Neural Network Partitioning for Resource-Limited Environments

by

Patrick Geel

to obtain the degree of Master of Science
at the Delft University of Technology, to be defended publicly on Tuesday May 2, 2023 at 13:30 (CEST).

Student number:	4552075	
Project duration:	June 1, 2022 - May 2, 2023	
Thesis committee:	Dr. Ir. Z. Al-Ars,	TU Delft, supervisor
	Dr. Ir. NP. van der Meijs,	TU Delft
	Ir. J. Petri-König,	AMD Research Engineer
	Ing. K. McElligott	Alten

TU Delft

Preface

During my master's study, I developed a strong interest in machine learning and hardware acceleration. My interest in these two fields led me to this master's thesis. I quickly realized that the thesis process differs vastly from studying for exams. Fortunately, I did not have to face this time on my own. I want to express my sincere appreciation to my parents, Björn & Esther; my sister Danielle and my girlfriend Lonneke; roommates Merel, Ivo,& Aileen; and all my friends for their unwavering support and encouragement throughout my academic journey.

Furthermore, I want to thank my supervisors, Zaid Al-Ars, Jakoba Petri-König, Kevin McElligott, and Christiaan Boerkamp, for their exceptional guidance, support, and motivation throughout my thesis project. I want to express extra gratitude to Zaid and Jakoba, who went above and beyond to support me, despite their busy schedules.

Patrick Geel
Delft, April 2023

Abstract

The demand for implementing neural networks on edge devices has rapidly increased as they allow designers to move away from expensive server-grade hardware. However, due to the limited resources available on edge devices, it is challenging to implement complex neural networks. This study selected the Kria SoM KV260 hardware platform due to its affordability and sufficient hardware capabilities for creating a resource-constrained environment. By leveraging the hardware acceleration capabilities of the FPGA for specific nodes of the MobileNetv1 model and offloading other nodes to the onboard quad-core ARM cortex-A53 CPU, it was feasible to implement a neural network on a hybrid combination of CPU and FPGA. Results showed that when executing the MobileNetv1 model in a hybrid configuration, a total runtime improvement of 2.8x over a pure CPU implementation can be achieved. The study concludes that node-wise partitioning of the MobileNetv1 model is a practical solution. This approach offers a cost-effective solution for users who seek an accessible way to run neural networks without the need for expensive server-grade hardware.

Contents

List of Figures

List of Tables

Acronyms

AI	artificial intelligence.
ASIC	application-specific integrated circuit.
CLB	configurable logic block.
CPU	central processing unit.
DNN	deep neural network.
DSP	digital signal processing.
FPGA	field programmable gate array.
GPU	graphics processing unit.
ICs	integrated circuits.
IOB	input/output block.
LUT	lookup table.
ML	machine learning.
NN	neural network.
PCB	printed circuit board.
PE	processing element.
SIMD	single input multiple data.
SoC	system on chip.
SoM	system on module.

1 Introduction

Artificial intelligence (AI) and machine learning (ML) are becoming increasingly important topics in science and technology. Researchers in academia and worldwide industries are solving everyday problems using ML, from medical imaging devices to detect tumor cells in scans[1]–[3]; to marketing to target which ads are best to show to which user. A specific type of machine learning algorithm modeled after the human brain's structure and function is a neural network (NN), which consists of interconnected nodes or neurons that process and transmit information—allowing the network to learn patterns and make predictions or classifications [4]. NNs are particularly effective for tasks that involve complex input-output mappings, such as image recognition and natural language processing.

As AI and ML become increasingly prevalent in various industries, the need for specific hardware to run these models has become more prominent [5]. The hardware required for AI and ML applications can vary depending on the specific needs of the task but typically includes graphics processing unit (GPU), field programmable gate array (FPGA), application-specific integrated circuit (ASIC), and central processing unit (CPU). To train a neural network model for image recognition, designers must teach such a network the correct solution for a given image. Such a process is called the training phase. Like the human brain, we learn to recognize patterns in images or structures in faces to identify an object or person. The structure of a NN works similarly to this; during the training phase, we input data to the neural network model, and the network learns the correct outcome and is corrected when it does not predict correctly. These decisions are made by large interconnects of neurons; each neuron performs mathematical operations; during training, we adjust the weights and biases of each neuron. Changing the weights and biases of a neuron affects how strongly the neuron responds. During the inference phase, the neural network is given unseen data, making predictions based on the data it was trained on.

Models are trained on server-grade hardware designed to handle large amounts of data and perform complex computations. Training a model is often done using GPUs as teaching a model costs significant amounts of memory and computational resources due to the many parameters that need to be learned [6]. Server-grade hardware is optimized for high performance and often has multiple processors, large amounts of memory, and other specialized hardware components. Inference uses pre-trained weights and biases, requiring less computational and memory load than training. Once a model has been trained, it can be deployed in the field. As many neural network models are designed to run near the data source to process data in real-time, models are often deployed to edge devices. These edge devices are typically smaller, less powerful, and less expensive than server-grade hardware [7]. The mobility of edge devices makes them an attractive alternative to server-grade hardware, enabling users to deploy neural networks in mobile applications such as drones, smart IoT, hospital patient monitoring, and more. Edge devices are optimized for low power consumption but contain limited processing power and memory. Edge devices are a low-cost, low-resource alternative to server-grade hardware; however, challenges do arise when performing inference on edge devices; this research aims to uncover a method to accelerate machine learning on edge devices. The following section will provide more context into the goals of this research.

1.1. Context

Current state-of-the-art research in machine learning on edge devices shows techniques such as pruning, quantization, and altering network architectures [8] to reduce the computational complexity of neural network models. While this reduces the computational and memory load, models often still can not be fully implemented on edge devices as the resources of the chosen platform limit them [9].

This research aims to find a method to make inference on edge devices possible. We specifically investigate how we can leverage the strengths of CPU and FPGA to perform inference of NN on edge devices. Using quantized neural networks reduces the computational complexity of the system [10]. By identifying which layers of a quantized neural network benefit from FPGA acceleration. We aim to devise a partitioning algorithm to split and deploy neural networks on resource-limited devices. By leveraging the strengths of both FPGAs and CPUs, this research aims to deliver a method for inference of complex neural network models on edge devices with limited resources, such as the Kria SoM KV260, which is a System on Module (SoM) that combines a quad-core Arm Cortex-A53 (referred to as CPU in this thesis) with a Zynq UltraScale+ MPSoC (XCK26), which is specifically designed for the platform. This platform is selected for this study as it offers a CPU + FPGA environment that constrains resource requirements allowing this research to experiment with a resource-limited environment. A dataflow-style architecture is desired to distribute a neural network to the Kria SoM KV260, as dataflow architectures are highly flexible and scalable in design. Such flexibility and scalability allow for implementation to be easily altered. This project collaborates with Advanced Micro Devices (AMD), formerly known as Xilinx Research Labs, who are interested in this work, to help them find a method to advance the world of AI and hardware acceleration for resource-limited devices. The AMD research team developed the FINN [11] (Framework for Fast, Scalable Binarized Neural Network Inference) project. FINN is a dataflow-style architecture used to build hardware platforms that can efficiently perform inference on binarized neural networks, which are a type of neural network that represent weights in low precision, rather than using floating-point values, making them well-suited for edge devices. Low precision is desirable for edge devices as it reduces the computational load while consuming less power and requiring less memory.

The proposed research aims to advance applications such as autonomous drones, smart cameras, and intelligent IoT devices in edge computing scenarios. We aim to display the possibility of executing models in resource-limited environments. By enabling the execution of neural networks in resource-limited environments, the opportunity arises to deploy neural networks on edge devices, bringing AI closer to the source and allowing for less dependency on server-grade hardware.

1.2. Problem statement & research questions

The main problem that this research address is as follows:

- *How can we leverage the unique strengths of CPUs and FPGAs in a hybrid computing architecture, to optimize the performance and efficiency of specific neural network models?*

To answer the problem statement, we need to addresses the following sub-questions:

1. Is a hybrid FPGA+CPU implementation a feasible and cost-effective solution for running neural networks compared to a pure CPU implementation?

2. What are techniques for partitioning and distributing neural networks between FPGAs and CPUs?

3. How do the partitioning and distribution of neural networks between FPGA and CPU affect performance metrics, such as throughput, latency, and cost, and what factors influence the optimal distribution strategy?

1.3. Thesis outline

The remainder of this thesis is broken up into the following chapters.

Chapter 2 presents essential background information to provide the foundation necessary to address the problem statement. It provides the necessary knowledge and understanding to help answer the problem statement and research questions.

Chapter 3 supplies the methodology necessary to answer the problem statement and research questions. This chapter will explore possible solutions and approaches to optimize performance and efficiency for neural network applications by effectively leveraging the unique strengths of CPUs and FPGAs in a hybrid computing architecture.

Chapter 4 discusses the steps taken to implement the potential solutions from the previous chapter. Here a detailed explanation is given to the steps taken in realizing the implementation.

In **Chapter 5** we discuss the proposed experiments that will be conducted to test the effectiveness of the proposed algorithms. The results of these experiments are presented in this chapter. It includes graphs and charts to illustrate the data obtained. The chapter describes the results, thoroughly discussing any limitations or implications of the research.

In **Chapter 6**, conclusions are drawn based on the previous chapters. The research questions are summarized and answered, along with recommendations for improvements that can be made in future works.

2 Background

This chapter contains background knowledge used in this research to help answer the problem statement and research questions outlined in Chapter 1. This section of the report is structured as follows, what an FPGA is and its general architecture is presented in Section 2.1. Heterogeneous computing is explained in Section 2.2, and information is provided on the difference between sequential and parallel programming. Section 2.3 briefly introduces what neural networks are and the common terminology used when discussing them. Section 2.4 presents Open Neural Network Exchange (ONNX), onnxruntime, and the FINN framework.

2.1. Field programmable gate array (FPGA)

The FPGA proposed by Xilinx in 1984 was introduced to expedite the design phase of application-specific integrated circuits (ASICs) [12]. By utilizing FPGAs, engineers could create, code, authenticate, and confirm hardware designs without fabricating tangible integrated circuits (ICs).

Since the 1980s, the use of FPGA has expanded beyond designing and testing ASICs. Today, FPGAs have diverse applications in digital signal processing, high performance computing, the internet of things, automotive, aerospace technology, security, encryption, and more. FPGAs allow designers to access customized low-power and high-performance solutions.

2.1.1. General FPGA architecture

Although FPGAs differ slightly depending on the model, they can be broken down into these 3 basic components as shown in Figure 2.1.

The basic building blocks of an FPGA are the configurable logic blocks (CLBs) (named logic blocks in Figure 2.1), comprised of lookup tables (LUTs), flip-flops, and multiplexers that can be configured to implement any digital logic function. The number and organization of CLBs can vary between different FPGA architectures, and the functionality of the LUT within each CLB can also vary. Input/output blocks (IOBs) interfaces the FPGA with external devices, providing the physical connection between the FPGA pins and the external peripherals. A designer can customize the configuration of the IOBs to meet specific requirements. Finally, programmable interconnects are the internal connections within an FPGA that allow different logic blocks to communicate. Hardware designers configure a routing resource matrix to create custom connections between different blocks of the FPGA, enabling complex signal paths between logic elements using hardware description

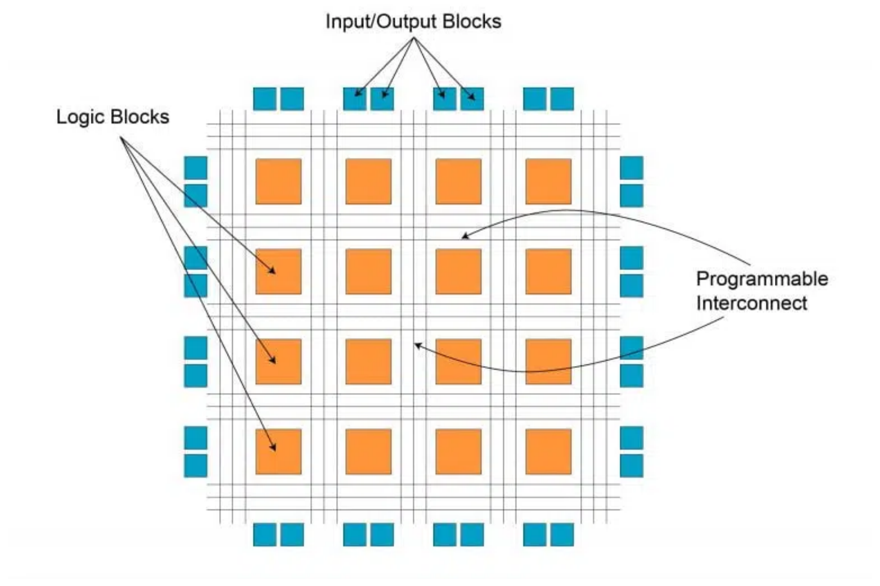


Figure 2.1: A simplified version of the FPGA architectures, figure from [13]

languages such as VHDL or Verilog. Designers can achieve high performance and low power consumption in custom digital circuits by utilizing FPGAs, which offer a key advantage through the flexibility of their programmable interconnects.

2.1.2. FPGA resources

This section explores the resources available on common FPGAs, including logic elements, memory blocks, and digital signal processors. The available resources for the Kria KV260 Vision AI Starter Kit [14] are also shown, as this is the chosen platform for this research. In Table 2.1, the resources for the KV260 starter kit can be found along with the PYNQ-Z2, Ultra96v2, and the Alveo U280, which is a server grade FPGA. These platforms depict the differences between edge devices and compare these to a server-grade FPGA.

Platform	**Available IOBs**	**LUTs (x1000)**	**FlipFlops (x1000)**	**Block RAMs**	**Ultra RAMs**	**DSPs**
Kria KV260	189	117.12	234.24	144	64	1248
PYNQ-Z2	125	53.2	106.4	140	0	220
Ultra96v2	82	70.56	141.12	216	0	360
Alveo U280	624	1304	2607	2016 (36Kb)	960 (288Kb)	9024

Table 2.1: FPGA resources for the Kria KV260, PYNQ-Z2, and the Ultra96v2.

As mentioned in Section 2.1.1, a CLB is made up of LUTs, flip-flops, and multiplexers. The resources available on a given FPGA differ based on the manufacturer. A designer selects a platform based on a neural network model and performance to meet the designer's requirements.

Block RAM is a specialized memory block within an FPGA that can store data or program code. Block RAMs can be configured to have different sizes and organizations depending on the neural network requirements. They can be used for implementing functions like first-in-first-out buffers (FIFOs) and state machines. Digital signal processing (DSP) blocks are specialized hardware blocks within an FPGA optimized for implementing signal processing algorithms like filtering, FFTs, and convolutions. DSP blocks often include features like high-speed multipliers, adders, and accumulators, which can implement complex digital signal processing pipelines with low latency and high throughput.

LUTs are essential components of FPGAs that allow digital logic functions to be implemented through pre-defined truth tables. Each LUT can be considered a small memory unit that receives input signals and returns the corresponding pre-defined value output. The LUTs are often connected to create complex logic functions, with the output of one LUT feeding into the input of another. Optimizing the use of LUTs is critical to achieving maximum performance and efficiency in FPGA designs.

2.1.3. System on module (SoM) FPGAs

As mentioned in Section 2.1.2 the selected platform for this research is the Kria KV260 Vision AI Starter Kit, a system on module (SoM). A SoM differs from a system on chip (SoC); therefore, this report section explains the difference.

A SoC and SoM are technologies used in embedded systems design but have different architectures and purposes. A SoC is a single integrated circuit containing all the components of a complete electronic system. It typically includes a processor, memory, peripherals, and other system components integrated onto a single piece of silicon. FPGAs are often used in SoCs as they allow for the creation of highly customized, application-specific computing systems that can perform complex tasks at high speeds. Note that the Ultra96 and the PYNQ-Z2 shown in Table 2.1 are SoC implementations.

A SoM is a production-ready printed circuit board (PCB) that integrates essential components of an embedded processing system, such as processor cores, communication interfaces, and memory blocks [15]. SoMs are designed to be easily embedded into various end systems, ranging from robots to security cameras. By using a modular approach, it is possible to provide a convenient and efficient way to integrate computing power into multiple applications. This allows designers and developers to focus on the specific features and functions of their products, without having to worry about the underlying hardware and software infrastructure.

2.2. Heterogeneous computing

Heterogeneous computing is a computing platform containing more than one type of computing resource, with each resource optimized for different tasks [16]. Combining the following homogeneous computing systems [17] multicore CPUs, graphics processing units (GPUs), FPGAs, and ASICs results in a heterogeneous computing system (i.e., FPGA+CPU). Each one of these platforms has its advantages and disadvantages. ASICs are the least reprogrammable as they are designed for specific tasks. CPUs are also not reconfigurable but are designed to perform a wide range of tasks, making them more versatile than ASICs. GPUs are platforms that handle the demands of computer graphics and video processing. They can perform many simple calculations in parallel, making them highly energy efficient. Regarding re-programmability, GPUs lie between CPUs and FPGAs, as they are not as reprogrammable as FPGAs, but more versatile than CPUs. FPGAs are highly reconfigurable and can be programmed to perform various tasks. The re-programmability and energy efficiency of FPGAs make them well-suited for prototyping. Figure 2.2 shows the platforms mentioned above and how they scale in terms of re-programmability and energy efficiency.

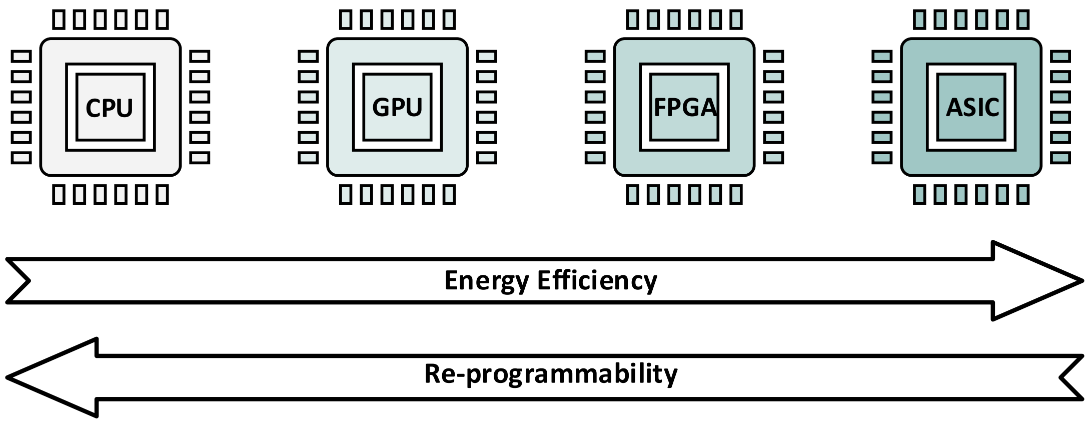
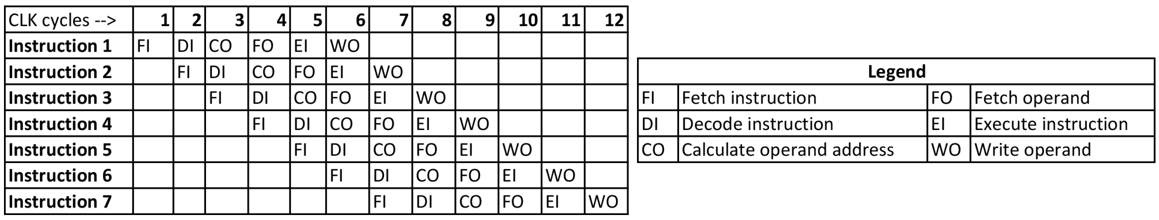


Figure 2.2: Different computing systems ordered in terms of re-programmability and energy efficiency

2.2.1. Sequential processing

Sequential processing refers to the execution of instructions in sequential order, also known as the instruction pipeline. The CPU fetches each instruction from memory, decodes it to determine the operation to be performed, executes the operation, and then stores the result in a register or memory location [18]. This process is repeated for each instruction in the program, with each instruction executed in sequence, one after the other. Methods such as pipelining exist, which improve the throughput of the CPU [19]. Figure 2.3 shows an example of a fully pipelined instruction pipeline. While sequential processing is generally more straightforward, it may not be well suited for large or complex tasks. The sequential nature of CPU processing allows complicated tasks to be broken down into simple operations that can be executed in a predetermined order.



CLK cycles -->	1	2	3	4	5	6	7	8	9	10	11	12
Instruction 1	FI	DI	CO	FO	EI	WO						
Instruction 2		FI	DI	CO	FO	EI	WO					
Instruction 3			FI	DI	CO	FO	EI	WO				
Instruction 4				FI	DI	CO	FO	EI	WO			
Instruction 5					FI	DI	CO	FO	EI	WO		
Instruction 6						FI	DI	CO	FO	EI	WO	
Instruction 7							FI	DI	CO	FO	EI	WO

Legend			
FI	Fetch instruction	FO	Fetch operand
DI	Decode instruction	EI	Execute instruction
CO	Calculate operand address	WO	Write operand

Figure 2.3: An example of an instruction pipeline, figure modified from [18].

2.2.2. Parallel processing

FPGAs allow for parallel processing, which allows for the simultaneous execution of multiple operations by dividing them into smaller tasks and distributing them in various processing units. Parallel processing in FPGAs is achieved by programming the logic gates on the FPGA to perform specific functions. This flexibility allows for custom processing units optimized for specific tasks, resulting in improved performance and energy efficiency. Parallel processing can achieve high levels of performance and efficiency but requires specialized hardware and software to implement [20].

2.2.3. CPU+FPGA

A CPU + FPGA computing architecture is used in this research. Combining these two platforms provides multiple benefits [21], discussed in this section regarding the heterogeneous platform. A combination of a CPU and FPGA can handle specific tasks in parallel, resulting in faster processing times and higher performance than a single CPU processor while utilizing the performance of each platform. The FPGA's re-programmability provides greater flexibility in the system, which is particularly useful in applications where requirements change frequently. Additionally, the CPU and FPGA can allocate resources efficiently, leading to better utilization of the available resources. The FPGA can also offload specific tasks to the CPU, freeing up resources on the FPGA and improving overall performance. An FPGA can improve power efficiency as it can be configured to perform specific tasks with low power consumption compared to a general-purpose CPU.

Applications for CPU + FPGA

Many use cases exist for CPU + FPGA hybrid models, such as video, image compression, automotive, autonomous vehicles, network processing, and high-performance computing. Besides these use cases, a CPU+FPGA hybrid may also be beneficial in a resource-limited environment. As stated in Section 2.1.2, there is a big difference in the type of FPGA used. A smaller FPGA, such as the Kria SoM KV260, may be unable to implement the application with the available resources. While a server-grade FPGA such as Alveo U280 can implement the application as it has more available resources. As a designer, it may not always be possible to use a server-grade FPGA; therefore, this scenario of a limited resource environment may present itself. The solution may be a hybrid combination of CPU+FPGA. In the NN domain, CPU and FPGA hybrid systems can be beneficial because they provide a powerful platform for accelerating NN computations. NNs require massive amounts of analysis; by combining CPUs and FPGAs, users can take advantage of the parallel processing capabilities of FPGAs to accelerate inference phases of NNs.

2.3. Neural Networks

A neural network (NN) can be seen as a function $f(x)$, which takes in some input x and returns an output [22]. A fundamental building block of the NN is the perceptron introduced in 1958 [23]. Figure 2.4 shows a snapshot of the perceptron. Each node is called a neuron; the associated edges are called synapses. Each edge has an associated weight that represents the strength of the connection between the two neurons. The weight of an edge determines the amount of influence one neuron has on another neuron. In a neural network, the edge passes data from the input to the output. The weights of the edges are trained during a phase known as training. During this process, the weights are adjusted so the neural network can accurately perform a task, such as an image classification.

Each neuron receives input from the previous layer and applies mathematical operations, including a dot product of the input and the weights, adding a bias term, and applying an activation function. The activation function is a non-linear operation that introduces non-linearity to the model, allowing the network to learn more complex relationships between inputs and outputs. The Rectified Linear Unit (ReLU) activation function is a popular choice in many neural networks because it is computationally efficient. Typically, neural network architectures consist of a series of layers, each containing multiple neurons connected by edges with associated weights. As shown on the right in Figure 2.4.

Fully connected layers

Fully connected layers are a layer that is common in neural network models. A fully connected layer consists of neurons, each neuron in one layer is connected to all the neurons in the next layer and all neurons in the previous layer, as shown in Figure 2.4. The purpose of a fully connected layer is to learn a non-linear mapping between the input and output of a given dataset. Each neuron in a fully connected layer applies a non-linear activation

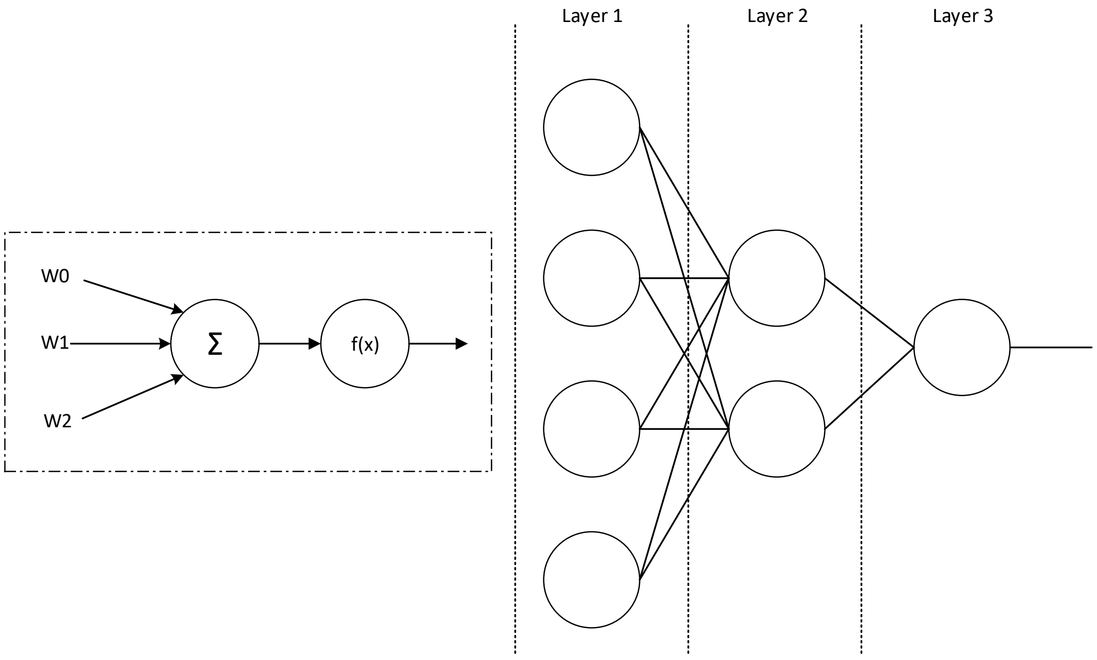


Figure 2.4: **Left**: is a snapshot of a single perception. **Right**: is an example of a single layer of a neural network

function to the weighted sum of its inputs, which allows the network to learn complex relationships between the input and output. Typically this activation function is the rectified linear unit (ReLu); non-linear operation replaces negative values with zero to introduce non-linearity in a neural network model, denoted by the equation:

$$y = max(0, x) \tag{2.1}$$

It is worth mentioning that fully connected layers are widely used in deep neural network (DNN) architectures. However, fully connected layers can be memory and computation intensive, especially for large datasets such as imagenet [24]. The number of neurons in the layer and the number of input features determine the computational complexity of a fully connected layer. It's typically O(m*n) for a fully connected layer with m input features and n neurons, where m and n are the input sizes and the number of neurons, respectively.

2.4. Software resources

This thesis section will introduce the software tools used in this research. First, an introduction will be given to Open Neural Network Exchange (ONNX), an open-source format for representing machine learning models. An introduction is given to onnxruntime, the software used to execute these ONNX models. Concluding with an introduction to the FINN framework and the build flow of the tool.

2.4.1. ONNX

ONNX (Open Neural Network Exchange) is an open standard for representing deep learning models in a format that is both flexible and computationally efficient [25]. This standard has been developed to encourage the exchange of deep learning models between different software frameworks and hardware platforms. The development of ONNX is a response to the fragmentation observed in the deep learning community [26], where models built in one framework may not be compatible with other tools and platforms.

ONNX is built using Google's Protocol Buffers (protobufs) [27]. These buffers provide a language and platform-independent way to structure data, similar to the JSON format. The protobufs format is ideal for dataflow architectures. Such buffers are designed to store data in a compact form that can be quickly parsed by many different programming languages enabling a seamless exchange between various programming languages. The ONNX standard is designed with both performance and compatibility in mind. It supports many deep learning models and operations, making it well-suited for deployment on various hardware platforms, including cloud-based systems, edge devices, and embedded systems. ONNX can be seen as a mathematical functions programming language, as the ONNX language contains all necessary operations to implement machine learning inference functions. An example of generating a linear regression can be seen in Listing 2.1. Such code can then be used to create an ONNX graph. The ONNX graph generated for the given linear regression can be seen in Figure 2.5.

```python
def onnx_linear_regressor(X):
    "ONNX code for a linear regression"
    return onnx.Add(onnx.MatMul(X, A), B)
```

Source Code 2.1: Example python code to generate a ONNX linear regression taken from [28].

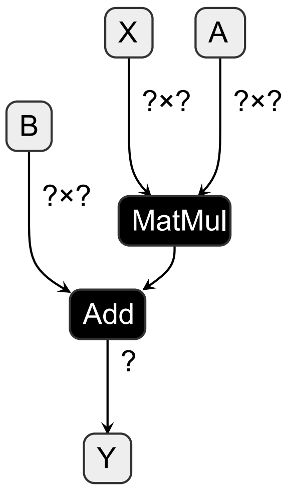


Figure 2.5: Linear regression ONNX graph taken from [28]. The nodes represent operations, while the edges represent the inputs and outputs of the operation. The question marks are where the shape of the data should be shown.

The above example can be expanded using frameworks such as PyTorch [29] or TensorFlow [30] to generate NN models. The trained model can be exported to the ONNX format, representing the model as an ONNX graph. A computational graph in ONNX consists of a set of nodes representing mathematical operations, and directed edges serve as the data flow between operations, as seen in Figure 2.5. Each node in such an ONNX graph is of the protobuf format, which contains information about the node's type, shape, and other parameters. The edges represent the inputs and outputs of the operation, allowing data to flow from one operation to another. Such a graph can then be serialized into one contiguous memory buffer that can be run using onnxruntime.

2.4.2. Onnxruntime

Onnxruntime is an open-source project by Microsoft to accelerate machine learning [31]. It is a cross-platform, high-performance inference engine for machine learning models in the

ONNX format. This allows users to leverage the best features and optimizations of different frameworks while still being able to use the exact model representation. ONNX runtime is a highly optimized and flexible system for executing ONNX models. It takes advantage of hardware-specific optimizations and provides advanced features to ensure that models are executed as efficiently as possible. Onnxruntime automatically parses through a model to identify optimization opportunities and provides access to the best hardware acceleration available.

2.4.3. FINN

FINN is a framework developed by AMD [11], which aims to investigate the implementation of DNN inference on FPGAs. The FINN framework provides designers an end-to-end flow that gives developers access to specialized hardware architectures. Dataflow architecture and the custom precision for few-bit weights and activations are two techniques that FINN leverage. Dataflow architecture in FINN provides dedicated hardware for each layer in a NN model. Using a dataflow architecture has the advantage that each layer can contain a different amount of hardware resources proportional to the compute resources of the device. The inputs can then be streamed through the device like a pipelined architecture. The advantage of this architecture is that only the resources required are utilized; in terms of latency, streaming architectures do not need to buffer data, which results in higher throughput. The FINN dataflow architecture is very specific for each model implemented using FINN, giving a designer a high level of control when it comes to tuning a model's performance on hardware. A designer can control the model performance by changing the folding setting of the FINN framework. Folding refers to the number of resources dedicated to a neural network that increases or decreases a model's throughput. A folding setting of 1 indicates fully unfolded, meaning that every neuron in a NN model has dedicated hardware. Resulting in maximum performance and means that the neural network will classify at the clock rate of the FPGA. A higher folding setting results in fewer resources for the neural network by proportionally removing resources from each layer, decreasing the network's throughput.

A designer can alter specific parameters, such as the number of processing elements (PEs) and the depth of the single input multiple data (SIMD) lane, to help control the number of resources used. Using more resources increases the throughput, with the result that a smaller portion of the neural network can be implemented on an FPGA. A Matrix–Vector–Threshold Unit (MVTU) forms the computational core for the FINN framework, shown in Figure 2.6.

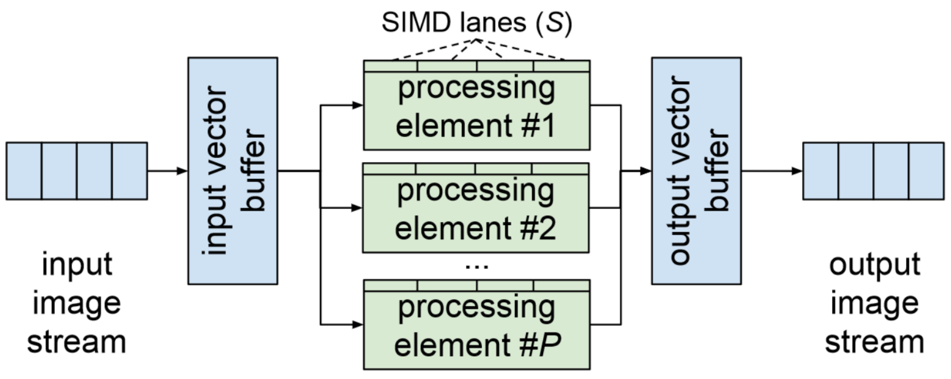


Figure 2.6: Overview of MVTU, figure is from paper [11]

Such a core comprises two components the SIMD lanes (S) and the processing elements (PE). Each PE is made up of SIMD lanes. Changing the folding setting for a given network alters the number of PEs and depth of SIMDs. Changing these two values affects the throughput of the neural network. The number of cycles needed to complete one matrix-vector multiply is Equation 2.2. X and Y refer to a matrix's height and width, respectively, and PE and SIMD are the set values.

$$F = \frac{X}{PE} x \frac{Y}{SIMD} \tag{2.2}$$

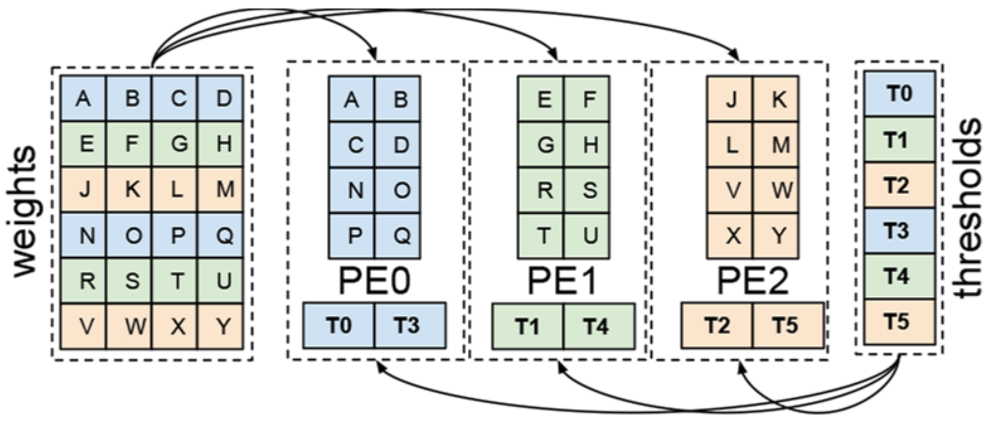


Figure 2.7: Neuron and weight folding for MVTU [11].

Figure 2.7 shows an example of the mapping of a 6x4 with SIMD equal to 2 and PE equal to 3. As changing the values of PE and SIMD affect performance, we, as the designers, can manually change these by providing a folding file to the FINN build flow. There is also the possibility to let FINN auto-set the folding values for a given network, each layer containing an MVTU will be set differently to maximize the resources used.

Custom precision for few-bit weights and activations is the second technique that FINN exploits. The idea is to reduce the weights and activation size during training while maintaining a respectable level of accuracy. Reducing the bit precision reduces the hardware footprint, allowing for more of a NN model to be implemented on a hardware platform. The end-to-end flow can be seen in Figure A.1, and each block is discussed below. The FINN end-to-end flow starts with exporting a neural network model in brevitas. The exported model then undergoes a network preparation step shown in Figure 2.8. Upon completion of the network preparation step, a hardware build can be made of the provided neural network. The network is then ready to be deployed to the FPGA platform.

Streamlining is a step in the network preparation step of the FINN framework, and it is the technique used to optimize models by eliminating floating-point operations. These transformations can include moving the operations around, collapsing them into a single operation, and converting them into multithresholding nodes. For more information on streamlining, refer to [32]. Once this step is completed, the resulting ONNX model will contain custom and standard nodes. During the convert to high-level synthesis (HLS) step, standard or custom layers are transformed into HLS layers, which map directly to a finn-hlslib function call. By converting the model to HLS layers, the transformation allows the resulting implementation to utilize specialized hardware circuits to perform these operations. Following the convert to the HLS step, the ONNX graph consists of HLS and non-HLS layers.

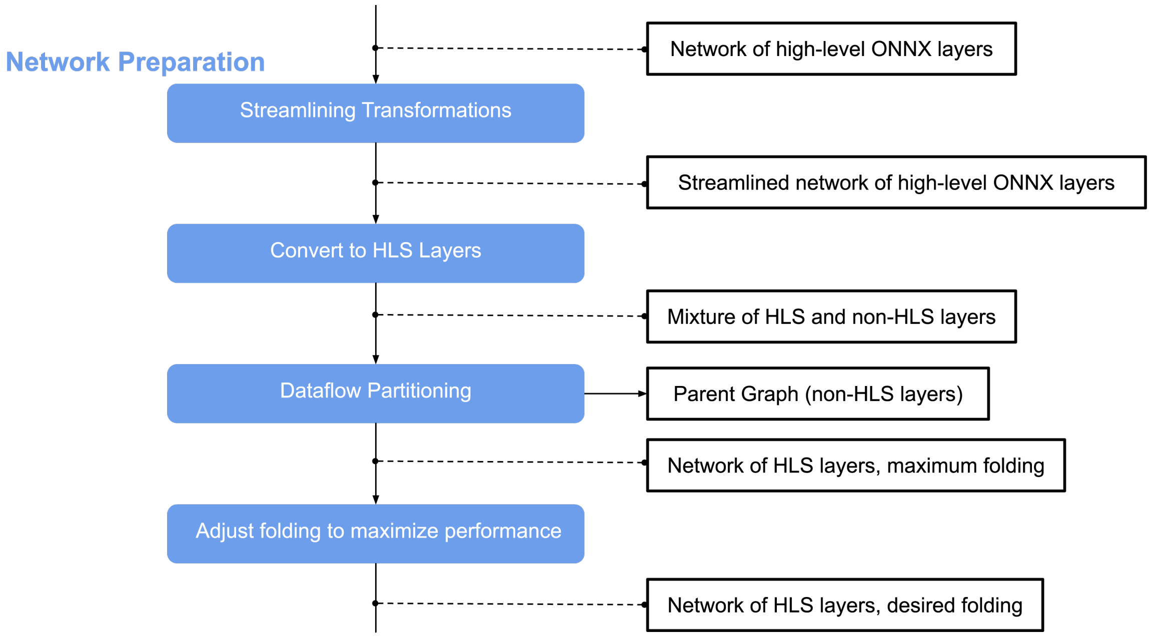

Network Preparation
Network of high-level ONNX layers
Streamlining Transformations
Streamlined network of high-level ONNX layers
Convert to HLS Layers
Mixture of HLS and non-HLS layers
Dataflow Partitioning
Parent Graph (non-HLS layers)
Network of HLS layers, maximum folding
Adjust folding to maximize performance
Network of HLS layers, desired folding

Figure 2.8: Network preparation steps, a snippet from Figure A.1

The dataflow partition step splits the graph to create two graphs, the parent and child (named partition graph in this thesis). The parent graph contains the non-HLS layers, while the HLS layers are in the partition graph. Folding is the final step in the network preparation step, and it is the process that sets the number of parallel operations in a node of the partition graph. Higher parallelization increases the neural network's throughput but increases resource usage. This trade-off creates a design space for the designer, enabling them to choose between implementing a neural network with a high throughput or a lower throughput with lower resource usage. Folding works by grouping multiple computations into a single operation, reducing the number of parallel operations required to compute the neural network. The FINN compiler can perform the folding step automatically, which uses a folding algorithm to optimize the hardware design. However, supplying a file that manually sets the folding constraints is also possible, allowing the designer to control how resources will be used.

To execute the DNN on the intended hardware platform, the hardware build step shown in Figure 2.9 involves generating an HLS IP for each layer, creating a stitched design, and performing hardware synthesis to produce a bitfile and PYNQ Python driver.

After creating HLS IP blocks per layer and stitching them into a single design, the FINN framework hardware build process results in an optimized hardware design. Creating HLS IP per layer step uses Vivado HLS tools to convert each layer from the graph into custom IP blocks. This approach leads to more resource-efficient usage, simplified design, and improved performance through pipelining and parallelization. These IP blocks can be reused across different accelerator designs, making it possible to build a library of standard IP blocks. In the stitched design step, the custom IP blocks are integrated into a single design using a dataflow-based stitching approach that enables efficient communication between different layers. The resulting design can be implemented on an FPGA using Vivado or Vitis. The hardware build process includes synthesis, place, route, and bitstream generation steps using Vivado. These steps involve translating the hardware description into a netlist, determining the optimal physical location and interconnect resources for logic blocks, and generating the configuration file for the FPGA. The FINN framework generates

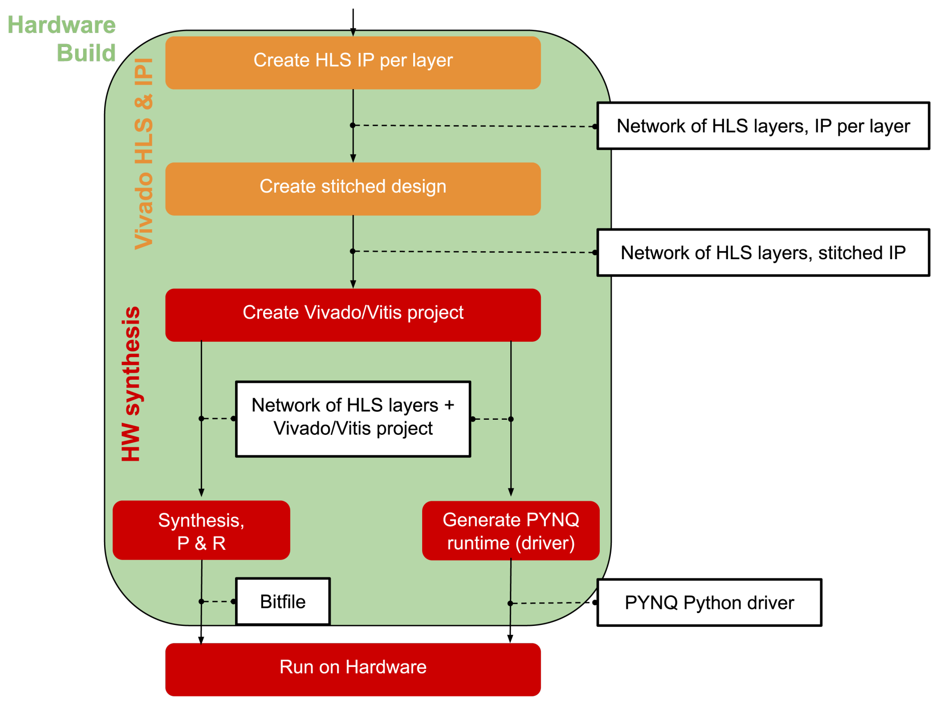

Hardware Build
Vivado HLS & IPI
HW synthesis
Create HLS IP per layer
Network of HLS layers, IP per layer
Create stitched design
Network of HLS layers, stitched IP
Create Vivado/Vitis project
Network of HLS layers + Vivado/Vitis project
Synthesis, P & R
Generate PYNQ runtime (driver)
Bitfile
PYNQ Python driver
Run on Hardware

Figure 2.9: Hardware build steps, a snippet from Figure A.1

a PYNQ python driver file that works with the generated bitfile to download the bitstream to the target FPGA and run it.

Estimation report FINN

Running for an estimate-only (no hardware synthesis) is possible in the FINN build flow. After setting the folding configurations of the model, FINN generates estimate reports. A designer can use these estimation reports to gauge metrics such as estimated resource usage, network performance, and clock cycles per layer. Such metrics allow a designer to decide if the design meets set requirements before performing hardware synthesis, which can take hours depending on the model's size.

2.5. Alternative solutions

To highlight and compare alternative solutions, this thesis section will present alternative solutions for heterogeneous computing and machine learning in resource-limited environments. We will delve into various solutions in this thesis section. Three end-to-end accelerators for deep learning on mobile devices will be discussed TVM, hls4ml, and Vitis AI [33].

TVM is an open-source deep-learning compiler for CPUs, GPUs, and specialized accelerators; that provides performance portability to deep-learning workloads across diverse hardware back-ends [34] by exposing graph/operator level optimization to deploy the workload to mobile devices such as embedded devices, FPGAs, and ASICs. TVM solves deep learning challenges by using high-level operator fusion, mapping to arbitrary hardware primitives, and memory latency hiding. TVM uses the high-level operations of a computational graph to perform optimization, such as operator fusion and data layout transformations. TVM offers users a general-purpose tool for optimizing machine learning models across various hardware platforms.

Hls4ml is a Python plugin for machine learning, designed to allow designers to quickly prototype a machine learning algorithm for implementation on FPGAs by taking a high-level representation of an ML model such as PyTorch or Tensorflow and generating synthesizable hardware description language (HDL) [35]. Enabling a seamless implementation of a machine learning model on FPGAs, hls4ml includes network optimization techniques, such as pruning and quantization-aware training. The goal of hls4ml is to translate machine learning algorithms for implementation on FPGAs and ASICs.

Vitis AI is a development environment that designers can leverage to accelerate AI models on Xilinx platforms [33]. It provides developers with a high-level programming interface and toolchain to accelerate their deep-learning applications. Vitis AI is made of the following key components. DPUs, which are computation engines designed to optimize a convolutional neural network, DPUs are efficient and scalable IP blocks that can be changed to meet an application requirement. Vitis AI contains a quantizer which is a tool that supports model quantization. The compiler of Vitis AI compiles the quantized model for the intended target, and runtime (VART) inferences the model on the embedded application.

The previous three presented works aim to design, optimize and deploy neural network models on dedicated hardware platforms. While each tool uses different techniques to achieve this, they all aim to improve performance, reduce latency and increase the throughput of a neural network model. These three presented works pose that advances in embedded devices (edge devices) have pushed researchers to develop methods to deploy neural networks to hardware devices in a performance-efficient manner.

As the Internet of Things (IoT) evolves, hardware is needed to infer DNN on edge. As edge devices often have limited computational resources and small storage. Currently, the industry heavily relies on cloud computing and server-grade hardware. This dependency presents several technical challenges, including latency, reliability, and privacy concerns [8]. There is a growing interest in on-device computation and storage to address these issues, which is essential for many time-critical industrial IoT applications that require real-time processing. Researchers are exploring three major research areas to deploy DNN models on edge devices: quantization, pruning, and network architecture design. These techniques help to reduce the computation and space complexity of DNN models, making them applicable to industrial IoT devices.

3 Methodology

This chapter supplies the methodology necessary to answer the problem statement and research questions 1, 2, and 3 as defined in Chapter 1. This chapter will explore possible solutions and approaches to optimize performance and efficiency in neural network models by effectively leveraging the unique strengths of CPUs and FPGAs in a hybrid computing architecture. As the goal is to develop an algorithm, Section 3.1 will elaborate on the design choices made in developing the two algorithms. The information discussed in this section is later used for the implementation chapter.

3.1. Design flow

In this section of the report, we present an overview of the design flow used to develop an algorithm that leverages the distinctive capabilities of both CPUs and FPGAs in a hybrid computing architecture. With the primary objective of optimizing performance and efficiency for a neural network model, as specified in the problem statement. Figure 3.2 shows the implementation eventually arrived at. The Kria SoM KV260 platform is chosen for this work as it is a cheaper alternative to server-grade hardware and provides a CPU + FPGA environment that we can use to experiment on. The Kria SoM KV260 contains fewer resources than a server FPGA making this a suitable option to experiment with resource-limited environments. We choose FINN as the software framework, as it is a framework that is structured in a dataflow-style architecture, making splitting more natural. This will significantly reduce the time required to develop and deploy custom hardware-accelerated applications. As explained in Section 2.4.3, the FINN framework is a tool used to convert machine learning models trained with high-precision floating-point arithmetic to low-precision fixed-point arithmetic. It enables the deployment of neural network models on edge devices with low power and low memory constraints while preserving model accuracy. We propose an algorithm to address research questions 1, 2, and 3 by partitioning and distributing a neural network in a resource-limited environment.

Design choices

After selecting the FPGA and FINN framework, the focus of this research must shift towards identifying the specific problem we aim to solve. As outlined in the research questions, we aim to develop a partitioning technique and distribute a neural network on a hybrid platform.

The primary objective of this research is to develop an algorithm that efficiently partitions a neural network to optimize its performance on a hybrid platform. To start, we must determine the inputs and outputs of the algorithm. We want to work with a model representation easily portable to various hardware or software platforms. ONNX graphs are easy to work with and simplify partitioning using pre-built Python packages. Thus the selected model representation for this research is a model in the ONNX format. With a model structure selected, we can now explain the overall framework of the algorithm. To distribute a single ONNX model to deploy on a hybrid platform, a partition must be made to enable its distribution. To identify the partition that offers the best performance in terms of total runtime. To obtain this objective, it is key to identify a function that can be used to identify when the partition satisfies the minimum runtime. The minimization function can be expressed using Equation 3.1, where T_{fpga} is defined in Equation 3.2.

$$\min\left(T_{\text{fpga}} + T_{\text{cpu}}\right) \tag{3.1}$$

$$T_{\text{fpga}} = T_{\text{data to device}} + T_{\text{inference time}} + T_{\text{data from device}} \tag{3.2}$$

To reach high-speed computation, low latency, and low power consumption, it is necessary to move data to FPGAs instead of reading it out of memory. Migrating data to an FPGA can improve system performance by reducing the amount of data transferred. Equation 3.2 considers the duration of data movement when evaluating FPGA time. When using an FPGA to execute a neural network, it is necessary to transfer data to the device to take advantage of its high performance, parallel processing, low power consumption, and cost-effectiveness.

Figure 3.1 depicts the three scenarios considered, where the baseline is denoted as a full CPU implementation as an assumption is made that the provided neural network model does not fit within the resources of the given platform.

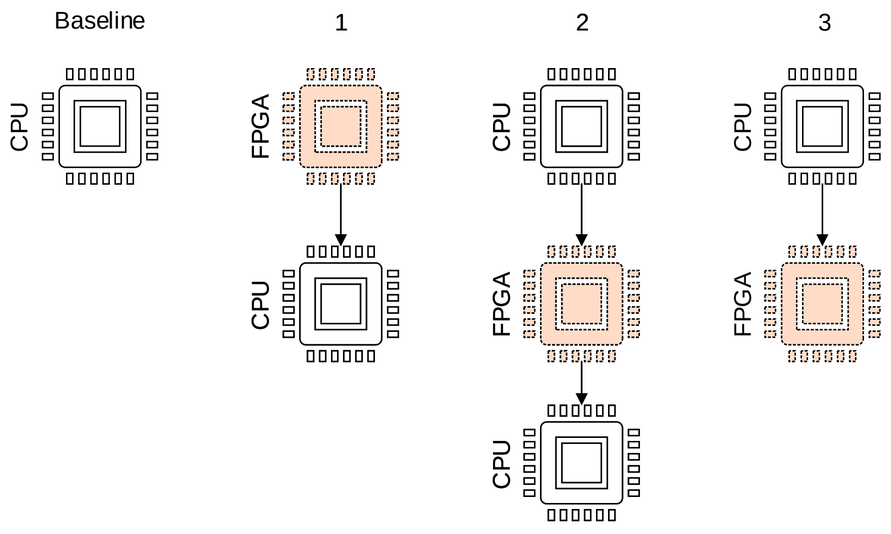


Figure 3.1: Three scenarios when creating subsets from a given ONNX model. The right three are the scenarios tested. The baseline is denoted as a full CPU implementation. The CPU is the quad-core Arm Cortex-A53 on the Kria SoM KV260.

Partitioning Techniques

Several techniques exist for partitioning and distributing neural networks between FPGAs and CPUs. Each partitioning technique has its advantages and disadvantages. Selecting the most appropriate approach can depend on various factors, including computational requirements, available resources, and the complexity of the network architecture.

One technique is layer-based partitioning, which involves partitioning each neural network layer to either the FPGA or the CPU based on computational requirements or available resources [36], [37]. This approach has the advantage of being relatively straightforward to implement. Furthermore, it allows for efficient use of resources and can result in improved performance by minimizing data movement between the FPGA and CPU. Another technique is node-based partitioning, where the neural network is divided into small subgraphs consisting of individual or small groups of layers. Each partition is then distributed to a specific FPGA or CPU based on the node's computational requirements and available resources. This approach can provide fine-grained control over the distribution of computations. For this research, we plan to perform node-based partitioning as we group layers based on the topology of the nodes in the ONNX graph.

Distributing a neural network between an FPGA and a CPU can be done using hardware/software co-design. The idea is to split the workload between the FPGA and the CPU, based on the strengths and weaknesses of the respective device, to achieve the best overall performance. An approach to distributing the neural network is partitioning a NN model into different sub-networks, with each sub-network optimized for execution on either the FPGA or the CPU. This requires analyzing the neural network's computational requirements and identifying which network parts can be executed most efficiently on which platform. For example, convolutional layers, which involve a lot of matrix multiplication, are typically well-suited for acceleration on an FPGA [38]. In contrast, fully connected layers and activation functions can be executed on a CPU.

3.2. Proposed algorithms

An outline of the two primary algorithms implemented in Chapter 4 is given using the information explained above. First, an introduction is given to the exhaustive search algorithm, followed by proposed changes for implementing the second algorithm.

3.2.1. Exhaustive search algorithms

This section explores the design choices and the approach used for the exhaustive search algorithm. An exhaustive search algorithm solves a computational problem by exploring all possible solutions among a finite set. The algorithm selects the solution that best meets the requirements or is closest to the desired outcome. An advantage of an exhaustive search algorithm is that it guarantees to find a global minimum within a finite set. The algorithm explores all possible solutions and evaluates each feasible option. One major drawback of an exhaustive search algorithm is that it can be computationally intensive, as the intensity grows exponentially with the size of the problem, which can quickly become computationally infeasible. The possible subset for the exhaustive search algorithm grows exponentially with the number of nodes ($n!$) with n equal to the number of nodes in a given ONNX graph. Although the algorithm is computationally intensive, it is relatively simple to implement, making it an appropriate starting point for this research.

3.2.2. Setup of algorithm 1

The first proposed solution is to implement an exhaustive search algorithm in a naive implementation, meaning that the algorithm tests all possible partitions. This implementation finds a global minimum, the best runtime for a given neural network model. Below in Figure 3.2, the exhaustive search algorithm for the primary algorithm can be found.

Experimental testing shows that utilizing an ONNX graph with a forking structure leads to complex partitioning due to the multiple paths in such a graph. Models of the non-sequential contain residual blocks [39], which is outside the scope of this research. There-

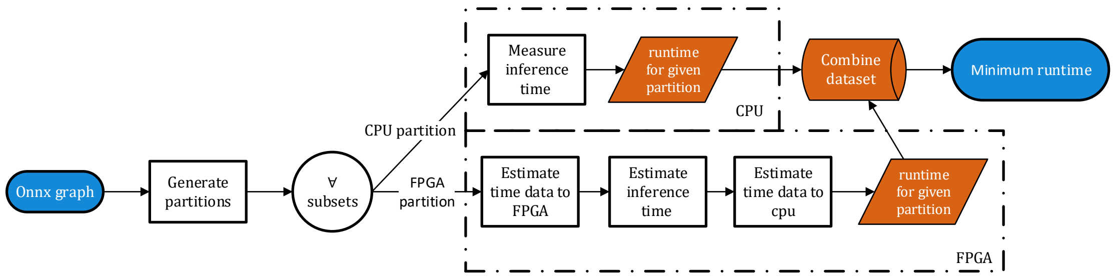


Figure 3.2: Design flow for algorithm 1. The input to the algorithm is a pre-trained ONNX graph, and the output is a dataset in which it is possible to find the best configuration based on design criteria.

fore, this research has selected a sequential ONNX graph structure as the preferred graph type for this study. To partition a sequential ONNX graph by grouping nodes (nodes are the operations as shown in Figure 2.5) into various sub-graphs, we either deploy the sub-graph to the FPGA or the CPU. To perform this partitioning, we use the QONNX (quantized ONNX), a Python package used to represent uniform quantization [40]. PartitionFromDict() is a function within the QONNX package that allows users to transform an ONNX model by passing a dictionary that defines partitions based on node indices. The resulting partitions each have a model attribute. Figure 3.3 illustrates the feasible partitions that can be placed on the FPGA. Note that the part placed on the CPU is not in the figure. It is important to note that the scenario with a single node is excluded by choice as it is not practically viable. The generated subset seen below are subsets for execution on the FPGA.

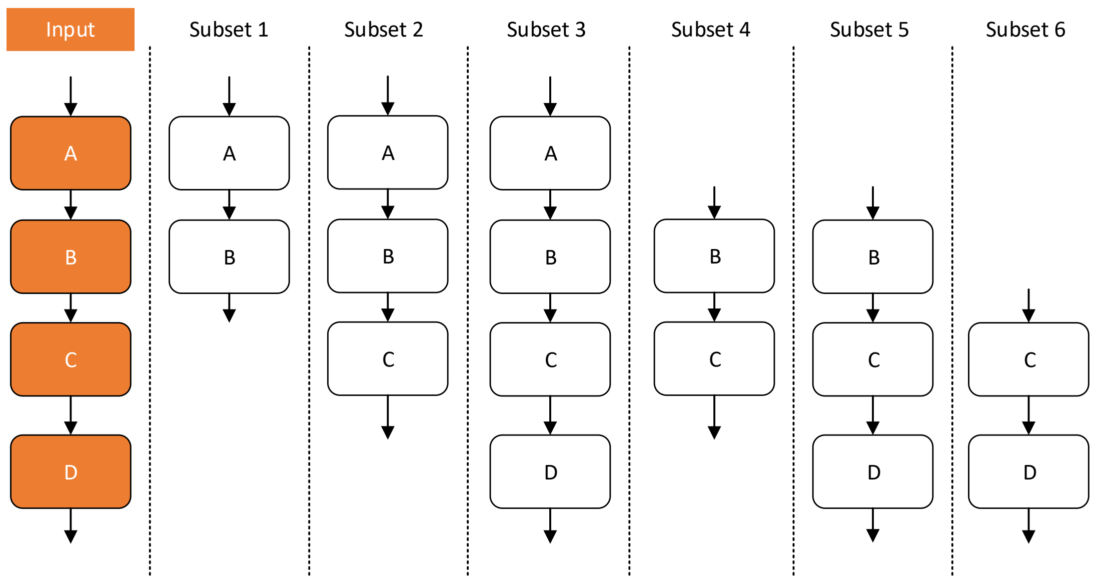


Figure 3.3: An example of the possible subsets for an ONNX graph of nodes to be placed on the FPGA, the nodes not shown in the subsets are placed on the CPU. Note that the scenarios with a single node are ignored.

3.2.3. FPGA estimation

To estimate the expected inference time of a neural network partition on the FPGA, it can be broken down into three parts as shown earlier in Equation 3.2. In the FINN flow, a step in the build process generates estimate reports containing information such as execution time, resource usage, and more. To estimate the execution time of the FPGA, we use the estimate report named *estimate_network_performance.json*; an example is below.

```
{
  ”critical_path_cycles”: 8718751,
  ”max_cycles”: 394272,
  ”max_cycles_node_name”: ”Thresholding_Batch_0”,
  ”estimated_throughput_fps”: 253.63201038876716,
  ”estimated_latency_ns”: 87187510.0
}
```

To estimate the execution time of our design, we can utilize the critical path cycles and the clock frequency of the FPGA. As the critical path cycles indicate the maximum clock cycles needed to output an answer, using this information, we can define the inference time as $T_{\text{inference time}} = \text{critical path cycles} * \text{CLK}^{-1}$. However, estimating the data movement times requires more effort since this information is not readily available in the FINN framework. A fit function will be derived to determine the movement times for data of a specific size in megabytes (MB). To do so, we pass data of different sizes (in MB) from the CPU to the FPGA and measure the time to transfer data. This process is repeated for data from the FPGA to the CPU, resulting in two functions defining the functions used to estimate the data movement times.

3.2.4. CPU measurements

As the ONNX graph is present for the CPU partition, it is possible to measure the execution time on the CPU. This can be done by using onnxruntime. Onnxruntime provides efficient and optimized execution of ONNX models on CPU platforms, leveraging the hardware and software capabilities of the system.

3.2.5. Setup of algorithm 2

The study proposes a second algorithm that builds on the previous algorithm described in Section 3.2.2 but incorporates a validation check to speed up the estimation process. Specifically, the assessment determines whether the FPGA partition fits within the resource limit of the given platform. A JSON file displays the utilized estimate resources during the FINN estimate report generation phase. Figure 3.4 shows that the algorithm adds a check for each generated subset to verify if the given partition's estimated resources fit on the Kria SoM KV260. If the estimation exceeds the total allowed resources of the Kria SoM KV260 (Table 2.1), the algorithm skips this partition. This pruning reduces the number of estimates that need to be made, resulting in a shorter estimation time for the same neural network.

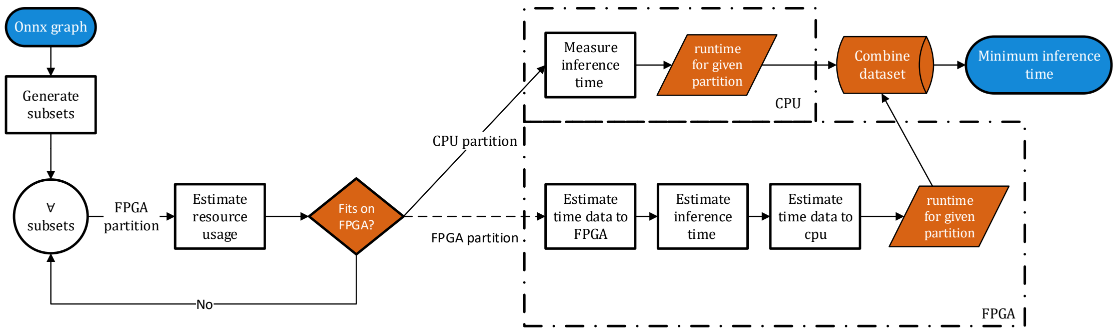


Figure 3.4: Design flow for algorithm 2. The input to the algorithm is a pre-trained ONNX graph, and the output is a dataset in which it is possible to find the best configuration based on design criteria. A check is added to see if it fits within the resources of the FPGA; if not, it skips and goes to the next partition.

4 Implementation

This chapter outlines the implementation of the previously introduced algorithms in Section 3.2.2 and Section 3.2.5. The code presented in this chapter is pseudo-code. The complete code can be found in Appendix B. We begin by introducing our implementation of algorithm 1, explaining how we set up an experiment to obtain the functions for estimating data movement, and discussing how we plan to approach the CPU implementation. After that, we describe how we modified the second iteration of the algorithm to check if a partition remains within the resource bounds.

4.1. Implementation algorithm 1

A practical implementation of algorithm 1 is provided below, Figure 3.2 refers to the outline used in this implementation. It explains how to partition the ONNX graph to generate the search space and experiments run to determine data movement times. It also outlines the approach for CPU inference times. Finally, it demonstrates how these components work together to identify the best partition.

4.1.1. Defining search space

Our implementation must partition a pre-trained ONNX graph at each custom operation ”MultiThreshold.” FINN uses the ”MultiThreshold” node as a custom operation to represent the quantization and binarization of neural network weights and activations. The node takes a tensor as input and applies a set of fixed threshold values to each element in the tensor. Without the MultiThreshold nodes, the FINN framework would not know which nodes in the neural network require quantization. It could not perform the necessary optimizations to generate a hardware design that efficiently implements the quantized neural network. Thus this node is critical to using the FINN framework, and therefore a graph is split at each MultiThreshold node.

Pseudo-code 1 generates a list of the indices where the MultiThreshold node is located. Our implementation shown in Pseudo-code 2 iterates over this list and compiles all possible subgraphs. This results in two lists, one for the nodes containing the ranges for CPU and one containing the nodes for the FPGA partition. These two lists can now be used with the function PartitionByDict to partition the CPU and FPGA partitions.

Algorithm 1 Generate partitions

```
procedure GeneratePartitions(model)
    wanted_Nodes ← empty list
    for each $node$ ∈ model.graph.node do
        if $node$.name == “MultiThreshold” then
            wanted_Nodes.append($node$.index)
        end if
    end for
    return wanted_Nodes
end procedure
```

Algorithm 2 Group sub graphs

```
$combinations\_lst\_fpga, combinations\_lst\_cpu \leftarrow \emptyset$
$max\_len$ ←length(model.graph.node)
for $i$ in $wanted\_nodes$ do
    for $l$ in $wanted\_nodes$ do
        if $i < l$ then
            if $i = 0$ and $l \neq max\_len$ then
                $fpga \leftarrow range(i, l)$
                $cpu \leftarrow range(l, max\_len)$
            else if $l = max\_len$ then
                $fpga \leftarrow range(i, l)$
                $cpu \leftarrow \emptyset$ if $i = 0$ else $range(0, i)$
            else
                $fpga \leftarrow range(i, l)$
                $cpu \leftarrow [range(0, i), range(l, max\_len)]$
            end if
            if $fpga$ and $cpu$ then
                $combinations\_lst\_fpga$.append($fpga$)
                $combinations\_lst\_cpu$.append($cpu$)
            end if
        end if
    end for
end for
```

4.1.2. FPGA estimation

For the above-created partitions, the FPGA estimation can begin, which is done in three steps illustrated in Figure 4.1. First, we use a function to estimate the data movement to the FPGA. Then, the FINN build flow is used to estimate the inference time, followed by an estimate of the time back to the device. Reading the critical path cycles from FINN build flow estimates report and dividing it by the clock frequency produces the estimated FPGA runtime. These three steps are essential in realizing the implementation.

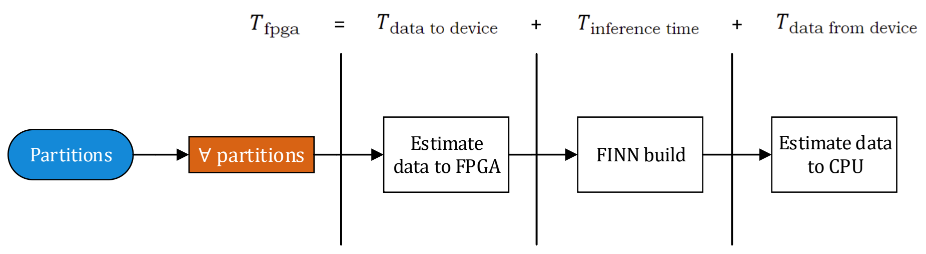

$T_{\text{fpga}} = T_{\text{data to device}} + T_{\text{inference time}} + T_{\text{data from device}}$
Partitions
∀ partitions
Estimate data to FPGA
FINN build
Estimate data to CPU

Figure 4.1: Implementation pipeline for estimation of the partition on FPGA

4.1.3. Finding data movement times

As data movement is a part of the total time for the FPGA it requires us to define a method to estimate these times for a given input size. The approach generalizes the duration of data movement to and from the device by considering the data size in MB and timing how long it takes to move x MB of data to and from the device. With this data, we can generate a function that returns the data movement time in seconds for a given data size. As this is an unknown in our algorithm, the following experiment derives a function that can be used to estimate the movement time.

1. Generate a bitfile for a model containing a single fully connected layer with a small input and output size. This model is used as we want to analyze the time it takes to move data to the device.
2. Vary the batch sizes to simulate different data sizes being passed to and from the device.
3. For each batch size, record the size of the data and the time taken to move the data. The data size is obtained using the formula $(\text{x.nbytes}) * 10^{-6}$ where x is a numpy array, and 10^{-6} makes the result size in MB.
4. Obtain movement times by measuring PYNQ functions $copy\_input\_data\_to\_device$ and $copy\_output\_data\_from\_device$.
5. Fit a function to obtain a relationship between data size and transfer time.

This process allows for a measurement setup to measure how long data movement takes based on the size of the data being transferred in MB. Measurements were taken using the Python time package to measure the PYNQ functions $copy\_input\_data\_to\_device$ and $copy\_output\_data\_from\_device$. Using the data collected for various sizes of MB and the time required to move, a linear fit function is fit to the data. As such, Equation 4.1 and Equation 4.2 are derived and will be used to estimate the time to move x MB of data to and from the device.

$$T_{\text{data to device}} = 1.0976 * x_{\text{input size mb}} + 0.3052 \tag{4.1}$$

$$T_{\text{data from device}} = 0.5606 * x_{\text{output size mb}} + 0.0954 \tag{4.2}$$

Using the above-defined linear equations, the implementation can be seen in Pseudocode 3 to estimate the time for data movement. The input and the output shape of a given FPGA partition are known, our implementation uses this knowledge to approximate the size of the tensors in MB, and using the functions defined above, the movement time can be estimated.

Algorithm 3 Calculate data movement times

```
1: function get_data_in($model\_path$)
2:     $model \leftarrow$ ModelWrapper($model\_path$)
3:     $inp\_name \leftarrow model.graph.input[0].name$
4:     $inpt\_size \leftarrow model.get\_tensor\_shape(inp\_name)$
5:     $input\_tensor \leftarrow$ np.random.randint($0, 255, inpt\_size$).astype($'uint8'$)
6:     $data\_to\_device\_time \leftarrow 1.0976256841119194 \times input\_tensor.nbytes \times 10^{-6} + 0.3052411426075403$
7:     return $data\_to\_device\_time$
8: end function
9: function get_data_out($model\_path$)
10:    $model \leftarrow$ ModelWrapper($model\_path$)
11:    $out\_name \leftarrow model.graph.output[0].name$
12:    $outpt\_size \leftarrow model.get\_tensor\_shape(out\_name)$
13:    $output\_tensor \leftarrow$ np.random.randint($0, 255, outpt\_size$).astype($'uint8'$)
14:    $data\_from\_device\_time \leftarrow 0.5606177488366392 \times output\_tensor.nbytes \times 10^{-6} + 0.09549860536045693$
15:    return $data\_from\_device\_time$
16: end function
```

4.1.4. CPU estimation

We use the PartitionByDict function from QONNX and the list of CPU partitions from Pseudocode 2. We measure the time required to execute each partition on the CPU. For this, we use onnxruntime for this purpose, as it allows the execution of an ONNX graph directly on the CPU. By measuring the time required for each partition, we can accurately measure its execution time rather than relying on an estimate. The exe_ort function shown in Pseudocode 4 is called five times to reduce measurement instabilities on the CPU and the average is taken of these five runs.

Algorithm 4 Measure time for CPU execution

```
1: function exe_ort(model_file)
2:     $model \leftarrow$ ModelWrapper(model_file)
3:     $sess \leftarrow$ ort.InferenceSession(model.model.SerializeToString(),so)
4:     $x\_test \leftarrow$ np.ones(model.get_tensor_shape(model.graph.input[0].name), dtype=np.float32)
5:     $start \leftarrow$ time.time
6:     sess.run(None, {model.graph.input[0].name: x_test})
7:     $stop \leftarrow$ time.time
8:     $runtime \leftarrow (stop - start) * 1e3$
9:     return $runtime$
10: end function
```

4.1.5. Summary implementation algorithm 1

In conclusion, the following implementation is realized for algorithm 1. Our implementation begins by partitioning a given ONNX graph into FPGA and CPU partitions, passing each partition to its respective device. To estimate the data movement time for the FPGA, we first examine the expected input tensor of the ONNX graph resulting in the estimated time to transfer data to the FPGA. Next, we generate a FINN estimate report using the FINN build flow. Finally, we use the function derived for data movement from the FPGA using the output tensor size to estimate the time needed to move data back. The sum of these three components results in the time required to run a given partition on the FPGA. For the CPU, the time is measured using Python package time and onnxruntime. This is because onnxruntime allows for easy deployment of an ONNX graph to the CPU. We are executing all the given partitions' results in the time required for each device for a given neural network. These two outputs together result in the following output for each partition (i.e., Table 4.1). All partitions are combined into one large dataset, which we can then apply an argmin function to the *total_runtime[ms]*, resulting in the best partition for a given neural network model.

Run Id	0
Number of nodes	146
CPU_range	range(0, 6),range(11, 146)
total_runtime_CPU[ms]	351.2269497
runtime_part_0[ms]	5.992269516
runtime_part_1[ms]	345.2346802
FPGA_range	range(6, 11)
total_runtime_FPGA[ms]	1.129702223
runtime_FPGA[ms]	0.07516352
data_in_FPGA[ms]	0.738004216
data_out_FPGA[ms]	0.316534486
total_runtime[ms]	352.3567

Table 4.1: An example of the output generated for a single partition. With an FPGA range of 6 to 11, the CPU contains two partitions, one from 0 to 6 and the other from 11 to the end. Total CPU runtime is the sum of the two CPU partitions the *total_runtime = total_runtime_cpu + total_runtime_fpga*

4.2. Implementation of algorithm 2

An outline for the implementation of the second proposed algorithm from Chapter 3 is given. This implementation builds on top of the algorithm explained in Section 4.1; therefore, our explanation will only add what has been added to this algorithm. The difference between the two algorithms is the added check to validate if the partition fits on the KV260. This can be seen in Figure 4.2 while the code implemented is shown in Pseudo-code 5

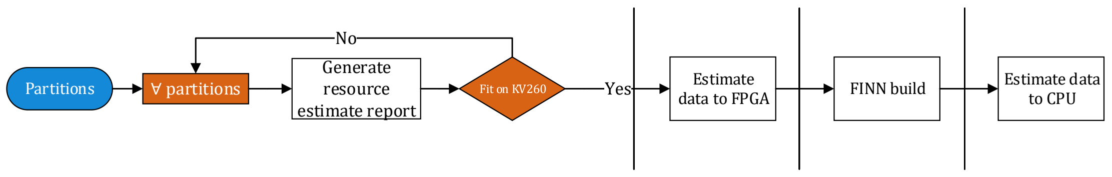


Figure 4.2: With respect to the previous implementation, a check is added to validate whether a given partition fits the Kria SoM KV260.

The rest of the algorithm is exactly the same as the implementation explained in Section 4.1, this implementation helps prune the search space as it skips the partitions which are too large for the given FPGA.

Algorithm 5 Check if resource estimate within limits of KV260

```
1: function check_resources
2:     $resource\_limit \leftarrow "BRAM\_18K" : 144, "LUT" : 117120, "URAM" : 64, "DSP" : 1248$
3:     Open file $"estimate\_layer\_resources.json"$ in read mode
4:     $resource\_rpt \leftarrow$ Load JSON data from file
5:     for $(k, v)$ in $resource\_rpt["total"]$ do
6:         if $v \leq resource\_limit[k]$ then
7:             return False
8:         end if
9:     end for
10:    return True
11: end function
```

5 Results

This chapter outlines the experimental setup in Section 5.1, explaining which neural network(NN) models are tested and the resources used to evaluate each proposed algorithm. The remainder of this chapter presents results obtained from each of these experiments; Section 5.2 presents the results for the first NN model tested, Section 5.3 presents experiments to examine if it is feasible to accelerate this NN further. Section 5.4 presents the results for the remaining two models. Each section discusses each algorithm in detail and concludes with a conclusion for the given model.

5.1. Experimental setup

To evaluate the performance of the implemented algorithms, a set of experiments are performed to measure the performance of each algorithm. The experiments remain constant for all algorithms to ensure an equal comparison can be made between each algorithm. The experimental setup contains the resources used, the models used to test the algorithms, and the metrics that must be captured to compare each algorithm. To test each algorithm, two resource platforms are used. The qce-alveo01 server generates and builds FPGA-accelerated applications for estimate report generation. The server is used to create the estimate reports as it has the FINN compiler setup and the required software tools, such as Vivado and Vitis. For the CPU measurements, the Kria SoM KV260 is used. Combining these two resources enables a complete analysis of the algorithm's performance and guides the selection of the most effective solution.

A crucial step in evaluating the performance of a given algorithm is defining specific metrics to gauge the algorithm's performance. Since these algorithms aim to minimize runtime on a hybrid platform, measuring the runtimes for each platform is essential. Another vital metric is the time required to estimate a given neural network. Considering these metrics, the algorithm's performance can be evaluated, and a conclusion can be drawn as to which algorithm generates the quickest estimation.

The following metrics must be captured for each model to evaluate which partition meets the runtime criteria. To make a fair comparison between each tested partition, PE and SIMD are set to 1, which is set by providing a folding configuration file to the FINN compiler. From the FPGA estimation, data movement times to and from the FPGA are derived, as well as the estimated runtime on the FPGA, which is read from the *estimate_network_performance.json*. The sum of these three is denoted below as *total runtime FPGA[ms]*. For the CPU time, the execution times are measured on the Kria SoM KV260; for a given partition to remove any measurement instabilities, we run a given partition five times and average over to calculate the runtime for the CPU denoted below as *total runtime CPU[ms]*.

Model selection

To demonstrate the effectiveness of the partitioning algorithm, we pass a pre-trained ONNX model to each algorithm to determine their respective partitioning locations for the model. As these algorithms should work with any sequential ONNX model, we test the following three models: MobileNetv1, cybersecurity, and CNV. All three models are pre-trained and ready for use in FINN. Each model is of a sequential structure. A design choice is made to ignore models with residual blocks, such as ResNet, as this makes partitions difficult due to the residual blocks in such graphs. For each model, a comparison is made between a pure CPU implementation where the entire network is run on the CPU. This is used as a baseline to demonstrate the performance difference using a hybrid implementation.

5.2. MobileNetv1 results

This section will present and discuss the results of the MobileNetv1 neural network, evaluated on both the first and second algorithms introduced in Chapter 3. For each algorithm, the top 5 best runtime estimates are presented. Each of these will be compared to the baseline implementation, where the entire network is run on the CPU. For each tested algorithm, a discussion will be held to highlight significant findings and pose possible modifications that can be made.

5.2.1. Results of algorithm 1

Here results for MobileNetv1 implemented on algorithm 1 are presented. Table 5.1 shows the top 5 best-estimated runtimes sorted in terms of total runtime. This table also contains the ranges and runtimes for the FPGA and CPU partitions.

			MobileNetv1 model		
Top k	**FPGA range**	**Runtime FPGA[ms]**	**CPU range**	**Runtime CPU[ms]**	**Runtime[ms]**
1	range(0, 146)	6.5116	None	None	6.5116
2	range(0, 136)	6.5275	range(136, 146)	3.0578	9.5853
3	range(6, 146)	6.6654	range(0, 6)	6.0182	12.6836
4	range(6, 136)	6.6813	(range(0, 6), range(136, 146))	9.0321	15.7134
5	range(0, 131)	6.0132	range(131, 146)	22.8128	28.826

Table 5.1: Top 5 estimated runtimes for the MobileNetv1 neural network algorithm 1.

The MobileNetv1 graph comprises a total of 146 nodes, which are evaluated in 406 different possible partitions. The outcome of this evaluation process reveals that implementing the complete model yields the lowest runtime of 6.51 ms. The other four best results involve implementing a large portion of the network on the FPGA and a small part on the CPU. However, resource usage shows that the top 5 results for the MobileNetv1 model cannot be accommodated on the Kria SoM KV260 platform, rendering them infeasible. This infeasibility motivated the need to implement algorithm 2, which includes a check to verify that the implementation fits the platform. Resource usage for the MobileNetv1 is depicted in Figure 5.1. The right figure reveals the utilization of BRAM exceeds the limit for the FPGA by 7x for the MobileNetv1 model, thereby making it infeasible to implement on the Kria SoM KV260.

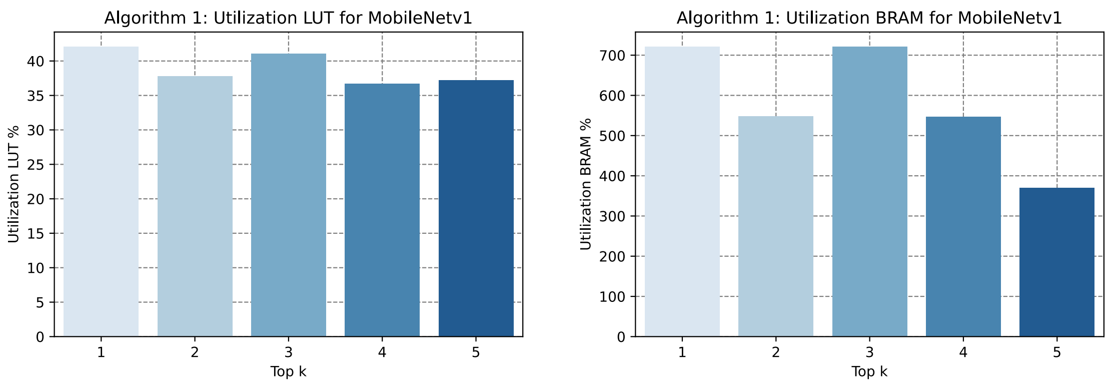


Figure 5.1: MobileNetv1 neural network's resource usage for algorithm 1, where the resource is normalized to maximum resource usage, revealing that the BRAM resource is being over-utilized, causing the top 5 to not fit on the FPGA. In contrast, the left figure illustrates the LUT usage, which meets the constraint since the maximum usage is 42.10%.

Figure 5.2 shows the improvement over this baseline for the MobileNetv1 graph for all tested partitions, with the baseline runtime of the MobileNetv1 on CPU of 356.46ms. The x-axis shows the percentage of the network implemented on the FPGA. The figure depicts that as the percent of the model on the FPGA increases, the higher the improvement as the total runtime decreases. In the best-case scenario where the entire network can be implemented on the FPGA, we show close to 100% improvement. Equation 5.1 shows the formula used to derive the improvement metric, indicating that as the estimated runtime approaches 0 ms, the total improvement approaches 100%.

$$\text{improvement \%} = \frac{t_{\text{CPU baseline}} - t_{\text{est. runtime}}}{t_{\text{CPU baseline}}} * 100 \tag{5.1}$$

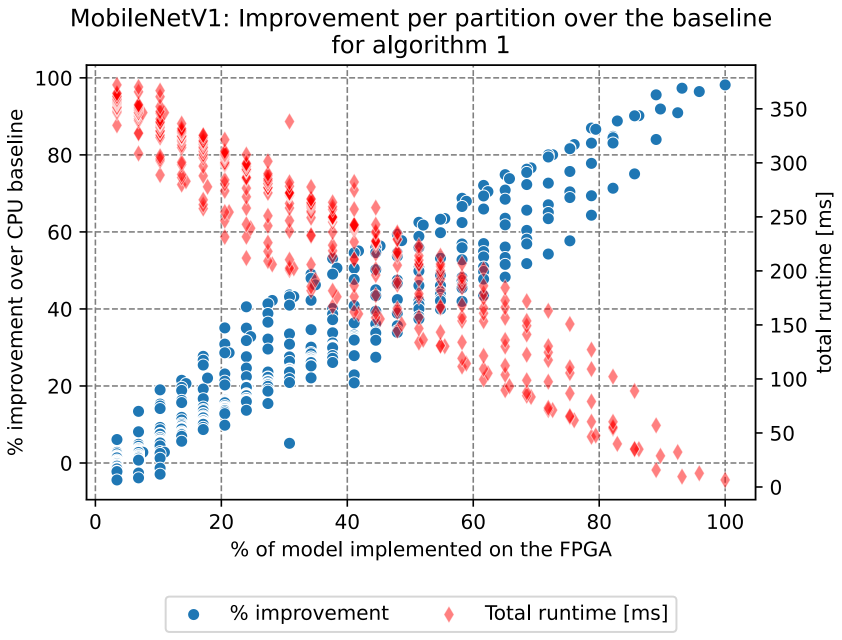


Figure 5.2: Algorithm 1 runtime for MobileNetv1 improvement with respect to a full CPU implementation. The diamonds represent the total runtime (right y-axis), and the dots represent the % improvement (left y-axis).

5.2.2. Results algorithm 2

For the implementation of algorithm 2, a check is added to verify that the partition generated fits on the Kria SoM KV260. Table 5.2 contains the top 5 best partitions for the MobileNetv1 graph. As this algorithm adds a check to verify the partition fitting within the resource limit, the top 5 partitions presented below are theoretically feasible. Figure 5.3 shows the resource usage for the top 5 best partitions. It demonstrates that the max of each partition remains below the resource limit of the Kria SoM KV260.

MobileNetv1 model					
Top k	**FPGA range**	**Runtime FPGA[ms]**	**CPU range**	**Runtime CPU[ms]**	**Runtime[ms]**
1	range(0, 81)	3.6453	range(81, 146)	122.9647	126.6101
2	range(6, 81)	3.7991	(range(0, 6), range(81, 146))	131.858	135.6571
3	range(0, 76)	3.6247	range(76, 146)	132.3175	135.9423
4	range(6, 76)	3.7785	(range(0, 6), range(76, 146))	144.696	148.4745
5	range(0, 71)	3.1099	range(71, 146)	147.5528	150.6628

Table 5.2: Top 5 estimated runtimes for the MobileNetv1 neural network algorithm 2

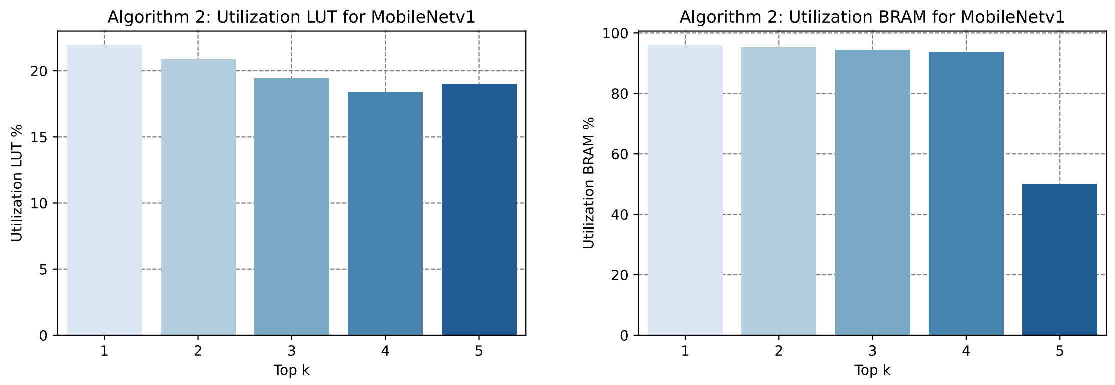


Figure 5.3: Estimated resource utilization for the MobileNetv1 neural network when implemented on algorithm 2.

An important observation that can be made between these results and the results from Section 5.2.1 is the fact that the runtime increases by 19.44x for the best case. The increase in runtime directly results in a larger portion of the neural network being implemented on the CPU. A comparison is again made with a pure CPU implementation to indicate the improvement achieved by implementing this algorithm. Figure 5.4 shows a scatterplot of the improvement made with respect to the pure CPU implementation with a runtime of 356.46 ms. The figure shows that as the percentage of the model implemented on the FPGA increases, the total runtime decreases.

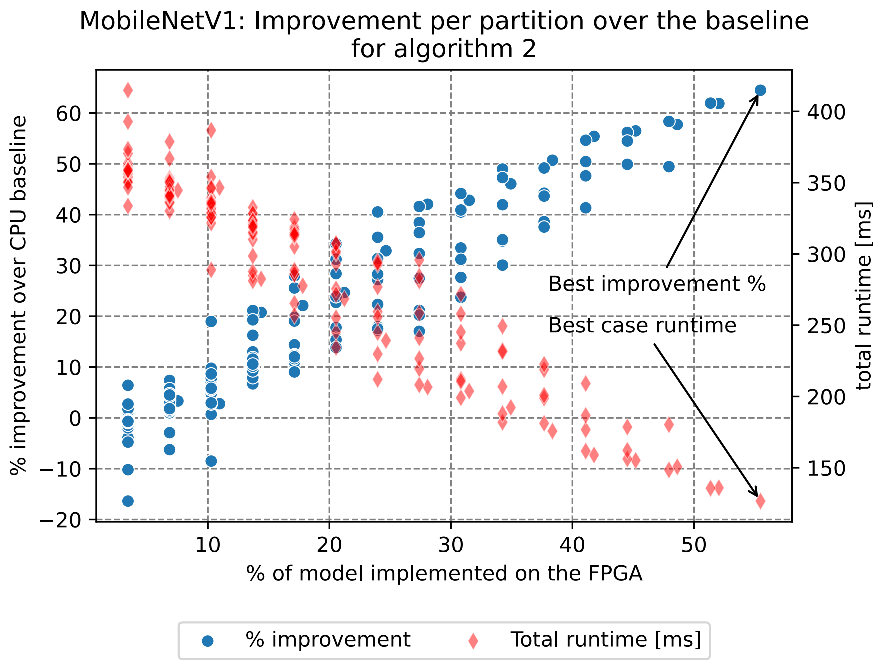


Figure 5.4: Runtime improvement compared to the full CPU implementation for the MobileNetv1. The diamonds represent the total runtime (right y-axis), and the dots represent the % improvement (left y-axis).

Compared to Figure 5.2, only around 60% of the MobileNetv1 model can be implemented on the FPGA and remain below the resource limits of the Kria SoM KV260, which corresponds with the best partition shown in Table 5.2. This is the theoretical maximum portion of the MobileNetv1 that will fit on the FPGA. As less of the model can be implemented on the FPGA, the best total improvement is 64.48% (calculated using Equation 5.1). When implementing MobileNetv1 on algorithm 2 we can show that the greatest performance increase occurs when the largest portion of the model is implemented on the FPGA. It is a similar result as when implementing the network on Algorithm 1.

Algorithm 2

Top k	FPGA range	Runtime FPGA[ms]	CPU range	Runtime CPU[ms]	Total runtime[ms]
1	range(0, 81)	3.6453	range(81, 146)	122.9647	126.6101
2	range(6, 81)	3.7991	(range(0, 6), range(81, 146))	131.858	135.6571
3	range(0, 76)	3.647	range(76, 146)	132.3175	135.9423
4	range(6, 76)	3.7785	(range(0, 6), range(76, 146))	144.696	148.4745
5	range(0, 71)	3.1099	range(71, 146)	147.5528	150.6628

First 5 that fit algorithm 1

Top k	FPGA range	Runtime FPGA[ms]	CPU range	Runtime CPU[ms]	Total runtime[ms]
51	range(0, 81)	3.6453	range(81, 146)	126.5804	130.2257
55	range(6, 81)	3.7991	(range(0, 6), range(81, 146))	129.9025	133.7016
58	range(0, 76)	3.647	range(76, 146)	132.6485	136.2732
61	range(6, 76)	3.7785	(range(0, 6), range(76, 146))	140.9487	144.7272
64	range(0, 71)	3.1099	range(71, 146)	147.5308	150.6407

Table 5.3: Comparision between algorithm 2 and the first five that fit from algorithm 1.

Table 5.3 shows that algorithm 1 and 2 both give the same results when looking at the first 5 that fit for algorithm 1. The total runtime differs slightly due to CPU measurements taken at different times. Top k shows where the first five fall in terms of runtime for algorithm 1, note that they are not sequential as these are the first 5 that have a BRAM utilization under the 100%. Showing that both algorithms come to the same solution if the constraint is resource utilization.

5.2.3. Estimation time for MobileNetv1

To measure the performance of an algorithm, measurements were taken to quantify the amount of time necessary to estimate the best runtime. Figure 5.5 depicts the time required to generate the above results. For the CPU and FPGA, the time package from python is used to measure. As we measure the CPU time for a given partition to remove any measurement instabilities, we run a given partition 5 times and average over to calculate the runtime.

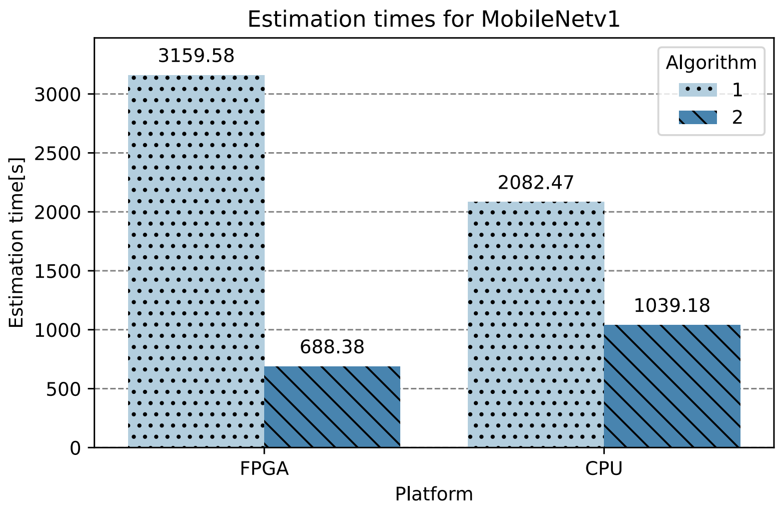


Figure 5.5: Estimation times for the MobileNetv1 neural network.

The figure shows that algorithm 2 has an improvement in estimation time. Implementing a check to verify if the model fits acts as a pruning technique for skipping partitions that are not feasible solutions. The implementation of algorithm 2 shows an improvement of 3x over algorithm 1 for MobileNetv1. While algorithm 2 does not reach a global minimum for MobileNetv1 as it does not search the entire search space. It does return the partition with the lowest total runtime that fits on the FPGA.

5.2.4. Finding for algorithm 1 and 2 for MobileNetv1

In conclusion, algorithm 1 shows promising results in partitioning a neural network for a hybrid platform. Implementing the entire neural network on the FPGA is the best option, as shown in the results above. While this may be the best solution, estimated resource usage shows that these top 5 are not always feasible to implement on an FPGA. Algorithm 2 solved this issue of infeasibility by validating if a given partition meets the resource constraints of the FPGA. Compared to a pure CPU implementation, an improvement of 54.74x is achieved for algorithm 1 as the best-case scenario for that algorithm is to implement the MobileNetv1 on the FPGA fully. While this is theoretically the best implementation, it is practically infeasible for the Kria SoM KV260, as the BRAM exceeds the platform's resource limits.

In contrast, for algorithm 2, we see an improvement of 2.82x for the best-case partition. These 5 all remain within the resource limits of the Kria SoM KV260, rendering them feasible solutions. While these may be feasible solutions, they do come at a cost. Runtime increases from 6.51ms to 126.61ms from algorithm 1 to 2 in the best-case scenario. While the solutions from algorithm 1 are not practically feasible, they indicate that a resource-limited environment decreases the total runtime for the MobileNetv1 compared to the best-case scenario. Regarding algorithm performance, algorithm 2 improved estimation time by 3x compared to algorithm 1, making algorithm 2 the better choice for estimation as it is quicker and yields results that can be implemented.

5.3. Further improvement MobileNetv1

To evaluate if we can reduce the total runtime for this hybrid setup, additional experiments are conducted to evaluate if a trade-off can be made on the FPGA to improve the overall runtime. We use the results obtained in Section 5.2.2 as these can be implemented on the FPGA. We will examine possible trade-offs on FPGA to hopefully improve the runtime of the MobileNetv1 network on a hybrid platform.

5.3.1. Varying SIMD and PE

As the experimental setup explains, SIMD and PE are set to 1 to allow for a fair comparison between partitions. However, setting SIMD and PE to 1 results in low throughput for MobileNetv1. Increasing the values of PE and SIMD enhances a model's throughput on an FPGA by enabling more parallel processing. Thus the first design optimization allows FINN to auto-configure the folding settings for the model by passing target frame rates to the FINN build flow setting SIMD and PE to values other than 1. For design optimization, only feasible partitions are evaluated. The same method is used as algorithm 2. Figure 5.6 shows that when increasing the target frame rate, the percentage of the MobileNetv1 model that can be implemented on the FPGA is reduced. As increasing the target FPS increases the resources needed, the percentage of models which can be implemented on the FPGA decreases. The right figure from Figure 5.6 shows that increasing target FPS decreases runtime on the FPGA. Overall, the total runtime still increases; such an increase is due to a greater portion of the model being implemented on the CPU, solidifying the case that the CPU is the bottleneck in this hybrid platform.

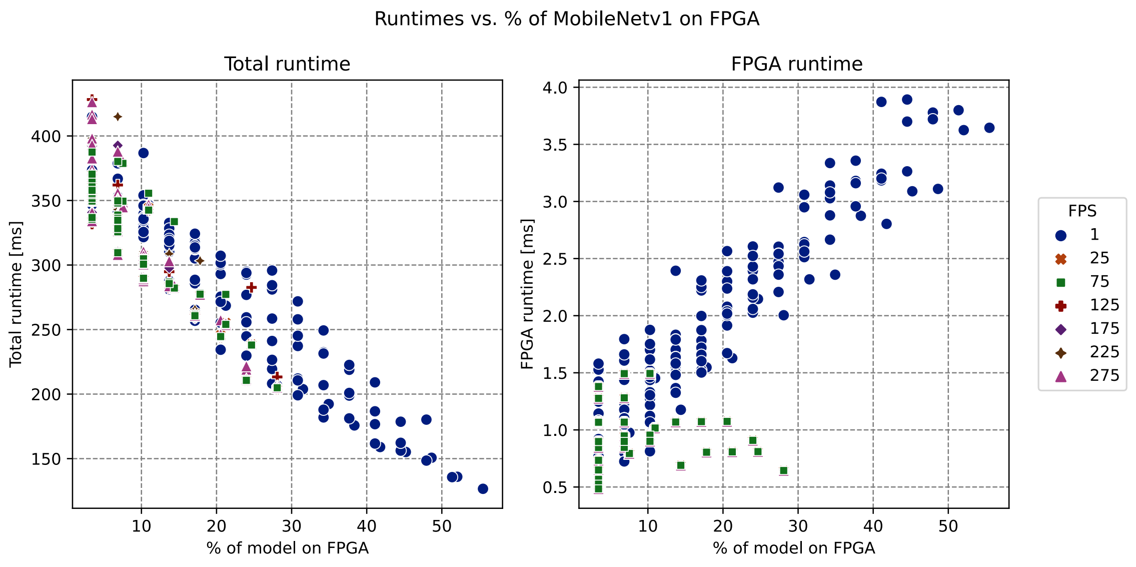


Figure 5.6: Varying target frame rate to evaluate if an improvement can be achieved for the total and FPGA runtime when setting SIMD and PE at higher values. Only evaluating partitions that fit the Kria SoM.

5.3.2. Theoretical improvement CPU

The previous experiment showed that the CPU of the Kria SoM KV260 is the bottleneck in the process. As such, calculations are made to see if improvements can be achieved if the CPU clock is faster than the current CPU clock of the Kria SoM KV260. This theoretical calculation demonstrates what is possible if the CPU platform were to have a higher clock speed. The calculation is done using Equation 5.2, where n is the factor speedup of the CPU, and $T_{\text{CPU runtime}}$ and $T_{\text{FPGA runtime}}$ are the runtime results from algorithm 2.

$$T_{\text{theoretical total runtime[ms]}} = \frac{T_{\text{CPU runtime}}}{n} + T_{\text{FPGA runtime}} \tag{5.2}$$

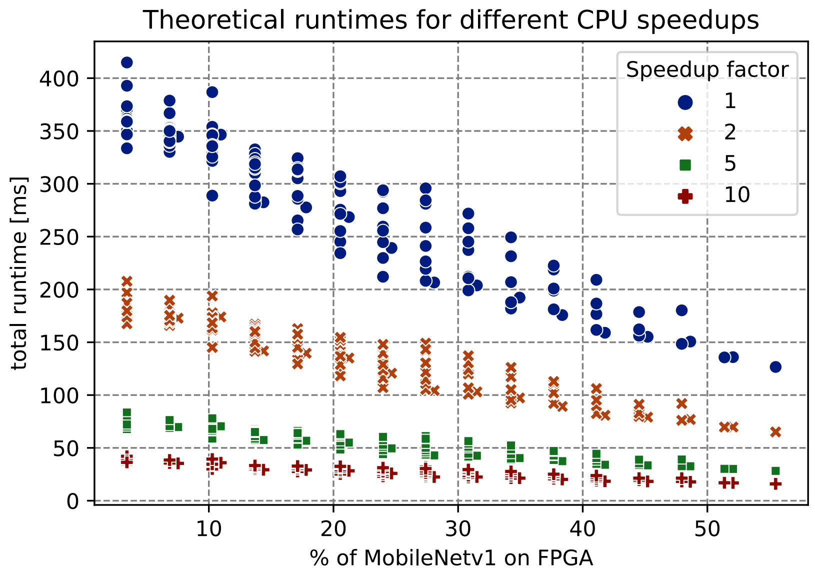


Figure 5.7: Theoretical total runtimes for different speedup factors for the MobileNetv1. Where the speedup factor in the legend corresponds to n in Equation 5.2.

Figure 5.7 shows how the speedup factor of the CPU affects the total runtime for the situation where SIMD and PE equal 1. We chose this scenario as it showed that the greatest partition of the model could be implemented on the FPGA. It can be seen that in the case of an improvement by a factor 10 the best case runtime becomes 15.94ms, an improvement of 7.94x that of the baseline (full CPU). A factor 10 improvement would mean that the CPU needs to have a clock speed of 3.3 GHz instead of 333.3 Mhz. Table 5.4 shows that as n increases, the difference in best and worst case decreases, as the CPU is the limiting factor where n=1 the effect of the CPU is much greater on the total runtime. Therefore when increasing the CPU clock speed, the slow-down effect of the CPU decreases resulting in a lower impact on the total runtime.

n	Min [ms]	Max [ms]	Δ [ms]
1	126.61	414.85	288.24
2	65.13	207.75	142.63
5	28.24	83.5	55.26
10	15.94	42.08	26.14

Table 5.4: Worst and best case total runtimes for theoretical speedup.

Using these theoretical calculations, a conclusion can be made that a faster CPU clock decreases the total runtime of the hybrid setup. Taking the CPU clock speed into consideration when selecting the hardware platform can significantly impact the total runtime.

5.3.3. Hardware resource usage vs. total runtime

We analyze how altering the target FPS affects resource utilization to investigate if trade-offs can be made to improve total runtime. With this information, we aim to present how FPS affects resource utilization. For this experiment, results from algorithm 1 are used to visualize what happens when we exceed the total resource limit.

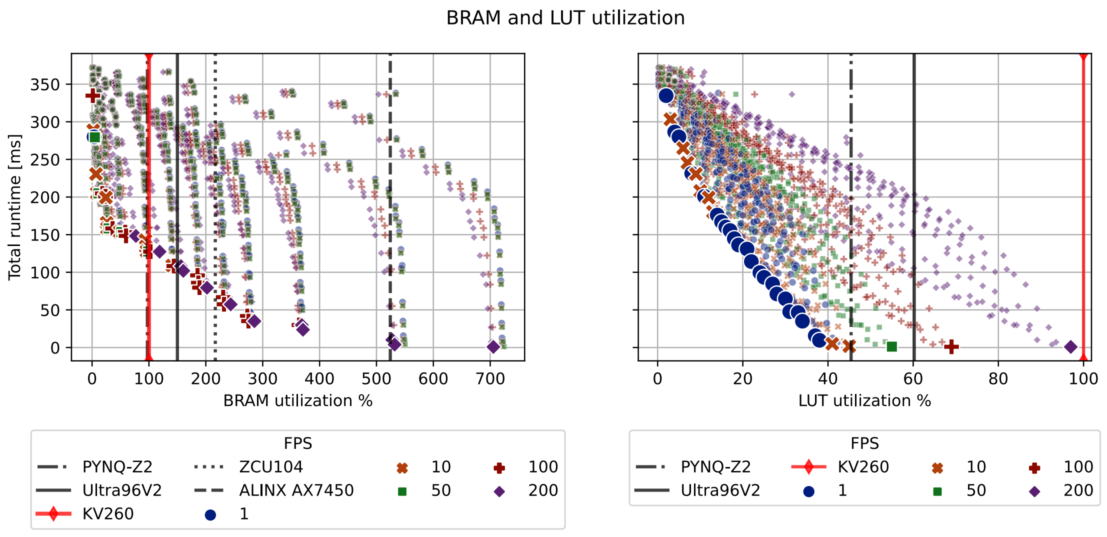


Figure 5.8: BRAM and LUT utilization for all partitions, a larger point indicates which target FPS yields the lowest runtime value for each resource. 100% utilization is the maximum of the Kria SoM KV260. Vertical lines indicate different hardware platforms (PYNQ-Z2, Ultra96v2, ZCU104, ALINX AX7450, and Alveo U280). If the vertical line is absent, it is outside the scale of the x-axis, indicating that the resource fits on the platform.

Figure 5.8 shows results for all possible partitions of the MobileNetv1 and the respective resource utilization. The right plot demonstrates that the MobileNetv1 implementation on the Kria SoM KV260 FPGA platform stays within the 117120 LUTs for all partitions. Additionally, the figure shows how the target frame rate and LUT usage can be adjusted. The figure identifies the best frame rate for a given LUT utilization since the minimum boundary indicates the minimum total runtime. On the other hand, the left plot indicates that the MobileNetv1 implementation on the Kria SoM KV260 has excessive BRAM utilization by 7 times, making it infeasible to implement the model fully. A platform with 7 times the BRAM of Kria SoM KV260 is needed to solve this, resulting in 1008 BRAM blocks. One such platform is the Alveo U280, which has 2016 (36Kb) BRAM. In the figure, vertical lines indicate other common FPGA platforms and where they fall respective to the Kria SoM KV260. The figures also reveal that higher BRAM utilization and target frame rate lead to lower total runtime due to less CPU slow-down and more of the model running on the FPGA. However, there's a trade-off between FPS and resource utilization, and lower utilization requires lower target FPS for BRAM and LUT. Therefore, the figures help select the lowest runtime value based on the partition and target FPS.

5.3.4. Comparison to other platforms

In this section, we will compare our results to other platforms. We compare the best-case runtime from algorithm 2 (FPGA range(0,81)) to the ZCU102 and the U280. These comparisons are made using FINN estimations for each new platform, respectively. We can only compare to the FPGA runtime because we cannot access these platforms. In Table 5.5, we show that the KV260 is the slowest of the three, with the U280 having the fastest estimated runtime. The ZCU102 and the U280 have folding files from `https://github.com/Xilinx/finn-examples/tree/main/build/mobilenet-v1/folding_config`.

Device	**FPS**	**Latency [ns]**
U280	2987.90198	10263252.0
ZCU102	469.68891	0.31194
KV260	1.94627	3.6453

Table 5.5: A comparison between the U280, ZCU102, and the KV260 for best case from algorithm 2

In terms of resource usage, we can see from Table 5.6. It can be seen that the resources for the U280 and ZCU102 remain well within the limits of the device. As these devices have a large number of resources available, the performance increase of these two devices is largely in part due to the fact that more resources can be utilized. More resource utilization results in more parallelization and a higher neural network application throughput. While higher FPS and shorter runtime is acheived for the U280 and ZCU102 these two platforms are much more expensive compared to the Kria SoM KV260. At the time of writing this report, the ZCU102 cost ≈ $\$3,200$, the U280 ≈ $\$8,500$, and the KV260 costs ≈ $\$250$. While the Kria SoM KV260 does lack performance it is a cheaper alternative.

Device	**BRAM**	**LUT**	**DSP**
U280	349	334344	96
Max resources U280	**2016**	**1304000**	**9024**
Device	**BRAM**	**LUT**	**DSP**
ZCU102	215(6.88Mb)	84631	48
Max resources ZCU102	**32.1Mb**	**653000**	**2520**
Device	**BRAM**	**LUT**	**DSP**
KV260	138	25662	0
Max resources KV260	**144**	**117120**	**1248**

Table 5.6: A comparison between the U280, ZCU102, and the KV260 in terms of resource usage for the best case from algorithm 2

5.3.5. Comparison to other work

In this section, we compare other works to this research; the comparison presented in Table 5.7 highlights some similarities and differences between the current research and other works aimed at improving AI in resource-limited environments. The other works primarily focus on simplifying the computational complexity by looking at methods to simplify convolution operations. In contrast, this work possesses an alternative solution using an HW/SW co-design to partition and distribute a neural network on edge devices. In terms of the cost, the devices presented cost thousands of dollars (in combination with an evaluation board) compared to the $250 of the Kria SoM KV260.

	Model	Platform	Method	Improvement over CPU	FPS
Ours	MobileNetv1	Kria SoM KV260	HW/SW co-design	2.82x	1.95
[41]	MobileNetv2	Arria 10 SoC	Dwsc	20x	266.6
[42]	MobileNet	ZU2 MPSoC	Dwsc	15.4x	205.3
[42]	MobileNet	ZU9 MPSoC	Dwsc	60.7x	809.8
[43]	RGB image of 384 × 384	VCU1525	FlexCNN	4.02x	23.8

Table 5.7: A comparison between other works and this work. **Dwsc=Depth wise separable convolution* fully on an FPGA platform.

5.4. Cybersecurity and CNV model results

Here, the cybersecurity and CNV model results will be shown and discussed. Both models are tested for each algorithm. The CNV model contains 46 nodes and has 54 possible partitions that are each evaluated. Table 5.8 shows the top 5 best runtimes for the CNV neural network implemented for algorithms 1 and 2. The cybersecurity model contains 25 nodes evaluated in 12 possible partitions. The top 5 best partitions are located in Table 5.9. Below, the results of each of these models will be discussed; the section is concluded with a summary of our findings.

Algorithm 1					
Top k	**FPGA range**	**Runtime FPGA[ms]**	**CPU range**	**Runtime CPU[ms]**	**Runtime[ms]**
1	range(0, 46)	1.0058	None	None	1.0058
2	range(2, 46)	1.0058	range(0, 2)	0.1951	1.2009
3	range(0, 39)	1.006	range(39, 46)	0.2604	1.2664
4	range(2, 39)	1.006	(range(0, 2), range(39, 46))	0.4671	1.4731
5	range(0, 35)	1.0034	range(35, 46)	0.7284	1.7318
Algorithm 2					
Top k	**FPGA range**	**Runtime FPGA[ms]**	**CPU range**	**Runtime CPU[ms]**	**Runtime[ms]**
1	range(0, 46)	1.0058	None	None	1.0058
2	range(2, 46)	1.0058	range(0, 2)	0.1928	1.1985
3	range(0, 39)	1.006	range(39, 46)	0.2571	1.2631
4	range(2, 39)	1.006	(range(0, 2), range(39, 46))	0.5639	1.5699
5	range(0, 35)	1.0034	range(35, 46)	0.6812	1.6846

Table 5.8: Top 5 estimated runtimes for the CNV neural network for algorithms 1 and 2.

Algorithm 1					
Top k	**FPGA range**	**Runtime FPGA[ms]**	**CPU range**	**Runtime CPU[ms]**	**Runtime[ms]**
1	range(0, 25)	0.4019	None	None	0.4019
2	range(0, 24)	0.4019	range(24, 25)	0.1612	0.563
3	range(0, 19)	0.4019	range(19, 25)	0.2186	0.6205
4	range(7, 25)	0.4009	range(0, 7)	0.3119	0.7127
5	range(0, 13)	0.4019	range(13, 25)	0.3177	0.7196
Algorithm 2					
Top k	**FPGA range**	**Runtime FPGA[ms]**	**CPU range**	**Runtime CPU[ms]**	**Runtime[ms]**
1	range(0, 25)	0.4019	None	None	0.4019
2	range(0, 24)	0.4019	range(24, 25)	0.1611	0.5629
3	range(0, 19)	0.4019	range(19, 25)	0.2358	0.6377
4	range(0, 13)	0.4019	range(13, 25)	0.2813	0.6832
5	range(7, 25)	0.4009	range(0, 7)	0.3	0.7009

Table 5.9: Top 5 estimated runtimes for the cybersecurity neural network algorithms 1 and 2

Table 5.9 show that the top implementation for both models is the partition where the entire network is implemented on the FPGA, similar to the result seen in Section 5.2.1. These models are small compared to the MobileNetv1; thus, each can be fully implemented on the Kria SoM KV260. Figure 5.9 show the estimated resource usages for the top 5 of each model. The figure depicts that estimated LUT and BRAM usages remain within the limits of the Kria SoM KV260 for both, proving that both models meet the resource constraints making each a feasible solution.

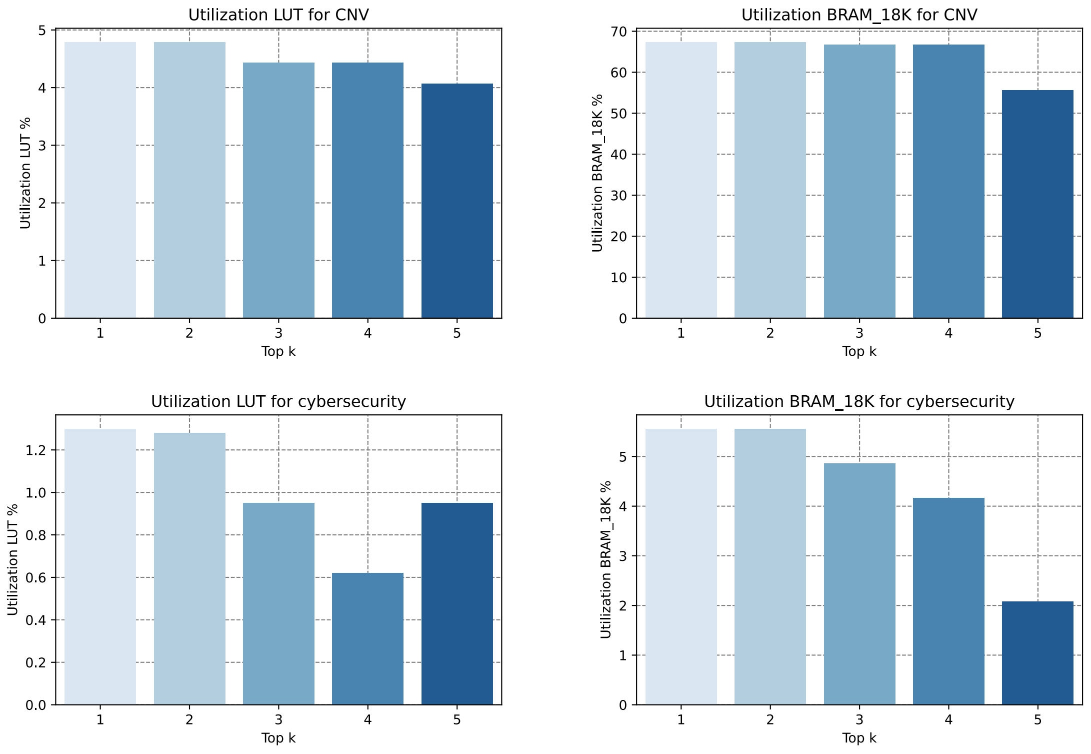


Figure 5.9: Top row depicts resource utilization for the CNV model and the bottom for the cybersecurity model.

5.4.1. Estimation time for cybersecurity and CNV model

Tabe 5.10 depicts the estimation times for each tested algorithm. Notable is that there is no improvement in estimation time, unlike MobileNetv1, where partitions do not fit on the FPGA. Both models meet the resource requirement. Therefore, implementing algorithm 2 does not affect the estimation time, as no partitions can be skipped.

Cybersecurity model		
	Algorithm 1 [s]	**Algorithm 2 [s]**
FPGA	8.66	8.65
CPU	2.02	2.11
CNV model		
	Algorithm 1 [s]	**Algorithm 2 [s]**
FPGA	146.06	142.33
CPU	31.85	31.86

Table 5.10: Estimation times for algorithm 1 and algorithm 2 for the cybersecurity and CNV models.

5.4.2. Findings for cybersecurity and CNV model

In conclusion, this section discusses the cybersecurity and CNV model results for various algorithms. The top implementation for both models is where the entire network is implemented on the FPGA, similar to the result in the previous section. The estimated resource usage for the top 5 of each model remains within the limits of the Kria SoM KV260, making each a feasible solution. Note that there is no improvement in estimation time, as the entire model fits on the Kria SoM KV260. These results indicate that the selection between algorithms 1 and 2 is invariant; choosing either algorithm will result in the same results within the same time for the given models.

6 Conclusion and future work

6.1. Conclusions

The field of edge AI is rapidly growing, driven by the need to bring AI capabilities to low-power edge devices. This field presents new challenges and opportunities for researchers and developers to design and optimize AI algorithms to run on edge devices. Edge AI has the potential to revolutionize a wide range of applications, from autonomous vehicles to IoT devices, and to unlock new capabilities that were previously impossible. This research has shown that partitioning and deploying a neural network model is feasible on the Kria SoM KV260 enabling the execution of neural networks in resource-limited environments, making inference of neural networks on edge devices possible, bringing AI closer to the data source and allowing for less dependency on server-grade hardware.

In this work, we studied the following main research question;

- *How can we leverage the unique strengths of CPUs and FPGAs in a hybrid computing architecture to optimize performance and efficiency for the neural network model?*

The sub-questions are listed as they were presented in the introduction:

1. Is a hybrid FPGA+CPU implementation a feasible and cost-effective solution for running neural networks compared to a pure CPU implementation? The question is answered in Section 6.1.1.
2. What are techniques for partitioning and distributing neural networks between FPGAs and CPUs? The question will be answered in Section 6.1.2.
3. How do the partitioning and distribution of neural networks between FPGA and CPU affect performance metrics, such as throughput and latency, and what factors influence the optimal distribution strategy? This is answered in Section 6.1.3.

6.1.1. Feasibility and cost-effectiveness

A hybrid implementation is more affordable than server-grade hardware, such hardware costs thousands of dollars. Cloud-based services like Google Cloud Platform (GPC), Amazon Web Services (AWS), or Microsoft Azure also require payment for access, and users need an internet connection to use them. A more cost-effective option is using a local hardware platform like the Kria SoM KV260, Ultra96v2, or the PYNQz2, typically costing a few hundred dollars. While these platforms do not have the same resources as server-grade hardware, they are a more affordable alternative.

By utilizing the FINN framework and the Kria SoM KV260. We have shown that implementing a neural network on a hybrid combination of CPU and FPGA is feasible by leveraging the hardware acceleration capabilities of the FPGA for specific nodes of the MobileNetv1 model and offloading other nodes to the onboard CPU. This approach offers a practical and cost-effective solution for users who seek an accessible way to run neural networks without the need for expensive server-grade hardware or cloud-based services.

6.1.2. Partitioning and distribution

Partitioning and distributing neural networks between FPGAs and CPUs is required to optimize performance and resource utilization in a hybrid setup. This research uses node-based partitioning to distribute neural network layers between the FPGA and CPU based on their computational requirements by grouping the nodes of an ONNX graph based on their topology in the graph. This approach can be practical for networks with highly parallelizable nodes in an ONNX graph.

6.1.3. Performance metrics

For this research, we analyzed the MobileNetv1 model in detail, as it is a neural network model which cannot be fully implemented on a Kria SoM Kv260 platform. Partitioning and distributing the MobileNetv1 neural network between FPGA and CPU significantly affected performance. Algorithm 1, which concludes that it is best to implement the entire MobileNetv1 on the FPGA, achieves a runtime of 54x faster than the CPU baseline of 356.5ms. However, these are not practically feasible on the Kria SoM KV260 due to the resource limitations of the chosen hardware platform. Algorithm 2 addresses this issue by validating if a given partition meets the resource constraints of the FPGA, resulting in an improvement of around 2x that of the CPU baseline. While these solutions are feasible, they come at the cost of increased runtime, as the sequential processing of the CPU is slow for the MobileNetv1.

Experiments were performed to analyze how changing PE and SIMD affects resource utilization. The results demonstrate a trade-off between FPS and resource utilization, where lower utilization requires lower target FPS for BRAM and LUT. Higher BRAM utilization and target frame rate lead to lower total runtime due to less CPU slow-down and more of the model running on the FPGA. However, excessive BRAM utilization may require a platform with higher BRAM, such as the Alveo U280. We conclude that for this setup changing the PE and SIMD values has a marginal improvement in total runtime, as the largest part of the total runtime is the CPU time.

Further analysis was performed to find if any trade-offs could be made on the FPGA to lower the total runtime. These experiments showed that the FPGA runtime increased when increasing the target FPS. However, increasing FPS results in more resource usage required. This led to less of the model being implemented on the FPGA and, in turn, reduced the total runtime as the CPU slows down the hybrid platform. Note that the CPU was the bottleneck of this setup. Theoretical analysis was performed to verify if a faster CPU would improve the overall performance. These calculations showed that increasing the CPU speed by a factor of 10 from 333.3 MHz to 3.33 GHz reduced the best-case theoretical runtime from 126.61 ms to 15.94 ms for the best-case scenario, which remains in the resource limits of the Kria SoM KV260. Such results show that increasing the CPU clock speed achieves a runtime that is 22x faster than the CPU baseline.

Two additional models, the cybersecurity and CNV model from FINN examples, validate that the implemented algorithms work for other models. These models both remain within the resource constraints of the Kria SoM KV260. These two models are small in terms of the number of nodes, they are fully implementable on the Kria SoM KV260 as they use a low bit precision to reduce the memory footprint of the model. The choice of algorithm did not make a difference as the second algorithm only removes partitions that do not fit on the hardware platform. Both of these algorithms indicate that the configuration that results in the best results is to implement all nodes on the FPGA and none on the CPU, as executing nodes on the CPU yields slower total runtime.

6.2. Future work

This study aimed to explore the potential of hybrid computing by using the strength of FPGA and CPU in a resource-constrained setting. To further expand on this research, some suggestions for future research are discussed in this section.

- A heuristic algorithm can be implemented which finds the largest portion of the neural network which fits on the FPGA. This research showed that implementing the largest part of the MobileNetv1 on the FPGA resulted in the lowest runtime. Thus an algorithm that can achieve this efficiently is beneficial.
- This study was conducted using the Kria SoM KV260 platform, and other platforms with varying levels of resources could yield different results. As shown the CPU slowed down the overall performance in this setup. Studying the effect of using a more powerful FPGA or CPU could improve the system's overall performance.
- Exploration of other neural network models using the implemented algorithms and partitioning techniques would be valuable as this study focused on MobileNetv1, cybersecurity, and CNV. The latter two are fully implementable on the Kria SoM KV260. Evaluating other models which are larger and more complex could provide insights into how well the algorithms and techniques generalize to other models and their specific characteristics. In this study, we restricted our design space to sequential model structures (i.e. non-forking), a future improvement using other partitioning techniques to allow for models with a forking structure that can be beneficial.
- Explore the effects of hybrid computing on non-convolutional networks, such as recurrent neural networks (RNN) or transformers. This will help determine if the partitioning and distribution techniques used for convolutional networks also apply to other types of neural networks. RNN networks are a network structure that the FINN framework cannot handle. As such, understanding how partitioning networks that contain RNN structures can significantly help implement quantized neural networks on edge devices by utilizing the FPGA capabilities for the FINN implementable nodes and offloading the non-finn layers to the CPU.
- Implementing NN applications using a hybrid solution requires further cost-benefit analysis for commercial implementation. A complete examination of the economic feasibility of deploying NN applications on a hybrid platform should be conducted to analyze whether such an implementation is feasible. This analysis should include the costs of developing and deploying the hybrid solution and the potential benefits of performance, power consumption, and resource utilization. Additionally, these metrics should be compared with a server-grade implementation to understand better if such an implementation is better for the given implementation.

A

Appendix A

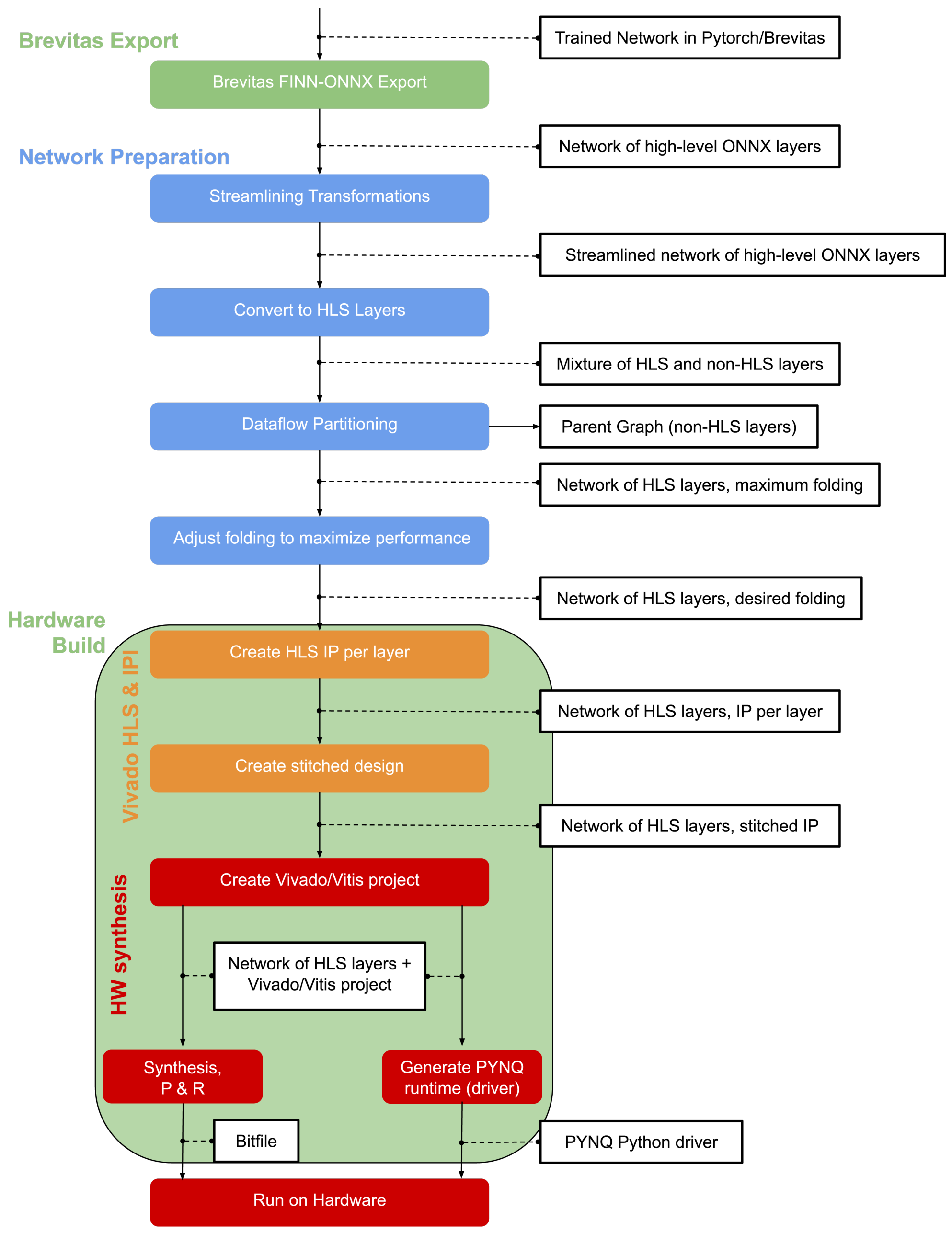

Brevitas Export
Trained Network in Pytorch/Brevitas
Brevitas FINN-ONNX Export
Network Preparation
Network of high-level ONNX layers
Streamlining Transformations
Streamlined network of high-level ONNX layers
Convert to HLS Layers
Mixture of HLS and non-HLS layers
Dataflow Partitioning
Parent Graph (non-HLS layers)
Network of HLS layers, maximum folding
Adjust folding to maximize performance
Network of HLS layers, desired folding
Hardware Build
Vivado HLS & IPI
Create HLS IP per layer
Network of HLS layers, IP per layer
Create stitched design
Network of HLS layers, stitched IP
HW synthesis
Create Vivado/Vitis project
Network of HLS layers + Vivado/Vitis project
Synthesis, P & R
Generate PYNQ runtime (driver)
Bitfile
PYNQ Python driver
Run on Hardware

Figure A.1: An end-to-end flow in FINN, starting from a trained PyTorch/Brevitas network up to deploying on FPGA [44]

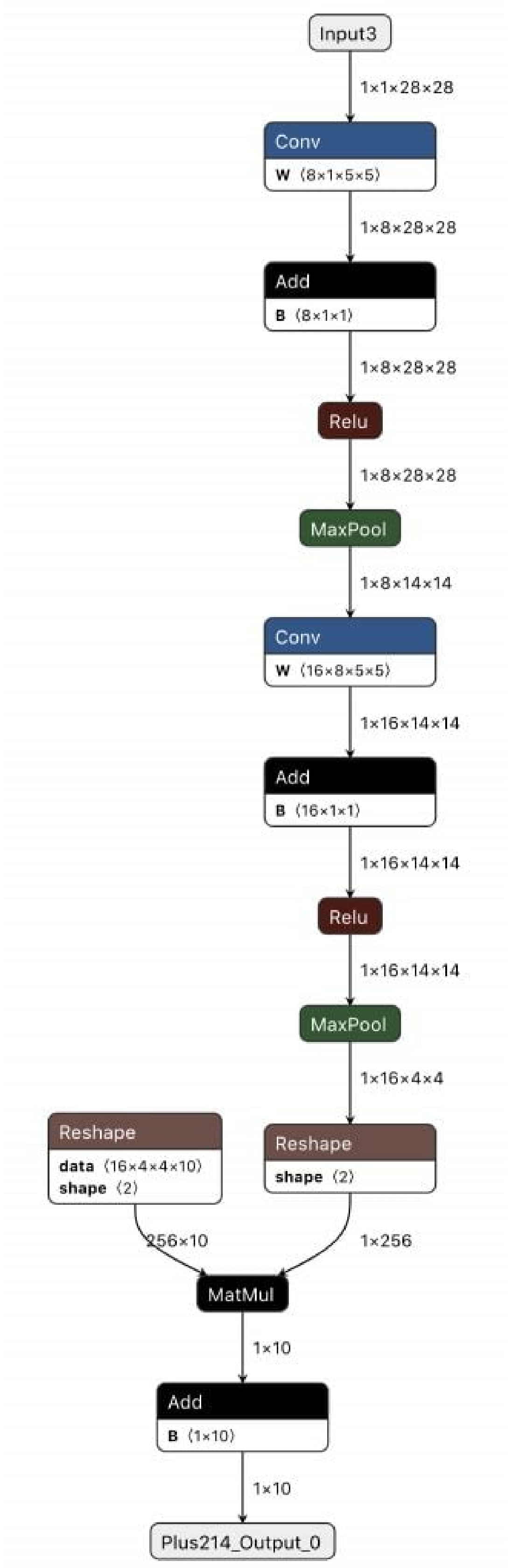


Figure A.2: An example of a ONNX graph

B Source code

B.1. Algorithm 1 & 2 source code FPGA

```
import numpy as np
from qonnx.core.modelwrapper import ModelWrapper
import finn.builder.build_dataflow as build
import finn.builder.build_dataflow_config as build_cfg
import os
from tqdm import tqdm
import shutil
from qonnx.util.cleanup import cleanup
from finn.transformation.streamline import Streamline
import argparse
from itertools import combinations
import json
from qonnx.util.basic import get_by_name

from custom_steps import (
    step_mobilenet_streamline,
    step_mobilenet_convert_to_hls_layers,
    step_mobilenet_convert_to_hls_layers_separate_th,
    step_mobilenet_lower_convs,
    step_mobilenet_slr_floorplan,
    custom_step_partition,
    custom_step_set_pe_simd
)


def generate_model_report(model_path,model_name):
    estimates_output_dir = f"output_estimates_{model_name}"
    custom_estimate_only_dataflow_steps = [
    "step_qonnx_to_finn",
    "step_tidy_up",
    "step_streamline",
    "step_convert_to_hls",
    "step_create_dataflow_partition",
    # "step_target_fps_parallelization",
    custom_step_set_pe_simd,
    "step_generate_estimate_reports",
]
    print("xczu3eg-sbva484-1-e")
    cfg_estimates = build.DataflowBuildConfig(
        output_dir          = estimates_output_dir,
        stop_step           = "step_generate_estimate_reports",
        # mvau_wwidth_max     = 80,
        # target_fps          = 1000000,
        synth_clk_period_ns = 10.0,
#         fpga_part           = "xck26-sfvc784-2LV-c", #KV260
        fpga_part           = "xczu3eg-sbva484-1-e", #PYNQ-Z1
#         steps               = build_cfg.estimate_only_dataflow_steps,
        steps               = custom_estimate_only_dataflow_steps,
        generate_outputs=[
            build_cfg.DataflowOutputType.ESTIMATE_REPORTS,
        ]
    )
    build.build_dataflow_cfg(model_path, cfg_estimates)
    cleanup(in_file = model_path,out_file = model_path)


    return estimates_output_dir

def generate_model_report_mobilenetV1(model_path,model_name):

    mobilenet_build_steps =  [
        step_mobilenet_streamline,
        step_mobilenet_lower_convs,
        step_mobilenet_convert_to_hls_layers_separate_th,
        "step_create_dataflow_partition",
        # "step_apply_folding_config",
        custom_step_set_pe_simd,
        "step_generate_estimate_reports",
        # "step_hls_codegen",
        # "step_hls_ipgen",
        # "step_set_fifo_depths",
        # "step_create_stitched_ip",
        # "step_synthesize_bitfile",
        # "step_make_pynq_driver",
        # "step_deployment_package",
    ]

    estimates_output_dir = f"output_estimates_{model_name}"

    cfg_estimates = build.DataflowBuildConfig(
        steps               = mobilenet_build_steps,
        output_dir          = estimates_output_dir,
        # mvau_wwidth_max     = 36,
        # target_fps          = 1000000,
        # folding_config_file = "folding.json",
        synth_clk_period_ns = 10.0,
        fpga_part           = "xck26-sfvc784-2LV-c",
        generate_outputs=[
            build_cfg.DataflowOutputType.ESTIMATE_REPORTS,
        ]
    )
    build.build_dataflow_cfg(model_path, cfg_estimates)
    cleanup(in_file = model_path,out_file = model_path)


    return estimates_output_dir

def get_subsets(model_file):
    # Instantiate a ModelWrapper object using the specified model file
    model = ModelWrapper(model_file)

    # Initialize lists for unwanted and wanted node indices
    unwanted_nodes, wanted_nodes = [], []

    # Store the maximum number of nodes in the graph
    max_len = len(model.graph.node)

    # Append the starting and ending indices of the graph to the wanted nodes list
    wanted_nodes.append(0)

    # Iterate through each node in the graph
    for ind, n in enumerate(model.graph.node):
        # If the node is a "MultiThreshold" node, add its index to the wanted nodes list
        if n.op_type == "MultiThreshold":
            wanted_nodes.append(ind)
    wanted_nodes.append(max_len)

    # Initialize empty lists for FPGA and CPU node combinations
    combinations_lst_fpga, combinations_lst_cpu = [], []

    # Iterate through each pair of starting and ending nodes in the wanted nodes list
    for i in wanted_nodes:
        for l in wanted_nodes:
            # If the starting node is not equal to the ending node and the ending node is not the last node in the graph
            fpga, cpu, cpu_0, cpu_1 = None, None, None, None
            if i < l:
                # If the starting node is the first node in the graph
                if i == 0 and l != max_len:
                    fpga = range(i, l)
                    cpu = range(l, max_len)
                # If equal to max len then
                elif l == max_len:
                    fpga = range(i,l)
                    if i != 0:
                        cpu = range(0,i)
                # If the starting node is not the first node in the graph
                else:
                    fpga = range(i, l)
                    cpu_0 = range(0, i)
                    cpu_1 = range(l, max_len)
                    cpu = (cpu_0, cpu_1)

                # Append the FPGA and CPU node combinations to their respective lists
                combinations_lst_fpga.append(fpga)
                combinations_lst_cpu.append(cpu)


    # Return the list of FPGA and CPU node combinations
    return combinations_lst_fpga,combinations_lst_cpu

def get_data_in(model_path):
    # Load the model specified by the given model_path using a ModelWrapper object
    model = ModelWrapper(model_path)

    # Get the name and size of the model's input tensor
    inp_name = model.graph.input[0].name
    inpt_size = model.get_tensor_shape(inp_name)
```

```python

    # Generate a random input tensor of the same size as the model’s input tensor
    input_tensor = np.random.randint(0, 255, inpt_size).astype(’uint8’)

    # Compute the size of the input tensor in MB
    input_size_mb = input_tensor.nbytes * 1e-6

    # Compute and return the input data transfer time based on the input tensor size
    return 1.0976256841119194 * input_size_mb + 0.3052411426075403

def get_data_out(model_path):
    # Load the model specified by the given model_path using a ModelWrapper object
    model = ModelWrapper(model_path)

    # Get the name and size of the model’s output tensor
    out_name = model.graph.output[0].name
    outpt_size = model.get_tensor_shape(out_name)

    # Generate a random output tensor of the same size as the model’s output tensor
    output_tensor = np.random.randint(0, 255, outpt_size).astype(’uint8’)

    # Compute the size of the output tensor in MB
    output_size_mb = output_tensor.nbytes * 1e-6

    # Compute and return the output data transfer time based on the output tensor size
    return 0.5606177488366392 * output_size_mb + 0.09549860536045693

def is_valid(model):
    if ”MultiThreshold” in [n.op_type for n in model.graph.node] and len(model.graph.node) > 1:
        return True
    else:
        return False

def check_resources(rpt_dir = ””, margin=1):
    KV_260_resource_limit = {
        ”BRAM_18K”: 144,
        ”LUT”: 117120,
        ”URAM”: 64,
        ”DSP”: 1248
    }

    with open(f”{rpt_dir}estimate_layer_resources.json”,’r’) as f:
        resource_rpt = json.loads(f.read())
    for k,v in resource_rpt[”total”].items():
        if not KV_260_resource_limit[k]*margin > v:
            return False
    return True
```

B.1.1. Algorithm 1 juypter notebook

```python
from FPGA_algorithm_0 import *
from qonnx.transformation.create_generic_partitions import PartitionFromDict
import time
import pandas as pd
import shutil

```

```
# Set path to the input model file
# model_file = "../models/mobilenetv1-w4a4_pre_post_tidy.onnx"
# model_file = "../models/end2end_cnv_w1a1_tidy.onnx"
model_file = "../models/cybsec-mlp-ready.onnx"

model_name = model_file.split("/")[-1].replace(".onnx","")

# Set the name for the outdir
outdir = f"../results"

if not os.path.exists(outdir):
    os.mkdir(outdir)

# Set platform name and algorithm number
platform = "FPGA"
Algorithm = "1"

# Clean up the input model file and write the cleaned version back to the same
     file
cleanup(in_file=model_file, out_file=model_file)

# Get all possible subsets of the input models
subset_models_FPGA = get_subsets(model_file)[0]
subset_models_CPU = get_subsets(model_file)[1]
subset_models = get_subsets(model_file)

# Instantiate a ModelWrapper object for the input model
model = ModelWrapper(model_file)

# Initialize an empty list to store all the subset metrics
all_subset_metrics = []

# Initialize a dictionary to store the best configuration
best_config = {}

# Initialize a variable to store the minimum runtime
min_runtime = float("inf")
passed =[]
# Iterate through the list of subset models
_start=time.time()
for ind,c in enumerate(tqdm(subset_models_FPGA)):
    print(c,subset_models_CPU[ind])
    # Transform the model to a partitioned version with the specified
    partitioning scheme
    parent = model.transform(PartitionFromDict(partitioning={0:c},
    partition_dir=f"tmp_models"))
    model_path = "tmp_models/partition_0.onnx"
    if is_valid(ModelWrapper(model_path)):
        # Estimate the runtime of the model on FPGA using the
    estimate_network_performance.json file
        if "mobilenetv1" in model_name:
            est_dir = generate_model_report_mobilenetV1(model_path,model_name)
        else:
            est_dir = generate_model_report(model_path,model_name)
        passed.append(ind)

        # Get the estimated data movement times
        dm_in = get_data_in(model_path)
        dm_out = get_data_out(model_path)

        # Load the estimated runtime information from the
    estimate_network_performance.json file
        with open (f"{est_dir}/report/estimate_network_performance.json",'r')
    as f:
            data = json.load(f)

        # Calculate the total runtime, and parse the runtime information to a
    dictionary
        runtime = data["critical_path_cycles"] * 1e-8 + dm_in + dm_out
        run_info = {
                "FPGA_range": str(c),
                "total_runtime_FPGA[ms]": runtime,
                "runtime_FPGA[ms]": data["critical_path_cycles"] * 1e-8,
                "data_in_FPGA[ms]": dm_in,
                "data_out_FPGA[ms]": dm_out,
                "CPU_range": str(subset_models_CPU[ind]),
        }
        all_subset_metrics.append(run_info)
        open(f"{outdir}/results_{model_name}_{platform}_{Algorithm}.json",'w')
    .write(json.dumps(all_subset_metrics))

_end = time.time()

_est_time = _end-_start
# parse estimation time to json file
with open(f"{outdir}/estimation_time_{model_name}_{platform}_{Algorithm}.json"
    ,'w') as f:
    f.write(json.dumps(
            {"Estimation time[s]":_est_time,
            "Platform": platform,
            "Algorithm": Algorithm})
           )
# Print the estimation time to terminal
print(f"Total runtime algorithm_1: {_est_time:.5f} s")
```

B.1.2. Algorithm 2 juypter notebook

```
from FPGA_algorithm_1 import *
from qonnx.transformation.create_generic_partitions import PartitionFromDict
import time
import pandas as pd
import shutil

# Set path to the input model file
# model_file = "../models/mobilenetv1-w4a4_pre_post_tidy.onnx"
model_file = "../models/end2end_cnv_w1a1_tidy.onnx"
# model_file = "../models/cybsec-mlp-ready.onnx"

# Get name of model
model_name = model_file.split("/")[-1].replace(".onnx","")

# Set the name for the outdir
outdir = f"../results_pynqZ1"

if not os.path.exists(outdir):
    os.mkdir(outdir)

# Set platform name and algorithm number
platform = "FPGA"
Algorithm = "2"

# Clean up the input model file and write the cleaned version back to the same
     file
```

```python
cleanup(in_file=model_file, out_file=model_file)

# Get all possible subsets of the input models
subset_models_FPGA = get_subsets(model_file)[0]
subset_models_CPU = get_subsets(model_file)[1]
subset_models = get_subsets(model_file)

# Instantiate a ModelWrapper object for the input model
model = ModelWrapper(model_file)

# Initialize an empty list to store all the subset metrics
all_subset_metrics = []

# Initialize a dictionary to store the best configuration
best_config = {}

# Initialize a variable to store the minimum runtime
min_runtime = float(”inf”)
fail_start = None
passed =[]
# Iterate through the list of subset models
_start=time.time()
for ind,c in enumerate(tqdm(subset_models_FPGA)):
    print(c,subset_models_CPU[ind])
    # Transform the model to a partitioned version with the specified partitioning scheme
    parent = model.transform(PartitionFromDict(partitioning={0:c}, partition_dir=f”tmp_models”))
    model_path = ”tmp_models/partition_0.onnx”
    # If the partition is valid
    if is_valid(ModelWrapper(model_path)):
        # If the system has failed then check if the start node is not present, no estimate will be generated.
        if fail_start != c[0]:
            if ”mobilenet” in model_name:
                est_dir = generate_model_report_mobilenetV1(model_path, model_name)
            else:
                est_dir = generate_model_report(model_path, model_name)
        # if check resources returns False then the split does not fit onto FPGA
        if not check_resources(rpt_dir = f”{est_dir}/report/”, margin=1):
            # Print statement to show where it doesn’t fit anymore
            print(f”{c} does not fit!”)
            # Set a fail point to prune the tree, as every below fail does not fit either
            fail_start = c[0]
        else:
            fail_start = None
            # Store the index that succeeded, will be used later.
            passed.append(ind)
            # Get the estimated data movement times
            dm_in = get_data_in(model_path)
            dm_out = get_data_out(model_path)

            # Load the estimated runtime information from the estimate_network_performance.json file
            with open (f”{est_dir}/report/estimate_network_performance.json”,’r’) as f:
                data = json.load(f)

            # Calculate the total runtime, and parse the runtime information to a dictionary
            runtime = data[”critical_path_cycles”] * 1e-8 + dm_in + dm_out
            run_info = {
                ”FPGA_range”: str(c),
                ”total_runtime_FPGA[ms]”: runtime,
                ”runtime_FPGA[ms]”: data[”critical_path_cycles”] * 1e-8,
                ”data_in_FPGA[ms]”: dm_in,
                ”data_out_FPGA[ms]”: dm_out,
                ”CPU_range”: str(subset_models_CPU[ind])
            }
            all_subset_metrics.append(run_info)
            open(f”{outdir}/results_{model_name}_{platform}_{Algorithm}.json”, ’w’).write(json.dumps(all_subset_metrics))

_end = time.time()
_est_time = _end-_start
with open(f”{outdir}/estimation_time_{model_name}_{platform}_{Algorithm}.json”,’w’) as f:
    f.write(json.dumps({”Estimation time[s]”:_est_time,
                        ”Platform”: platform,
                        ”Algorithm”: Algorithm}))
print(f”Total runtime algorithm_1: {_est_time:.5f} s”)
```

B.2. CPU

B.2.1. CPU estimtate

```
from qonnx.transformation.double_to_single_float import DoubleToSingleFloat
from onnx import helper
from onnxruntime_extensions import get_library_path
import onnxruntime as ort
import numpy as np
from qonnx.core.modelwrapper import ModelWrapper
import time
import os
from tqdm import tqdm
import shutil
import qonnx.core.onnx_exec as oxe
import argparse
from itertools import combinations
import json
import pandas as pd
from qonnx.transformation.create_generic_partitions import PartitionFromDict
from qonnx.util.basic import get_by_name


def dataflow_parent_setup(model_file):
    model = ModelWrapper(model_file)
    for n in model.graph.node:
        if n.domain == ”finn.custom_op.fpgadataflow”:
            n.domain = ”ai.onnx.contrib”
    model.save(”dataflow_parent_run.onnx”)

def set_multithreshold_default(model):#,save_model):
    ’’’
    Pass a modelproto model and the save file
    ’’’

    model = model.transform(DoubleToSingleFloat())
    dl = ””
    for n in model.graph.node:
        if n.op_type == "MultiThreshold":
            if len(model.get_tensor_shape(n.input[0])) == 2:
                dl = "NC"
            elif len(model.get_tensor_shape(n.input[0])) == 4:
                dl = "NCHW"
            else:
                assert dl == "", "NON valid Data layout"

    new_attr = [helper.make_attribute("out_scale", 1.0),
                helper.make_attribute("out_bias", 0.0),
                helper.make_attribute("data_layout",dl)]

    for n in model.graph.node:
        if n.op_type == "MultiThreshold":
            out_scale,bias,datalayout = False,False,False
            for na in n.attribute:
                if na.name == "out_scale": out_scale = True
                if na.name == "out_bias": bias = True
                if na.name == "data_layout": datlayout = True
            if not out_scale: n.attribute.append(new_attr[0])
            if not bias: n.attribute.append(new_attr[1])
            if not datalayout: n.attribute.append(new_attr[2])

            n.domain = "ai.onnx.contrib"
    return model

def revert_quantAvgPool(model):
    nodes = [n for n in model.graph.node if n.op_type == 'QuantAvgPool2d']
    attrs = [n.attribute for n in model.graph.node if n.op_type == '
    QuantAvgPool2d']
    for node,attr in zip(nodes,attrs):
        for a in attr:
            if a.name == "stride":
                s = a.i
            elif a.name == "kernel":
                k = a.i
        update = helper.make_node(
            "AveragePool",
            inputs=[node.input[0]],
            outputs=[node.output[0]],
            kernel_shape=[k,k],
            strides=[s,s],
        )

        model.graph.node.remove(node)
        model.graph.node.append(update)
    return model


def get_subsets(model_file):
    model = ModelWrapper(model_file)
    unwanted_nodes,wanted_nodes = [],[]
    max_len = len(model.graph.node)
    combinations_lst_fpga,combinations_lst_cpu = [],[]
    wanted_nodes.append(0)
    for ind,n in enumerate(model.graph.node):
        if n.op_type == "MultiThreshold":
            wanted_nodes.append(ind)
    wanted_nodes.append(max_len)

    for i in range(len(wanted_nodes)):
        for l in range(len(wanted_nodes)):
            if i<l:
                if wanted_nodes[l]+1 < max_len  and wanted_nodes[i] == 0:
                    fpga = range(wanted_nodes[i],wanted_nodes[l]+1)
                    cpu = range(wanted_nodes[l]+1,max_len+1)
                    combinations_lst_fpga.append(fpga)
                    combinations_lst_cpu.append(cpu)
                else:
                    fpga = range(wanted_nodes[i],wanted_nodes[l]+1)
                    if fpga != range(0,max_len+1):
                        cpu = {”part”: 2, ”ranges”: [range(0,wanted_nodes[i]), range(wanted_nodes[l]+1,max_len+1)]}

                    combinations_lst_fpga.append(fpga)
                    combinations_lst_cpu.append(cpu)
    return combinations_lst_fpga,combinations_lst_cpu

so = ort.SessionOptions()
so.register_custom_ops_library(get_library_path())

def exe_ort(model_file):
    model = ModelWrapper(model_file)
    if [True for n in model.graph.node if n.op_type == ”QuantAvgPool2d”] != []:
        model = revert_quantAvgPool(model)
    model = set_multithreshold_default(model)

    sess = ort.InferenceSession(model.model.SerializeToString(),so)

    x_test = np.ones(model.get_tensor_shape(model.graph.input[0].name), dtype=np.float32)

    idict = {model.graph.input[0].name: x_test}
    start = time.time()
    try:
        sess.run([],idict)
    except:
        pass
    stop = time.time()
    runtime = (stop-start)*1e3

    return runtime

def get_input_file(model_name,file_directory):
    input_file = [os.path.join(file_directory,i) for i in os.listdir(file_directory) if ”results” in i and model_name in i and i.endswith(”json”)]
    if len(input_file) == 1:
        with open(input_file[0],’r’) as f:
            data = json.loads(f.read())
        return data, input_file[0]
    else:
        return None,None
def write_estimation_time_to_file(model_name,file_directory, est_dict):
    estimation_file = [os.path.join(file_directory,i) for i in os.listdir(file_directory) if ”estimation_time_” in i and model_name in i and i.endswith(’.json’)]
    if len(estimation_file) == 1:
        with open(estimation_file[0],’r’) as f:
            data = json.loads(f.read())
        pd.DataFrame([data,est_dict]).to_csv(estimation_file[0].replace(”json”
    ,"csv"),index=False)


def get_top_k_result(k,result_file,model_name,outdir):
    df = pd.read_csv(result_file)
    min = df.sort_values("total_runtime[ms]",ascending=True)
    min = min[['FPGA_range', 'total_runtime_FPGA[ms]','CPU_range', '
    total_runtime_CPU[ms]','total_runtime[ms]']].head(k).reset_index(drop=True
    )
    min.to_csv(f"{outdir}/top_{k}_{model_name}.csv",index=False)
```

B.2.2. Algorithm 1 & 2 juypter notebook CPU

```
import pandas as pd
import shutil
from CPU_estimate import *
import os

# print(os.listdir("../results"))
# Get all possible subsets loaded from the FPGA splits possible
# model_file = "../models/mobilenetv1-w4a4_pre_post_tidy.onnx"
model_file = "../models/end2end_cnv_w1a1_tidy.onnx"
# model_file = "../models/cybsec-mlp-ready.onnx"
for model_file in models:

    model_name = model_file.split('/')[-1].replace(".onnx","")
    # Define the location where the result file is located
    outdir = f"../results"
    if not os.path.exists("results"):
        os.mkdir("results")
    # Get the input data file
    [rng_data,result_dir] = get_input_file(model_name,outdir)

    platform,Algorithm = "CPU", "2"

    _base_model = ModelWrapper(model_file)
    all_subset_metrics = []
    min_runtime = float("inf")

    n = 5
    _start = time.time()
    for c in tqdm(rng_data):
        rng = eval(c["CPU_range"])
        runtime, runtime_0, runtime_1 = 0,0,0
        if type(rng) == range:

            part_0 = None
            part_1 = rng
            parent = _base_model.transform(PartitionFromDict(partitioning={1:
    part_1},partition_dir=f"tmp_models"))
            for _ in range(n):
                runtime += exe_ort("tmp_models/partition_1.onnx")
            c["total_runtime_CPU[ms]"] = runtime/n

        elif type(rng) != range and rng != None:

            part_0 = rng[0]
            part_1 = rng[1]
            parent = _base_model.transform(PartitionFromDict(partitioning={0:
    part_0,1:part_1},partition_dir=f"tmp_models"))
            for _ in range(n):
                runtime_0 += exe_ort("tmp_models/partition_0.onnx")
                runtime_1 += exe_ort(”tmp_models/partition_1.onnx”)

            c[”runtime_part_0[ms]”] = runtime_0/n,
            c[”runtime_part_1[ms]”] =  runtime_1/n,
            c[”total_runtime_CPU[ms]”] = (runtime_0+runtime_1)/n

        with open(f”results/results_{model_name}_{platform}_{Algorithm}.json”, ’w’) as file:
            json.dump(c,file)

    _end = time.time()
    _est_time = _end - _start

    est = {”Estimation time[s]”:_est_time,
    ”Platform”: platform,
    ”Algorithm”: Algorithm}
    write_estimation_time_to_file(model_name,outdir,est)

    print(f”Total estimation time for CPU {_est_time} s”)

    df = pd.DataFrame(rng_data)
    df[”total_runtime[ms]”] = df[”total_runtime_FPGA[ms]”] + df[”total_runtime_CPU[ms]”]
    df.to_csv(f”{outdir}/results_{model_name}_complete.csv”)

    get_top_k_result(5,f”{outdir}/results_{model_name}_complete.csv”, model_name,outdir)
    shutil.rmtree(”tmp_models”)
```

Bibliography

[1] K. Suzuki, *Machine learning in computer-aided diagnosis: Medical imaging intelligence and analysis: Medical imaging intelligence and analysis*. IGI Global, 2012.

[2] P. Yan, K. Suzuki, F. Wang, and D. Shen, *Machine learning in medical imaging*, Pages: 1327–1329 Publication Title: Machine Vision and Applications Volume: 24, 2013.

[3] K. Suzuki, “Overview of deep learning in medical imaging,” *Radiological physics and technology*, vol. 10, no. 3, pp. 257–273, 2017, Publisher: Springer.

[4] P. Picton and P. Picton, *What is a neural network?* Springer, 1994.

[5] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, “Hardware for machine learning: Challenges and opportunities,” in *2017 IEEE Custom Integrated Circuits Conference (CICC)*, IEEE, 2017, pp. 1–8.

[6] J. Lee and H.-J. Yoo, “An Overview of Energy-Efficient Hardware Accelerators for On-Device Deep-Neural-Network Training,” *IEEE Open Journal of the Solid-State Circuits Society*, vol. 1, pp. 115–128, 2021. doi: `10.1109/OJSSCS.2021.3119554`.

[7] M. G. S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, and F. Hussain, “Machine Learning at the Network Edge: A Survey,” *ACM Comput. Surv.*, vol. 54, no. 8, Oct. 2021, Place: New York, NY, USA Publisher: Association for Computing Machinery, issn: 0360-0300. doi: `10.1145/3469029`. [Online]. Available: `https://doi.org/10.1145/3469029`.

[8] S. Liu, D. S. Ha, F. Shen, and Y. Yi, “Efficient neural networks for edge devices,” en, *Computers & Electrical Engineering*, vol. 92, p. 107 121, Jun. 2021, issn: 00457906. doi: `10.1016/j.compeleceng.2021.107121`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/S0045790621001257` (visited on 04/14/2023).

[9] M. Zhang, F. Zhang, N. D. Lane, *et al.*, “Deep Learning in the Era of Edge Computing: Challenges and Opportunities,” en, in *Fog Computing*, A. Zomaya, A. Abbas, and S. Khan, Eds., 1st ed., Wiley, May 2020, pp. 67–78, isbn: 978-1-119-55169-0 978-1-119-55171-3. doi: `10.1002/9781119551713.ch3`. [Online]. Available: `https://onlinelibrary.wiley.com/doi/10.1002/9781119551713.ch3` (visited on 04/18/2023).

[10] Srivatsan Krishnan, *Quantization for Fast and Environmentally Sustainable Reinforcement Learning – Google AI Blog*, Sep. 2022. [Online]. Available: `https://ai.googleblog.com/2022/09/quantization-for-fast-and.html#:%20:text=Quantization%20can%20save%20memory%20storage,models%20and%20achieve%20faster%20training.`.

[11] Y. Umuroglu, N. J. Fraser, G. Gambardella, *et al.*, “FINN: A Framework for Fast, Scalable Binarized Neural Network Inference,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, arXiv:1612.07119 [cs], Feb. 2017, pp. 65–74. doi: `10.1145/3020078.3021744`. [Online]. Available: `http://arxiv.org/abs/1612.07119` (visited on 07/06/2022).

[12] S. M. S. Trimberger, “Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology: This Paper Reflects on How Moore’s Law Has Driven the Design of FPGAs Through Three Epochs: The Age of Invention, the Age of Expansion, and the Age of Accumulation,” *IEEE Solid-State Circuits Magazine*, vol. 10, no. 2, pp. 16–29, 2018, issn: 1943-0582. doi: `10.1109/MSSC.2018.2822862`. [Online]. Available: `https://ieeexplore.ieee.org/document/8392473/` (visited on 02/22/2023).

[13] *FPGA Design, Architecture and Applications [2023]*. [Online]. Available: `https://www.logic-fruit.com/blog/fpga/fpga-design-architecture-and-applications/`.

[14] *Kria KV260 Vision AI Starter Kit Data Sheet (DS986)*, Mar. 2022. [Online]. Available: `https://docs.xilinx.com/r/en-US/ds986-kv260-starter-kit/Summary`.

[15] *What’s a SOM*. [Online]. Available: `https://www.xilinx.com/products/som/what-is-a-som.html`.

[16] A. Khokhar, V. Prasanna, M. Shaaban, and C.-L. Wang, “Heterogeneous computing: Challenges and opportunities,” *Computer*, vol. 26, no. 6, pp. 18–27, Jun. 1993, issn: 0018-9162. doi: `10.1109/2.214439`. [Online]. Available: `http://ieeexplore.ieee.org/document/214439/` (visited on 02/07/2023).

[17] Q. Liu and W. Luk, “Heterogeneous Systems for Energy Efficient Scientific Computing,” in *Reconfigurable Computing: Architectures, Tools and Applications*, O. C. S. Choy, R. C. C. Cheung, P. Athanas, and K. Sano, Eds., vol. 7199, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 64–75, isbn: 978-3-642-28364-2 978-3-642-28365-9. doi: `10.1007/978-3-642-28365-9_6`. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-28365-9_6` (visited on 02/07/2023).

[18] I. Pipelining, “Instruction Pipelining,” *Computer*, 2003.

[19] P. M. Kogge, *The architecture of pipelined computers*. CRC press, 1981.

[20] D. B. Thomas, L. Howes, and W. Luk, “A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation,” en, in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, Monterey California USA: ACM, Feb. 2009, pp. 63–72, isbn: 978-1-60558-410-2. doi: `10.1145/1508128.1508139`. [Online]. Available: `https://dl.acm.org/doi/10.1145/1508128.1508139` (visited on 04/04/2023).

[21] Y. Li, X. Zhao, and T. Cheng, “Heterogeneous Computing Platform Based on CPU+FPGA and Working Modes,” in *2016 12th International Conference on Computational Intelligence and Security (CIS)*, Wuxi, China: IEEE, Dec. 2016, pp. 669–672, isbn: 978-1-5090-4840-3. doi: `10.1109/CIS.2016.0161`. [Online]. Available: `http://ieeexplore.ieee.org/document/7820552/` (visited on 02/07/2023).

[22] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, *What is the State of Neural Network Pruning?* arXiv:2003.03033 [cs, stat], Mar. 2020. [Online]. Available: `http://arxiv.org/abs/2003.03033` (visited on 01/26/2023).

[23] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain.,” en, *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958, issn: 1939-1471, 0033-295X. doi: `10.1037/h0042519`. [Online]. Available: `http://doi.apa.org/getdoi.cfm?doi=10.1037/h0042519` (visited on 01/25/2023).

[24] S. Han, H. Mao, and W. J. Dally, *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*, arXiv:1510.00149 [cs], Feb. 2016. [Online]. Available: `http://arxiv.org/abs/1510.00149` (visited on 02/01/2023).

[25] J. Bai, F. Lu, K. Zhang, *et al.*, *ONNX: Open Neural Network Exchange*, Publication Title: GitHub repository, 2019. [Online]. Available: `https://github.com/onnx/onnx`.

[26] Faith Xu, "Faster Scalable ML Model Deployment Using ONNX and Open Source Tools," in *2020 IEEE Infrastructure Conference*, San Francisco, CA, USA: IEEE, Oct. 2020, pp. i–i, isbn: 978-1-72817-728-1. doi: `10.1109/IEEECONF47748.2020.9377615`. [Online]. Available: `https://ieeexplore.ieee.org/document/9377615/` (visited on 04/04/2023).

[27] Google, *Protocolbuffers/protobuf: Protocol Buffers - Google's data interchange format*. [Online]. Available: `https://github.com/protocolbuffers/protobuf`.

[28] ONNX Runtime developers, *ONNX Concepts - ONNX 1.15.0 documentation*. [Online]. Available: `https://onnx.ai/onnx/intro/concepts.html`.

[29] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[30] M. Abadi, P. Barham, J. Chen, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th Symposium on Operating Systems Design and Implementation*, 2016, pp. 265–283.

[31] ONNX Runtime developers, *ONNX Runtime*, 2021. [Online]. Available: `https://onnxruntime.ai/`.

[32] Y. Umuroglu and M. Jahre, *Streamlined Deployment for Quantized Neural Networks*, arXiv:1709.04060 [cs], May 2018. [Online]. Available: `http://arxiv.org/abs/1709.04060` (visited on 07/06/2022).

[33] *Xilinx/Vitis-AI: Vitis AI is Xilinx's development stack for AI inference on Xilinx hardware platforms, including both edge devices and Alveo cards.* [Online]. Available: `https://github.com/Xilinx/Vitis-AI`.

[34] T. Chen, T. Moreau, Z. Jiang, *et al.*, "TVM: End-to-end optimization stack for deep learning," *arXiv preprint arXiv:1802.04799*, vol. 11, no. 20, 2018, Publisher: CoRR.

[35] FastML Team, *Fastmachinelearning/hls4ml*, 2023. doi: `10.5281/zenodo.1201549`. [Online]. Available: `https://github.com/fastmachinelearning/hls4ml`.

[36] S. B. Akintoye, L. Han, H. Lloyd, *et al.*, *Layer-Wise Partitioning and Merging for Efficient and Scalable Deep Learning*, arXiv:2207.11019 [cs], Jul. 2022. [Online]. Available: `http://arxiv.org/abs/2207.11019` (visited on 04/11/2023).

[37] A. Parthasarathy and B. Krishnamachari, *Partitioning and Placement of Deep Neural Networks on Distributed Edge Devices to Maximize Inference Throughput*, arXiv:2210.12219 [cs], Oct. 2022. [Online]. Available: `http://arxiv.org/abs/2210.12219` (visited on 04/11/2023).

[38] R. Ding, G. Su, G. Bai, W. Xu, N. Su, and X. Wu, “A FPGA-based Accelerator of Convolutional Neural Network for Face Feature Extraction,” in *2019 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, 2019, pp. 1–3. doi: `10.1109/EDSSC.2019.8754067`.

[39] K. He, X. Zhang, S. Ren, and J. Sun, *Deep Residual Learning for Image Recognition*, arXiv:1512.03385 [cs], Dec. 2015. [Online]. Available: `http://arxiv.org/abs/1512.03385` (visited on 04/10/2023).

[40] A. Pappalardo, Y. Umuroglu, M. Blott, *et al.*, *QONNX: Representing Arbitrary-Precision Quantized Neural Networks*, arXiv:2206.07527 [cs, stat], Jun. 2022. [Online]. Available: `http://arxiv.org/abs/2206.07527` (visited on 04/19/2023).

[41] L. Bai, Y. Zhao, and X. Huang, “A CNN Accelerator on FPGA Using Depthwise Separable Convolution,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1415–1419, 2018. doi: `10.1109/TCSII.2018.2865896`.

[42] D. Wu, Y. Zhang, X. Jia, *et al.*, “A High-Performance CNN Processor Based on FPGA for MobileNets,” in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 136–143. doi: `10.1109/FPL.2019.00030`.

[43] A. Sohrabizadeh, J. Wang, and J. Cong, “End-to-End Optimization of Deep Learning Applications,” en, in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Seaside CA USA: ACM, Feb. 2020, pp. 133–139, isbn: 978-1-4503-7099-8. doi: `10.1145/3373087.3375321`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3373087.3375321` (visited on 04/19/2023).

[44] *End-to-End Flow — FINN documentation*. [Online]. Available: `https://finn.readthedocs.io/en/latest/end_to_end_flow.html`.