Interconnecting Multiple FPGAs for Distributed Machine Learning Inference

LINUS CARLSSON & FELIX MALMSJÖ
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Interconnecting Multiple FPGAs for Distributed Machine Learning Inference

Linus Carlsson
li1633ca-s@student.lu.se, linus.carlsson@inceptron.io
Felix Malmsjö
fe7501ma-s@student.lu.se, felix.malmsjo@inceptron.io

Inceptron AB

Supervisors: Erik Larsson (LTH), Steffen Malkowsky (Inceptron), Lucas Ferreira (Inceptron)

Examiner: Pietro Andreani

13th June 2024

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Machine learning has been growing in popularity for many years now and is currently popular among both businesses and private individuals. Running machine learning models takes a lot of computational resources, which means a large amount of energy is consumed.

The aim of this thesis is to investigate the usage of machine learning model specific hardware on Field Programmable Gate Arrays (FPGAs) as a potential solution to low-latency machine learning inference. However, as FPGAs have limited resources, not all models can fit on one FPGA. Therefore, the model has to be partitioned to be run over two or more FPGAs to distribute the need for resources.

It was found that partitioning the machine learning model over multiple FPGAs introduces a few challenges, such as an unbalanced resource utilization over the FPGAs and a possible bottleneck of inter-FPGA communication when the goal is high throughput. The main difficulty when designing the hardware accelerator is how to store all the millions of parameters. On the other hand, the introduced latency of the inter-FPGA communication is negligible, which means focus can be placed on reducing the computational latency. Top-1 inference accuracy is also close to the fully floating-point model, while top-5 accuracy is slightly higher when implementing the convolutional neural network in hardware.

Acknowledgements

We would like to extend a massive thank you to Inceptron AB for the opportunity to work on this thesis. Everyone on the Inceptron AB team has been more than welcoming of all types of questions that have led to some fantastic discussions. We would like to thank Lucas Ferreira, Patrik Persson, and Steffen Malkowsky especially for their continued support during the thesis.

Contents

List of Figures

List of Tables

List of Acronyms

AI Artificial Intelligence. 40

AMD Advanced Micro Devices. 2, 7, 12, 22, 24, 37, 38

API Application Programming Interface. 25

ASIC Application Specific Integrated Circuit. 8, 9, 12, 13, 27

AWS Amazon Web Services. 2, 7, 8, 12, 19–22, 24, 33, 38

AXI4 Advanced eXtensible Interface 4. 22–26, 37, 38

BRAM Block RAM. 7, 9, 17, 20, 21, 27, 33, 34, 39

Chisel Constructing Hardware in a Scala Embedded Language. 2, 13

CLI Command Line Interface. 2

CNN Convolutional Neural Network. vii, 3, 5, 6, 17, 19, 40

CPU Central Processing Unit. ix, 8, 11, 13, 20–25, 27, 29, 33, 39, 40

DDR4 Double Data Rate 4. 20

DMA Direct Memory Access. v, ix, 7, 12, 21–25, 29, 31, 38, 39

DNN Deep Neural Network. 3, 20

DRAM Dynamic Random Access Memory. 20

DS Depthwise Separable. 4

DSP Digital Signal Processor. 7–9, 12, 15, 16, 19, 21, 27, 30, 34, 39

FC Fully-Connected. 4, 11, 17, 20, 26, 40

FF Flip-Flop. 27

FIFO first in, first out buffer. 20, 22–26, 31, 38

FPGA Field Programmable Gate Array. v, vii, ix, 1–3, 7–9, 11–13, 17, 19–27, 29, 31–33, 35, 37–40

FPS Frames Per Second. 21, 32

GCC GNU Compiler Collection. 2

GPT Generative Pre-trained Transformer. 5, 6

GPU Graphics Processing Unit. 7–9, 12, 13, 27, 39, 40

HBM High Bandwidth Memory. 20

HLS High Level Synthesis. 3, 13

IEEE Institute of Electrical and Electronics Engineers. 7, 17

ILA Integrated Logic Analyzer. 2

IP Intellectual Property. 22, 34

JTAG Joint Test Action Group. 2, 37

LLM Large Language Model. 12

LUT Look Up Tables. 8, 19, 27, 30, 39

LUTRAM Look Up Table RAM. 27

ML Machine Learning. 1, 3, 7, 8

NPU Neural Processing Unit. 9

PCIe Peripheral Component Interconnect Express. 7, 12, 21–26, 29, 37, 38

PCIM PCI Master. 38

QSFP Quad Small Form-factor Pluggable. 7

R/W Read/Write. 22–24

RAM Random Access Memory. 29, 34, 38

ReLU Rectified Linear Unit. 17

RO Read Only. 22–24

ROM Read Only Memory. 20

RTL Register Transfer Level. 3, 13

sbt Simple Build Tool. 2

Scala Scalable Language. 2

TFLite Tensorflow Lite. v, 15

TPU Tensor Processing Unit. 8

URAM UltraRAM. 7, 9, 20, 27, 39

VHDL Very High Speed Integrated Circuit Hardware Description Language. 13

WO Write Only. 22–24

XDMA Xilinx Direct Memory Access. 2, 22, 24, 38

Chapter 1
Introduction

1.1 Thesis goals and questions

The goal of this thesis is to design and investigate a scalable system that would allow for running Machine Learning (ML) inference on Field Programmable Gate Arrays (FPGAs) in the cloud. This includes the system to send data to and between the FPGAs, the machine learning model itself implemented on one or multiple FPGAs, and a way to receive the inference results of the machine learning model.

1.1.1 Questions and challenges

During the planning phase, it was predicted that a few challenges would arise.

- How can hardware implemented on FPGAs in the cloud be debugged?
- What protocols should be implemented to reduce latency and keep high throughput while keeping flexibility in the partitioning of the models?
- Multiple cloud providers — Different architectures and topologies lead to different interconnect setups.
- Latency calculation and optimization.

1.1.2 Thesis scope

The scope of this thesis is to develop, benchmark, and investigate the viability of a system to send data from a host system to the FPGA, to create a hardware design for a ML model that can be factored to occupy multiple FPGAs, and similarly, create a system that can receive the inference result from the hardware design. Furthermore, a system that can transfer data between FPGAs is to be realized.

Focus is placed on the implementation of the hardware accelerator, transfer between host and FPGA, and inter-FPGA communication. Latency calculations for these sections are as such in the scope of this thesis. Other performance metrics for the neural network and transfer system are included. Cloud debugging is explored briefly. Due to hardware limitations by the cloud providers, specific protocol investigation is outside the scope of this work. Exploring different cloud provider architectures and topologies is also left outside the scope.

1.1.3 Tools and infrastructure

All hardware implementations are done in Constructing Hardware in a Scala Embedded Language (Chisel). Simple Build Tool (sbt) is used to run the Scalable Language (Scala)/Chisel code which generates Verilog that can be imported into Vivado where the block diagram feature is used to connect the design to the cloud FPGA shell. Vivado is also used to perform the synthesis and implementation. Because of the restrictions placed on the cloud FPGAs, bit stream generation from the implementation checkpoints is done using the cloud Command Line Interface (CLI).

Ninja is used as the primary build system. It is used to generate verification data and manage scripts that depend on other scripts. Several shell scripts are also created to reduce manual input of repeatedly used commands. GNU Compiler Collection (GCC) is used to compile the software driver to interact with the FPGA. The software driver utilizes the `fpga_pci` library from Amazon Web Services (AWS) and the Xilinx Direct Memory Access (XDMA) drivers provided by Advanced Micro Devices (AMD) through the cloud provider.

An instance of Vivado hosted in the cloud is also used to debug the implemented hardware using the System Integrated Logic Analyzer (ILA). To enable debugging on the FPGAs attached to the virtual instances in the cloud, a virtual Joint Test Action Group (JTAG) interface is used. This virtual interface is deployed on the instance hosting the FPGA and attached to a specific FPGA slot. Vivado connects to this virtual JTAG interface to read and configure the System ILA. The System ILA is configured to trigger on certain conditions.

1.2 Previous work

During the course of the thesis, some relevant literature was reviewed. The literature relates to using a hardware construction language like Chisel to create parameterizable hardware generators [1]. How to interconnect multiple FPGAs was also investigated [2], [3], along with the partitioning of neural networks and distribution of computation workload [4].

1.2.1 Chisel

Using Chisel to create hardware generators that generate hardware based on input parameters has been done previously and shows promise [1]. Chisel allows for the usage of parameters propagating through multiple layers of such generators to create highly customizable logic. Chisel also helps avoid excess boilerplate and contains a substantial amount of ready-made components that can be instantiated in a design, speeding up development. Since Chisel is based on Scala, it allows for the use of both object-oriented and functional programming when creating the hardware constructors.

1.2.2 Interconnecting multiple FPGAs

Previous work has proposed custom interconnect strategies that allow for high throughput and low latency. PlasticNet is one such system. When compared to Ethernet, PlasticNet promises lower latency [2]. However, PlasticNet integrates into a High Level Synthesis (HLS) environment, while the goal presented in this thesis requires an integration into Register Transfer Level (RTL). Furthermore, these systems would not work in the cloud environment where only certain interfaces are available for use.

1.2.3 Factoring neural networks

There have been multiple studies done on distributing ML models on FPGAs. Mazraeli et al. explored distributing a transformer model over multiple FPGAs and found success in optimizing for different cost functions [4]. Different ways of distributing Deep Neural Network (DNN) models over multiple FPGAs have been explored by Johnson et al. [5]. Johnson et al. made use of a reconfigurable general accelerator and did not explore inter-FPGA communication.

1.2.4 Contributions of this thesis

This thesis explores a purpose-built neural network accelerator with full register level control over the hardware design, which is in contrast with the previous work, where further layers of abstraction are present. Furthermore, the goal of this thesis is to explore the interconnection of FPGAs and to investigate an optimal way to factor the neural network, taking into account multiple parameters to maximize the utilization of the FPGAs and the servers they are connected to by also investigating heterogeneous computing.

1.3 History of neural networks

Artificial neural networks are models inspired by the layout of biological neural networks, such as the ones found in animal brains. These networks typically have a number of input and output nodes. Hidden between these interfaces, there are layers of nodes that connect to each-other in various configurations. Historically, neural networks were simpler in design and at first they were used to describe logical expressions [6]. The method called back-propagation, used for training modern neural networks, was first discovered in 1974, but popularized years later in 1986 [7]. One of the first Convolutional Neural Networks (CNNs) was presented in 1998 where it was used for handwriting recognition [8].

1.3.1 Convolutional neural networks

CNNs implement the neural network layout where the hidden layers are primarily made up of convolutional layers. Pooling layers are typically present and is one way to reduce the amount of data through down-sampling. Another way to down-sample is to use convolutional layers with a stride larger than one. Following

all the convolutional and pooling layers, there are typically one or more Fully-Connected (FC) layers that do the actual classification after all the features have been extracted [9].

VGG

VGG is a very deep convolutional neural network used for classification. The architecture of VGG is built up of blocks of a few convolutional layers followed by a max-pooling layer, reducing the feature map size. To achieve high accuracy, 16 or 19 weight layers are used. Following the convolutional layers, there are three FC layers [10].

The final FC layer contains 1000 channels used for representing 1000 classes, each indicating a different subject. VGG16 contains 138 million parameters and is one of the larger convolutional neural networks. VGG16 uses a configuration which consists of two main block types, as shown in Figure 1.1a. The network consists of three Type 1 blocks and two Type 2 blocks. For the implementation presented in this thesis, the final FC layer representing 1000 classes has been replaced with a FC layer representing ten classes to correspond to the Imagenette dataset [11]. See Figure 3.1 for an overview of the full model.

ResNet

ResNet, similarly to VGG, makes use of a very deep network architecture. One major difference in the design is the utilization of the residual path [12]. The residual path can be seen as a skip connection, bypassing a few convolutional layers before being added back to the feature map. It was found that using the residual path in the network reduced the degradation problem seen when training very deep neural networks, and as such a deeper network could be achieved. The residual path block is illustrated in Figure 1.1b.

ResNet50 contains 25 million parameters, but uses 48 convolutional layers, far deeper than VGG. With the degradation problem solved, deeper models that perform better than for example VGG can be built, while keeping the parameter count down. ResNeXt was built on top of the ResNet architecture and introduced ”cardinality”, increasing classification performance [13].

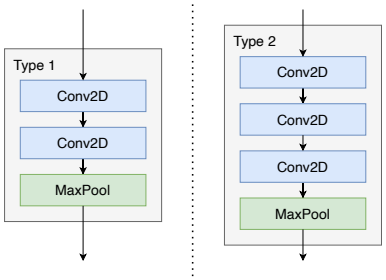
MobileNet

MobileNet is designed to be used in mobile applications and edge devices. MobileNet makes use of depthwise convolutions followed by pointwise convolutions. The depthwise convolution does the filtering and the pointwise convolution does the combining. The structure of one of these Depthwise Separable (DS) blocks is depicted in Figure 1.1c.

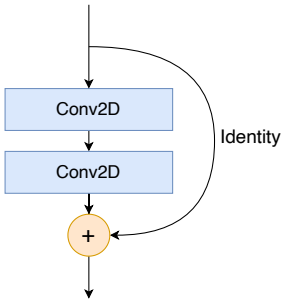
The architecture proposed as MobileNet in [14] reduces both the parameters and operations needed by about 33 times, while the accuracy loss is around one percentage point when compared to VGG16 in a general benchmark.

Several other network designs have been built on top of the MobileNet architecture. MobileNetV2 introduced inverted residual blocks to the network [15]. EfficientNet further enhanced the performance of mobile-sized networks while showing

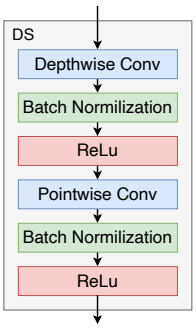
state-of-the-art accuracy [16]. While mobile neural networks tend to have substantially fewer parameters than networks like VGG and ResNet, they often have more complicated operations and architectures.



(a) VGG Block Structure.



(b) ResNet Block Structure.



(c) MobileNet Block Structure.

Figure 1.1: CNN architectures.

1.3.2 Transformer neural networks

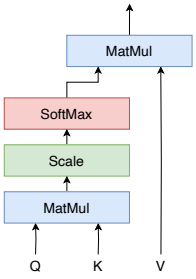
Transformer neural networks were first proposed by researchers at Google and utilize multi-head attention to relate tokens to each other. The transformer structure is showcased in Figure 1.2c. Previous to the solely attention based transformer, convolutional and recurrent neural networks were used in conjunction with encoders and decoders in order to achieve similar results.

The scaled dot-product, as seen in Figure 1.2a, is used in the multi-head attention block depicted in Figure 1.2b. The transformer architecture lowers training cost and improves performance when compared to previous machine transduction models [17]. This is because it does not need to do recurrent evaluation during the training forward pass. Thanks to the self-attention, it also obtains a global perspective field that would otherwise require deep CNNs to be implemented. Transformer networks are a precursor to the Generative Pre-trained Transformer (GPT).

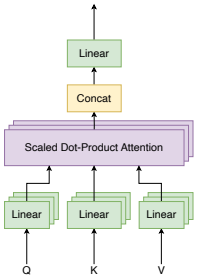
The architecture of transformer neural networks is more complex than the CNNs presented in Section 1.3.1, which is the main reason for not investigating transformer models further in this thesis.

Generative Pre-trained Transformer — GPT

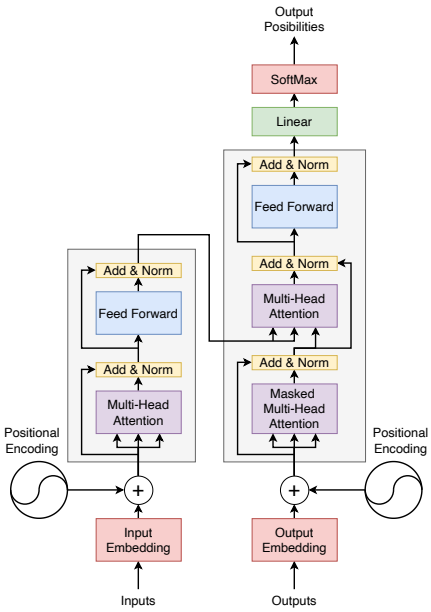
In 2018, OpenAI introduced the first widely recognized GPT. The idea behind making a Transformer model generative is a two stage training process that begins with training on unlabeled data, before being fine-tuned to solve specific tasks [18].



(a) Scaled Dot-Product Attention. Adapted from [17]



(b) Multi-head Attention. Adapted from [17]



(c) Transformer Structure. Adapted from [17]

Figure 1.2: Transformer architecture.

1.3.3 Quantization of neural networks

Neural networks are typically computed using Institute of Electrical and Electronics Engineers (IEEE) 754 floating-point 32 which is appropriate for use on Graphics Processing Units (GPUs). It is possible to quantize the models to use a smaller data-width for the parameters in order to reduce the memory footprint. Full 8-bit integer quantization is supported by most ML frameworks, which is a good starting point when quantizing a model. A representative dataset is needed to fully quantize a model to 8 bits. Furthermore, lower-resolution integer quantization is showing promising results [19], and can aid in reducing resource utilization when constructing custom hardware. Quantization can sometimes improve accuracy, but it is typical to see a slight loss in accuracy when the quantization is pushed to a substantially lower data-width than the original floating-point representation.

1.4 Cloud providers

Cloud providers such as AWS and Microsoft Azure have instances with powerful FPGA devices connected to servers. This makes it possible to rent FPGAs and upload custom hardware onto them. Moreover, the FPGAs are interconnected, allowing for inter-FPGA communication and transfer of data. The architecture can be slightly different between different providers. AWS has instances with one, two, or eight FPGAs, while Azure has instances with one, two, or four FPGAs. Other providers can have different setups and architectures of inter-FPGA communication, server hardware, and amount of FPGAs connected.

1.4.1 Amazon Web Services — AWS

AWS provides instances with one, two, and eight FPGAs. The FPGA is AMD Virtex VU9+. For the instances with two FPGAs, the FPGAs are connected together in a group, allowing for high-speed Direct Memory Access (DMA) inter-FPGA communication. Instances with eight FPGAs have two groups of four FPGAs. Within each group, high-speed DMA is available. For communication between groups, higher latency and lower throughput can be expected [20]. AWS advertises that their FPGAs are connected in a ring structure with high-speed transceivers, but this feature is unfortunately not yet supported. AWS provides two GitHub repositories with information and support [20], [21].

1.4.2 Azure

Azure FPGA instances make use of AMD Alveo U250 FPGA accelerators connected to the instances via Peripheral Component Interconnect Express (PCIe). Up to four FPGA devices are connected together in a single instance. These FPGAs have around twice as many Digital Signal Processors (DSPs), slightly more UltraRAM (URAM), and a similar amount of Block RAM (BRAM) as the ones offered by AWS. Azure refers to AMD documentation for most information about the FPGAs. The accelerators have two Quad Small Form-factor Pluggable (QSFP) ports that are not available for use.

1.4.3 VMAccel

VMAccel is a cloud provider that has more customizability when it comes to FPGA device choice compared to both AWS and Azure, providing many models. The increased flexibility allows for more precise factoring of the ML model. The storage heavy part of the network can be placed on an FPGA with a large amount of memory, while the compute intensive part can be placed on an FPGA with more computational resources, such as an FPGA with more DSP blocks and Look Up Tables (LUTs). This allows for a more tailored system architecture. VMAccel is a FPGA- and GPU-focused cloud platform and is considerably smaller than the likes of AWS and Azure.

1.5 Different inference hardware types

There are different types of hardware that can be used for neural network inference. The differences mainly lie in how general the hardware is, how energy efficient it is, and the unit cost.

1.5.1 Central Processing Units — CPUs

Central Processing Units (CPUs) are one of the more general pieces of hardware. They can do many tasks well, but do not excel at any specific task. CPUs typically have an order of magnitude fewer cores than GPUs, which leads to GPUs outperforming CPUs when running inference with many operations in parallel and for large batch sizes. Since neural networks consist of mostly arithmetic operations and almost no branching, inference makes poor use of the CPU's power.

1.5.2 Graphics Processing Units — GPUs

GPUs are less general than CPUs, and can provide performance and efficiency gains with the tasks they are designed for. GPUs are traditionally designed for computer graphics, but both training and inference of neural networks can make use of most of the same instructions that are used in computer graphics, resulting in higher performance compared to using CPUs. GPUs excel with large batch size processing during neural network inference, but lack when the batch size is small.

1.5.3 Application Specific Integrated Circuit — ASIC

Creating an Application Specific Integrated Circuit (ASIC) especially made for a single neural network implementation, would most likely result in exceptional performance at the expense of flexibility, upfront cost, and time to market. If a change is made to the neural network model, an entirely new ASIC will have to be made, with the same cost and time to market as before. Developing for ASICs is slightly different than developing for FPGAs, since a lot of focus has to be placed on area utilization, while for FPGA design focus has to be placed on unit utilization. ASICs like the ones found in Google's Tensor Processing Units (TPUs) have a very high power efficiency and show promising performance, but are

limited by performance in floating point arithmetic [22]. However, since floating point is mostly useful during training, this limitation does not influence the usage of ASICs in edge devices performing inference.

1.5.4 Field Programmable Gate Arrays — FPGAs

FPGAs are customizable hardware that are composed of different units for data storage, computation, and inputs and outputs. These units can be connected together to create custom hardware, but there is a limited amount of each unit. There are, for example, a limited number of BRAM, DSP, and URAM units that can be instantiated. This results in having to design with respect to the available number of units whereas with ASIC design, focus lies more on area utilization. However, compared to both GPUs and Neural Processing Units (NPUs), FPGAs can be made to have higher performance and lower energy utilization because they can be specialized in one type of computation. Compared to GPUs, FPGAs can provide higher performance with lower batch sizes and are especially efficient with a batch size of 1 since GPUs are made to process data in parallel with many cores. Within the restrictions of utilization and routing, FPGAs can reach performance close to that of ASICs, with a lower barrier of entry and more flexibility in case of a neural network model upgrade.

Chapter 2
Background

2.1 Neural networks

With the recent explosion in neural networks, trying to design hardware to keep up with the ever evolving complexity and scale of the networks is not an easy task. As networks become larger, the need for new techniques to handle the vast amount of layers and parameters increases with every new model. Since the architectures of neural networks change rapidly, a streamlined process to create and deploy hardware accelerators is necessary to keep up.

2.1.1 Selecting an appropriate network for multi FPGA inference

Selecting a network that is big enough where multiple FPGAs could justifiably be used while keeping the implementation minimal is not easy, every network has different requirements and requires a tailored approach when implementing.

- ResNet18 is a small network that has a repeating pattern of convolutional layers where every second or third layer has a skip connection. Implementing the skip connections in hardware means that data has to be temporarily stored in memory before being added back to the output of a convolution. The skip connections could be challenging to implement in hardware due to the low amount of available memory.
- MobileNetv2 is another small network with very few parameters, making it a little too small to be implemented as a single network across two FPGAs. It also uses strode convolution as its method to reduce feature map size, which is a harder problem to solve than simple convolution since utilizing all clock cycles efficiently can become an issue.
- VGG16 is a quite large network consisting mostly of simple convolutions and a few FC layers. VGG uses max-pooling layers to reduce the size of the feature maps, which is simpler to implement efficiently than strode convolution. It has many convolutions, each of which produce small feature maps that are perfect for transmitting between FPGAs. The FC layers can however be difficult to implement because of the large number of weights.

VGG16 is selected for its simple architecture and its large size. However, the FC layers are omitted and instead computed using a CPU.

2.1.2 Quantization

One way to introduce noise into the network is to quantize the model. It has been shown that introducing noise can improve accuracy by avoiding local minima [23]. By quantizing a model, resolution is reduced and subsequently noise is introduced. Even low bit quantization can improve classification accuracy when compared to the full-precision model [24]. One issue with the noise introduced by quantization is that it often follows a pattern and can introduce bias in the network, which can greatly be reduced by using heuristic based quantization methods [25].

Since computers operate on finite quanta of information, the smallest being the bit, quantization is well suited for computers, since it reduces the amount of data the computer has to store and operate on. It also reduces the memory pressure, both in terms of space and throughput. Quantization removes the complex floating-point arithmetic used in GPUs and makes use of integer arithmetic, which is not as complex and can be implemented with high efficiency on FPGAs.

2.2 FPGA

Since most computer architectures are addressed in bytes, the smallest quanta of information a computer can compute with at a reasonable speed is usually 8 bits; a byte. This is an inherent limit in modern computer architectures for neural networks. Many studies have shown that neural networks can operate as good or even better on fewer than 8 bits for certain tasks [26], [27]. The FPGA, in contrast to the classical computing approach, has no architectural limit when it comes to word width, as they can be programmed to operate on 2 bit integers just as well as 8 bit, if not better [28].

In comparison to ASICs, FPGAs have some limitations inherent to their design. These limitations include, but are not limited to, the memory and DSP blocks are fixed resources and have to be utilized efficiently. The internal structure is also fixed, so routing between blocks that are distant from each other has to be pipelined more to compensate for the added path delay in the critical path. These limitations have to be kept in mind when designing logic for the FPGAs to ensure efficient utilization of the FPGA resources.

2.3 Cloud infrastructure

Even if a model can be quantized down to a single bit, the larger models like some Large Language Models (LLMs) will still not fit on the FPGAs of today. Some way to interconnect FPGAs is needed to leverage the power of multiple FPGAs working together to solve an inference problem. FPGAs available in the cloud solve just that problem, they can be interconnected to solve inference together [20]. Depending on the cloud provider, there are different methods to interconnecting the FPGAs, each with their upsides and downsides. The interconnect methods available typically either use DMA to transfer data over PCIe or some other protocol like AMD Aurora [29]. The FPGAs available in the AWS cloud are used in this thesis.

Running FPGAs on-premise is great for some tasks but can become an issue as soon as the deployment needs to be scaled up. Having the ability to scale up as needed is imperative for keeping up with the rate in which neural networks evolve. The ability to scale puts pressure on the cloud providers to provide efficient ways to interconnect FPGAs to be able to leverage their potential in the cloud compute market.

2.4 Choosing an appropriate hardware design language

Developing large scale digital systems can get convoluted. For example, if a small change that has a drastic impact on the architecture has to propagate through the entire system. Traditional RTL design languages like Very High Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog do not offer a scalable enough way to parameterize a design like a neural network. Neural networks come in many shapes and sizes, but usually consist of the same core components. Being able to truly customize these components to fit any model while still being able to easily connect components is crucial.

Chisel is an RTL language, much like Verilog or VHDL, but utilizes a software programming language to do the parameterization to generate the hardware. Chisel removes boilerplate that is taken for granted to reduce development time. Chisel also retains the designer's the ability to have cycle-accurate control over the design. Unlike HLS, Chisel does not abstract away the underlying wires and registers. This gives the designer the ability to control every connection in the hardware, while still providing larger composite building blocks than Verilog or VHDL. Chisel is the language of choice in this thesis.

2.5 Selecting the right hardware to perform inference

The hardware of choice to implement neural network inference in this work are FPGAs. Unlike GPUs and CPUs which are general compute architectures, FPGAs provide a mix of flexibility and performance by giving the designer the ability to create hardware that is specific to the intended use case while maintaining a level of re-programmability that ASICs do not.

FPGAs also provide a faster time to market compared to ASICs since no new chip fabrication is needed when implementing a new design. Time to market is especially important when designing for neural networks, since the landscape is evolving rapidly. Furthermore, since FPGAs have a fixed number of resources, novel techniques need to be applied to maximize performance within the resource constraints. Because FPGAs can be programmed to suit each network individually, the latency of inference can be reduced drastically compared to conventional approaches such as GPUs.

Chapter 3
Implementation

In this section, the different parts of the neural network inference system are presented. Some design considerations like utilization constraints, network partitioning, and scheduling are introduced and illustrated along with the performance metrics and other figures that will be presented in the results section.

3.1 Deconstructing the quantized Tensorflow Lite (TFLite) model

Extracting the weights and biases from the TFLite model can be done using tools like Netron, which provides a graphical interface to view the computational graph of the network and the ability to download the weights and biases from all the layers. While this approach works, it is tedious and not programmatic, which means it quickly becomes unfeasible when implementing large networks like VGG. Each node that represents a layer in the graph also contains scaling values for each input, output, weight, and bias, which can only be copied by hand from Netron.

Looking through the TFLite models' structure using a script provided by the developers of TFLite, the structure and relations in the TFLite file can be reverse-engineered to extract all the data necessary and compile the numbers into a format better suited for the implementation.

3.2 Quantization

When quantizing a neural network model using the TFLite framework, all the weights and biases are quantized into integers. For the network not to lose too much accuracy, each layer has some floating-point scales that need to be applied for the network to keep the results in the same domain. There are scales for the weights, the bias, the input, and the output. Applying these scales can be done in a few different ways. One way is pre-scaling the weights and biases to a fixed point number using the floating-point scale provided by TFLite. Another way is to create a hardware quantizer that quantizes the 32-bit convolution result into an 8-bit integer. By pre-scaling the weights and biases, there is no need for a hardware quantizer, which saves DSP-slices at the expense of increased memory

usage. The quantizer method uses more DSP-slices, but does not require as much memory to store the parameters.

3.2.1 Pre-scaling the weights and biases

To pre-scale the weights and biases to a fixed-point number, first the input, weight, and output scales are applied to all the weights and biases. The smallest absolute weight and bias after the scaling is found for each layer. Then the resulting floating-point number is scaled by some power of two and rounded until the error between this new fixed-point representation and the floating-point representation is less than 0.1 %. With errors less than 0.1 %, the bit-width explodes in size. Furthermore, extracting the largest absolute values from the newly created fixed-point representation yielded the maximum number of bits required to store the weights and biases. The resulting scaled weights and biases can then be used as fixed-point numbers in hardware with the extracted binary-point and bit-width.

3.2.2 Quantizer

When using a quantizer, the floating-point scales have to be converted to a fixed-point number to avoid implementing costly floating-point hardware. By realizing the constraints on the combined scale, referred to as M in [30], the floating-point number can be scaled by a constant factor and turned into a fixed-point representation of the floating-point scale. The method proposed in the paper combines the input, output, bias, and weight scales into one using Equation 3.1, it then multiplies the combined scales by some power of two in order to turn it into a fixed-point number representation using Equation 3.2. These fixed-point numbers can later be used in hardware to perform fixed-point multiplication, and a fixed-point rounding algorithm can be used to remove the decimals to produce the quantized result. The method proposed is much cheaper in terms of utilization and faster in terms of logic delay than using integer-floating-point multipliers.

$$M := \frac{S_i S_w}{S_o} \tag{3.1}$$

$$M = 2^{-n} M_0 \tag{3.2}$$

Rounding

Rounding by truncation is a trick used to perform rounding quickly in hardware while only introducing a small bias. The rounding is performed on fixed point numbers by first adding the equivalent of 0.5 and then truncating the decimals. This achieves rounding that rounds towards infinity when the decimal places are $[0.5, 1)$ and rounds towards zero when $[0, 0.5)$, when the number is positive. The same is true for negative numbers, except for when the decimal is 0.5, which rounds towards zero. The small difference this method introduces can subsequently introduce a small bias in the calculations and is an undesired side effect from this method of rounding.

In classical computing using the floating-point IEEE 754 standard, the rounding method mentioned above is not used. The IEEE 754 uses the "bankers' rounding" algorithm to avoid introducing bias [31]. The method works by rounding towards the closest even number instead of rounding towards zero or infinity. Other methods, like stochastic rounding, can be used to reduce bias even further. Stochastic rounding is done using a weighted probability to round the fraction towards whichever whole number the fraction is closest to, thus removing any systematic bias [32].

3.3 Hardware Design

The VGG16 CNN is composed of several similar blocks, consisting of convolution layers and max-pooling layers. Following these blocks, there are three fully-connected blocks before a soft-max activation function. In Figure 3.1, the architecture of VGG16 is shown with the feature map sizes labeled (Batch Size x Width x Height x Depth) in parameters. The convolutional layers each have a Rectified Linear Unit (ReLU) activation function.

Due to dataset limitations, the final FC layer with dimensions of 4096 x 1000 is replaced by a FC layer with dimensions of 4096 x 10, classifying images to ten classes. This influences the accuracy and will give a higher accuracy than expected for VGG16 with 1000 classes [33].

3.3.1 Scheduling

To minimize back-pressure in the design, it is important to create a hardware schedule such that each following layer can process all the output from the previous layer, which allows for a more consistent flow of data through the graph. The schedule is set by considering the input data rate on a 24-bit bus and making sure each layer is able to process a sufficient amount of data by changing the parallelism of the hardware convolutional blocks.

As can be seen in Figure 3.1, the feature maps decrease in size while the number of feature maps increase. The max-pooling layers reduce the feature map size to one quarter of their original size. The locations in the graph after max-pooling layers make for perfect places to factor the model to spread across multiple FPGAs, while keeping the data rate needed to transfer the feature maps between the FPGAs low.

3.3.2 Partitioning the network

When partitioning the network, it is also important to consider the BRAM required to store the weights of the layers. As the number of channels grow, the storage required for the weights increases exponentially. The storage required for the weights in the final three layers (block 5) is around the same as the storage required for the rest of the convolutional blocks (blocks 1-4) of the network. This is shown in Figure 3.2.

The exponential increase in storage requirement causes the place to partition the network to shift further down the graph to account for the limited BRAM

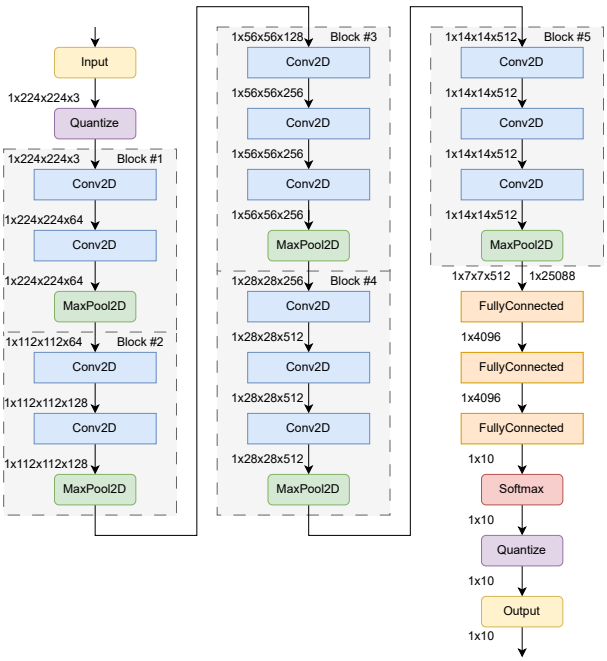


Figure 3.1: Architecture of VGG16.

resources. This can result in one of the FPGAs being more utilized than the other, especially with respect to DSP and LUT utilization. For this thesis, the network is split after block 4, making use of the small feature map after the max-pooling layer and taking the memory constraints into consideration.

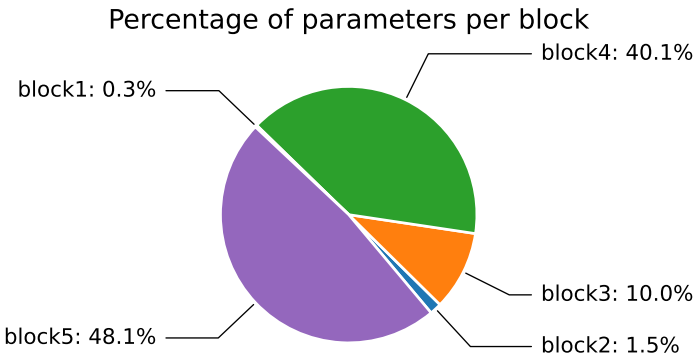


Figure 3.2: Storage required to store the convolution layer kernel values.

3.3.3 Utilization

To utilize the most of the FPGA it is important to take FPGA-specific design considerations into account when designing. The FPGA consists of different blocks that each contain some logic that can be routed together to create some circuit. These blocks mostly consist of DSPs, LUTs, and different kinds of memories.

DSP

Each FPGA has a limited number of DSP units that can be used to perform a few different operations at high speeds, especially for numbers with high bit-width. Since the convolution part of the CNN is compute-heavy, a lot of DSP units are required, and as such, careful attention has to be paid to the utilization of the DSP units. When creating the schedule, the DSP utilization has to be taken into consideration as to not over-utilize the available resources. Any left-over operations after all the DSP units have been depleted will be mapped into the LUTs which can jeopardize timing if the operations are large.

To more efficiently utilize the DSPs, it is possible to use the same DSP unit to perform multiple low-bit operations. The DSP units in the FPGA available in AWS can support two 8-bit operations per DSP. These optimization methods are not implemented in this thesis.

Memory

Memory is sparse on most FPGAs. Even high-memory FPGAs only provide a few hundred megabits of fast, low-latency memory. Because of this, storing numerous

parameters becomes a challenge. Because the weights are stored in BRAM, only the URAM is left over for buffers such as first in, first out buffers (FIFOs) and other buffers needed to keep the pipeline fed at all times [34].

Because the DNN part of VGG requires a lot of memory, in the order of hundreds of megabytes to store the parameters, the small amount of BRAM available is not enough to store these parameters. The memory required to store the parameters of the FC layers is around 8 times larger than what is needed to store the parameters for the convolutional layers, as seen in Figure 3.3. Since the URAM cannot be used as a Read Only Memory (ROM) out of the box, some logic has to be implemented to be able to load the URAMs with values as soon as the design is loaded into the FPGA.

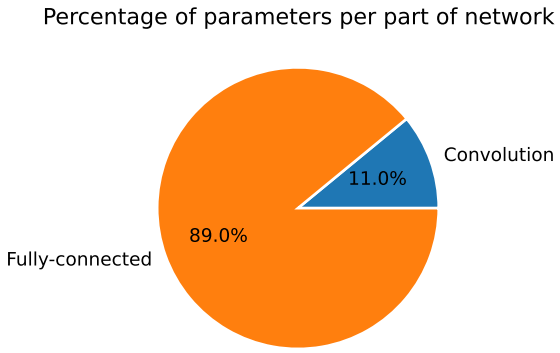


Figure 3.3: Comparison of memory required to store weights for the convolutional and fully-connected parts.

The FPGAs in AWS are equipped with 64 GB of Double Data Rate 4 (DDR4) Dynamic Random Access Memory (DRAM) which is ideal for use in the DNN because of the DNN memory access pattern being completely linear. However, the DNN is not implemented on an FPGA to limit the scope of this thesis. Instead, the DNN part is implemented in software on the host server CPU.

There are FPGAs available with High Bandwidth Memory (HBM) that have higher memory bandwidth. Such devices could aid in the design of DNNs if the DRAM does not deliver high enough performance. FPGAs with HBM are not currently available in the cloud providers discussed.

3.3.4 Routing

Because FPGAs have a fixed layout, routing can become a difficult task for the implementation tool when the design grows large. To reduce eventual routing issues, designing units that are easily scaled without introducing long control paths is essential. It is also important to make sure that each and every path is sufficiently pipelined in order to achieve high clock speed. When paths are adequately pipelined, the tool can perform re-timing, which can reduce timing issues for long logic chains inside the FPGA by moving the registers between logic. Introducing

pipelines will also reduce the route and logic delay, which in turn decrease the clock period, thus allowing for a higher clock frequency.

3.3.5 Multiple FPGA design

Due to the limited amount of BRAM resources on FPGAs, storing many parameters can be difficult. The limited amount of DSP units is also a huge constraint when designing for only one FPGA. One way to solve this is to factor the network over multiple FPGAs and, with that, distribute the need for BRAM and DSPs.

It is important to select a suitable point for partitioning the network, since the feature maps can be quite large in a neural network. In the case of VGG16, splitting after a pooling layer seems to be optimal. As mentioned in Section 3.3.2, in the case of VGG16, a good place to partition the network would be after the fourth max-pooling layer, as can be seen in Figure 3.1. This split incurs a transfer of about 100 $kB/frame$ which even at 1000 Frames Per Second (FPS) would only use 100 MB/s of throughput, a fraction of the available PCIe bandwidth.

When designing for more than four FPGAs on AWS, it is important to consider the grouping of FPGAs. When transferring data between two groups of four FPGAs, the transfer will need to be routed via the CPU in order to reach the second group due to the AWS system architecture, resulting in reduced throughput and increased latency. The logical layout of a system with eight FPGAs is shown in Figure 3.4

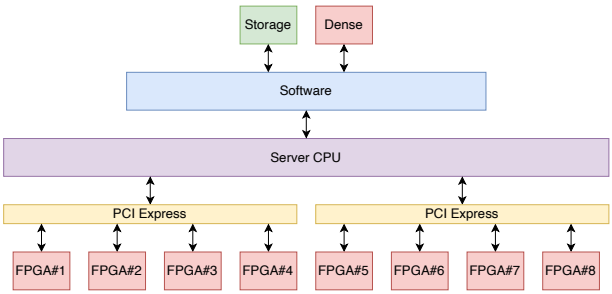


Figure 3.4: Logical overview of system with eight FPGAs.

3.4 DMA

Since the FPGA is only used for doing model specific inference, loading the images into the FPGA needs to be done on a general compute platform such as a CPU with an accompanying operating system. To transfer data fast and efficiently, DMA is used to map parts of the CPU's virtual memory into the FPGA, enabling the CPU to perform writes and reads from this memory region to interact with the FPGA. The FPGA is attached to the computer with a 16x PCIe Gen-3 interface, enabling high-speed data transfer between the FPGA and the CPU.

3.4.1 Xilinx Direct Memory Access — XDMA

AMD has an already available and supported XDMA driver with an accompanying XDMA subsystem which can translate PCIe into Advanced eXtensible Interface 4 (AXI4) transactions. The XDMA Subsystem can be configured to support many modes, such as AXI4-Stream or AXI4 memory-mapped, with varying data widths. The XDMA driver used in the thesis is loaded in interrupt mode. In order for the driver to function with the FPGA, it has to be compiled with options to match the vendor- and device-id of the FPGA.

Since the FPGA found in the F1 instances on AWS have the XDMA subsystem pre-configured as 512-bit interface and in AXI4 memory-mapped mode, an AXI4 memory-mapped to AXI4-Stream converter and a bit-width converter are implemented in this work. These provide the necessary conversions between the memory-mapped interface that the CPU uses to communicate with the hardware and the streaming interface that the accelerator expects. The width converter converts the 512-bit interface down to a 24-bit interface used by the accelerator.

3.4.2 Hardware core

The hardware core, as shown in Figure 3.5a, consists of a converter between AXI4 memory-mapped and AXI4-Stream, an input and output FIFO, and some configuration registers. The FIFO blocks are AXI4-Stream compatible and serve as temporary storage for the input and output data of the core. The DMA core can both convert from AXI4-Stream to AXI4 memory-mapped and from AXI4 memory-mapped to AXI4-Stream. A similar Intellectual Property (IP) core is already available through the AMD IP catalog but would not meet the requirements of this thesis.

The configuration and status registers are read from and written to by the CPU, as shown in Figure 3.5a. The registers, presented in Table 3.1, have different access restrictions, which are Read Only (RO), Read/Write (R/W), or Write Only (WO).

The `RX_FIFO` register is polled by the software driver, and once it is filled enough, a transfer can commence. The `TX_FIFO` register works in the same way. The occupancy reported by the `TX_FIFO` and `RX_FIFO` is measured in words.

The `FLAGS` register is read from and written to by the software driver. This register contains information and status of the system, such as, if the system has received a last signal from the user logic. This indicates that the data inside the `RX_FIFO` contains the last part of the packet. Before any more data can be received into the `RX_FIFO` the CPU has to clear the `FLAGS` register by writing a zero to it.

The `TX_LEN` register is written to by the software driver and signals the packet length of the transfer. The last word of data remains in the transmit FIFO until the `TX_LEN` register has been written to and the amount of words transmitted matches the register value. The `TX_LEN` register can be written to either before or after a transfer.

In order to issue a user reset to the neural network, the `RESET` register can be written to by the software. Once something other than zero is written to this register, a reset is sent to the connected system, before the value of the register is set back to zero, awaiting a new write.

The final register in the DMA core is the `SIGNATURE` register. This register is read from by the software driver to make sure that the DMA core has been instantiated properly by performing a signature check. The software can only be run once the signature check has completed successfully.

Table 3.1: Configuration and status registers for the DMA core.

Address	Name	Mode	Function
0x00	RX_FIFO	RO	Receive FIFO occupancy in words
0x04	TX_FIFO	RO	Transmit FIFO occupancy in words
0x08	FLAGS	R/W	Flags indicating the state of the DMA
0x0C	TX_LEN	WO	Total packet length in words
0x10	RESET	WO	Generates a user-reset signal
0x14	SIGNATURE	RO	Unique signature: 0x62696E67

3.4.3 Inter-FPGA transmitter

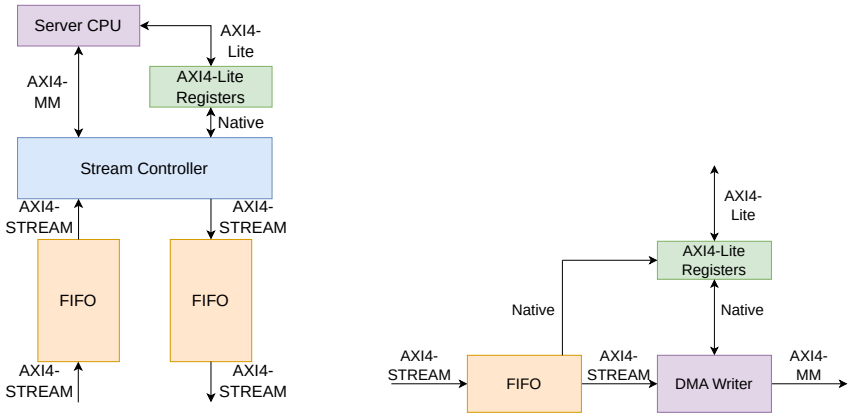
The inter-FPGA transmitter takes in data via an AXI4-Stream input and outputs AXI4 memory-mapped packed data to a specified address. Figure 3.5b shows the block diagram for this system. The input to this block is stored in an AXI4-Stream compatible FIFO. The inter-FPGA transmitter block converts the incoming stream to AXI4 memory-mapped and is made to emulate DMA transfers from the CPU. The receiving FPGA receives the memory-mapped data transfer as if it would have come from the CPU, like in Figure 3.5a.

The CPU can access AXI4-Lite configuration and status registers in the design. In Table 3.2 the configuration registers available are listed. The inter-FPGA transmitter will enter its running state once the `TX_LEN` is written to, the controller will assert bit 0 in the `STATUS` register to indicate that the transmitter is in a running state. The `STATUS` register is polled by the software driver to know when it is time to schedule another transmission.

The `PTR` register is divided into two addresses with big-endian ordering. This register is set to the physical address of the receiving FPGA. Since all FPGAs are PCIe bus masters, they have complete access to the system's physical memory space. Great care has to be taken when writing to the PCIe bus, to ensure that there is no address write violation. Writing to the wrong memory region can corrupt the system.

It is important that the inter-FPGA transmitter can perform a full burst, since no other transfer is allowed to happen on the bus in the middle of a burst. To avoid stalling the whole bus, one transfer must be done as quickly as possible. To allow this, the design waits for the FIFO to fill enough for one burst, even if it has been commanded to start transmitting. The `BURST_LEN` register sets this threshold and can be varied to accommodate different AXI4 requirements.

The `SIGNATURE` register indicates to the software driver which device it is. If the register contents do not match the expected value, the whole process should be aborted to ensure that nothing goes wrong. The `SIGNATURE` register also serves as an indicator to unaware software drivers where the device is mapped. By



(a) DMA core. **(b)** Inter-FPGA Transmitter.

Figure 3.5: DMA hardware blocks.

performing a sweep of set intervals in the bar space, the software driver can in theory locate the device. This enables the software to be generic and portable between different address configurations.

Table 3.2: Configuration and status registers for the inter-FPGA transmitter.

Address	Name	Mode	Function
0x00	PTR[63:32]	R/W	High bits of endpoint address
0x04	PTR[31:0]	R/W	Low bits of endpoint address
0x08	TX_LEN	WO	Total packet length in words
0x0C	TX_FIFO	RO	Transmit FIFO occupancy in words
0x10	BURST_LEN	R/W	Burst length of the AXI4 transaction
0x14	STATUS	RO	Transmitter status
0x18	SIGNATURE	RO	Unique signature: 0x464D4C43

3.4.4 Software driver

To send data over PCIe from the CPU to the FPGA, a DMA software driver has to be used. Since the AWS platform has the XDMA subsystem pre-configured with an AXI4 memory-mapped interface, the XDMA driver available from AMD could not directly be used to send data into the network. A software driver must be implemented that can interact with the hardware core mentioned in Section 3.4.2.

The driver has to control the registers of the hardware core and be able to read and write from the core at the same time to eliminate the need for large FIFOs in the hardware core.

The Application Programming Interface (API) for the driver provides an easy interface for transferring data from the CPU to the neural network on the FPGA.

Architecture

The driver consists of three threads that run concurrently. The main thread that calls the API for transferring data generates two buffers that must be page-aligned in memory. This is done through a provided API call. These buffers are later fed into the transfer call. Once the call has been issued, the API launches two separate threads that have a reference to the page-aligned memory region for either reading or writing. This ensures that the DMA can read and write at the same time without blocking itself. It also makes it easy for the developer to run the transfer without having to think about coherence.

When the transmit thread is in high performance mode the transmit thread transmits the whole buffer in one transfer, which exposes the risk of the transfer timing out if the hardware is not ready to receive the whole buffer, in such a scenario, the API can be compiled with a flag indicating a slower performance. Great care has to be given to ensure that the software is aware of the current state of the hardware to not overfill the FIFOs, causing a stall in the DMA transfer. This is solvable by reducing the DMA chunk size, which also impedes performance. The receiving thread can be compiled with two options that specify if it should try to read the whole buffer in one go or if it should read only the contents of the FIFO, checking how much is available at the time before reading. This carries with it the same issues as with the transmit thread.

Another API is also in place to handle the mediation of data between the FPGAs. Since the CPU assigns the memory regions to the different PCIe devices, the FPGAs have to inform each other of which memory region they got assigned. This is done through the mediation API. The API retrieves the physical address of the receiving FPGA and informs the transmitting FPGA of the destination address by updating some registers in the inter-FPGA transmitter. The API is then setup to transfer a chunk of data from one FPGA to another. When the inter-FPGA DMA transmitter has been informed of the size of this transfer, it enters transfer mode, where it waits for its FIFO to fill. The FIFO has to fill so that there is enough for one full AXI4 burst. This is controlled by the burst length register in the circuit. The burst length depends on the specific targeted PCIe interface.

If more than two FPGAs are connected together, multiple mediation threads have to be started, one for every inter-FPGA connection. For example, if three FPGAs are to be connected, one mediation thread is needed between FPGA #1 and #2, and one between FPGA #2 and #3. The software architecture is shown in Figure 3.6.

3.4.5 Inter-FPGA data transfer

To transfer the feature map of the final layer in the first FPGA, the inter-FPGA transmitter mentioned in Section 3.4.3 is used. The output from the final layer is streamed into the transmitter FIFO and transferred to the next FPGA once

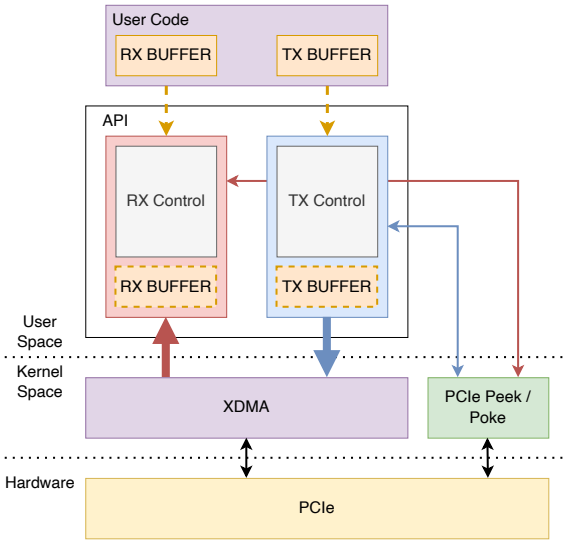


Figure 3.6: DMA Software Architecture.

enough data is in the FIFO. The second FPGA receives the data through the PCIe interface. The data is converted to AXI4-stream before reaching the subsequent layer in the network. Once the data reaches the final pooling layer, it is either further processed through the FC layers or sent back to the CPU via the hardware core described in Section 3.4.2. It is important that this transfer introduce as little latency as possible to reduce the impact of stretching a network across multiple FPGAs. Since one of the major drawbacks of interconnecting FPGAs is the overhead and added latency from whatever communication protocol is used to transfer the data.

3.5 Figures of merit

To evaluate the design and algorithms, some figures of merit need to be established. There are a few things that define the performance of a neural network, such as throughput, accuracy, latency, and resource utilization.

3.5.1 Throughput

The throughput of the network is decided by the network schedule and the clock frequency. Since the schedule is fixed to use a nominal amount of resources suitable for slicing; only the clock frequency can change the throughput. The clock frequency is chosen to be low to reduce the need for heavy pipelining, which leads to extra work. This thesis is not aiming to improve on throughput, so these optimizations are left out.

3.5.2 Accuracy

The accuracy of a neural network can realistically only be measured in end-to-end accuracy. Creating an implementation that is 100 % bit accurate with the TensorFlow model is not a usable merit, as explained in Section 3.2. The accuracy reported is end-to-end accuracy, meaning how accurately the network can perform inference. Accuracy is measured with the Imagenette validation dataset over 1000 images [11]. Top-1 accuracy is defined as when the predicted result is the same as the correct result. Top-5 accuracy is defined as when the correct result is within the top-5 of the predicted results. The dataset contains 10 classes instead of the 1000 classes VGG16 was trained on initially, increasing the classification accuracy [33].

3.5.3 Latency

Doing any kind of computing will introduce latency. The latency introduced is a combination of clock frequency, architecture, and schedule. The frequency of the network will be locked at 125 MHz, so any latency will be inherent to the architecture and schedule. The latency is measured as the time from the first byte being transferred from the CPU into the FPGA until the first byte is received by the CPU from the FPGA.

3.5.4 Resource utilization

When using FPGAs there are good definitions of resource utilization since the resources are limited in function and not in area as with ASIC design. These definitions are LUT, Flip-Flop (FF), BRAM, URAM, Look Up Table RAM (LUT-RAM), and DSP. These are represented as percentages of the maximum utilization of the used FPGA.

3.5.5 Power consumption

The FPGAs draw a specified amount of power during runtime depending on the implemented design. However, the tools used can only provide an estimated consumption figure. While the server the FPGAs are connected to also consumes power, getting an accurate representation of its power consumption is not trivial. Moreover, the server power consumption, while dependent on server design, will always be present even with other types of inference hardware such as GPUs. As such, power consumption will not be presented in this report.

Chapter 4
Results

In this section, first the final system for inference is presented. Results from the two different quantization techniques are presented and compared. The performance of the hardware created for transferring data from the host system to the FPGA is showcased along with the cross FPGA transfer. Figures of merit of the neural network are presented along with FPGA utilization.

4.1 Final inference system

The complete system for performing inference on multiple FPGAs is shown in Figure 4.1a and Figure 4.1b. The input images are stored in Random Access Memory (RAM) on the server. Through the software driver, the images are transferred to the first FPGA via DMA over PCIe. The data is received by the input processing blocks of the network and the first convolutions can begin. Once enough data has been processed by layers one through ten in FPGA one, the inter-FPGA transmitter, as seen in Figure 3.4.3, will begin transferring data to the second FPGA where the final three layers of convolution take place, before the data is sent back to the server CPU for further processing.

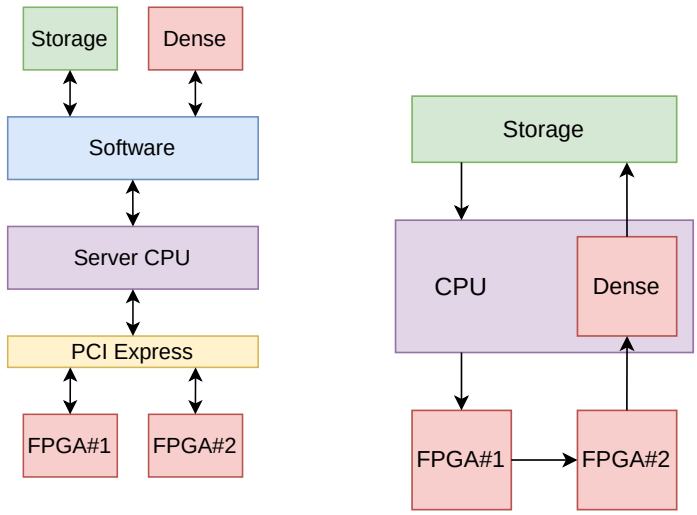
4.2 Quantization

Two different quantization schemes have been examined. The first scheme pre-scales all the weights and biases with the M factor mentioned in Section 3.2.2. The second scheme utilizes a quantizer which takes in the 32 bit values from the convolution result and quantizes the result using the M factor in hardware instead of software.

4.2.1 Pre-scaling the weights and biases

Scaling the weights and biases to fixed point numbers with a bit-width that achieves an error less than 0.1 % results in poor accuracy in the implemented design compared to using the quantizer method as described in Section 3.2.2. Moreover, the pre-scaled weights and biases have an average bit-width of 12 bits, resulting in an approximately 50 % increase in memory utilization for storing the

weights and biases when compared to storing 8-bit weights and 32-bit biases. Due to the limited memory resources, this method is not adequate.



(a) Block layout. **(b)** Data flow.

Figure 4.1: Inference System.

4.2.2 Quantizer

The quantizer method used in this thesis results in both fewer and smaller quantization errors when compared to the pre-scaling method. Since this method requires only storing 8-bit weights and 32-bit biases, it reduces the amount of memory needed in the design substantially. One drawback of this method is that the quantizer block requires at least one DSP-slice, reducing the DSP resources available to the rest of the design. 8-bit multiplication can be mapped to LUTs, making up for the loss in DSP resources when using the quantizer block. A comparison of the errors caused by the quantization method compared to the floating-point version is shown in Figure 4.2. The error rate is shown in a logarithmic scale. As can be seen, the quantizer method performs orders of magnitude better than the pre-scaled method.

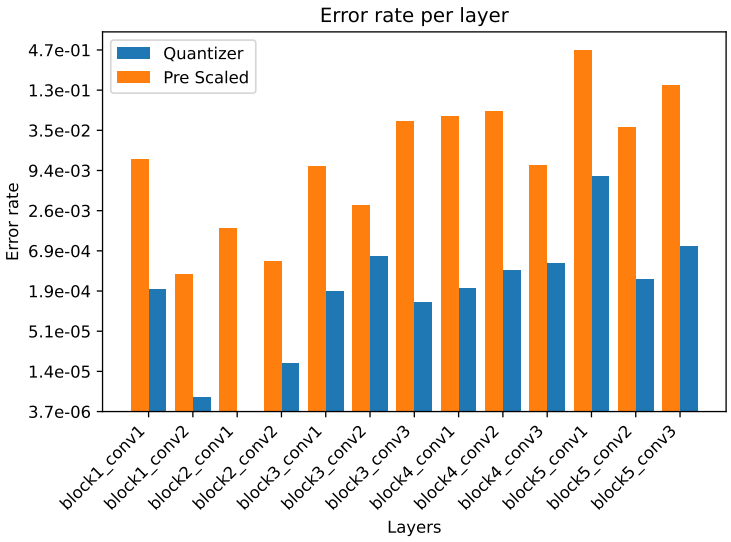


Figure 4.2: Overview of quantization accuracy per layer.

4.3 DMA

Transferring data from the system memory to the FPGA devices connected to the system is done over Direct Memory Access. The performance achieved is presented in Table 4.1.

Table 4.1: Overview of DMA performance.

Performance Target	CPU to FPGA	FPGA to FPGA
Normal	0.6 GB/s	265 MB/s
High Performance	6.4 GB/s	428 MB/s

4.3.1 Host to FPGA

Transferring data from the host system to the FPGA reaches speeds of 6.4 GB/s at most. The transfer speed depends on the chunk size used and which performance target is selected. A chunk size of 8 MiB is seen to be the optimal for high speeds, as shown in Table 4.2.

4.3.2 Inter-FPGA

The inter-FPGA throughput suffers due to limitations in the FIFO sizes of the receiving DMA core and limitations in the shell logic. However, the inter-FPGA performance is more than adequate to transfer the feature maps of the neural network as long as a few blocks of convolution and pooling have been performed

Table 4.2: Comparison between chunk size and transfer speed.

Chunk Size	Normal	High Performance
32 KiB	0.16 GB/s	0.16 GB/s
64 KiB	0.61 GB/s	1.00 GB/s
256 KiB	0.59 GB/s	2.19 GB/s
512 KiB	0.61 GB/s	3.20 GB/s
1 MiB	0.58 GB/s	4.57 GB/s
4 MiB	0.56 GB/s	5.96 GB/s
8 MiB	0.60 GB/s	6.38 GB/s
16 MiB	0.58 GB/s	6.29 GB/s
32 MiB	0.57 GB/s	6.25 GB/s
64 MiB	0.59 GB/s	5.87 GB/s
128 MiB	0.58 GB/s	5.86 GB/s

to reduce the feature map size. If the network is split after the fourth block as described in Section 3.3.2, 100 KB per FPS would be the transfer amount. At for example 100 FPS, the transfer speed requirement would be 10 MB/s. The inter-FPGA performance reaches 428 MB/s, 40 times the requirement. As seen with the inter-FPGA transfer in Table 4.3, performance depends on the chunk size and the performance target. Due to the limitations, the chunk size is limited to 64 KiB.

Table 4.3: Comparison between chunk size and transfer speed for the cross FPGA transfer.

Chunk Size	Normal	High Performance
4 KiB	25 MB/s	25 MB/s
8 KiB	54 MB/s	50 MB/s
16 KiB	94 MB/s	102 MB/s
32 KiB	202 MB/s	198 MB/s
64 KiB	265 MB/s	428 MB/s
128 KiB	N/A	N/A

4.4 Performance metrics

The performance achieved by factoring the network over multiple FPGAs is more than adequate. The introduced latency as a result of transmission of data between FPGAs is minimal and insignificant in the scope of the latency of the whole network. The factoring allows for larger networks to be run on FPGA devices that could not previously due to a limitation in FPGA resources.

4.4.1 Throughput

The throughput achieved by the split system is low enough to not be limited by the inter-FPGA bandwidth constraint. It is also low enough to not be limited by the input / output data rate. The throughput achieved by the implementation is in accordance with the selected schedule and the fixed clock frequency.

4.4.2 Accuracy

The top-1 end to end accuracy of VGG16 as implemented is 87.7 % whereas the top-5 accuracy is 98.9 %. Compared to the TensorFlow model, this is a slight decrease in top-1 accuracy, caused mainly by quantization. On the other hand, the top-5 accuracy sees a slight increase.

Table 4.4: VGG16 accuracy compared on FPGA and CPU.

VGG16	FPGA INT8	CPU FP32
Top-1	87.7 %	88.5 %
Top-5	98.9 %	98.5 %

4.4.3 Latency

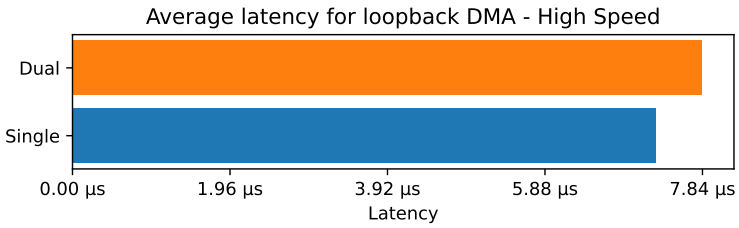
The latency introduced by the cross FPGA transfer is minimal. As seen in Figure 4.3, the latency for the inter-FPGA loop back is slightly higher than the single FPGA loop back. The latency using the high speed setup is substantially lower than when using the normal polling setup, as can be seen when comparing the results in Figure 4.3a and Figure 4.3b. It is clear from this data that the overhead of connecting multiple FPGAs is insignificant, both when comparing to the overhead of data transfer to a single FPGA and especially when considering the latency of the network itself.

4.4.4 Utilization

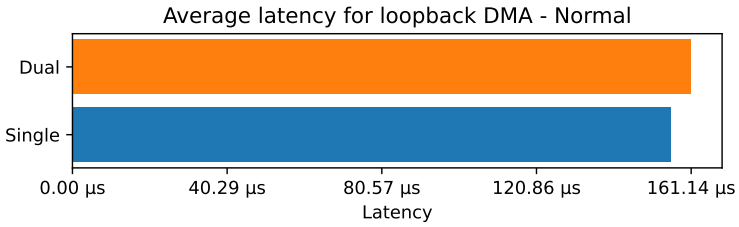
The utilization of the FPGA resources is not evenly distributed as illustrated in Figure 4.4. The foremost resource constraint is both the Block RAM and Ultra RAM. The first FPGA has a lot higher utilization as shown in Figure 4.4a when compared to the second FPGA shown in Figure 4.4b.

Block RAM

Due to the large number of 8-bit weights, the BRAM usage is the main limiting factor of how many layers fit on a single FPGA. Since AWS has split up the floor plan of the FPGA into two sections, one for the user logic and one for the shell, only 78 % of the BRAM is available for use in custom logic.



(a) Average latency for loop back DMA — High speed.



(b) Average latency for loop back DMA — Normal.

Figure 4.3: Latency comparison in different configurations and modes.

Ultra RAM

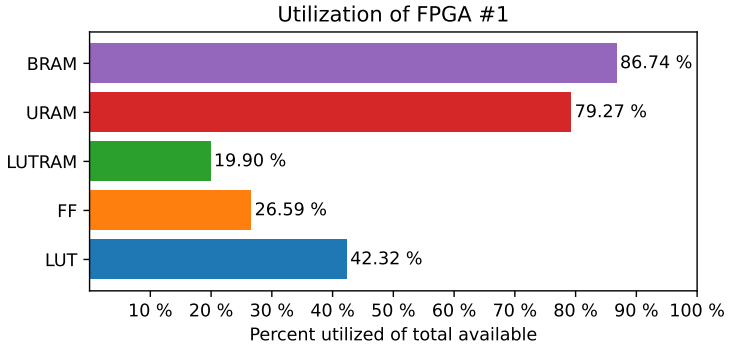
The Ultra RAM resources are rapidly exceeded and are primarily used for temporary storage. This is due to this model's architecture and the IP blocks that were available. With some optimizing, this bottleneck can be almost completely eliminated.

LUT RAM

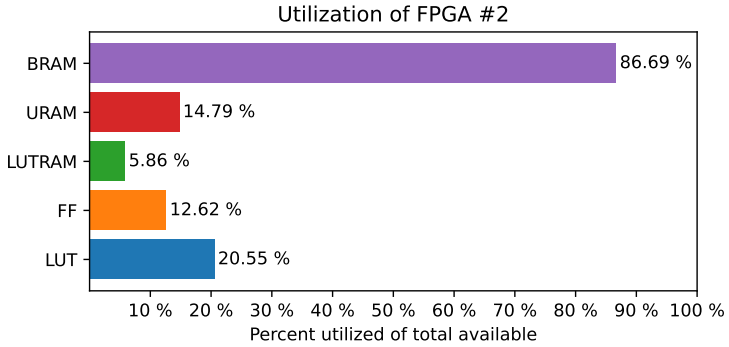
Some weights can be mapped to LUT RAM when the BRAM utilization exceeds implementable margins.

DSP

The number of DSP slices used in the design depends on the desired throughput and the selected schedule, and with it, the number of calculations done in parallel. Because of the implementation details, the number of operations that can be run in parallel depends on the schedule of the layers and has to be an integer multiple of the base schedule implemented in this design. This metric is not shown in Figure 4.4.



(a) Resource utilization of FPGA #1.



(b) Resource utilization of FPGA #2.

Figure 4.4: Resource utilization of the two FPGAs.

Chapter 5
Discussion

5.1 Conclusions

This work has shown that interconnecting multiple FPGAs can allow for larger networks to be run on FPGAs that would previously not fit on a single FPGA. The solution implemented shows great promise, with low overhead for inter-FPGA communication. There are a few points of improvement that will be discussed further.

5.1.1 How can hardware implemented on FPGAs in the cloud be debugged?

Debugging FPGAs in the cloud can be done similarly to how it would be done locally. Using an Integrated Logic Analyzer placed in the design, which can later be attached to using a JTAG interface. The JTAG interface of the FPGA in the cloud is virtual, and requires some special configuration to connect to. However, once everything is set up, debugging in the cloud is set up is no more difficult than doing it locally.

5.1.2 What protocols should be implemented to reduce latency and keep high throughput while keeping flexibility in the partitioning of the models?

In this thesis, the AXI4 memory-mapped and AXI4-Stream protocols have been explored together with PCIe. Using PCIe to communicate between systems works well, but leaves room for improvement. Using AMD's Aurora protocol for inter-FPGA communication could possibly reduce latency and improve performance compared to PCIe as mentioned in Section 2.3.

5.1.3 Latency calculation and optimization

Distributing the neural network over multiple FPGA introduces a small, insignificant latency to the total inference time. Reducing the inference time can be done a few different ways. It is important to focus attention where the largest portion

of latency originates from. As such, creating a more latency optimized hardware pipeline could drastically reduce first batch inference latency.

5.2 Points of improvement

- While the inter-FPGA communication implemented in this work has a high enough throughput to support transferring the feature maps between FPGAs, with larger feature maps and a shorter inference time, it is possible that future models could saturate the current link between the FPGAs.
- The hardware DMA core is designed with the available FPGA infrastructure in mind. With future or different DMA implementations, the DMA core might have to altered or completely removed. With a different DMA setup, for example, re-configuring the XDMA subsystem to AXI4-Stream the DMA core could be removed almost entirely.
- Making the inter-FPGA transmitter module smart by being able to read the receiving DMA core's FIFO occupancy. This could enable higher throughput by removing the chunk size limitation discussed in Section 4.3.2 from the software.

5.2.1 Inter-FPGA communication

The solution implemented in this work is created with the infrastructure surrounding the AWS cloud FPGAs in mind. However, there are other implementation that also allow for high speed and low latency transfer between FPGAs, such as AMD's Aurora protocol mentioned in Section 5.1.2. Such implementations are dependent on the cloud providers and AMD allowing access to the multi-gigabit transceivers available in the FPGAs either through some type of shell or through directly exposing these systems to the user. Implementing an inter-FPGA communication system using this approach has the benefit of being lower overhead while not using up the limited PCIe bandwidth available.

One of the main issues with the current method is that data seems to drop when back-pressure is generated by the receiving FIFO. Since the design uses quite small FIFOs to limit the amount of RAMs used, they get filled up quickly, especially when trying to send a lot of data through at once. For some reason, the PCI Master (PCIM) interface in the shell does not de-assert its ready signal when the receiving FIFO on the other FPGA is full.

5.2.2 DMA core

With further support from the leading cloud providers, the DMA core implemented in this work could be slimmed down. Since the XDMA Subsystem is locked to AXI4 memory-mapped mode by default in AWS F1, the implemented DMA core is required. However, if the XDMA subsystem could be configured to support AXI4-Stream, the entire process could be significantly simplified.

5.2.3 Software

The DMA performance could possibly be improved if more time and energy were to be spent optimizing the software driver. Moreover, the software driver could be streamlined if a streaming DMA was made available, since most of the software is in place to control the DMA Core.

5.3 Neural network hardware design

The design and architecture of the hardware implementation for the neural network could be improved to increase throughput, reduce inference latency, and reduce resource utilization. By employing some strategies mentioned in Section 3.1 like DSP sharing and further model quantization, the throughput, inference latency, and utilization can be improved.

5.3.1 Utilization

One of the main limiting factors when designing accelerators for neural networks on FPGAs is, where to store the weights. Networks with many parameters such as VGG quickly make use of all the available BRAM resources. To help with this, quantization to 4-bit weights can reduce the memory requirement by half for the same number of parameters. Further research into finding efficient ways to initialize values into URAM could also be of great interest, since it is much more abundant.

5.3.2 Throughput

There are a few ways to increase the throughput of the network. One way is to increase the clock frequency of the FPGA. With increased clock frequency comes additional work to make sure no paths fail timing. Furthermore, going down to 4-bit quantization can allow for more operations done in parallel by way of DSP re-utilization and mapping more multiplications to LUTs. Looking into fully unrolling loops by employing a faster schedule could also be of interest in certain scenarios. Larger FPGAs can allow for more operation level parallelism, resulting in higher throughput.

5.3.3 Latency

One of the main benefits of running neural network inference of batch size one on FPGAs is that the latency can be reduced compared to GPUs. When splitting neural networks over multiple FPGAs, the latency increases slightly. However, the latency of inference computation is larger than that imposed by the data transfer between FPGAs. This makes multiple FPGA neural network inference a worthwhile endeavor. With even larger models that require more FPGAs, the inter-FPGA communication might have a larger impact, especially if the data has to be routed through the CPU to reach the second grouping as explained in Section 3.3.5.

With future support from cloud and FPGA vendors, the data transfer between FPGAs could be streamlined with tighter integration with the provided design suite. If this leads to lower latency is difficult to predict, as it would depend on the specific implementation the vendors provide.

5.4 Further research

In this thesis, simple CNNs with many parameters have been explored. Newer neural networks for image classification make use of more complex architectures with, for example, skip connections. Implementing such operations efficiently on FPGAs could be an interesting research topic. Various types of transformer networks are quickly entering the market. Investigating the possibility of accelerating such networks on FPGAs could prove worthwhile due to the growing need for energy efficient computing in the Artificial Intelligence (AI) race.

5.4.1 Neural network design for efficient hardware implementation

Designing neural networks with hardware in mind could possibly aid in further improvement. For example, using rounding algorithms that are cheap to implement in hardware. One could look into doing stochastic rounding instead of banker's rounding. Stochastic rounding could, for example, be implemented in hardware using a Linear Feedback Shift Register. This would most likely require retraining the model with the new rounding method.

5.4.2 Heterogeneous computing

In order to maximize the computational efficiency, it could be interesting to look into distributing the network over various different types of acceleration media. Some operations could be most efficiently done on FPGAs, while others could be mapped to GPUs or maybe even use the server CPU of the FPGA instance for some operations. In this work, the FC layers are placed on the CPU to finalize the classification.

References

[1] M. C. Madineni, M. Vega and X. Yang, "Parameterizable design on convolutional neural networks using chisel hardware construction language", *Micromachines*, vol. 14, no. 3, 2023, ISSN: 2072-666X. DOI: 10.3390/mi14030531. [Online]. Available: https://www.mdpi.com/2072-666X/14/3/531.

[2] C. Salazar-García, J. González-Gómez, K. Alfaro-Badilla *et al.*, "Plasticnet: A low latency flexible network architecture for interconnected multi-fpga systems", in *2020 IEEE 3rd Conference on PhD Research in Microelectronics and Electronics in Latin America (PRIME-LA)*, 2020, pp. 1–4. DOI: 10.1109/PRIME-LA47693.2020.9062749.

[3] C. Salazar-García, R. García-Ramírez, R. Rímolo-Donadío, C. Strydis and A. Chacón-Rodríguez, "Plasticnet+: Extending multi-fpga interconnect architecture via gigabit transceivers", in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5. DOI: 10.1109/ISCAS51556.2021.9401058.

[4] M. Mazraeli, Y. Gao and P. Chow, "Partitioning large-scale, multi-fpga applications for the data center", in *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, 2023, pp. 253–258. DOI: 10.1109/FPL60245.2023.00043.

[5] H. Johnson, T. Fang, A. Perez-Vicente and J. Saniie, "Reconfigurable distributed fpga cluster design for deep learning accelerators", in *2023 IEEE International Conference on Electro Information Technology (eIT)*, 2023, pp. 1–5. DOI: 10.1109/eIT57321.2023.10187228.

[6] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity", *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, Dec. 1943, ISSN: 1522-9602. DOI: 10.1007/BF02478259. [Online]. Available: https://doi.org/10.1007/BF02478259.

[7] B. Macukow, "Neural networks – state of art, brief history, basic models and architecture", in *Computer Information Systems and Industrial Management*, K. Saeed and W. Homenda, Eds., Cham: Springer International Publishing, 2016, pp. 3–14, ISBN: 978-3-319-45378-1.

[8] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[9] L. Stankovic and D. Mandic, *Convolutional neural networks demystified: A matched filtering perspective based tutorial*, 2022. arXiv: 2108.11663 [cs.IT].

[10] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2015. arXiv: 1409.1556 [cs.CV].

[11] J. Howard, *Imagenette: A smaller subset of 10 easily classified classes from imagenet*, Mar. 2019. [Online]. Available: https://github.com/fastai/imagenette.

[12] K. He, X. Zhang, S. Ren and J. Sun, *Deep residual learning for image recognition*, 2015. arXiv: 1512.03385 [cs.CV].

[13] S. Xie, R. Girshick, P. Dollár, Z. Tu and K. He, *Aggregated residual transformations for deep neural networks*, 2017. arXiv: 1611.05431 [cs.CV].

[14] A. G. Howard, M. Zhu, B. Chen *et al.*, *Mobilenets: Efficient convolutional neural networks for mobile vision applications*, 2017. arXiv: 1704.04861 [cs.CV].

[15] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L.-C. Chen, *Mobilenetv2: Inverted residuals and linear bottlenecks*, 2019. arXiv: 1801.04381 [cs.CV].

[16] M. Tan and Q. V. Le, *Efficientnet: Rethinking model scaling for convolutional neural networks*, 2020. arXiv: 1905.11946 [cs.LG].

[17] A. Vaswani, N. Shazeer, N. Parmar *et al.*, *Attention is all you need*, 2023. arXiv: 1706.03762 [cs.CL].

[18] A. Radford, K. Narasimhan, T. Salimans and I. Sutskever. "Improving language understanding by generative pre-training". (2018), [Online]. Available: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf (visited on 15/04/2024).

[19] A. Zhou, A. Yao, Y. Guo, L. Xu and Y. Chen, *Incremental network quantization: Towards lossless cnns with low-precision weights*, 2017. arXiv: 1702.03044 [cs.CV].

[20] A. Labs, *Aws fpga app notes*, https://github.com/awslabs/aws-fpga-app-notes, 2023, (commit: 21a4e9f).

[21] AWS, *Aws fpga*, https://github.com/aws/aws-fpga, 2023, (commit: 863d963).

[22] N. P. Jouppi, C. Young, N. Patil *et al.*, *In-datacenter performance analysis of a tensor processing unit*, 2017. arXiv: 1704.04760 [cs.AR].

[23] M. Plappert, R. Houthooft, P. Dhariwal *et al.*, *Parameter space noise for exploration*, 2018. arXiv: 1706.01905 [cs.LG].

[24] C. Leng, H. Li, S. Zhu and R. Jin, *Extremely low bit neural network: Squeeze the last bit out with admm*, 2017. arXiv: 1707.09870 [cs.CV].

[25] A. Finkelstein, U. Almog and M. Grobman, *Fighting quantization bias with bias*, 2019. arXiv: 1906.03193 [cs.LG].

[26] B. D. Rouhani, R. Zhao, A. More *et al.*, *Microscaling data formats for deep learning*, 2023. arXiv: 2310.10537 [cs.LG].

[27] Z. Liu, B. Oguz, A. Pappu, Y. Shi and R. Krishnamoorthi, *Binary and ternary natural language generation*, 2023. arXiv: 2306.01841 [cs.CL].

[28] C. Wu, M. Wang, X. Chu, K. Wang and L. He, *Low precision floating-point arithmetic for high performance fpga-based cnn acceleration*, 2020. arXiv: 2003.03852 [eess.SP].

[29] AMD. "Aurora 8b/10b". (), [Online]. Available: https://www.xilinx.com/products/intellectual-property/aurora8b10b.html (visited on 31/05/2024).

[30] B. Jacob, S. Kligys, B. Chen *et al.*, *Quantization and training of neural networks for efficient integer-arithmetic-only inference*, 2017. arXiv: 1712.05877 [cs.LG].

[31] "Ieee standard for floating-point arithmetic", IEEE Computer Society, New York, NY, USA, Standard IEEE Std 754-2008, Aug. 2008. [Online]. Available: https://web.archive.org/web/20160806053349/http://www.csee.umbc.edu/~tsimo1/CMSC455/IEEE-754-2008.pdf.

[32] E.-M. E. Arar, D. Sohier, P. de Oliveira Castro and E. Petit, *Stochastic rounding variance and probabilistic bounds: A new approach*, 2023. arXiv: 2207.10321 [math.NA].

[33] Z. Akata, F. Perronnin, Z. Harchaoui and C. Schmid, "Good practice in large-scale learning for image classification.", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 3, pp. 507–520, 2014, ISSN: 01628828.

[34] AMD. “Amd ultrascale+ fpgas product selection guide”. (), [Online]. Available: https://docs.amd.com/v/u/en-US/ultrascale-plus-fpga-product-selection-guide (visited on 13/05/2024).

Printed by Tryckeriet i E-huset, Lund 2024

LUND
UNIVERSITY

Series of Master´s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2024-992
http://www.eit.lth.se