticFPN conseguem efetuar a segmentação semântica. Embora seja possível executar estes algoritmos computacionalmente dispendiosos e altamente paralelos em Graphics Processing Units (GPU), Field Programmable Gate Arrays (FPGA) podem ser uma alternativa com melhor desempenho e maior eficiencia energética. O desempenho das CNNs podem ainda ser melhoradas aplicando a poda de canais, à custa da precisão.

Esta dissertação implementa o accurate and automatic channel pruning (AACP), um algoritmo genético evolutivo que efectua automaticamente a poda de canais a qualquer CNN utilizando grafos de dependência. Em seguida, aplica o fluxo de trabalho Vitis AI para quantizar e gerar modelos executaveis de várias versões podadas e não podadas de SalsaNext e SemanticFPN para serem executados na Deep-Learning Processing Unit (DPU) numa placa Xilinx Versal ACAP VCK190 FPGA. As implementações de DPU resultantes mantêm o desempenho em SalsaNext e alcançam em média um desempenho 2.25× melhor em SemanticFPN, sendo 10× mais eficientes em termos energéticos, quando comparadas com uma GPU de computador.

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao Professor João Canas Ferreira por estar sempre disponível para me motivar quando tudo parecia que não ia funcionar, e por me indicar o caminho para encontrar as soluções. Posso dizer com toda a certeza que sem o seu apoio, não teria acabado este documento.

Estou eternamente também à minha mãe, aos meus avós e ao meu irmão, que me aturaram todos estes anos a falar sobre assuntos que não entendiam, sempre me motivaram a trabalhar por melhor e me ofereceram a oportunidade de estudar.

Um especial carinho aos meus companheiros de curso, compinchas e amigos para a vida, Armando, Catarina, Cláudia, Filipe, Inês, Matilde, Miguel e Pedro. O curso teria sido uma jornada muito mais escura sem a vossa companhia.

Em particular, obrigado Catarina. És a melhor amiga que alguma vez podia ter pedido, e com este documento concluído, finalmente posso-te parar de chatear com as minhas figuras do TikZ e voltar ao meu nível de nerd normal.

This work is supported by European Structural and Investment Funds in the FEDER component, through the Operational Competitiveness and Internationalization Programme (COMPETE 2020) [Project nº 047264; Funding Reference: POCI-01-0247-FEDER-047264]

André Campanhã

"You miss 100 % of the shots you don't take."
- Wayne Gretzky

- Michael Scott

Contents

List of Figures

List of Tables

Abbreviations and Symbols

AACP	Accurate and Automatic Channel Pruning
ABC	Artificial Bee Colony
ACAP	Adaptive Compute Acceleration Platform
ADAS	Advanced Driver Assistance System
AI	Artificial Inteligence
ANN	Artificial Neural Network
API	Application Programing Interface
ASIC	Application Specific Integrated Circuit
BEV	Birds Eye View
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSV	Comma-separated Values
DE	Diferential Evolution
DNN	Deep Neural Network
DPU	Deep-Learning Processing Unit
DSP	Digital Signal Processor
FLOP	Floating-point Operation
FPGA	Field Programmable Gate Array
FPN	Feature Pyramid Network
FPS	Frame Per Second
GPU	Graphics Processing Unit
HLS	High Level Synthesis
IDE	Improved Diferential Evolution
IP	Intelectual Property
LiDAR	Light Detection and Ranging
LUT	Look-up-table
MAC	Multiply And Accumulate
mIoU	Mean Intersection-over-union
NoC	Network on Chip
OCR	Optical Character Recognition
PL	Programmable Logic
PS	Processing System
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RV	Range-view
SAE	Society Of Automotive Engineers

SDK	Software Development Kit
SoC	System on a Chip
SRAM	Static Random Access Memory
TOF	Time Of Flight
VART	Vitis AI Runtime
VRAM	Video Random Access Memory
XIR	Xilinx Intermediate Representation

Chapter 1

Introduction

Studies carried out by the World Health Organization show that 1.25 million deaths occur every year due to road traffic accidents, which lead to a decrease of 1 % to 2 % of gross domestic product from every country in the world [1]. Due to these high fatality rates and monetary costs, it is increasingly evident that advanced driver assistance systems (ADAS) — such as lane keeping, automatic braking, adaptive cruise control, autonomous driving, and many others — need to be an integral part of a modern vehicle's safety apparatus.

ADAS can be categorized by the type of sensor they use (figure 1.1) or the level of automation — as defined by the Society of Automotive Engineers (SAE) J3016 standard, which can be seen in table 1.1 [1].

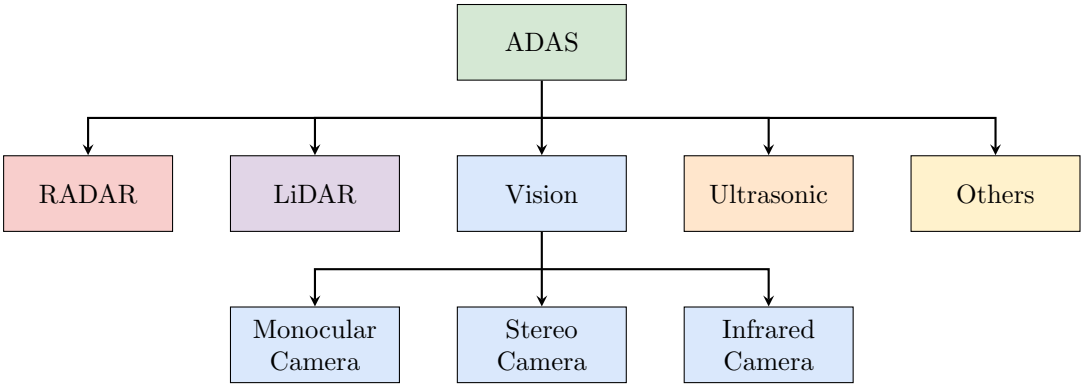


Figure 1.1: Taxonomy of ADAS based on the type of sensor (adapted from [1]).

Even though fully autonomous ADAS are still years away [2], level three systems are already seeing use in production vehicles [1].

Traffic scene detection, an integral part of an ADAS pipeline, can be separated into two subtasks: object detection, which aims at distinguishing variously defined objects — such as vehicles, pedestrians, and traffic lights —, and road/lane detection, which results in a deeper understanding of the drivable area, including road marking detection, lane detection, and road segmentation [2].

This work focuses on semantic road segmentation since it helps better understand the scene by predicting a meaningful class label for each sensor point [3].

Table 1.1: SAE J3016 levels of driving automation (adapted from [1]).

Level	**Definition**	**Monitoring of Driving Environment**
Zero	No automation	Human driver
One	Driver assistance	
Two	Partial automation	
Three	Conditional automation	System
Four	High automation	
Five	Full automation	

For this task, several sensors can be used to gather information about the scene. Camera-based systems (monocular or stereo) are the most common vision systems because of their low cost and similarity to human sight. Nonetheless, they are often unreliable, given the variety of road appearances, problems with image clarity, and poor visibility conditions. By employing active sensors such as light detection and ranging (LiDAR) sensors, that measure the distance from the reflection of emitted laser beams by the time of flight (TOF), semantic road segmentation can become robust to diverse environmental illumination while achieving higher accuracy [2].

Since ADAS perform safety-critical functions in a mobile environment, accuracy is necessary while not sacrificing real-time performance and power consumption. To evaluate and compare different methods of semantic segmentation, openly available datasets exist, such as the SemanticKITTI, which provides over 43 000 point-wise annotated full 3D LiDAR scans [4], and CityScapes, with 5000 pixel-level annotated RGB images [5].

With developments in advanced deep neural networks (DNN), it was recently possible to achieve a massive leap in accuracy and reliability in semantic segmentation. However, these models are computationally heavy, making them unsuitable for real-time deployments. Significant performance gains can be made using model compression techniques, such as pruning and quantization.

Graphics Processing Units (GPU) are a prominently used platform for parallel computing (suitable for tasks such as deep neural networks). However, their power consumption is very high, making them unsuitable for vehicles with limited power supply. Field Programmable Gate Arrays (FPGA) present a better alternative because of their lower power consumption and ability to perform massively parallel processing and data on-chip communication [2]. Tools such as Xilinx Vitis AI aim to provide an integrated development environment that can expedite the implementation of artificial inteligence (AI) inference on Xilinx FPGAs. The toolchain provides optimized intelectual property (IP) cores — Deep-Learning Processing Unit (DPU) —, libraries, and tools for this [6].

Objectives

The main objective of this dissertation is to develop hardware implementations of semantic segmentation DNNs with characteristics suitable for autonomous driving.

In this context, SalsaNext [3], a LiDAR-based semantic segmentation network, is implemented in a Xilinx Versal ACAP VCK190 FPGA board to obtain real-time performance (10 Hz in the case of LiDAR, which is the usual output rate of the sensors) utilizing Vitis AI 3.0. To further improve performance, channel pruning is applied to the model while reducing its impact on accuracy.

In order to verify the workflow, another model, Semantic Feature Pyramid Network (FPN) [7] with the MobileNetV2 backend, was also implemented.

Furthermore, these FPGA implementations' performance (latency and accuracy) and energy efficiency are evaluated and compared against a traditional GPU implementation.

Organization of the dissertation

Chapter 2 presents needed background information about how LiDAR works and the different interpretations of such signals, and a brief overview on Artificial Neural Networks (ANN) and FPGAs. It also lays out the state-of-the-art DNNs for semantic segmentation models, methods for implementing these networks in hardware, and introduces techniques for optimizing them, such as quantization and pruning.

Having laid out the theoretical background needed, chapter 3 addresses the problems with pruning a complex DNN and describes in detail the chosen pruning algorithm, accurate and automatic channel pruning (AACP), an evolutionary genetic algorithm. It also highlights implementation details, such as an efficient estimation of the computational complexity of a given pruning configuration and optimizations for managing the system memory when pruning.

Chapter 4 outlines the full workflow to implement the DNN in hardware by going through modifications made to the chosen models and the different pruning experiments done to them. Then, it describes the quantization process and the problems encountered when applying it. Finally, the hardware implementation of the quantized network and the methods for evaluating the final results are elaborated.

Chapter 5 shows the pruning results obtained from both models. Later, the quantization accuracy loss of the models' different pruned and unpruned versions is analyzed. Finally, the DPU implementation of the quantized models' throughput performance and energy efficiency of these different configurations is compared against the floating-point GPU versions.

Chapter 6 concludes the dissertation, taking away the main points of the research and suggests future work for further improvements for the implementation of FPGA-based inference of semantic segmentation neural network models.

Chapter 2

Background and State of the Art

2.1 Light Detection and Ranging

LiDAR works by emitting a laser beam at an object and then measuring its TOF, in order to calculate the distance to an object [1]. By firing such beams in all directions, a 360° 3D point cloud can be generated, typically at 10 Hz [3]. Figure 2.1 shows a visualization of a LiDAR output.

Figure 2.1: An illustration of a camera view and the corresponding LiDAR points (from [2]).

This unstructured data is commonly processed with two methods:

- Point-wise representation, that directly processes the raw 3D points without applying any additional transformation or pre-processing. These tend to perform well for small point clouds but do not scale well due to their processing capacity and memory requirement.
- Projection-based rendering, which transforms the 3D point cloud into another format like voxel cells, multi-view representation, lattice structure, and rasterized images. A common method is to project the 3D point clouds into 2D image space in a top-down or range-view (RV), i.e. panoramic view, format. Contrasted with point-wise representation-based approaches, 2D-rendered image representations are denser, more compact, and easier to perform computations, as 2D convolutional layers can be taken advantage of [3].

In order to obtain a 2D RV image, each 3D LiDAR point $\hat{p} = (x, y, z)$ can be mapped to spherical coordinates, and finally to image coordinates (u, v) as

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \frac{1}{2}[1 - \arctan(y,x)\pi^{-1}]w \\ [1 - (\arcsin(z, r^{-1}) + f_{\text{down}})f^{-1}]h \end{pmatrix}, \tag{2.1}$$

where the height and width of the projected image are given by h and w respectively, $f = f_{\text{up}} + f_{\text{down}}$ is the vertical field-of-view of the LiDAR sensor, and $r = ||\hat{p}||$ is the range of said point.

This image can be further enhanced by combining for each of its (u, v) pixels the original x, y, z values, the intensity and the range r as different channels, resulting in a $[w \times h \times 5]$ tensor, thus reducing the problem from point-cloud segmentation to image segmentation [8].

2.2 Artificial Neural Networks

ANNs are a type of computational model that tries to mimic the way the human (or, more generally, the mammal) brain works by layering interconnected nodes called perceptrons (basic neuron-like structures, commonly called neurons), which process and transmit information [9]. Each neuron receives input from other perceptron and applies a transformation, such as a nonlinear activation function, to produce an output. The output of one layer of neurons is passed as input to the next layer until the final layer produces the desired output of the network. This structure is shown by the 3-layer example in figure 2.2.

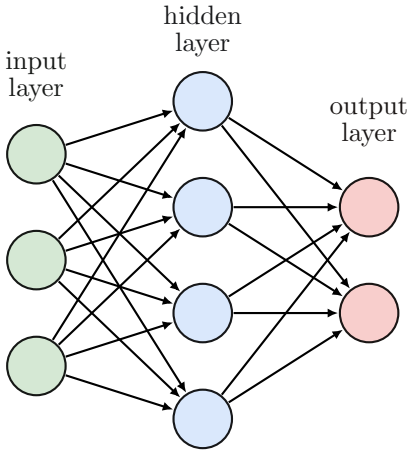


Figure 2.2: Visual representation of a fully connected ANN with three input neurons, four hidden neurons (given this name as their output value is not visible in the network's output), and two output neurons.

Every perceptron connection has an associated weight ω_i and bias b_i, which are linearly combined with the outputs of the previous neurons x and fed into an activation function f, resulting in the output y as is shown in equation (2.2).

$$y(x, \omega, b) = f\left(\sum_i x_i \omega_i + b\right) \tag{2.2}$$

These activation functions have to be non-linear because the opposite would result in being able to compute each output by a simple linear expression. Therefore generalization of the model would be impossible to achieve.

Machine learning can be achieved by adjusting these variables to minimize a predefined cost function. In 1986 a breakthrough in the field came under the name of the *backpropagation algorithm* [9], which allowed for the training of the neural networks with input-output samples. Since a neural network can be seen as a nonlinear parametric function, training a neural network does not significantly differ from training other parametric prediction models. The necessary components include training examples, a loss function to measure the model's accuracy, and an iterative optimization method like gradient descent to optimize the cost function. Training neural networks can be challenging due to their complex multilayer structure and difficulty in calculating gradients, which are necessary for optimization. Initially, the Heaviside activation function was commonly used in ANNs, which is discontinuous. To address this issue, a differentiable approximation of the Heaviside function, such as the logistic sigmoid function (shown in figure 2.3), was used for the development of a practical algorithm for training a neural network.

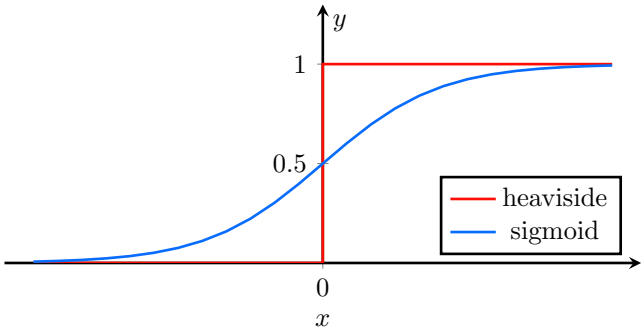


Figure 2.3: The Heaviside function (in red) and a sigmoid logistic function (in blue). As can be seen, the Heaviside function shows a discontinuity at $x = 0$.

Other activation functions have been proposed, such as the rectified linear unit (ReLU), defined as

$$f(z) = \max\{0,\ z\}, \tag{2.3}$$

with its derivative begin equal to one for $z > 0$ and equal to zero for $z < 0$. It has been shown that the ReLU's non-linearity can speed up the training process when applied to hidden layers of deep networks due to the derivate not slowing down when approaching small values and due to not saturating for positive values [9]. However, training freezes when $z < 0$. To solve this, the *leaky* ReLU was proposed, changing the function to have a negative slope α for inputs $z < 0$, usually set to a small value such as 1×10^{-2} [10], and resulting in

$$f(z) = \max\{0,\ z\} + \alpha \min \max\{0,\ z\}. \tag{2.4}$$

With the huge leaps in computer power and parallelization due to the use of GPUs, ANNs with several hidden layers, called DNNs, became widespread. Although more computationally expensive, they allow for much more complex models, which have allowed them to dominate the machine learning field since the 2010s [9].

2.2.1 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a type of neural network that learns features from data during the training process, rather than using pre-defined feature vectors like traditional ANNs. They gained recognition for their success in optical character recognition (OCR) tasks in recognizing handwritten digits [9]. They are named "convolutional" because they use convolutions in the first layers instead of the inner products used in traditional fully connected networks.

One reason CNNs were developed is that raw data can be too complex and large to be directly inputted into a network. For example, a one-channel image array with a resolution of 256×256 would produce an input vector of approximately 65 000 elements. If the first layer of the network has 1000 nodes, the number of parameters connecting the input to the first layer would be around 65 million. This number would increase even more for high-resolution images or images with more channels (such as color images). Such a large number of parameters can lead to overfitting, meaning the network performs poorly on new data. Additionally, vectorizing the image data also causes a loss of information about the relationships between pixels (i.e., locality), which can be crucial for learning [9].

Convolutional networks address these issues by combining multiple convolution layers. These layers perform their eponymous mathematical operation, which involves passing a filter or kernel over the data, effectively applying element-wise multiplication, and then summing the results. This process helps not only to extract crucial localized features from the raw data but also to preserve the spatial relationships between elements. Each layer can have any number of input and output channels. As shown in figure 2.4, a window of input data corresponding to the output value is multiplied element-wise with the kernel associated with the output channel, and the results are accumulated. Each input channel has as many kernels as the layer's output channels, and the result of each input channel is summed into a single set of output channels [9].

The network can learn patterns and features from the data without requiring many parameters by using them. Some operations can be added to improve the results. The nonlinearity step involves applying a nonlinear function to the output of the convolution step, and the pooling step involves reducing the data size by summarizing it in some way, such as taking the maximum or average value. These steps allow the network to learn more complex patterns and features from the data while maintaining a manageable number of parameters [9].

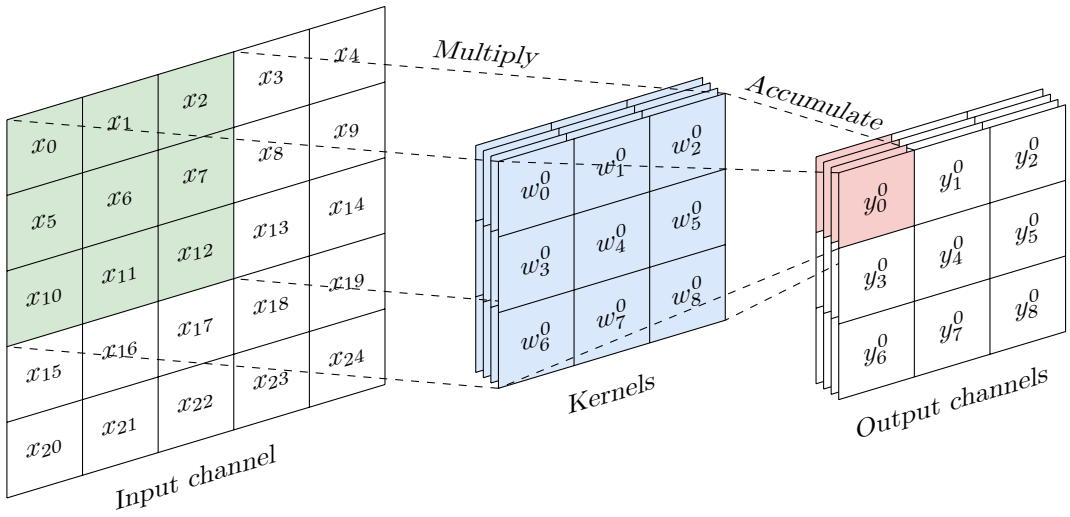


Figure 2.4: Illustration of the convolutional part of a 2D convolutional layer with a single input channel and three output channels.

2.2.2 Residual Networks

DNNs have become increasingly deep to cope with more complex problems. However, due to the multiplicative nature of the gradient calculation used in the backpropagation algorithm, if the values of the parameters get either too small or too big, it can cause phenomena known as vanishing and exploding gradients. If the gradients get vanishingly small, training can get extremely slow. On the other hand, if the gradient values explode, the network weights get pushed past their optimal region [9].

To simplify the weights' optimization, so-called residual building blocks, shown in figure 2.5, that combat the vanishing/exploding gradient problems were introduced [9]. These blocks group several layers and create two parallel paths, one with the standard transformations and one with the identity operation — often called shortcuts, skip connections, or residual connections. These two paths are merged together into a single output of the group. If the outputs of both paths do not have equal dimensions, the shortcut has to be changed from an identity to a matrix multiplication of the appropriate dimension.

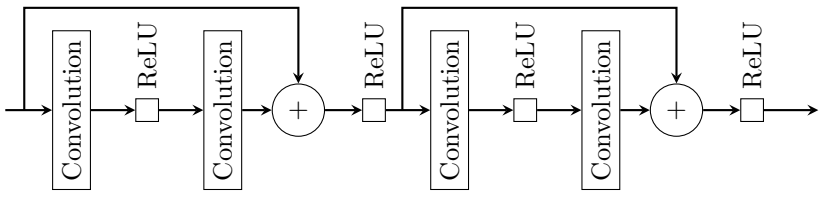


Figure 2.5: Layout of a CNN with the use of two residual blocks (adapted from [9]).

2.2.3 Batch Normalization

An essential contrast in the training process of DNN architectures versus single-layer models is the dynamic nature of the input distribution for each layer, as it changes through each iteration of parameter updates. This results in a slower training pace as it necessitates lower learning rates. It also increases the sensitivity of the training to initial parameter settings. One mitigation strategy for such issues is the application of batch normalization. It is commonly recommended during preprocessing to standardize input variables to have zero mean and unit variance, as this helps avoid significant dynamic range variations across the inputs. Batch normalization employs this idea, enforcing similar scaling on the activation values produced by all the network layers. Variations of this concept exist, with some implementing normalization before the nonlinearity, others after [9].

More concretely, this operation employs the transformation on the input x

$$f(x) = \gamma \frac{x - \mu_x}{\sigma_x + \varepsilon} + \beta, \tag{2.5}$$

with γ and β being the learnable parameters usually set to vectors of ones and zeroes, respectively and ε a very small value to avoid divisions-by-zero (in the order of 1×10^{-5}). μ_x and ε_x are the running estimates of the mean and standard deviation of the input x [9], [11].

2.3 Field Programmable Gate Arrays

FPGAs are configurable integrated circuits that can perform massively parallel computation and high throughput on-chip data communication. Their main components are configurable logic blocks (CLB) and look-up-tables (LUT). CLBs contain several LUTs, as well as a configurable switch matrix, selection circuitry (multiplexers), registers, and flip-flops. LUTs are essentially hardware versions of truth tables used in CLBs to implement any desired boolean function. They can also be used as small memories or small random access memorys (RAM). Static Random Access Memorys (SRAM) blocks can be found throughout the programmable logic (PL) fabric and can be connected to create larger, deeper memories or RAMs [12].

One of their main advantages is their flexibility. Unlike traditional application specific integrated circuits (ASIC), designed to perform a specific function and cannot be easily reprogrammed, FPGAs can be reconfigured to perform a wide range of tasks. This allows for faster design cycles and the ability to make changes to a system without the need for expensive redesigns of ASICs.

Despite their many benefits, FPGAs have some limitations. They are generally more expensive than general-purpose microprocessors and have a longer development time than traditional programming, requiring specialized hardware description languages such as VHDL or Verilog. In addition, FPGAs are not as easily scalable as ASICs, which can be more efficient for large-scale production.

FPGAs also offer faster performance than a microprocessor — central processing unit (CPU) or GPU — in certain applications, as they can be optimized to perform specific tasks in parallel rather than sequentially. They also consume less power and generate less heat than a general-purpose

microprocessor for specific computational tasks, making them well-suited for use in portable or battery-powered devices.

Given this, they are ideal for the processing of LiDAR 3D point clouds and the inference of DNN in a power-constrained real-time environment such as the one in ADAS [2].

There has been a trend towards the integration of PL, CPU — commonly called processing system (PS) —, and software-programmable acceleration engines into a single device, known as an Adaptive System on a Chip (SOC). These devices combine the flexibility and performance benefits of FPGAs with the scalability and cost advantages of ASICs, making them an attractive choice for a wide range of applications [13].

One such example is Xilinx's Versal™ Adaptive Compute Acceleration Platform (ACAP) architecture [13] (see figure 2.6), which offers a large PL (with 900 000 LUTs and 158 Mbit of RAM), a PS with two dual-core ARM CPUs, a Cortex-A72 application processor and a Cortex-R5 real-time processor, digital signal processor (DSP) engines, network on chip (NoC) capabilities, and 400 AI engines [14].

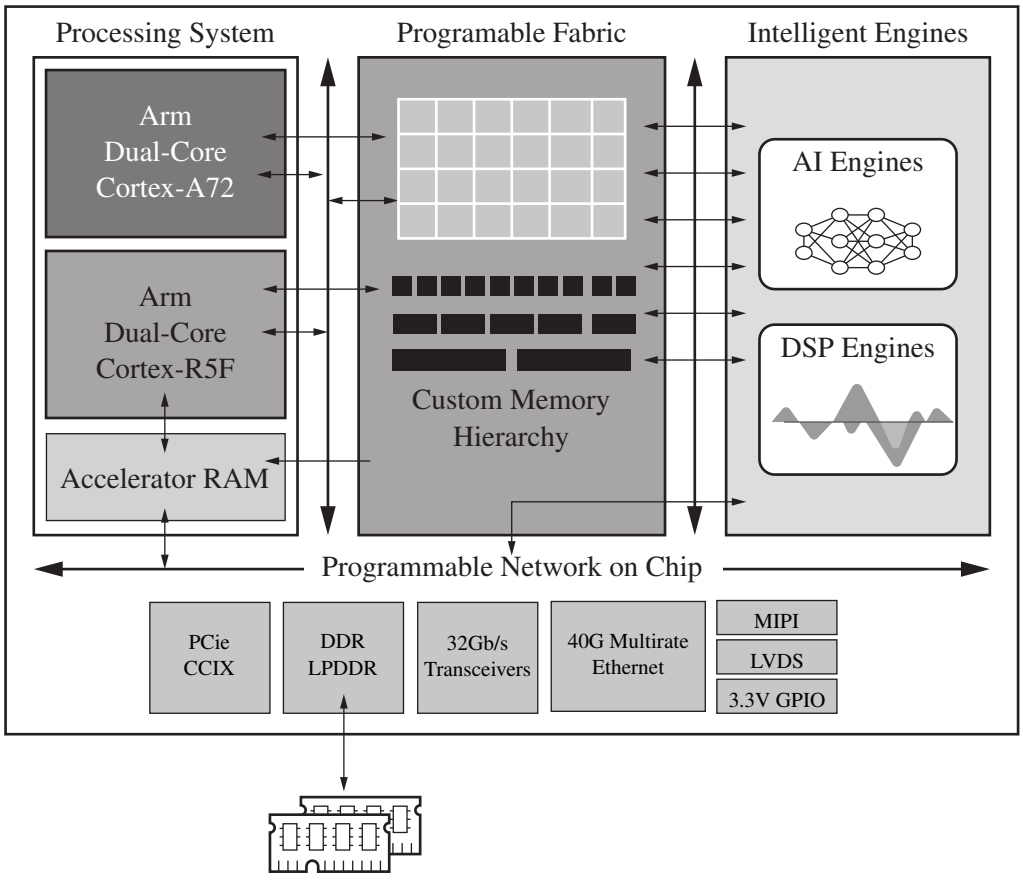


Figure 2.6: Versal architecture diagram (adapted from [15]).

2.4 LiDAR 3D Point Cloud Semantic Segmentation

Semantic segmentation of 3D LiDAR point clouds has seen significant progress in recent years, with notable improvements in both the accuracy and speed of processing [3], [8], [16]–[20]. However, the field continues to grapple with the challenge of achieving the optimal balance between these two conflicting objectives.

The availability of large datasets is essential for the progress of computer vision. Since there was a lack of such a dataset for LiDAR point cloud semantic segmentation in automotive scenes, Behley *et al.* [4] introduced a dense point-wise annotated version of the KITTI Vision Odometry Benchmark, called SemanticKITTI, which consists of 22 sequences of real LiDAR scans (using the Velodyne HDL-64E sensor) of roads in and around Karlsruhe, Germany, making up 43 000 scans. These scenes were annotated with 28 classes of interest for ADAS. This dataset is relevant not only for developing new models but also for their benchmarking to properly compare them in speed and accuracy under the same conditions.

The most widely used metric in this field, popularized in [21], is the mean intersection-over-union (mIoU), is defined as

$$\text{mIoU} = \frac{1}{C}\sum_{c=1}^{C}\frac{\text{TP}_c}{\text{TP}_c + \text{FP}_c + \text{FN}_c}, \tag{2.6}$$

where TP_c, FP_c and FN_c are, respectively, the number of true positive, false positive, and false negative predictions for a given class c, and with C being the total number of classes [4].

The methods for LiDAR point cloud semantic segmentation differ in their network structure and how they represent the data, as was explained in section 2.1. Point-wise methods benefit from the fact that they do not require pre-processing of the data and are shown to achieve good results with the original PointNet [22] network and its subsequent version, PointNet++ [23]. However, these were shown not to scale computationally well enough for high-resolution scans.

Projection-based methods, on the other hand, involve transforming the 3D point cloud into a 2D representation. Some methods, like PolarNet [23], use birds eye view (BEV) projection in polar coordinates, while others, like the SqueezeSeg familly [17]–[19], RangeNet++ [8] and SalsaNext [3] use a spherical projection to an image, which is then processed using standard 2D convolutional networks for semantic segmentation. To attain state-of-the-art results in 3D LiDAR semantic segmentation, SalsaNext [3] implemented several improvements to the SalsaNet [20] network, such as the incorporation of a new global context block and enhancements on the encoder-decoder blocks.

As seen in the accuracy comparison in table 2.1, projection-based approaches tend to perform better, and SalsaNext's results stand out.

Table 2.1: LiDAR-based semantic segmentation mIoU results on the SemanticKITTI test dataset (adapted from [24]).

Type	Model	mIoU (%)
Point-based	PointNet	14.6
	PointNet++	20.1
	SPGraph	20.0
	SPLATNet	22.8
	TgConv	37.0
	RLNet	50.3
Projection-based	SqueezeSeg	29.5
	SSG-CRF	30.8
	SSGV2	39.7
	SSGV2-CRF	39.6
	BiseNet	41.4
	SalsaNet	45.4
	RangeNet21	47.4
	Xie *et al.*	47.9
	RangeNet53	49.9
	RangeNet53++	52.2
	SSGV3	55.9
	SalsaNext	59.5

2.5 Hardware implementation of Semantic Segmentation Networks

Due to the real-time requirements of ADAS, they are often implemented in embedded platforms such as mobile CPUs/GPUs processors, FPGAs, and ASICs [24]. However, the inference process of DNNs is very computationally expensive, as it depends on a vast number of multiply and accumulate (MAC) operations — equation (2.2) —, and requires at least two input values (the connection weight and the output value of the preceding neuron) that must be fetched from memory. Similarly, the inference of convolutional layers is costly due to its matrix-multiplication nature. Therefore, CPUs are not enough for large models since they perform operations sequentially. GPUs, which were initially designed to perform massive matrix operations for graphical rendering, can be very performant for this use case, especially on the high end, but exceed in many cases the power envelope for embedded applications [25].

On the other hand, FPGAs and ASICs are very flexible, which can lead to considerable gains in performance when compared to the former two. Great research effort has been targeted at FPGA implementations of DNNs due to their reconfigurable nature and ease of design when compared to ASICs.

Several high level synthesis (HLS) tools were developed for synthesizing DNN inference accelerators to improve performance and further explore the possible design space of such implementations, such as *Haddoc2* [26], *fpgaConvNet* [27], *FINN-R* [28], *hls4ml* [29], *NVDLA* [30] and *Vitis AI* [6]. These aim at turning a high-level description of the network (*FINN-R*, *NVDLA*, and *Vitis AI* can even take models built with standard machine learning frameworks like *PyTorch* and *TensorFlow*) into optimized deployable accelerators.

Xilinx's *Vitis AI* [6] is designed to fully utilize the potential of the AI engines present in the Versal ACAP architecture and provides optimized IP cores — DPUs —, tools, libraries, models, and example designs to simplify the development process. It also contains tools such as the *Vitis AI Optimizer*, which can, with a combination of pruning and finetuning, reduce the model's complexity without degrading its accuracy; and *Vitis AI Quantitizer*, which reduces the model's precision from 32-bit floating-point weights to fixed-point that not only require less bandwidth but also can better leverage the resources present in the FPGA. It offers a complete design flow as seen in figure 2.7.

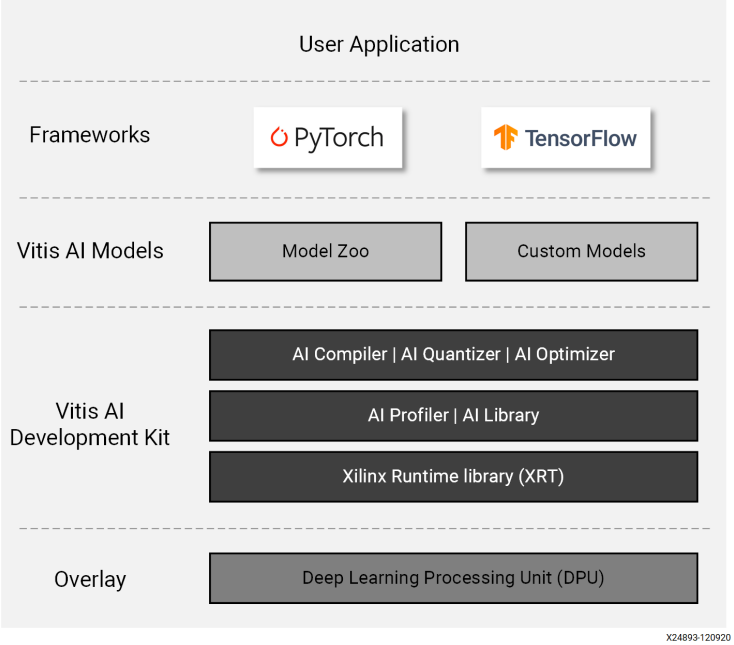


Figure 2.7: Vitis AI development stack (from [6]).

The DPUCVDX8G is engineered to optimize inference on general-purpose CNN on Versal ACAP devices. It is made of a combination of both AI Engines and PL components. The convolution process is entrusted to the AI Engines in the FPGA, with data transfers between the AI Engine memory tiles and the PL facilitated by the AI Engine interface tiles. In certain high-performance Versal devices, groups of contiguous AI Engines make up the AI Engine groups for enhanced computational performance. The PL component, on the other hand, provides a scheduler module, global memory dedicated to shared weights, and batch handlers for some basic operations. Figure 2.8 shows a top-level block diagram of the DPU.

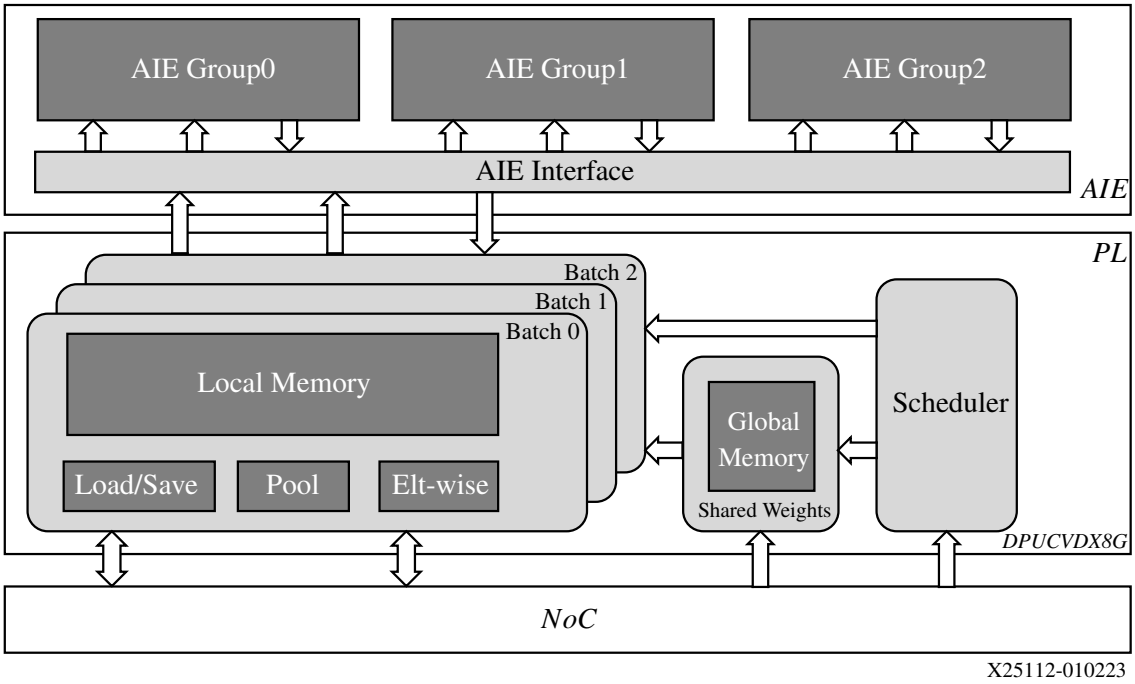
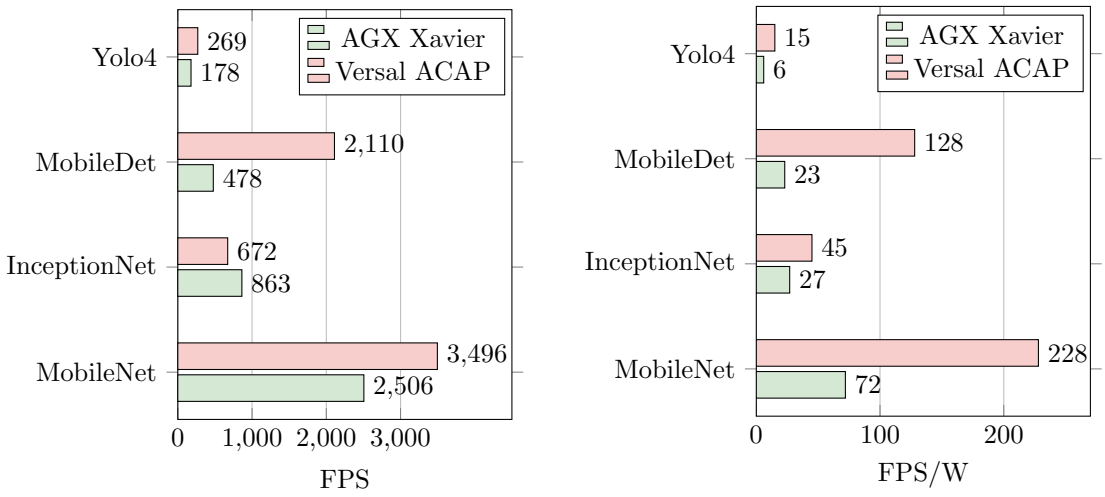


Figure 2.8: Top-level block diagram of the DPUCVDX8G (from [31]).

Szántó *et al.* [32] benchmarked *Vitis AI* accelerators for the MobileNet, InceptionNet, MobileDet, and Yolo4 models on a Versal ACAP device, and compared them to the same models running on an NVIDIA Jetson AGX Xavier, a GPU-based embedded system on a module for AI inference. From the results shown in figure 2.9, it can be concluded that depending on the model, the Versal-based implementations can have extensive performance and power efficiency gains over the AGX Xavier ones. However, this is less than half the theoretical potential of the AI engines. It should be noted that these models only used the quantization process provided in *Vitis AI*, and did not employ any pruning techniques.

Xie *et al.* [24] with an FPGA-targeted (Xilinx Zynq UltraScale+ MPSoC ZCU104) CNN using an *NVDLA* [30] architecture, were able to achieve a mIoU of 46.4 % with a frame processing time of 17.76 ms — an improvement by a factor of 2.17 when compared to the same network running on a high-end desktop GPU — when evaluating on the SemanticKITTI benchmark. Inspired by ContextNet, BiSeNet, and SalsaNext, the proposed model used a multi-branch design, combining extracted spatial and contextual features. The input pre-processing (projecting the raw LiDAR points) was done in software running in the PS and then passed to the inference accelerator via the AXI4 bus. This work shows great promise for the further development of such an approach, since Xilinx's more recent Versal ACAP, with its AI engines, could increase the gains in performance and allow for more complex networks, leading to better accuracy.



(a) Perfomance comparison in frames per second (FPS).

(b) Power efficiency comparison in FPS per Watt.

Figure 2.9: Xilinx Versal vs NVIDIA AGX Xavier AI inference benchmarking results (adapted from [32]).

2.6 Deep Neural Network Optimization for Hardware Implementations

Several optimization techniques exist for accelerating neural networks on FPGAs and ASICs, including weight pruning, quantization, and model compression. They aim to reduce the computational complexity and power consumption of neural networks, allowing them to meet real-time goals and making them more suitable for deployment in resource-constrained environments such as embedded systems for ADAS [25].

2.6.1 Quantization

Quantization is a technique used to decrease memory and computing requirements by converting continuous or large sets of data into a smaller, discrete set through the use of a linear or non-linear function. By reducing the precision of values, the number of bits needed to represent them is reduced, making it possible to reduce memory consumption and bandwidth and utilize more efficient logic [25].

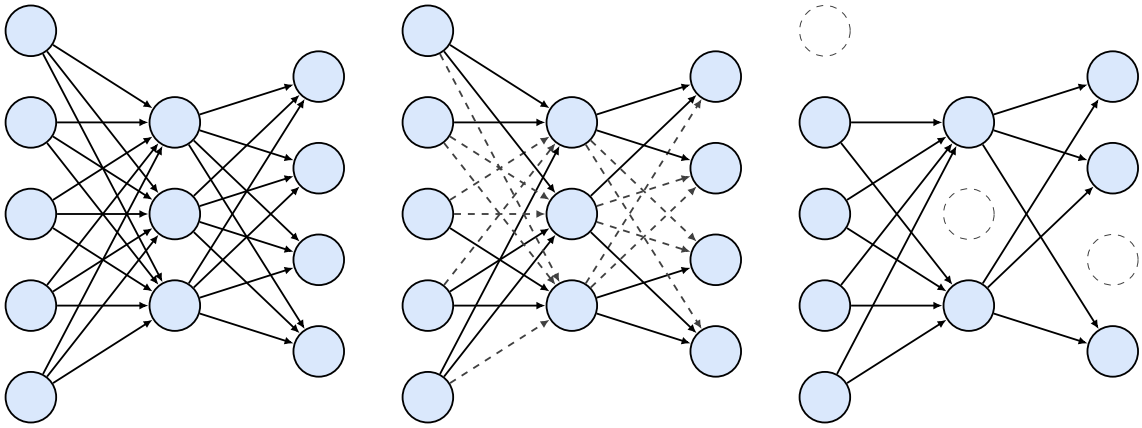
According to the IEEE 754 standard, a 32 bit floating-point value $(-1)^s \cdot m \cdot 2^{e-127}$ is represented by the bit string made up of the 1 bit sign s, the 23 bit exponent e and the 28 bit mantissa m. This makes it possible to represent numbers ranging from 10^{-38} to 10^{38}. On the other hand, fixed-point representations in two's complement have an integer part (of width I) and decimal part (of width D, called the scale factor), where D determines the position of the decimal point and the precision of the representation. This makes them able to represent numbers from -2^I to $(2^{I-1} - 2^{-D})$, with a quatitization step of 2^{-D}. Since the size of D is flexible, this representation can be tailored for

specific ranges. This is useful for neural networks because it allows adaptation to different ranges of weights and activations in different layers.

However, quantization can lead to a decrease in the model's accuracy. Therefore, the quantized weights and the scale factors must be fine-tuned after the quantization of the model.

2.6.2 Pruning

Some connections and neurons in an ANN may have a negligible contribution to the final output. Network pruning works by removing such connections, thus reducing the computational complexity of inference while trying to minimize accuracy loss as much as possible. There are essentially two pruning methods, seen in figure 2.10: fine-grained and coarse-grained pruning [25], [33].



(a) Original network (b) Fine-grained pruning (c) Coarse-grained pruning

Figure 2.10: Visualization of fine-grained and coarse-grained pruning. Connections and neurons in gray dashes represent pruned elements.

Fine-grained pruning, also known as sparse pruning, removes redundant weights, which results in the creation of sparse matrices. However, it requires specialized hardware and techniques for weight skipping and compression. At this time, *Vitis AI* has not implemented this type of pruning [33] due to their DPU not supporting weight skipping.

Coarse-grained pruning, some times referred to as structured pruning, on the other hand, involves the elimination of neurons that do not significantly contribute to the accuracy of the network. When applied to convolutional layers, it is called channel pruning. This method involves pruning entire convolution channels, i.e., changing the number of either input or output channels. Its advantage is that it does not require the use of specialized hardware for the acceleration of inferences.

Pruning always reduces the accuracy of the original model. Retraining or fine-tuning can be used to adjust the remaining weights to recover the accuracy.

Kang [34] introduced a new pruning scheme for CNNs that takes into account ASIC and FPGA *sparse* accelerator architectures, since inefficiencies may arise when not taken into consideration,

such as internal buffer misalignments and load imbalances. Therefore, it aims at balancing weight fetching for simultaneous activations by keeping the same number of weights in each group.

Li *et al.* [35] proposed AdaPrune, a novel approach for the same problem that promises to improve both performance and energy efficiency while not degrading accuracy. It uses two techniques: input and output channel group pruning. By alternating between these two, it aims at evenly distributing the zeros in each weight-fetching group.

MetaPruning [36] is a channel pruning approach that requires the training of a *PruningNet*, a meta-network that can generate weights for any network pruning configuration. Then, by applying an evolutionary procedure, it tries to maximize the inference accuracy of the population.

Lin *et al.* [37] proposes ABCPruner, an Artificial Bee Colony (ABC) algorithm that tries to find the optimal channel pruning structure by first shrinking the search space by randomly pruning an upper-bounded percentage of the channels of each layer and then using the ABC algorithm to optimize the structure while fine-tuning each candidate for better inference accuracy estimation.

Li *et al.* [38] introduces EagleEye, a pruning algorithm that randomly generates pruning strategies and chooses the winner based on *Adaptive Batch Normalization*, which works by recalibrating the model's batch normalization layers and then calculating the accuracy on the dataset. As such, it avoids training and substantially speeds up the candidate's fitness calculation since no backpropagation is needed to update the batch normalization statistics. Xilinx Vitis AI Optimizer implements this method, although it is proprietary, and no results have been published.

AACP [39] implements a proposed improved diferential evolution (IDE) genetic algorithm to search through the possible configurations of the model. The method prunes the model until reaching a target number of floating-point operations (FLOP) and parameters by randomly selecting a layer and removing the channel one with the lowest L_1 norm — the sum of all weights — to generate an individual. Then, to determine the fitness of a said individual, Lin *et al.* proposes the *Pruned Structure Accuracy Evaluator*. This technique follows the same procedure as *Adaptive Batch Normalization* in [38].

Most pruning algorithms have to be implemented or customized for a specific model, which can be difficult for large ones with many interdependencies, such as residual connections. Fang *et al.* [40] tackles the problem of structurally pruning an arbitrary architecture by introducing *Dependency Graph*, a fully automatic method for implementing a generic pruner for any network. The paper also proposes a pruning method called GroupNorm, which prunes channels based on the L_n norm of an interdependent group of layers instead of a single one.

The comparison in table 2.2 shows that AACP obtains good accuracy while providing a flexible architecture that can be further explored. GroupNorm also stands out since, despite having a simple method for searching the optimal architecture, it provides good results due to being group-based by implementing *Dependency Graph*. The results of EagleEye could not be directly compared since the paper does not provide the original accuracy of the models. However, it shows promising results since it achieves 77.1 % and 76.4 % and 74.2 % with pruning rates of 25 %, 50 % and 75 % on the ResNet-50 model with the ImageNet dataset.

Table 2.2: Pruning methods comparison with ResNet-50 and MobileNetV2 models on the ImageNet Dataset (adapted from [39], [40]).

CNN	Method	FLOPs(%)	Base Top-1(%)	ΔTop-1(%)	Base Top-5(%)	ΔTop-5(%)
ResNet50	ThiNet-70	36.80	72.88	-0.84	91.14	-0.47
	AACP	42.00	75.94	-0.06	92.74	-0.1
	FPGM	42.20	76.15	-0.56	92.87	-0.24
	HRank	43.80	76.15	-1.17	92.87	-0.54
	SRR-GR	44.10	76.13	-0.37	92.86	-0.19
	DMCP	46.30	76.60	-0.40	—	—
	PFS	51.20	77.20	-1.60	—	—
	AutoSlim	51.20	76.10	-0.50	—	—
	MetaPruning	51.20	76.60	-1.20	—	—
	AACP	51.70	75.94	-0.48	92.74	-0.43
	GroupNorm	51.82	76.15	-0.32	—	—
	FPGM	53.50	76.15	-1.32	92.32	-0.55
	ABCPruner	54.30	76.01	-2.15	92.96	-1.27
	SRR-GR	55.10	76.13	-1.02	92.86	-0.51
	ThiNet-50	55.80	72.88	-1.87	90.02	-1.12
MobileNetV2	MetaPruning	27.70	72.00	-0.80	—	—
	AMC	29.70	71.80	-1.00	—	—
	AutoSlim	29.70	74.20	-1.20	—	—
	DMCP	29.70	74.60	-1.10	—	—
	Uniform 0.75x	30.00	71.80	-2.50	90.5	-1.7
	PFS	30.00	72.10	-1.20	—	—
	AACP	30.20	71.80	-0.69	90.5	-0.6
	GroupNorm	54.55	71.87	-3.41	—	—
	MetaPruning	57.58	74.70	-6.50	—	—

Chapter 3

Development of the Channel Pruning Algorithm

A custom version of the AACP [39] algorithm was implemented for pruning the DNN models. Lin *et al.* provide a repository containing the source code of their implementation. However, the authors tailored this code to work with the models they studied in their original paper, and, therefore, it could not be used for the development of the work of the dissertation, even though it proved helpful to have a deeper understanding of the algorithm.

This algorithm uses an improved version of diferential evolution (DE) (an evolutionary genetic algorithm) to find an optimal model architecture in a search space given by a set of constraints, such as the number of FLOPs required for the model's inference or the number of parameters in its weights.

The model's channel architecture represents the solutions generated by the AACP algorithm, which is expressed in vector form as $C = [c_1 \;\; c_2 \;\; \dots \;\; c_L]$, with c_i referring to the output channel number of the i-th convolutional layer of the model. This representation allows for more general and straightforward manipulation of the underlying architecture instead of dealing with the complex representations of the layers in the commonly used machine learning frameworks (such as *PyTorch* or TensorFlow).

For the example given in figure 3.1, the architecture of the model is represented by the vector $C_1 = [32 \;\; 32 \;\; 64]$. By pruning the first layer in half (i.e., going from 32 to 16 output channels), the architecture vector would be transformed to $C_1' = [16 \;\; 32 \;\; 64]$.

Most complex ANN models do not follow an exclusively sequential structure. Instead, they often incorporate residual connections (section 2.2.2) or other interdependency mechanisms, creating dependencies among the multiple layers. If these dependencies are not adequately addressed, the resulting representation vector of the network, post-pruning, may be rendered invalid. More concretely, for the example in figure 3.2, the initial architecture vector remains $C_2 = C_1 = [32 \;\; 32 \;\; 64]$. With the same pruning strategy as the previous example, the result would be $C_2' = C_1' = [16 \;\; 32 \;\; 64]$. However, the input of the third layer is an aggregate of the outputs of the first and second layers.

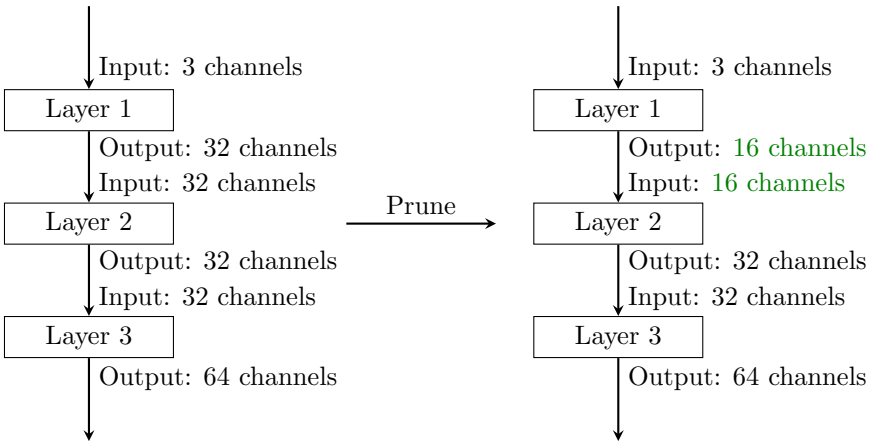


Figure 3.1: Example of pruning of a simple sequencial CNN model. Layer 1's number of output channels is pruned from 32 to 16.

This creates a discrepancy in the dimensionality of the outputs (16 channels cannot be summed with 32 channels), rendering the model invalid.

Moreover, it is typical for other parameterized layers to be present in the model, be they in the middle of convolutional layers, such as batch normalization layers, or at the end, as is usual with fully connected layers. These layers also need to be resized to keep the model's integrity.

In order to solve this problem in the original AACP, Lin *et al.* [39] studies the ResNet and VGG models, whose well-defined repeating structure is easy to distill into a configuration array. Such an approach makes the method not very generalizable since DNNs have become increasingly complex structures and might be made of different sub-networks, as in transfer learning methods, thus limiting its ease of use.

A more general approach involves creating a graph of the dependencies of every layer in the network. The recently-released *VainF/Torch-Pruning* [41] *PyTorch* library provides an easy-to-use application programing interface (API) for breaking down a complex DNN into independent groups of inter-dependent layers by constructing and analyzing a dependency graph of the ANN. In addition, it also provides methods to prune specific channels of a group, dealing with the complexity of keeping the model dimensionally coherent, and methods to determine the importance of the group's channels by calculating the L_n-norm. This library was instrumental in implementing the pruning algorithm.

Some inference platforms like the DPU for the Xilinx Versal ACAP can process multiple channels of a layer in parallel [42]. However, pruning the layer to yield a number of output channels not divisible by the platform's parallel-processing capability leads to inefficiencies. For example, a platform capable of processing 16 channels in parallel will not speed up if a 32-channel layer is pruned to 17 channels since it still requires two processing passes, one for 16 channels and another for the remaining channel. Therefore, the genetic algorithm's search space is confined to pruning configurations that adhere to these considerations.

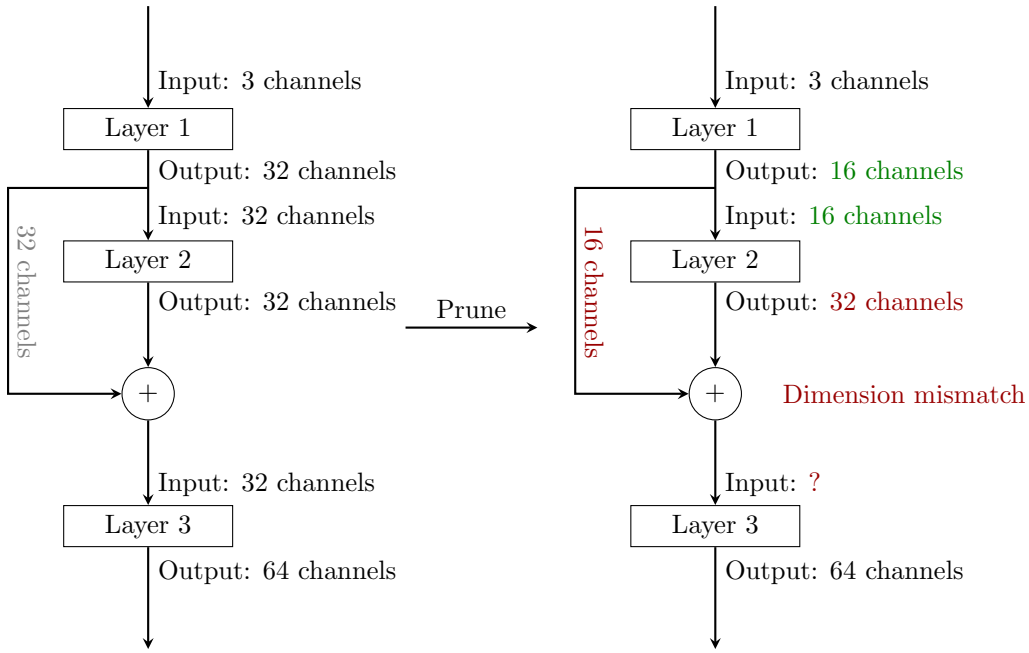


Figure 3.2: Example of a naïve approach to pruning a CNN model with a residual connection. Layer 1's output is pruned from 32 to 16 channels. However, Layer 3's input depends on both Layer 1 and Layer 2. Therefore, this pruning strategy is incorrect since summing 32 channels with 16 channels is impossible.

3.1 Description of the Algorithm

The AACP algorithm, described in algorithm 1, is a genetic algorithm using IDE [39]. The population is a list of individuals P^i, with i being the current iteration (i.e., generation) of the genetic algorithm:

$$P^i = \{ P_1^i \;\; P_2^i \;\; \dots \;\; P_N^i \} \tag{3.1}$$

Each individual has a set of genes corresponding to each interdependent layer group's number of output channels.

In the following sections 3.1.1 to 3.1.6, each step of the algorithm is explained.

3.1.1 Initialization

Every individual at iteration $i = 0$ is initialized to have the same configuration vector as the original model and is then pruned as described in algorithm 2.

The L_1 norm was used as the metric to choose which channel to prune in a group to achieve a lower drop in accuracy.

Algorithm 1 Implemented AACP algorithm

Input: FLOPs prune ratio R_f, population size N, number of iterations I, differential weight F, crossover rate CR, reinitialization rate R, pruning step S, original model M
Output: Optimal pruned model M'

$P^0 \leftarrow$ INITIALIZEPOPULATION(M,N,R_f,S) — section 3.1.1
for $i = 1,\ldots,I$ **do**
  for $n = 1,\ldots,N$ **do**
    MUTATION — section 3.1.2
    CROSSOVER — section 3.1.3
    BOUNDING — section 3.1.4
    SELECTION — section 3.1.5
    REINITIALIZATION — section 3.1.6
return $M' = \underset{p \in P^I}{\text{argmax}}\ \text{FITNESS}(p)$

3.1.2 Mutation

From three randomly selected individuals from the previous iteration, P_x^{i-1}, P_y^{i-1} and P_z^{i-1}, a candidate for the iteration i is generated:

$$A_n^i = P_x^{i-1} + F \times (P_y^{i-1} - P_z^{i-1}), \tag{3.2}$$

with $F \in [0,2]$ being the diferential weight.

3.1.3 Crossover

Each gene only has a probability CR of crossing over to the individual of the next iteration, thus introducing gene-level variety. By generating a random number RANDOM() from a uniform distribution between 0 and 1, the gene j is only passed on with a probability $CR \in (0,1]$,

$$B_n^i[j] = \begin{cases} A_n^i[j], & \text{if RANDOM()} < CR \\ P_n^{i-1}[j], & \text{otherwise.} \end{cases} \tag{3.3}$$

Algorithm 2 Prune until target

Input: Individual p, FLOPs prune ratio R_f, pruning step S, original model M
Output: Pruned individual p_{pruned}

while FLOPs(p) > $(1-R_f)\times$ FLOPs(M) **do**
  $j \leftarrow$ RANDOM(0, LEN(M))
  if $p[j] > S$ **then**
    $p[j] \leftarrow p[j] - S$
return $p_{\text{pruned}} = p$

3.1.4 Bounding

The generated candidate might not be inside the search space, i.e., with the channel number not being a multiple of S, the pruning step, or not be below the targeted number of FLOPs. Therefore, the number of output channels in each group j is rounded down to the nearest positive integer multiple of S and then pruned until the targeted number of FLOPs is reached if the candidate is above the threshold:

$$C_n^i[j] = \max\left\{\left\lfloor \frac{B_n^i[j]}{S} \right\rfloor \times S,\ S\right\}, \quad \forall j \tag{3.4}$$

$$D_n^i = \begin{cases} C_n^i, & \text{if } \text{FLOPs}(C_n^i) < (1-R_f) \times \text{FLOPs}(M) \\ \textsc{PruneUntilTarget}(C_n^i, R_f, S), & \text{otherwise.} \end{cases} \tag{3.5}$$

3.1.5 Selection

The candidate is compared to the previous generation's individual, and the one with the highest fitness value is chosen. The calculation of the fitness is done in two steps:

1. Zero the batch normalization layer's parameters and do a forward pass of a few thousand samples without needing to calculate the gradients for backpropagation, but keeping the model in training mode so that the running means and standard deviations are calculated.

2. Do another forward pass of the dataset and calculate the model's fitness based on accuracy or any other metric. Cross entropy loss was chosen for this, with lower values being interpreted as fitter than higher ones.

$$P_n^i = \begin{cases} D_n^i, & \text{if } \textsc{Fitness}(D_n^i) > \textsc{Fitness}(P_n^{i-1}) \\ P_n^{i-1}, & \text{otherwise.} \end{cases} \tag{3.6}$$

3.1.6 Reinitialization

In order to explore new locations in the search space and not get stuck in local optima, AACP, as a final step in the iteration, reinitializes the individual to a random position if it has remained unchanged for the past R iterations:

$$P_n^i = \begin{cases} \textsc{InitializeIndividual}(M, R_f, S), & \text{if } P_n^i = P_n^{i-1} = \cdots = P_n^{i-R} \\ P_n^i, & \text{otherwise.} \end{cases} \tag{3.7}$$

3.2 Implementation Considerations

The algorithm was implemented in *Python* with the target of pruning *PyTorch* models. It was given a modular interface, allowing it to be used for any CNN model without any modifications. For this reason, the pruner expects some model-specific parameters:

- the unpruned *PyTorch* model;
- a sample input in order to create the dependency graph with *VainF/Torch-Pruning* [40] and calculate the number of FLOPs;
- a function to calculate of a given *PyTorch* model;
- a function to update the batch normalization layers;
- a function that generates a list of layers that should be untouched by the pruner.

The latter parameter must be a function and not simply a list since, internally, the pruner creates a copy of the model. If this were not the case, the modules referenced by the list would point to the original model, not the copy.

Appendix A shows the full algorithm implementation.

3.2.1 FLOPs Estimation

The initialization and bounding step proved to be unexpectedly computationally expensive due to the necessity of recalculating the number of FLOPs for every pruning step when performing the *prune until target* operation, as described in algorithm 2. Since *PyTorch* models are defined in runtime and are independent of the size of the input, the most common way to calculate the number of FLOPs performed when inferring is to attach a pre-hook function (a function that runs before a module's execution) to each module. This pre-hook function calculates the FLOPs based on the type of module. Then, a forward pass of the model is performed, and the results of the pre-hook function calls are accumulated. Since it occupied a significant percentage of the algorithm's runtime, an estimator for the change in the number of FLOPs after pruning an arbitrary amount of channels in an interdependent group of layers was devised.

Assuming that the majority of the inference computation of a CNN is spent on convolutional layers, targeting a pruning ratio of the number of FLOPs of the whole model is approximately the same as targeting the same ratio of just the convolutional layers.

In a 2D convolutional layer j, the result of the channel i is

$$\text{out}(C_{\text{out}_i}^{j}) = \text{bias}^{j}(i) + \sum_{k=0}^{C_{\text{in}}^{j}-1} \text{weight}^{j}(i,k) \star \text{input}^{j}(k), \tag{3.8}$$

where C_{in}^{j} and C_{out}^{j} are the number of input and output channels, respectively [43] and $\star$ is the 2D cross-correlation operator. Also assuming that the bias summations are much lesser than the FLOPs of the cross-correlation part, the whole layer's amount of FLOPs is approximately linearly dependent to the number of input and output channels.

Therefore, starting from a complete calculation of the original model's layer j's quantity FLOPs, f_j, using the pre-hook method, a constant factor

$$F_j = \frac{f_j}{C_{\text{in}}^{j} C_{\text{out}}^{j}} \tag{3.9}$$

can be calculated. With this constant, the estimated number of FLOPs after pruning is

$$f'_j = F_j(C^j_{\text{in}} - \Delta C^j_{\text{in}})(C^j_{\text{out}} - \Delta C^j_{\text{out}}). \tag{3.10}$$

The estimator was implemented with the *Flopth* [44] library by using its internal methods and classes that generate a list of leaf modules and the corresponding quantity of FLOPs. A dictionary indexed by the module's name stores the quantity of input and output channels, the original number of FLOPs, and the calculated constant factor F_j of each convolutional leaf module. When pruning a group, the estimator goes through every model, updates the internal channel counters if they are to be pruned (determined by the *VainF/Torch-Pruning* library), and calculates the FLOPs estimation with equation (3.10).

In order to verify the performance and effectiveness of the estimator in comparison to the method of adding the pre-hook functions and doing the forward pass, a test was devised in which the *prune until target* function (algorithm 2) was run 50 times for both methods and varying the target prune ratio on the ResNet18 and ResNet101 models [45] with the CIFAR-10 dataset [46] input size. For each run, the achieved pruned ratio and the process time (the sum of system and CPU times of the process, measured with *Python*'s time library [47]) were measured and averaged, with the results shown in table 3.1. The tests were performed on a computer with an AMD Ryzen Threadripper 1920X 12-Core Processor, with 64 GiB of system memory and an NVIDIA GeForce RTX 2070 8 GB.

Table 3.1: FLOPs estimator verification test results on the ResNet-18 and ResNet-101 models using the CIFAR-10 dataset.

Model	**Method**	$R_f(\%)$	**Process Time**(s)	**Achieved** $R_f(\%)$	**Relative Error**(%)	**Speedup**
ResNet-18	Forward pass	25	4.45	25.11	-0.42	1×
		50	10.24	50.08	-0.15	1×
		75	21.67	75.02	-0.02	1×
	Estimator	25	0.21	24.80	0.80	20.82×
		50	0.45	49.52	0.97	22.81×
		75	0.96	74.29	0.94	22.55×
ResNet-101	Forward pass	25	234.11	25.01	-0.03	1×
		50	548.96	50.00	-0.01	1×
		75	1007.53	75.00	0.00	1×
	Estimator	25	3.06	24.65	1.40	76.47×
		50	7.12	49.31	1.37	77.15×
		75	13.16	73.98	1.35	76.53×

As expected, the speedup achieved by using the estimator is very substantial. This is even more noticeable in the ResNet101 network because it is deeper than the ResNet18 and thus has a

more considerable inference latency. Even though the estimator causes the method to undershoot the target, due to the average relative error being relatively small (under 2 %), it was deemed an acceptable tradeoff.

3.2.2 Memory Management

In each iteration and for every individual, at least one copy of the original *PyTorch* model is made (if the reinitialization step is executed, an additional one is made) and uploaded to the GPU when using CUDA for accelerating the updating of the batch normalization layer's running means and standard deviations, and the fitness calculations. Therefore, after the fitness calculation, the model and all references to it should be deleted to avoid running out of Video Random Access Memory (VRAM). Through testing, it was noted that memory consumption in the GPU was very high and tended to increase up to a point, and for larger models such as SalsaNext, it exhausted the VRAM of an NVIDIA A100 40 GB.

To fix this issue, a routine to free the model from the GPU was implemented. Before deleting the model, it transfers it to the CPU and empties the CUDA cache. This procedure forced the GPU to remove the model from its memory and thus reduce memory consumption tremendously.

Chapter 4

Design and Evaluation Methodology

This chapter focuses on the methodology and implementation of the inference for semantic segmentation CNNs in the target FPGA Xilinx Versal ACAP VCK190, as well as the evaluation of the outputs. After training, the two chosen models, SalsaNext and SemanticFPN, are pruned in order to increase performance. These models are then quantized and compiled using Vitis AI, and their executable model for the inference platform is generated. Figure 4.1 shows this workflow.

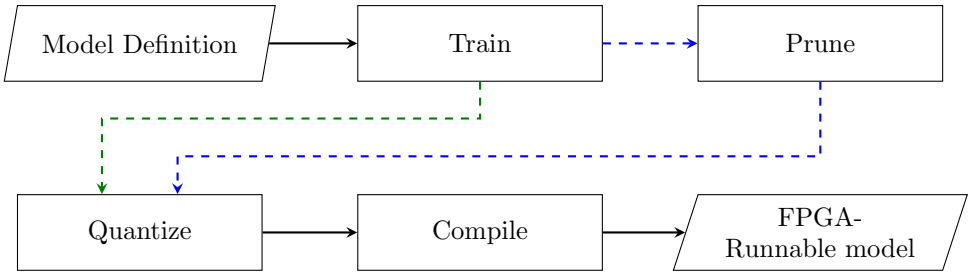


Figure 4.1: Operations order to get an FPGA-runnable model. The blue and green paths show the workflow with and without pruning, respectively.

Section 4.1 explores the chosen models, SalsaNext and SemanticFPN, discussing their implementation and subsequent modifications necessary for hardware inference. Section 4.2 provides a detailed insight into the employed pruning process and parameter selection. Next, section 4.3 outlines the quantization procedure.

Section 4.4 details the implementation of the pruned and quantized models onto the Xilinx Versal ACAP VCK190 platform using Vitis AI. Lastly, section 4.5 delineates the evaluation metrics and methods, providing an understanding of the performance and efficiency of the implemented models.

4.1 Models

Two models were selected for study in this dissertation, the camera-based SemanticFPN with the MobileNetV2 backend and the LiDAR-based SalsaNext. They were chosen due to having good accuracy, being complex models, and having their source code and results available on Xilinx's

Vitis AI Model Zoo [48], which guaranteed the feasibility of the project. However, the quality of the source code of both models differed tremendously.

Models on the Model Zoo repository have a list of versions, each targeted for a specific platform: the GPU version, which has the original source code used to train, infer and evaluate the model, the floating-point weights of the trained model (for *PyTorch* models, the `.pth` file) and the quantized model; and multiple for the various target FPGA models (such as the VCK190), containing the compiled Xilinx Intermediate Representation (XIR) *xmodel* files, which have the model description and quantized weights ready to be deployed in the DPU. Therefore, the GPU version was used as the basis for the implementation in this dissertation.

4.1.1 SalsaNext

Vitis AI Model Zoo provides a version of SalsaNext with a proclaimed pruning rate of 75 %. However, the repository, whose code is based on the original paper, was not in a working state out of the box since it relied on older *PyTorch* libraries than the ones distributed in the Vitis AI development environment. Moreover, the provided floating-point weights appeared to be random, having a near-zero mIoU score, requiring the model's training from scratch, meaning the results in the Model Zoo performance table could not be replicated precisely.

Comparing the original model with the one provided by Xilinx, it was clear that some changes were made:

- All *LeakyReLU* layers had its negative slope changed from the *PyTorch*'s default 0.1 to 0.101 565. This is due to Xilinx's DPUs only supporting this value [6];
- The order of all *BatchNorm* and *LeakyReLU* operations was swapped. Because of the way Vitis AI Quantizer fuses layers, Xilinx recommends doing the operations in the following order: *Convolution*, *BatchNorm* and finally *ReLU*/*LeakyReLU* [49];
- Xilinx's DPUs don't support the *PixelShuffle* operation used in SalsaNext. To remedy this, they replaced it with a transposed convolution operator, followed by *BatchNorm* and a *LeakyReLU*.
- All output channels of all layers were halved from multiples of 32 to multiples of 16, except for the last convolution, whose output channels represent the value of each label class. Xilinx also added a convolutional layer before this last one to maintain the same number of input channels of the last convolutional layer.

From the latter, it can be concluded that the pruning was done manually and not using the proprietary Xilinx Vitis AI Optimizer. This conclusion meant that the provided model could not be used for testing the pruning algorithms since they would be pruning an already pruned network. As such, the results would not be representative of their capabilities.

Thus, the original model source code provided by Cortinhal *et al.* [3] was altered using all the modifications made by Xilinx, except for the manual pruning change, thereby obtaining an

unpruned model that can be deployed to a DPU. This model was then trained from scratch on the SemanticKITTI dataset for 150 epochs with the same parameters used in the original publication, serving as the basis for all pruning experiments on SalsaNext.

4.1.2 SemanticFPN with MobileNetV2 Backend

For the SemanticFPN model, Model Zoo's repository provided fully working source code and unpruned floating-point weights trained on the CityScapes dataset [5] with both a ResNet18 and a MobileNetV2 backend. The latter was chosen since it had a better mIoU score, despite worse throughput performance. The repository also contained code to perform the quantization step, which made this model's implementation very straightforward.

4.2 Pruning

In both implemented models, SalsaNext and SemanticFPN, two distinct pruning methods were employed: the AACP algorithm, whose details and implementation are described in chapter 3, and the GroupNorm pruning algorithm from the *VainF/Torch-Pruning* library. The latter one was used in order to have a baseline for comparison. Experiments were conducted at pruning rates of 25%, 50%, and 75%, each representing the proportion of FLOPs removed from the models.

For the implementation of the AACP algorithm, specific parameters were selected based on the ones used in the original paper [34]: with a population size of 10 individuals, 200 iterations were made, doing the mutations with a differential weight 0.5 and a crossover rate of 0.8. The reinitialization rate was set to 5 instead of 3 due to the overly frequent reinitializations observed under the original parameter setting, which suggested that the neighborhood of the position of an individual in the search space was not being properly explored. The fitness function applied in this process is the inverse of the loss function used for the training of each model, and in order to improve the speed of the process, it was used a subset of 1000 samples of the training set of SemanticKITTI for the SalsaNext model, and 2000 samples of the training set of CityScapes for SemanticFPN.

Concerning the GroupNorm pruning algorithm, provided by the *VainF/Torch-Pruning* library, was used for global pruning without sparsity learning. This algorithm works by iteratively removing the channels with the lowest importance calculated by the L_1 norm from the whole model until the target accuracy is reached. The algorithm keeps the model's integrity thanks to the Dependency Graph [40] implemented in the same library and also used in AACP.

Both the AACP and GroupNorm pruning methods were conducted with the pruning step set to 16. This ensured that the number of output channels remained a multiple of 16 after pruning, optimizing the resulting models for performance on the Xilinx Versal ACAP VCK190. The DPU is capable of processing the outputs of 16 channels in parallel, and as such, this parameter guarantees that pruning without improving throughout does not take place.

After pruning, both models were fine-tuned in order to regain lost accuracy. For this, the models were trained for 10 epochs, and the one with the highest validation accuracy was picked.

4.3 Quantization

Using the Vitis AI Quantizer [6], the weights of the network are quantized from 32 bit floating-point numbers to signed 8 bit integers. Its post-training quantization method utilizes the cross-layer range equalization algorithm proposed in [50] and the batch normalization tuning and AdaQuant algorithms proposed in [51].

The quantizer first analyses the model by analyzing the modules that compose it. Each module is then replaced with an equivalent quantized version defined in the library (that can be directly mapped to a DPU instruction), quantizes the weights, fuses the convolutional layers with the proceeding batch normalization layers, and optimizes the model for the target DPU. The model then needs to be calibrated with only a small subset of the training data through a single forward pass and without needing to perform backpropagation by utilizing batch normalization tuning. This outputs the calibration data that is loaded on subsequent runs of the quantizer. An evaluation of the new quantized and calibrated model is done to determine the loss in accuracy, and the model can be exported to a quantized *PyTorch* model or an uncompiled XIR model (a graph-based intermediate representation of a neural network model that is framework-independent). Figure 4.2 shows a flow chart describing the aforementioned workflow.

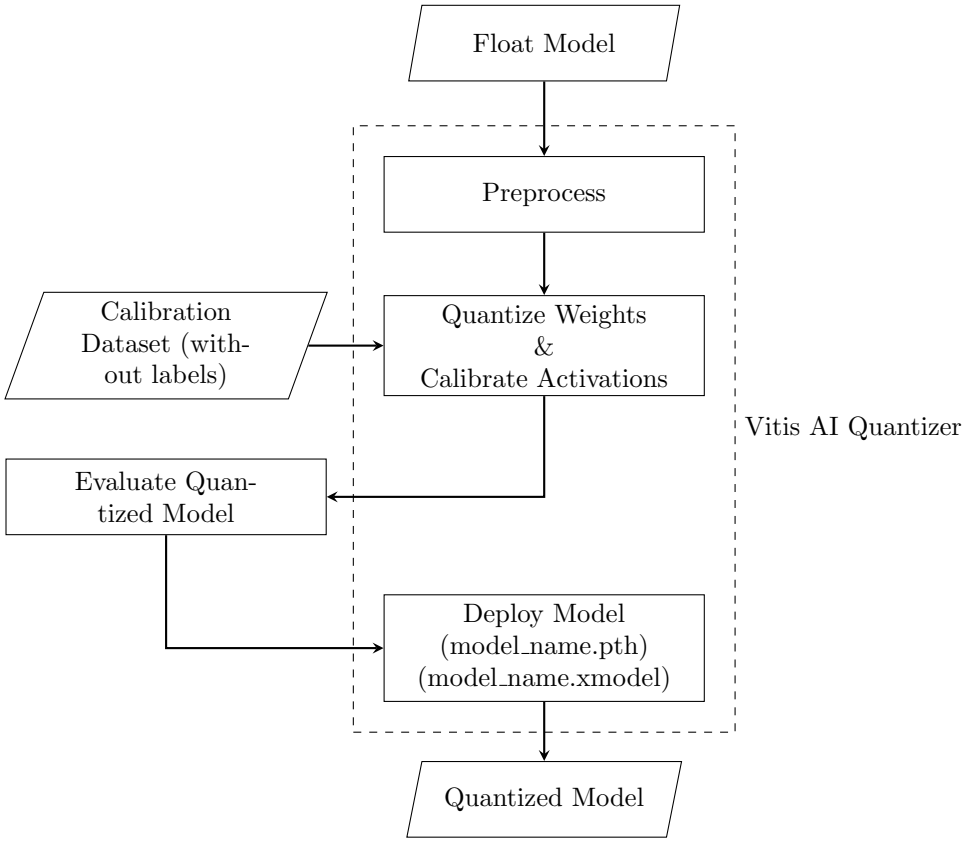


Figure 4.2: Vitis AI Quantizer workflow (adapted from [6]).

However, SalsaNext's operations could not be allocated to the DPU, having the *padding* instructions mapped to the CPU. This is not ideal since the model would be divided into multiple

sub-graphs, some assigned to the DPU and others assigned to the CPU. Not only would the CPU execution be slower, a prohibitive throughout cost would be paid due to the transfer of data between the DPU and CPU in each change of subgraph. Through debugging and investigation, it was discovered that *PyTorch* changed the implementation of the *padding* operation to be fully implemented in C. There are four padding modes: `constant`, `reflect`, `replicate` and `circular`. Previously, when calling the `pad` operator, *PyTorch* had a *Python* implementation that would route each mode to a corresponding C implementation. Vitis AI Quantizer would translate the layer to the quantized version through the names of these mode-specific *padding* functions (although currently only supports `constant` and `replicate` padding modes). However, after the change, the routing happens in C, and therefore the quantizer only detects the generic `pad` function, which it can not parse. Therefore, a patch to the Vitis AI Quantitizer parser had to be made by adding the routing function shown in listing 1.

Listing 1 Patch for Vitis AI Quantitizer to handle new *padding* operator implementation, applied onto the module `pytorch_nndct.parse.op_dispatcher`.

```
def pad(self, input, pad, mode, value=0.0):
    if mode == 'constant':
        return self.constant_pad_nd(input, pad, value)
    elif mode == 'replicate':
        return self.replication_pad2d(input, pad)
    else:
        raise ValueError('Only supports constant and replicate modes')
```

Vitis AI Quantitizer also provides fast finetuning, which implements the AdaQuant algorithm, which tries to optimize the weights and the quantization parameters [51]. This slow process requires large amounts of RAM, especially for large models. Because of this, it was not possible to execute this technique for the SalsaNext model, being only able to perform the operation on the SemanticFPN model.

Therefore, for each given model, quantized models were obtained from the several pruning experiments described in section 4.2 and from the unpruned version. For SemanticFPN, fast fine-tuning was also applied on an additional quantization run.

4.4 Hardware Implementation

The implementation was made on a Xilinx Versal ACAP VCK190 Evaluation Board, running the DPUCVDX8G targeted reference design. This board image contains the DPU and a PetaLinux installation with the Vitis AI Runtime (VART).

In order to run the quantized models on the DPU, the XIR models have to be compiled using the Vitis AI Compiler. The compiler first splits the graph into several subgraphs if some operations can not be executed on the DPU. It then performs architecture-aware optimizations of the inference execution of the model by efficiently scheduling the instructions in order to promote parallelism and

data reuse. Finally, the refined graph containing essential data and directives for VART is converted into a compiled XIR *xmodel* file [6].

The compiled models were executed on C++ programs made with the Vitis AI Library, a high-level API for AI inference built on top of VART, with algorithm libraries for multiple applications, such as 2D and 3D segmentation, and benchmarking facilities [6]. The test programs were cross-compiled with the PetaLinux Software Development Kit (SDK) for the Cortex A72/Cortex A53 SoC.

4.5 Evaluation

In order to compare the effectiveness of the usage of dedicated inference hardware and the applied model compression techniques, several evaluations were made, such as measurements of accuracy metrics (global accuracy and mIoU) of the models in the various pruning states (no pruning and 25 %, 50 %, and 75 % pruning rates) and before and after quantization, as well as the inference's throughput performance, power consumption, and power efficiency on the targeted FPGA and on a GPU baseline. The models were evaluated using the SemanticKITTI and the CityScapes dataset for SalsaNext and SemanticFPN, respectively.

For measuring the throughput performance and power consumption on the GPU, two threads are run concurrently, one that executes the inference of the model continuously in a loop and measures the time taken for each execution, and one that measures the power consumption of the GPU every 5 s with the CUDA API. Then, the averages of the inference time and power of a 5 min run are calculated. Since the time measurement is taken using the CUDA Event API, it is only relative to the processing time of the GPU. It, therefore, is not influenced by the power measurement thread's execution. An NVIDIA RTX 2070 TI GPU with 8 GB of VRAM was used for this benchmark.

A similar approach is used for the Xilinx Versal ACAP VCK190 measurements. However, the power consumption data is only available on the board's separate System Controller SoC [52]. As such, while running the performance benchmarking program on the ACAP processor for 5 min, a program running the System Controller logs to a comma-separated values (CSV) file the total power consumption of the whole system, as well as the consumption of the PL power domain, every 5 s, using the built-in *poweradvantage Python* library. The PL power domain contains both the power consumption of the programmable logic fabric, in which the DPU is implemented, as well as the fixed-function building blocks, such as the AI engines [53]. It thus encompasses the power consumption of the neural network's inference.

Chapter 5

Results and Discussion

5.1 Pruning

Tables 5.1 and 5.2 show the pruning results of the SalsaNext and SemanticFPN with MobileNetV2 backend, respectively, with both the custom group-based implementation of AACP and the GroupNorm algorithm. With post-pruning finetuning included, on SalsaNext, AACP took on average 26 h and GroupNorm 5 h.

Table 5.1: Comparison of SalsaNext pruning results with AACP and GroupNorm, with target pruning ratios of 25 %, 50 % and 75 %. The starting model had 88.10 % accuracy and 54.70 % mIoU when evaluated on the SemanticKITTI dataset.

Method	**FLOPs**	**Parameters**	**FLOPs(%)**	**Params(%)**	**ΔAcc(%)**	**ΔmIoU(%)**
—	131.17 G	6.94 M	0.00	0.00	0.0	0.0
AACP (25 %)	94.64 G	5.42 M	27.85	21.98	0.6	6.8
GN (25 %)	96.59 G	4.75 M	26.36	31.56	2.1	8.0
AACP (50 %)	61.15 G	3.94 M	53.38	43.22	2.4	6.8
GN (50 %)	51.42 G	2.38 M	60.80	65.75	4.9	7.0
AACP (75 %)	30.73 G	2.44 M	76.57	64.85	6.4	17.3
GN (75 %)	32.05 G	1.45 M	75.57	79.13	5.3	14.1

It is clear that AACP, when compared to GroupNorm, achieves good results since it consistently outperforms its peer in terms of accuracy and mIoU. The only outlier is in the pruning of SalsaNext with the 75 % target, where GroupNorm shows lower degradation. This can be explained by the fact that both algorithms are stochastic, and thus performing multiple runs of the algorithms or increasing the number of iterations on AACP could improve the reliability of the results.

However, when compared to the manually-pruned SalsaNext model trained from scratch available on Vitis AI ModelZoo, with a reduction of 75 % of the FLOPs, with 54.3 % mIoU and 87.3 % of accuracy, both of the automatic methods result in worse solutions. Nonetheless, the time and computing power required to train the model (it is not stated in the documentation, but from

Table 5.2: Comparison of SemanticFPN pruning results with AACP and GroupNorm, with target pruning ratios of 25 %, 50 % and 75 %. The starting model had 94.57 % accuracy and 68.67 % mIoU when evaluated on the CityScapes dataset.

Method	**FLOPs**	**Parameters**	**FLOPs(%)**	**Params(%)**	**ΔAcc(%)**	**ΔmIoU(%)**
—	9.51 G	3.78 M	0.00	0.00	0.00	0.00
AACP (25 %)	7.26 G	2.85 M	23.68	24.47	2.49	11.50
GN (25 %)	6.97 G	2.58 M	26.74	31.67	5.29	22.87
AACP (50 %)	5.37 G	2.76 M	43.47	27.00	19.08	48.00
GN (50 %)	4.31 G	1.79 M	54.69	52.66	32.42	55.42
AACP (75 %)	3.52 G	1.89 M	62.93	49.89	38.13	58.56
GN (75 %)	1.1 G	1.45 M	88.47	61.72	78.67	67.18

the training configuration present in the repository, the training was performed for 400 epochs), and the need for specialized manual intervention, makes this approach expensive.

The relationship between accuracy and mIoU is not linear since, for example, by analyzing the results from AACP with pruning targets of (25 %) and (50 %) in SalsaNext, an almost 2 % drop in accuracy occurs, and the mIoU is not changed. So, AACP could achieve better results by modifying the fitness function of AACP to optimize the mIoU instead of the accuracy.

In SalsaNext, in contrast with SemanticFPN, the expected overshooting of the FLOPs estimation used for AACP was not present. This can be attributed to the convolutional part of SalsaNext being much more pronounced than the non-convolutional part, and thus, the estimator error is minor.

5.2 Quantization

For the quantization of the models, tables 5.3 and 5.4 show the impact of the process in the accuracy and mIoU on the various pruning experiments from section 5.1. SemanticFPN is not significantly affected much by the quantization, and the fast-finetuning algorithm further reduces the loss. Moreover, in the pruning configurations with the highest pruning rate, the quantization improves the accuracy performance, probably due to the low starting accuracy of the pruned floating-point models.

SalsaNext's accuracy and mIoU suffer more by the quantization of the model. This could be due to the deepness of the model, causing the layer-wise quantization error to be compounded. Using batch normalization tuning with the fast-finetuning method could yield more favorable results at the cost of a computationally-heavy and memory-intensive process. This is confirmed by the values reported in the Vitis AI ModelZoo, where no quantization loss happens in their manually-pruned version of SalsaNext.

Table 5.3: Impact of the quantization of SalsaNext for the unpruned model and the various pruning experiments.

				Quantization Loss		Total Loss	
Experiment	**Quant.**	**Acc**(%)	**mIoU**(%)	**ΔAcc**(%)	**ΔmIoU**(%)	**ΔAcc**(%)	**ΔmIoU**(%)
Unpruned	Float32	88.1	54.7	0.0	0.0	0.0	0.0
	Int8	82.5	46.7	5.6	8.0	5.6	8.0
AACP (25 %)	Float32	87.5	47.9	0.0	0.0	0.6	6.8
	Int8	80.1	41.1	7.4	6.8	8.0	13.6
GN (25 %)	Float32	86.0	46.7	0.0	0.0	2.1	8.0
	Int8	82.3	44.5	3.7	2.2	5.8	10.2
AACP (50 %)	Float32	85.7	47.9	0.0	0.0	2.4	6.8
	Int8	75.8	40.7	9.9	7.2	12.3	14.0
GN (50 %)	Float32	83.2	47.7	0.0	0.0	4.9	7.0
	Int8	69.6	39.6	13.6	8.1	18.5	15.1
AACP (75 %)	Float32	81.7	37.4	0.0	0.0	6.4	17.3
	Int8	78.9	35.5	2.8	1.9	9.2	19.2
GN (75 %)	Float32	82.8	40.6	0.0	0.0	5.3	14.1
	Int8	72.8	34.4	10.0	6.2	15.3	20.3

Table 5.4: Impact of the quantization of SemanticFPN for the unpruned model and the various pruning experiments. The original Float32 model is compared with the quantized Int8 model with and without using the Fast-Finetuning (FT) method.

				Quantization Loss		**Total Loss**	
Experiment	**Quant.**	**Acc**(%)	**mIoU**(%)	**ΔAcc**(%)	**ΔmIoU**(%)	**ΔAcc**(%)	**ΔmIoU**(%)
Unpruned	Float32	94.6	68.7	0.0	0.0	0.0	0.0
	Int8	94.3	67.3	0.3	1.4	0.3	1.4
	Int8 (FT)	94.5	68.3	0.1	0.4	0.1	0.4
AACP (25 %)	Float32	92.1	57.2	0.0	0.0	2.5	11.5
	Int8	91.3	54.0	0.8	3.1	3.3	14.6
	Int8 (FT)	91.9	55.6	0.2	1.5	2.7	13.0
GN (25 %)	Float32	89.3	45.8	0.0	0.0	5.3	22.9
	Int8	88.5	43.9	0.8	1.9	6.1	24.7
	Int8 (FT)	89.1	44.8	0.1	1.0	5.4	23.8
AACP (50 %)	Float32	75.5	20.7	0.0	0.0	19.1	48.0
	Int8	74.8	20.2	0.7	0.5	19.8	48.5
	Int8 (FT)	76.0	20.6	-0.5	0.0	18.6	48.0
GN (50 %)	Float32	62.2	13.3	0.0	0.0	32.4	55.4
	Int8	61.5	13.2	0.7	0.1	33.1	55.5
	Int8 (FT)	62.3	12.8	-0.2	0.4	32.3	55.8
AACP (75 %)	Float32	56.4	10.1	0.0	0.0	38.1	58.6
	Int8	56.9	9.9	-0.4	0.2	37.7	58.8
	Int8 (FT)	56.4	10.1	0.1	0.0	38.2	58.6
GN (75 %)	Float32	15.9	1.5	0.0	0.0	78.7	67.2
	Int8	16.9	1.6	-1.0	-0.1	77.7	67.1
	Int8 (FT)	16.4	1.5	-0.5	0.0	78.2	67.1

5.3 Throughput and Efficiency

Running the throughput performance and power consumption benchmarks on the DPU for the Xilinx Versal ACAP VCK190 and on an NVIDIA RTX 2070 8 GB GPU yielded the results seen in tables 5.5 and 5.6 for SalsaNext and SemanticFPN, respectively. Comparing the measured speedups relative to the unpruned models with the expected from the FLOPs reduction, it can be seen that it does not follow a linear relation. This is because there are operations that are independent of the number of channels, such as data transfers and other layer-specific execution overheads. Moreover, it can be noted that the power consumption of the models through the different pruning rates remains somewhat consistent on the DPU and decreases with higher rates on the GPU — more noticeable in table 5.6.

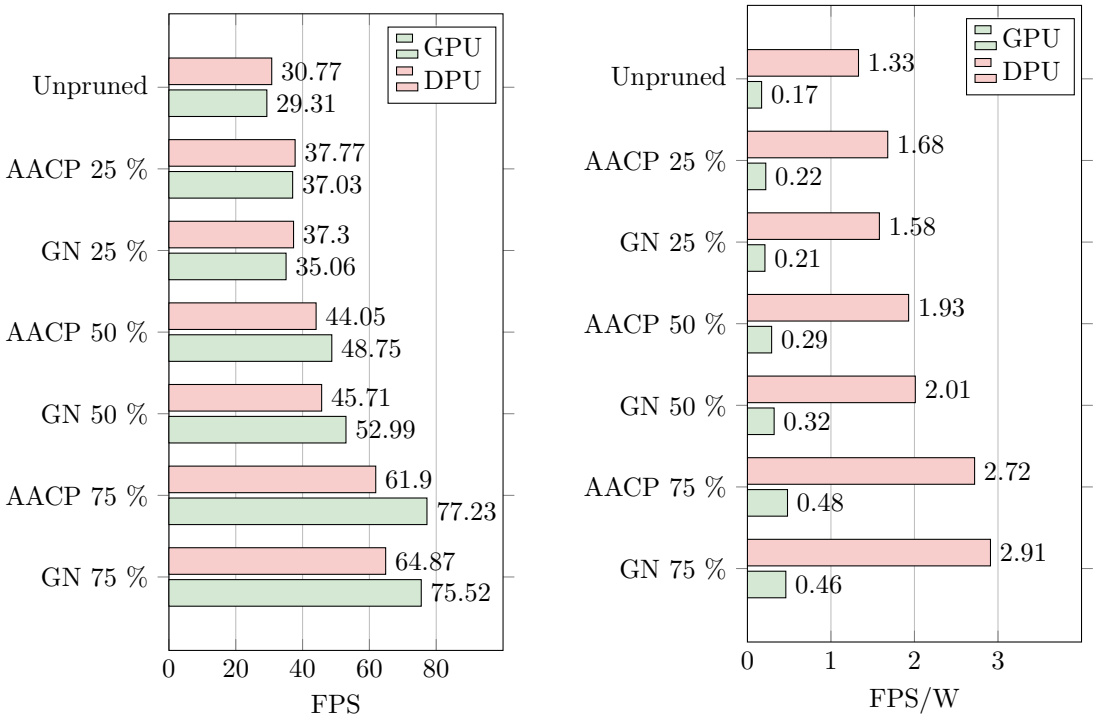
Table 5.5: SalsaNext's performance and power consumption inference benchmark results on the DPU for the Xilinx Versal ACAP VCK190 and on an NVIDIA RTX 2070 8 GB GPU for the various pruning experiments.

		DPU on Versal ACAP VCK190			RTX 2070 GPU		
Experiment	**FLOPs Speedup**	**Throughput (FPS)**	**Power (W)**	**Speedup**	**Throughput (FPS)**	**Power (W)**	**Speedup**
Unpruned	1.00	30.77	23.10	1.00	29.31	171.52	1.00
AACP (25 %)	1.39	37.77	22.54	1.23	37.03	166.84	1.26
GN (25 %)	1.36	37.30	23.64	1.21	35.06	167.68	1.20
AACP (50 %)	2.15	44.05	22.79	1.43	48.75	165.41	1.66
GN (50 %)	2.55	45.71	22.74	1.49	52.99	165.18	1.81
AACP (75 %)	4.27	61.90	22.75	2.01	77.23	162.18	2.64
GN (75 %)	4.09	64.87	22.26	2.11	75.52	162.44	2.58

Table 5.6: SemanticFPN's performance and power consumption inference benchmark results on the DPU for the Xilinx Versal ACAP VCK190 and on an NVIDIA RTX 2070 8 GB GPU for the various pruning experiments.

		DPU on Versal ACAP VCK190			RTX 2070 GPU		
Experiment	**FLOPs Speedup**	**Throughput (FPS)**	**Power (W)**	**Speedup**	**Throughput (FPS)**	**Power (W)**	**Speedup**
Unpruned	1.00	279.62	21.61	1.00	140.02	122.65	1.14
AACP (25 %)	1.31	334.29	21.55	1.20	154.80	85.77	1.80
GN (25 %)	1.36	344.63	21.10	1.23	159.48	79.39	2.01
AACP (50 %)	1.77	519.55	21.74	1.86	196.76	75.81	2.60
GN (50 %)	2.21	518.10	21.09	1.85	232.44	74.00	3.14
AACP (75 %)	2.70	643.94	20.96	2.30	272.27	71.41	3.81
GN (75 %)	8.68	793.71	21.47	2.84	353.86	68.34	5.18

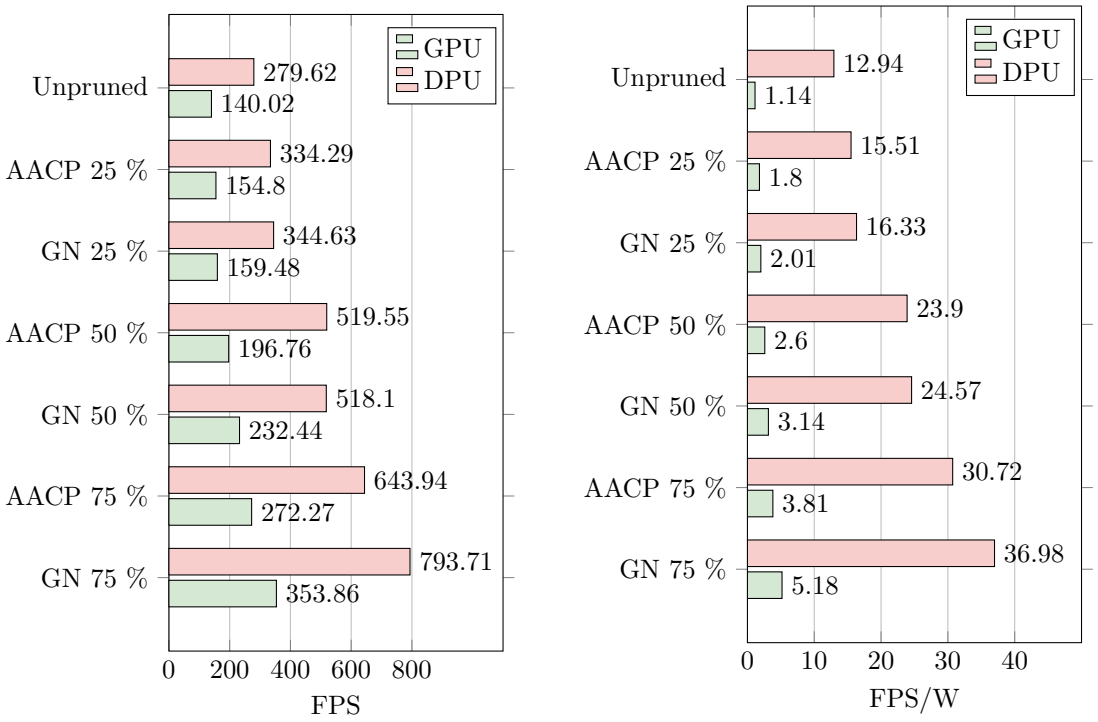
As illustrated in figures 5.1 and 5.2, the power efficiency of the DPU in the Xilinx Versal ACAP VCK190 is far superior to the NVIDIA RTX 2070 8 GB GPU. In SalsaNext, the two devices have similar performances. However, for the SemanticFPN model, the DPU consistently outperforms the GPU by a large margin.



(a) Throughput performance comparison in FPS.

(b) Power efficiency comparison in FPS per Watt.

Figure 5.1: Comparison between SalsaNext's inference throughput and power efficiency performance for the unpruned models and the various pruned configurations, running on the DPU in a Xilinx Versal ACAP VCK190 (red) and an NVIDIA RTX 2070 8 GB (green).



(a) Throughput performance comparison in FPS.

(b) Power efficiency comparison in FPS per Watt.

Figure 5.2: Comparison between SemanticFPN's inference throughput and power efficiency performance for the unpruned models and the various pruned configurations, running on the DPU in a Xilinx Versal ACAP VCK190 (red) and an NVIDIA RTX 2070 8 GB GPU (green).

Chapter 6

Conclusions and Future Work

FPGA-based inference implementations of semantic segmentation CNNs based on Vitis AI show great promise for ADAS applications. The Xilinx Versal ACAP VCK190 with the DPUCVDX8G IP core-based implementations of the SalsaNext and SemanticFPN models resulted in performance that matches (SalsaNext) or surpasses by 2.25× (SemanticFPN) a desktop-class GPU while providing more than 10× better power efficiency. The achieved inference throughput results for SalsaNext successively exceed the typical frame rate of LiDAR sensors, 10 Hz [3].

Channel pruning further increases the throughput capabilities of such implementations at the cost of accuracy. These model compression techniques yield better results when applied to larger models such as SalsaNext, suggesting more redundancies in the neural network. Through the implementation of group-based pruning with the help of a layer-dependency graph, channel pruning can be applied to any CNN without manually describing the model's internal dependencies, making this process much more streamlined and removing the need to have specialized knowledge of the model's inner architecture. Using an evolutionary genetic algorithm (AACP) resulted in better pruning strategies than a simple norm-based approach, indicating the methodology's effectiveness.

However, Vitis AI is not yet a mature tool to generate these hardware implementations, requiring manual modifications to the source models due to unsupported operations and, thus, necessitating computationally-expensive training. The quantization process needed for inference on the DPUs introduces some accuracy loss, which the fast-finetuning method tries to remedy. The amount of memory needed for this correction process gets prohibitive for larger models with heavy datasets, such as SalsaNext, limiting its applicability.

6.1 Future Work

The following future research vectors were identified during the development of the dissertation:

1. Exploration of the impact of different fitness functions for the pruning genetic algorithm. Since different problems have different key target metrics, such as the mIoU for semantic segmentation, targeting AACP to optimize for specific metrics could further improve results.

2. Tune the parameters of the pruning algorithm. These were chosen to match the original paper. However, they could be optimized to provide faster (i.e., require fewer iterations to reach an optimal solution) and increase the pruning accuracy results.

3. Research DPU-specific throughput estimation, taking into account its architecture. Although this dissertation has already considered the number of channels that can be processed in parallel, other improvements could be made, such as optimizing the size of the layers' weights to facilitate more efficient internal transfers.

References

[1] V. K. Kukkala, J. Tunnell, S. Pasricha, and T. Bradley, "Advanced Driver-Assistance Systems: A Path Toward Autonomous Vehicles," *IEEE Consumer Electronics Magazine*, vol. 7, no. 5, pp. 18–25, Sep. 2018, ISSN: 2162-2256. DOI: 10.1109/MCE.2018.2828440.

[2] Y. Lyu, L. Bai, and X. Huang, "Real-Time Road Segmentation Using LiDAR Data Processing on an FPGA," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351244.

[3] T. Cortinhal, G. Tzelepis, and E. Erdal Aksoy, "SalsaNext: Fast, Uncertainty-Aware Semantic Segmentation of LiDAR Point Clouds," in *Advances in Visual Computing*, G. Bebis, Z. Yin, E. Kim, *et al.*, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, Dec. 2020, pp. 207–222, ISBN: 978-3-030-64559-5. DOI: 10.1007/978-3-030-64559-5_16.

[4] J. Behley, M. Garbade, A. Milioto, *et al.*, "SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences," arXiv, Aug. 16, 2019. DOI: 10.48550/arXiv.1904.01416. arXiv: 1904.01416 [cs].

[5] M. Cordts, M. Omran, S. Ramos, *et al.* "The Cityscapes Dataset for Semantic Urban Scene Understanding." arXiv: 1604.01685 [cs]. (Apr. 7, 2016), [Online]. Available: http://arxiv.org/abs/1604.01685 (visited on Jun. 25, 2023), preprint.

[6] Xilinx, Inc. "Vitis AI User Guide (UG1414)." (Jun. 15, 2022), [Online]. Available: https://docs.xilinx.com/r/en-US/ug1414-vitis-ai/Vitis-AI-Overview (visited on Dec. 5, 2022).

[7] A. Kirillov, R. Girshick, K. He, and P. Dollár, "Panoptic Feature Pyramid Networks," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019, pp. 6392–6401. DOI: 10.1109/CVPR.2019.00656.

[8] A. Milioto, I. Vizzo, J. Behley, and C. Stachniss, "RangeNet ++: Fast and Accurate LiDAR Semantic Segmentation," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Nov. 2019, pp. 4213–4220. DOI: 10.1109/IROS40897.2019.8967762.

[9] S. Theodoridis, *Machine Learning : A Bayesian and Optimization Perspective*. London: Academic Press, 2020, vol. 2nd edition, ISBN: 978-0-12-818803-3.

[10] PyTorch Contributors. "LeakyReLU — PyTorch 2.0 documentation." (2023), [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.LeakyReLU.html (visited on Jun. 28, 2023).

[11] PyTorch Contributors. "BatchNorm2d — PyTorch 2.0 documentation." (2023), [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html (visited on Jun. 28, 2023).

[12] O. Terzo, K. Djemame, A. Scionti, and C. Pezuela, Eds., *Heterogeneous Computing Architectures: Challenges and Vision*. Boca Raton: CRC Press, Sep. 10, 2019, 338 pp., ISBN: 978-0-429-39960-2. DOI: 10.1201/9780429399602.

[13] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx Adaptive Compute Acceleration Platform: Versal ^{TM} Architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside CA USA: ACM, Feb. 20, 2019, pp. 84–93, ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293906.

[14] S. Ahmad, S. Subramanian, V. Boppana, *et al.*, "Xilinx First 7nm Device: Versal AI Core (VC1902)," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, Aug. 2019, pp. 1–28. DOI: 10.1109/HOTCHIPS.2019.8875639.

[15] Xilinx, Inc. "ACAP at the Edge with the Versal AI Edge Series (WP518)." (Jun. 9, 2021), [Online]. Available: https://docs.xilinx.com/api/khub/documents/Xz0szg2HiN1YFYfaJVXcrQ/content?Ft-Calling-App=ft%2Fturnkey-portal&Ft-Calling-App-Version=4.1.3&filename=wp518-ai-edge-intro.pdf (visited on Jan. 9, 2023).

[16] Y. Wang, T. Shi, P. Yun, L. Tai, and M. Liu, "PointSeg: Real-Time Semantic Segmentation Based on 3D LiDAR Point Cloud," arXiv, Sep. 25, 2018. DOI: 10.48550/arXiv.1807.06288. arXiv: 1807.06288 [cs].

[17] B. Wu, A. Wan, X. Yue, and K. Keutzer, "SqueezeSeg: Convolutional Neural Nets with Recurrent CRF for Real-Time Road-Object Segmentation from 3D LiDAR Point Cloud," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018, pp. 1887–1893. DOI: 10.1109/ICRA.2018.8462926.

[18] B. Wu, X. Zhou, S. Zhao, X. Yue, and K. Keutzer, "SqueezeSegV2: Improved Model Structure and Unsupervised Domain Adaptation for Road-Object Segmentation from a LiDAR Point Cloud," in *2019 International Conference on Robotics and Automation (ICRA)*, May 2019, pp. 4376–4382. DOI: 10.1109/ICRA.2019.8793495.

[19] C. Xu, B. Wu, Z. Wang, *et al.*, "SqueezeSegV3: Spatially-Adaptive Convolution for Efficient Point-Cloud Segmentation," in *Computer Vision – ECCV 2020*, A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 1–19, ISBN: 978-3-030-58604-1. DOI: 10.1007/978-3-030-58604-1_1.

[20] E. E. Aksoy, S. Baci, and S. Cavdar, "SalsaNet: Fast Road and Vehicle Segmentation in LiDAR Point Clouds for Autonomous Driving," in *2020 IEEE Intelligent Vehicles Symposium (IV)*, Oct. 2020, pp. 926–932. DOI: 10.1109/IV47402.2020.9304694.

[21] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes Challenge: A Retrospective," *International Journal of Computer Vision*, vol. 111, no. 1, pp. 98–136, Jan. 1, 2015, ISSN: 1573-1405. DOI: 10.1007/s11263-014-0733-5.

[22] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation," presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 652–660.

[23] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space," in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017.

[24] X. Xie, L. Bai, and X. Huang, "Real-Time LiDAR Point Cloud Semantic Segmentation for Autonomous Driving," *Electronics*, vol. 11, no. 1, p. 11, 1 Jan. 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11010011.

[25] M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, and M. Shafique, "Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead," *IEEE Access*, vol. 8, pp. 225 134–225 180, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3039858.

[26] K. Abdelouahab, M. Pelcat, J. Serot, C. Bourrasset, J.-C. Quinton, and F. Berry. "Hardware Automated Dataflow Deployment of CNNs." arXiv: 1705.04543 [cs]. (Jun. 29, 2017), preprint.

[27] S. I. Venieris and C.-S. Bouganis. "fpgaConvNet: A Toolflow for Mapping Diverse Convolutional Neural Networks on Embedded FPGAs." arXiv: 1711.08740 [cs]. (Nov. 23, 2017), preprint.

[28] M. Blott, T. B. Preußer, N. J. Fraser, *et al.*, "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 3, 16:1–16:23, Dec. 15, 2018, ISSN: 1936-7406. DOI: 10.1145/3242897.

[29] F. Fahim, B. Hawks, C. Herwig, *et al.* "Hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices." arXiv: 2103.05579 [physics]. (Mar. 23, 2021), preprint.

[30] NVIDIA Corporation. "NVIDIA Deep Learning Accelerator." (2018), [Online]. Available: http://nvdla.org/ (visited on Jan. 6, 2023).

[31] Xilinx, Inc. "DPUCVDX8G for Versal ACAPs Product Guide (PG389)." (Jan. 21, 2023), [Online]. Available: https://docs.xilinx.com/r/en-US/pg389-dpucvdx8g (visited on Jun. 29, 2023).

[32] P. Szántó, T. Kiss, and K. J. Sipos, "Energy-efficient AI at the Edge," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, Jun. 2022, pp. 1–6. DOI: 10.1109/MECO55406.2022.9797178.

[33] Xilinx, Inc. "Vitis Ai Optimizer User Guide (UG1333)." (Jun. 15, 2022), [Online]. Available: https://docs.xilinx.com/r/en-US/ug1333-ai-optimizer (visited on Jan. 8, 2023).

[34] H.-J. Kang, "Accelerator-Aware Pruning for Convolutional Neural Networks," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 7, pp. 2093–2103, Jul. 2020, ISSN: 1558-2205. DOI: 10.1109/TCSVT.2019.2911674.

[35] J. Li and A. Louri, "AdaPrune: An Accelerator-Aware Pruning Technique for Sustainable CNN Accelerators," *IEEE Transactions on Sustainable Computing*, vol. 7, no. 1, pp. 47–60, Jan. 2022, ISSN: 2377-3782. DOI: 10.1109/TSUSC.2021.3060690.

[36] Z. Liu, H. Mu, X. Zhang, *et al.*, "MetaPruning: Meta Learning for Automatic Neural Network Channel Pruning," presented at the Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 3296–3305. [Online]. Available: https://openaccess.thecvf.com/content_ICCV_2019/html/Liu_MetaPruning_Meta_Learning_for_Automatic_Neural_Network_Channel_Pruning_ICCV_2019_paper.html (visited on Jun. 28, 2023).

[37] M. Lin, R. Ji, Y. Zhang, B. Zhang, Y. Wu, and Y. Tian. "Channel Pruning via Automatic Structure Search." arXiv: 2001.08565 [cs]. (Jun. 28, 2020), [Online]. Available: http://arxiv.org/abs/2001.08565 (visited on Jun. 28, 2023), preprint.

[38] B. Li, B. Wu, J. Su, and G. Wang, "EagleEye: Fast Sub-net Evaluation for Efficient Neural Network Pruning," in *Computer Vision – ECCV 2020*, A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 639–654, ISBN: 978-3-030-58536-5. DOI: 10.1007/978-3-030-58536-5_38.

[39] L. Lin, S. Chen, Y. Yang, and Z. Guo, "AACP: Model Compression by Accurate and Automatic Channel Pruning," in *2022 26th International Conference on Pattern Recognition (ICPR)*, Aug. 2022, pp. 2049–2055. DOI: 10.1109/ICPR56361.2022.9956562.

[40] G. Fang, X. Ma, M. Song, M. B. Mi, and X. Wang. "DepGraph: Towards Any Structural Pruning." arXiv: 2301.12900 [cs]. (Mar. 23, 2023), [Online]. Available: http://arxiv.org/abs/2301.12900 (visited on Apr. 7, 2023), preprint.

[41] G. Fang, *VainF/Torch-Pruning*, Jun. 12, 2023. [Online]. Available: https://github.com/VainF/Torch-Pruning (visited on Jun. 12, 2023).

[42] "DPUCVDX8G for Versal ACAPs Product Guide (PG389) • Reader • Documentation Portal." (), [Online]. Available: https://docs.xilinx.com/r/en-US/pg389-dpucvdx8g/Introduction (visited on Dec. 5, 2022).

[43] PyTorch Contributors. "Conv2d — PyTorch 2.0 documentation." (2023), [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html (visited on Jun. 22, 2023).

[44] Y. Wang, *Flopth*, Jun. 21, 2023. [Online]. Available: https://github.com/vra/flopth (visited on Jun. 22, 2023).

[45] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition." arXiv: 1512.03385 [cs]. (Dec. 10, 2015), [Online]. Available: http://arxiv.org/abs/1512.03385 (visited on Jun. 22, 2023), preprint.

[46] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Aug. 4, 2009.

[47] Python Software Foundation. "Time — Time access and conversions," Python documentation. (Jun. 21, 2023), [Online]. Available: https://docs.python.org/3/library/time.html (visited on Jun. 22, 2023).

[48] Xilinx, Inc. "Vitis AI Model Zoo Details & Performance table." (Feb. 9, 2023), [Online]. Available: https://xilinx.github.io/Vitis-AI/docs/reference/ModelZoo_VAI3.0_Github_web.htm (visited on Jun. 19, 2023).

[49] Xilinx, Inc. "Developing a Model — Vitis™ AI 3.0 documentation." (Feb. 9, 2023), [Online]. Available: https://xilinx.github.io/Vitis-AI/docs/workflow-model-development.html (visited on Jun. 25, 2023).

[50] M. Nagel, M. van Baalen, T. Blankevoort, and M. Welling, "Data-Free Quantization Through Weight Equalization and Bias Correction," presented at the Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 1325–1334. [Online]. Available: https://openaccess.thecvf.com/content_ICCV_2019/html/Nagel_Data-Free_Quantization_Through_Weight_Equalization_and_Bias_Correction_ICCV_2019_paper.html (visited on Jun. 25, 2023).

[51] I. Hubara, Y. Nahshan, Y. Hanani, R. Banner, and D. Soudry. "Improving Post Training Neural Quantization: Layer-wise Calibration and Integer Programming." arXiv: 2006.10518 [cs, stat]. (Dec. 14, 2020), [Online]. Available: http://arxiv.org/abs/2006.10518, preprint.

[52] Xilinx, Inc. "Versal Adaptive SoC Power Management - Xilinx Wiki - Confluence." (May 2, 2023), [Online]. Available: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/1299382289/Versal+Adaptive+SoC+Power+Management (visited on Jun. 27, 2023).

[53] Xilinx, Inc. "Versal ACAP Technical Reference Manual (AM011)." (Dec. 16, 2022), [Online]. Available: https://docs.xilinx.com/r/en-US/am011-versal-acap-trm (visited on Jun. 27, 2023).

Appendix A

AACP implementation

```
import copy
import time

import numpy as np

import torch
import torch.nn as nn

import torch_pruning as tp
from torch_pruning.importance import GroupNormImportance

from loguru import logger

from flops_estimator import PrunedFlopsEstimator
from avgmeter import TimeEstimate

import gc

class AACPruner:

    def __init__(self, model: nn.Module, dummy_input: torch.Tensor,
                 device: torch.device, calc_fitness, update_batchnorm,
                 log_dir: str, checkpoint_dir, ignored_layers):
        """
        Construct a new Prunner object

        Args:
            model: pytorch model to be pruned
            dummy_input: dummy input for calculating the complexity
            device: device to run the model
            calc_fitness: function to calculate the fitness of the model
            update_batchnorm: function to update the batchnorm
            log_dir: directory to save the log
```

```
            checkpoint_dir: directory to load the checkpoint
            ignored_layers: function to determine the ignored layers of
                            given model
        """
        self.original_model = model
        self.dummy_input = dummy_input
        self.device = device
        self.update_batchnorm = update_batchnorm
        self.calc_fitness = calc_fitness
        self.ignored_layers = ignored_layers

        dep_graph = tp.DependencyGraph().build_dependency(
            self.original_model, example_inputs=self.dummy_input)

        groups = list(
            dep_graph.get_all_groups(
                ignored_layers=ignored_layers(model),
                root_module_types=(tp.ops.TORCH_CONV,))
        )

        self.arch = np.zeros(len(groups), dtype=int)
        for i, group in enumerate(groups):
            self.arch[i] = len(group.items[0].idxs)

        self.flops_estimator = PrunedFlopsEstimator.from_model(
            self.original_model, [self.dummy_input])

        self.log_dir = log_dir
        self.checkpoint_dir = checkpoint_dir



    def run(self, pr_flops: float, iterations: int=200, pop_size: int=10,
            F: float=0.5, CR: float=0.8, R: int=3, step: int=16) -> nn.Module:
        """
        Run the prunning algorithm

        Args:
            pr_flops: target prunning rate
            iterations: number of iterations
            pop_size: population size
            F: differential weight
            CR: crossover probability
            R: reinitialization interval
            step: step size

        Returns:
            pruned_model: pruned model
        """
```

```
assert pr_flops <= 1 and pr_flops >= 0
assert iterations > 0
assert pop_size > 0
assert F >= 0 and F <= 2
assert CR >= 0 and CR < 1
assert R > 0
assert step > 0

logger.info("="*20)
logger.info("Running AACP")
logger.info(f"Prunning rate: {pr_flops}")
logger.info(f"Iterations: {iterations}")
logger.info(f"Population size: {pop_size}")
logger.info(f"Differential weight: {F}")
logger.info(f"Crossover probability: {CR}")
logger.info(f"R: {R}")
logger.info(f"Step size: {step}")
logger.info("="*20)

self.individual_init_e = TimeEstimate(pop_size)
self.individual_iter_e = TimeEstimate(iterations*pop_size)

self.pr_flops = pr_flops

flops = self.flops_estimator.flops

target_flops = flops * (1 - pr_flops)


if self.checkpoint_dir:
    logger.info("Loading checkpoint")
    iteration, population, global_best, global_best_arch = \
        self.load_checkpoint(self.checkpoint_dir)

else:
    iteration = 0
    population = self.init_population(
        pop_size, target_flops, flops, step)

    global_best = np.inf
    global_best_arch = None

for i in range(iteration, iterations):
    logger.info("Iteration {}/{} ({})",
                i, iterations, self.individual_iter_e.get_estimate())
    idx, fitness = self.iterate(
        population, target_flops, flops, F, CR, R, step)
    if fitness < global_best:
        global_best = fitness
```

```
                global_best_arch = population[idx].genes.copy()

            logger.info("Global Best: {:.3f}", global_best)

            self.save_checkpoint(i + 1, population, global_best, global_best_arch)

        individual = self.Individual(pruner=self)
        individual.genes = global_best_arch
        individual._update_model()
        individual._update_batchnorm()
        pruned_model = individual.model
        pruned_model.eval()

        return pruned_model

    def init_population(self, pop_size, target_flops, original_flops, step):
        """
        Initialize the population

        Args:
            pop_size: population size
            target_flops: target flops
            original_flops: original flops
            step: step size

        Returns:
            population: population array with individuals
        """
        population = []

        logger.info("Initiating population")
        for i in range(pop_size):
            self.individual_init_e.start()
            logger.info("Individual {}/{} ({})",
                        i, pop_size, self.individual_init_e.get_estimate())
            individual = self.Individual(pruner=self)
            individual.init_individual(
                target_flops, original_flops, step=step)
            individual.free_model()
            population.append(individual)
            self.individual_init_e.stop()

        return population

    def iterate(self, population, target_flops, original_flops, F, CR, R, step):
        """
        Runs one iteration of the algorithm

        Args:
```

```
        population: population array with individuals
        target_flops: target flops
        original_flops: original flops
        F: differential weight
        CR: crossover probability
        R: reinitialization interval
        step: step size

    Returns:
        best_idx: index of the best individual
        best_fitness: fitness of the best individual
    """
    old_gen = np.array([individual.genes.copy()
                        for individual in population])
    best_fitness = np.inf
    best_fitness_idx = 0


    for i in range(len(population)):
        start_time = time.time()
        logger.info("Individual {}/{} ({})",
                    i, len(population), self.individual_iter_e.get_estimate())
        r1, r2, r3 = np.random.choice(len(population), 3, replace=False)

        # mutate
        new_genes = (old_gen[r1] + F *
                     (old_gen[r2] - old_gen[r3])).astype(int)

        # crossover
        for j in range(len(new_genes)):
            if np.random.rand() < CR:
                new_genes[j] = old_gen[i][j]

        # bound
        for j in range(len(new_genes)):
            if new_genes[j] <= step:
                new_genes[j] = step
            elif new_genes[j] % step != 0:
                new_genes[j] = round(new_genes[j] / step) * step

        new_individual = self.Individual(pruner=self)
        new_individual.set_genes(new_genes, recalculate=False)
        new_individual.prune_until_target(
            target_flops, original_flops, step=step)
        new_genes = new_individual.genes

        # selection
        new_individual.set_genes(new_genes)
```

```
            if new_individual < population[i]:
                population[i] = new_individual
            else:
                new_individual.free_model()
                population[i].stay += 1

                # reinitialize
                if population[i].stay >= R:
                    logger.info("Reinitializing individual {}", i)
                    new_individual = self.Individual(pruner=self)
                    new_individual.init_individual(
                        target_flops, original_flops, step=step)
                    population[i] = new_individual

            population[i].free_model()
            if population[i].fitness < best_fitness:
                    best_fitness = population[i].fitness
                    best_fitness_idx = i

            self.individual_iter_e.update(time.time() - start_time)

            logger.info("Current Best: {:.3f}", best_fitness)

        return best_fitness_idx, best_fitness

    def save_checkpoint(self, iteration, population, global_best, global_best_arch):
        """
        Saves a checkpoint of the algorithm

        Args:
            iteration: current iteration
            population: population array with individuals
            global_best: global best fitness
            global_best_arch: global best architecture
        """
        checkpoint = {
            'iteration': iteration,
            'population': list(map(
                lambda individual: individual.state_dict(), population
            )),
            'global_best': global_best,
            'global_best_arch': global_best_arch
        }
        logger.info("Saving checkpoint to {}", f'{self.log_dir}/aacp_checkpoint')
        torch.save(checkpoint, f'{self.log_dir}/aacp_checkpoint')

    def load_checkpoint(self, checkpoint_path):
        """
        Loads a checkpoint of the algorithm
```

```
        Args:
            checkpoint_path: path to the checkpoint

        Returns:
            iteration: current iteration
            population: population array with individuals
            global_best: global best fitness
            global_best_arch: global best architecture
        """
        checkpoint = torch.load(f'{checkpoint_path}/aacp_checkpoint')
        iteration = checkpoint['iteration']
        global_best = checkpoint['global_best']
        global_best_arch = checkpoint['global_best_arch']
        population = []
        for individual_state in checkpoint['population']:
            individual = self.Individual(pruner=self)
            individual.load_state_dict(individual_state)
            population.append(individual)

        return iteration, population, global_best, global_best_arch


    class Individual:
        def __init__(self, pruner):
            """
            Constructor for the Individual class

            Args:
                pruner: pruner object
            """
            self.original_model = pruner.original_model
            self.arch = pruner.arch
            self.dummy_input = pruner.dummy_input
            self.device = pruner.device
            self.original_flops_estimator = pruner.flops_estimator
            self.flops_estimator = copy.deepcopy(pruner.flops_estimator)
            self.stay = 0
            self.model = None
            self.genes = None
            self.dep_graph = None
            self.groups = None
            self.pruner = pruner

        def free_model(self):
            """
            Frees the model from memory (CPU and GPU)
            """
            if hasattr(self, 'model') and self.model is not None:
```

```
            self.model.cpu()
            del self.model
        if hasattr(self, 'dep_graph') and self.dep_graph is not None:
            del self.dep_graph
        if hasattr(self, 'groups') and self.groups is not None:
            del self.groups
        torch.cuda.empty_cache()
        gc.collect()

    def init_individual(self, target_flops, original_flops, step):
        """
        Initializes the individual

        Args:
            target_flops: target flops
            original_flops: original flops
            step: step size
        """
        if self.model is not None:
            self.model.cpu()
            del self.model
        self.genes = self.arch.copy()
        self.model = copy.deepcopy(self.original_model)
        self.model.to(self.device)

        self.dep_graph = tp.DependencyGraph().build_dependency(
            self.model, example_inputs=self.dummy_input)
        self.groups = list(
            self.dep_graph.get_all_groups(
                ignored_layers=self.pruner.ignored_layers(self.model),
                root_module_types=(tp.ops.TORCH_CONV,)
            )
        )

        self.prune_until_target(
            target_flops, original_flops, step=step)

        self.set_genes(self.genes)

    def prune_until_target(self, target_flops, original_flops, step):
        """
        Prunes the individual until the target flops is reached

        Args:
            target_flops: target flops
            original_flops: original flops
            step: step size
        """
        ops = self.flops_estimator.flops
```

```
        logger.info("Pruning until target ratio: {}",
                    1 - target_flops / original_flops)

        while ops > target_flops:
            i = np.random.randint(0, len(self.genes))
            if self.genes[i] <= step:
                continue
            self.genes[i] -= step
            idxs = np.arange(
                self.genes[i], self.genes[i] + step, dtype=int).tolist()
            self.groups[i].prune(idxs)
            ops = self.flops_estimator.prune(self.groups[i], step)

        logger.debug("Pruning finished at ratio {}",
                     1 - ops / original_flops)

    def set_genes(self, genes, recalculate=True):
        """
        Sets the genes of the individual. If recalculate is True, the
        model' fitness is recalculated
        """
        self.genes = genes
        self._update_model(dumb=not recalculate)
        if recalculate:
            self._update_batchnorm()
            self._update_fitness()

    def _update_model(self, dumb=False):
        if hasattr(self, 'model') and self.model is not None:
            self.model.cpu()
            del self.model
        if hasattr(self, 'dep_graph') and self.dep_graph is not None:
            del self.dep_graph
        if hasattr(self, 'groups') and self.groups is not None:
            del self.groups

        self.model = copy.deepcopy(self.original_model)
        gnorm = GroupNormImportance(p=2)

        self.dep_graph = tp.DependencyGraph().build_dependency(
            self.model, example_inputs=self.dummy_input)
        self.groups = list(
            self.dep_graph.get_all_groups(
                ignored_layers=self.pruner.ignored_layers(self.model),
                root_module_types=(tp.ops.TORCH_CONV,)
            )
        )

        self.flops_estimator = copy.deepcopy(self.original_flops_estimator)
```

```
        groups = enumerate(self.groups)
        if not dumb:
            for i, group in groups:
                importances = gnorm(group)
                # get the indices of the channels to prune (lowest importance)
                idxs = torch.argsort(importances, descending=True)[
                    self.genes[i]:].tolist()
                # prune the channels
                group.prune(idxs)
                self.flops_estimator.prune(group, len(idxs))
        else:
            for i, group in groups:
                idxs = np.arange(
                    self.genes[i], len(group.items[0].idxs), dtype=int).tolist()
                group.prune(idxs)
                self.flops_estimator.prune(group, len(idxs))

    def _update_batchnorm(self):
        logger.debug("Updating batchnorm")
        self.pruner.update_batchnorm(self.model)

    def _update_fitness(self):
        logger.debug("Updating fitness")
        self.fitness = self.pruner.calc_fitness(self.model)

    def __lt__(self, other):
        return self.fitness < other.fitness

    def __gt__(self, other):
        return self.fitness > other.fitness

    def __eq__(self, other):
        return self.fitness == other.fitness

    def state_dict(self):
        return {
            "genes": self.genes,
            "fitness": self.fitness,
            "stay": self.stay,
        }

    def load_state_dict(self, state_dict):
        self.genes = state_dict["genes"]
        self.fitness = state_dict["fitness"]
        self.stay = state_dict["stay"]
```