THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE

FAST AND RESOURCE EFFICIENT IMAGE SEGMENTATION ON FIELD-PROGRAMMABLE GATE ARRAYS

JOSEPH KANG
SPRING 2025

A thesis
submitted in partial fulfillment
of the requirements
for baccalaureate degrees
in Engineering Science and
Computer Science with honors
in Engineering Science

Reviewed and approved* by the following:

Mohamed Almekkawy
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Lucas Passmore
Teaching Professor of Engineering Science and Mechanics
Honors Advisor

*Signatures are on file in the Schreyer Honors College.

Abstract

In recent years, deep learning has achieved state-of-the-art results across a range of domains, revolutionizing several fields. However, deploying neural networks in real-time systems with stringent latency and resource constraints remains a significant challenge owing to the high computational demands of conventional hardware such as CPUs and GPUs. Field-Programmable Gate Arrays (FPGAs) offer a promising alternative because their reconfigurability and high degree of parallelism can deliver extremely low latency and reduced power consumption.

Image segmentation, especially in applications such as autonomous vehicles and medical imaging, benefits from FPGA-based neural network inference because of its time-sensitive nature and the frequent use of power-constrained devices. In this study, we demonstrate the feasibility of FPGA deployment for image segmentation by converting a Keras-based UNet model into a Hardware Description Language (HDL) using high-level synthesis (HLS).

We trained a compact version of the UNet architecture on three relevant datasets: the Oxford-IIIT Pet dataset, Cardiac Acquisitions for Multi-structure Ultrasound Segmentation (CAMUS) dataset, and Multimodal Brain Tumor Segmentation (BraTS) dataset. Despite the model's reduced size, it maintained high segmentation performance, as measured by the Dice coefficient, even after conversion to fixed-point arithmetic for HLS. The resulting model achieved Dice scores of 0.8396 (BraTS), 0.7615, and 0.7352 for BraTS (Oxford Pets), and 0.7352 (CAMUS).

The model was synthesized into HDL (Verilog) using the *hls4ml* Python library targeting the *xcu250-figd2104-2L-e* FPGA. Compared to execution on Google Colab's CPU, T4 GPU, and TPU v2-8, the FPGA implementation achieved dramatically reduced inference latency. For the BraTS and Oxford Pets datasets, inference took less than one millisecond—over 100 times faster than on traditional hardware. For CAMUS, the latency was approximately 8 ms, which was more than 13 times faster than the baseline results. While LUT, FF, URAM, and BRAM_18K usage remained within acceptable limits, the high number of multiplications used in the network caused DSP usage to exceed the available capacity, suggesting the need for further optimization for this specific FPGA device.

Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to thank my Thesis Supervisor, Dr. Mohamed Almekkawy, for his support throughout the creation of this thesis. This specific topic was undertaken as a direct result of his suggestion, and I am truly grateful for the opportunity to explore it. Researching UNet neural network acceleration on FPGAs allowed me to explore the intersection of many different cutting edge fields, including image segmentation, deep learning, and hardware specialization. Dr. Almekkawy has given me great guidance throughout this entire process and I am more than happy to have written my thesis with him.

I would also like to thank Dr. Almekkawy's Ph.D. student Ahmed Al-Qurri for his help. His knowledge of medical imaging data sets and UNet architectures was very helpful in researching the specific datasets and networks I used. Finally, I would like to thank my honors advisor, Dr. Lucas Passmore, who has supported me over four years as an undergraduate. Dr. Passmore has been very helpful to me in both scheduling my classes and planning out the researching and writing of this thesis.

Chapter 1 Introduction

The following section introduces concepts necessary to understand this thesis and its potential impact in both the fields of image segmentation and hardware acceleration. Section 1.1 introduces the current challenges of low-latency/resource constrained deep learning applications and why Field-Programmable Gate Arrays (FPGAs) could offer a potential solution. Section 1.2 introduces techniques to reduce the size of complex neural networks without significantly impacting the accuracy of such models. Section 1.3 details the workflow of hls4ml, an open-source Python library for turning deep learning python code into a Hardware Description Language (HDL) which can be used on an FPGA. Section 1.4 introduces image segmentation and the UNet–the most widely used neural network architecture in the field. Finally, section 1.5 introduces various metrics that are used in evaluating the accuracy of a model on image segmentation tasks.

1.1 Deep Learning and FPGAs

Deep learning is a subfield of machine learning that uses artificial neural networks containing many layers, to model complex patterns in data [25]. Inspired by the structure and function of the human brain, deep learning systems can automatically learn hierarchical representations, making them particularly effective for tasks such as image recognition, natural language processing, and time series prediction [15]. Unlike traditional algorithms that require manual feature engineering, deep learning models can extract relevant features directly from raw data, allowing them to outperform classical approaches in many domains [39].

Deep learning and neural networks have completely revolutionized many fields due to their ability to perform exceptionally well on complex tasks where traditional techniques fall short [25]. Despite the overwhelming success, deploying deep neural networks in

situations with tight resource and latency constraints remains a significant challenge [11]. Many real-time applications such as autonomous systems, edge computing, and high-energy physics experiments require not only high accuracy but also low power consumption and extremely low latency [29]. While traditional hardware such as GPUs can be incredibly effective, their fixed architecture limits flexibility in optimizing for latency or power efficiency [11].

Field-Programmable Gate Arrays (FPGAs) are a type of integrated circuit that consists of an array of programmable logic blocks with a connecting grid, as seen in Figure 1.1.

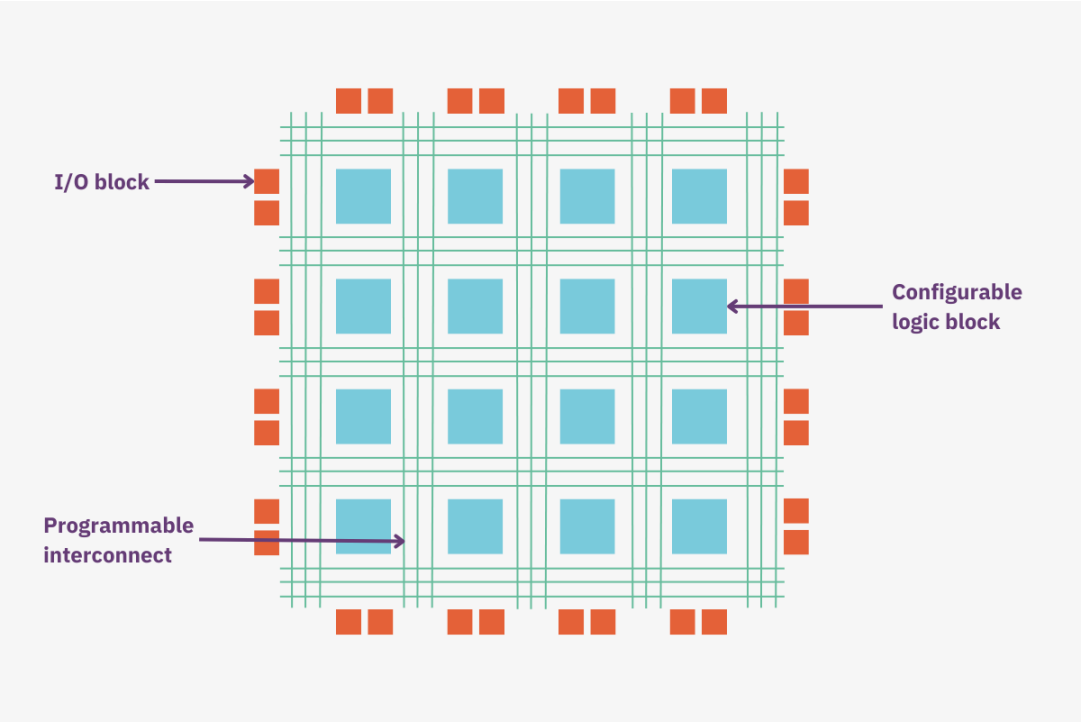

Figure 1.1. Basic Structure of an FPGA: Highlighting Configurable Logic Blocks (CLBs), Programmable Interconnects, and I/O Blocks. Source: [26].

The interconnects between logic blocks can be repeatedly programmed after manufacturing using Hardware Description Languages (HDLs) [11]. As opposed to other programming languages used to build software, HDLs are used to design and simulate the behavior of hardware. Their flexibility in specifying hardware for different tasks often allow for independent operations to run fully in parallel, achieving trillions of operations per second at a relatively low power cost with respect to CPUs and GPUs [11]. Due to their flexibility, low latency, and energy efficiency, FPGAs have been utilized for a

variety of applications that require high performance with low power consumption [35]. More recently, there has been increased interest in using FPGAs for deep learning tasks. Of these include computer vision [14], low-powered devices [13], medical imaging [30], and particle physics [11].

Despite the potential for FPGAs, there are a variety of challenges with using them for complex tasks such as deep learning. Perhaps the most prominent of these challenges are the limited on-chip resources and lack of scalability [46]. A significant challenge in creating an optimal FPGA implementation is to balance FPGA resource usage while achieving the latency, throughput, and accuracy goals of the target algorithm [40]. FPGAs consist of a finite number of logic blocks, Digital Signal Processing slices, and on-chip memory [29]. In more detail:

- **BRAM (Block RAMs)**: Blocks of RAM that provide distributed, low-latency memory resources throughout the FPGA fabric. They are commonly used to store intermediate feature maps, weights, and activation values during neural network inference [44].
- **DSP (Digital Signal Processing)**: Dedicated Digital Signal Processing slices optimized for fast arithmetic operations such as multiplication and accumulation, making them essential for implementing deep learning layers like convolutions and dense layers [45].
- **FF (Flip-Flops)**: Basic sequential elements used to store binary state information and facilitate data pipelining and synchronization within digital designs [9].
- **LUT (Look-Up Tables)**: Core building blocks of FPGA logic that implement arbitrary Boolean functions, enabling reconfigurable combinational logic design [9].
- **URAM (UltraRAM)**: High-capacity memory blocks available on some FPGAs, offering significantly more storage than BRAM and suited for memory-intensive applications [48].

Large or complex neural networks may exceed the available resources of most, if not all FPGA chips [46]. Furthermore, compared to traditional hardware such as GPUs, scaling involves significant engineering effort to design and optimize logic, as the hardware must be manually tailored to fit a specific task [50]. FPGAs are also comparatively more expensive for equivalent performance of more general purpose hardware, making them currently unsuitable for extremely computationally expensive tasks [40].

1.2 Compression Techniques

Due to the aforementioned constraints, implementing complex neural networks on FPGAs almost always involves compression techniques–ways to reduce the size or complexity of a network without significantly reducing the accuracy [5]. There are several ways to approach this.

First, one can simply reduce the number of parameters in a given network. Oftentimes this involves either reducing the size of certain layers and/or the number of layers of a network. Thorough experimentation is needed to choose a less complex network without significantly reducing accuracy [5]. Tools such as Automated Machine Learning (AutoML) have begun to allow for much easier selection of models, but this technology is still in its infancy [17]. A related, but more structured approach is pruning–the act of systematically removing neurons that are less important for making accurate predictions [5]. After initial training, weights and connections deemed less important to the network's predictions are removed. The network is then usually retrained to cover any potential accuracy losses [5]. While challenges exist, most modern deep learning libraries provide support for pruning. If done correctly, the technique can be implemented without significant accuracy loss [36]. An illustration of pruning can be seen in Figure 1.2.

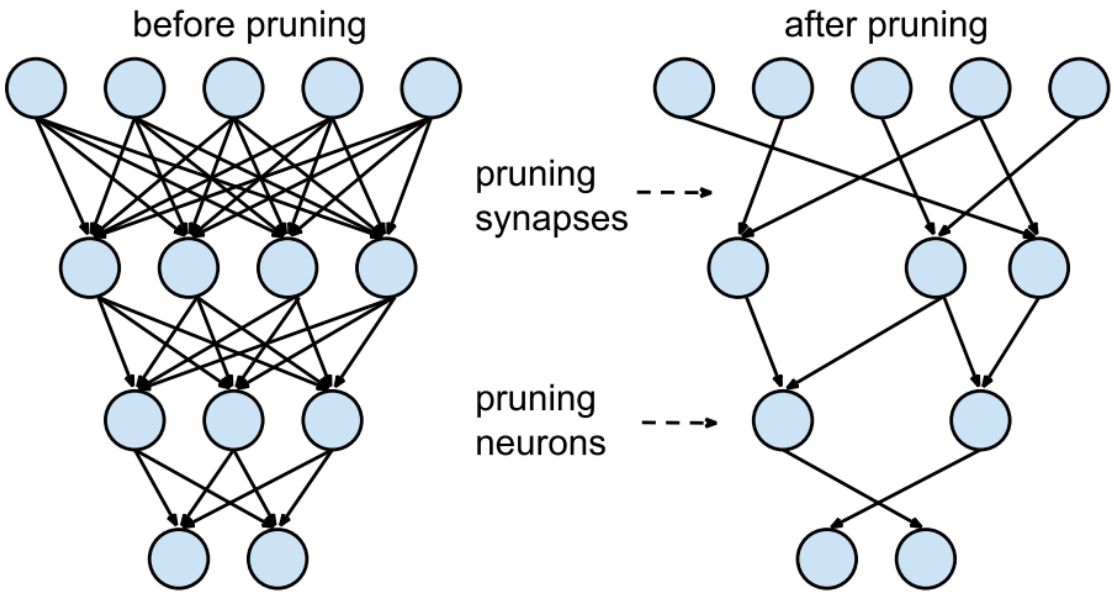

Figure 1.2. Illustration of Neural Network Pruning. Weights deemed less important are removed before retraining to cover accuracy losses. Source: [42].

Lastly, quantization involves reducing the bit accuracy of the weights of a network [5]. Due to requiring fewer hardware resources, FPGAs often use fixed-point arithmetic, where the number of digits representing the fractional part is fixed, as opposed to the 32-bit floating point (float32) representation that is used in most deep learning libraries. An illustration of fixed-point representation of numbers when compared to standard floating-point representation is shown in Figure 1.3.

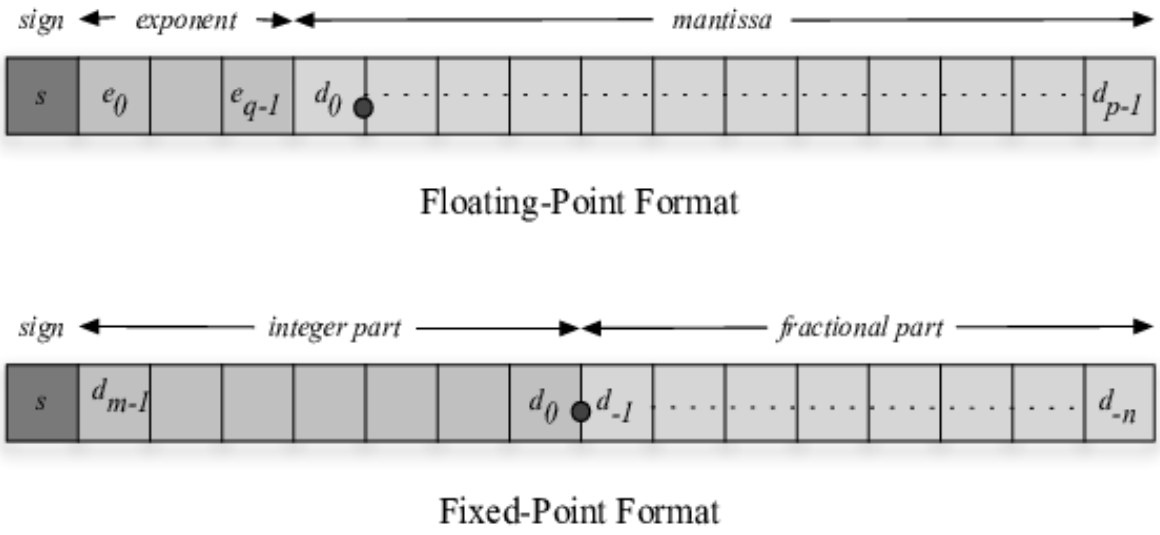

Figure 1.3. Illustration of floating-point vs. fixed-point notation. The decimal point is free to move in floating-point whereas it is fixed in place for fixed-point, resulting in a constant number of bits allocated for the integer and fractional parts of the number. Source: [27].

Reducing the precision of the fixed-point representations used on an FPGA can significantly reduce its resource consumption [11]. Quantization-Aware Training (QAT), where quantization is introduced into the training process, allows for a model to maintain high accuracy by having the weights adapt to the reduced precision during training, before they are converted into FPGA hardware [29]. Moreover, open-source libraries such as QKeras offer simple ways to train models using fixed-point arithmetic instead of 32-point floating arithmetic, leading to a reduction or elimination of accuracy loss when converting to fixed-point systems required on specialized hardware [8]. Finding optimal bit-precision for a model requires experimentation specific to each task. Maintaining high accuracy on FPGA hardware has been demonstrated with low bit-precision networks, and even with binary neural networks, where floating point numbers are replaced with binary ones [31].

1.3 Hls4ml

Another issue of using FPGAs for machine learning is the design complexity [11]. Hardware description languages such as Verilog are necessary when working with FPGAs. However, these languages often are tedious and time consuming to work with due to

having very low levels of abstraction [11]. To address this, tools such as High-Level Synthesis (HLS) have been developed, which allow one to describe FPGA functionality using a higher level language like C++, and then automatically synthesize the functionality into an HDL, targeting a specific FPGA part [11]. HLS tools also allow simulation of hardware performance, including correct functionality, latency, and resource usage [11]. This allows for an easy way to test intended performance without the need for physical hardware, further simplifying the design cycle. Vitis HLS, developed by AMD as a part of the Vitis unified software platform, is one such example.

Taking this a step further in the field of deep learning, hls4ml is an open source tool that translates high level python code into C++, and then uses HLS tools to synthesize this into an HDL [11]. Hls4ml was initially built for detecting particles at the Large Hadron Collider, where sub-microsecond latency is necessary due to the extremely high volume of data that needs to be filtered and processed [11]. Traditional hardware such as CPUs and GPUs, are unable to meet these latency requirements [11]. Since then, hls4ml has been applied to a wide variety of applications, including autonomous vehicles [14], edge devices [13], and image segmentation [30]. Initially, only fully connected layers were supported in hls4ml. However, support for CNNs [1], RNNs [23], GNNs [12] and others have since been added, allowing for the acceleration of increasingly complex networks into FPGA hardware. Most recently, the Multi-Headed Attention layer that underlies the transformer architecture has also been added [22] [21]. While hls4ml supports PyTorch and several HLS backends, Keras with Vitis HLS currently has the most support. The general workflow for hls4ml is shown in Figure 1.4.

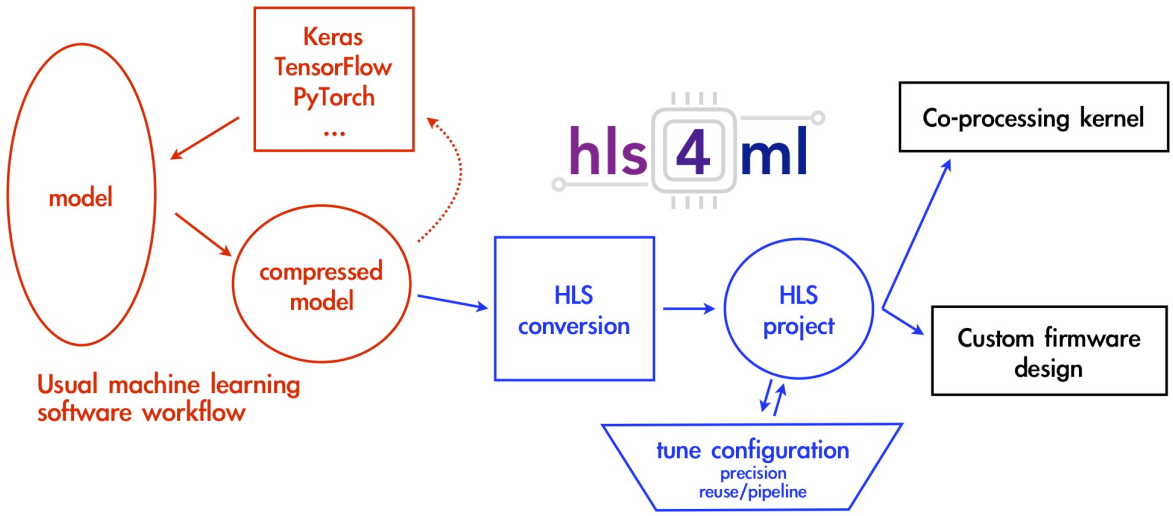

Figure 1.4. Workflow of hls4ml: Converting and Optimizing Machine Learning Models for FPGA Deployment. The usual machine learning workflow is used to achieve a high accuracy model, which is then converted into an HLS representation for further tuning. Hls4ml then converts this into a custom firmware design on an FPGA, using an HDL. Source: [11].

In addition to the significant decrease in design time that hls4ml allows, it also gives users control of several aspects of the resulting FPGA implementation, allowing for the balancing of accuracy, latency, and resource usage specific to each application [11]. For one, the bit precision that is used for each layer can be adjusted, with higher bit precision leading to increased accuracy but also higher resource usage [11]. This flexibility gives even more incentive for quantizing the original model, as almost all accuracy loss can be mitigated when translating from python to C++. Each layer also has the option of prioritizing latency or resource usage, with larger layers requiring the resource option to be used [11]. For most time-sensitive applications, the latency option is preferred when possible. Finally, hls4ml offers a "reuse factor" for each layer, which determines the number of times each multiplier is used in order to compute a layer of neuron's values [11]. Low reuse factor achieves the lowest latency and highest throughput but uses the most resources, while high reuse factor saves resources at the expense of longer latency and lower throughput [11].

1.4 Image Segmentation and UNet

Image segmentation is an image processing and computer vision technique that partitions images into discrete groups of pixels, often called masks [37]. The primary goal of this is to simplify the representation of the image to make it easier to analyze [28]. Image segmentation is vital to many modern-day technologies involving deep learning [16]. Image segmentation, especially as it pertains to computer vision and medical imaging, proves to be a suitable task for deep learning on FPGAs [30]. Many imaging applications require real-time processing that require low latency [46]. In addition, the power efficiency of FPGAs make them ideal for portable or battery-operated medical devices [40]. The inherent flexibility of FPGAs also allows them to be tailored to meet specific requirements of different imaging systems. The use of FPGAs in imaging tasks is still quite new, which warrants demonstration of its potential efficacy in the field.

UNet is a type of convolutional neural network introduced in 2015 that was specifically designed for image segmentation tasks [37]. The architecture consists of an encoder and decoder path. The encoder path consists of repeated application of two 3×3 convolutional layers, each followed by a rectified linear unit (ReLU) and a 2×2 max pooling operation with stride 2 for downsampling [37]. This is done to reduce spatial dimensions while increasing feature representation. The bottleneck consists of convolutional layers that extract the most abstract features of the input, and serves as a bridge between the encoder and decoder [37]. UNet then transitions to a decoder path that involves an upsampling of the feature map, followed by a 2×2 convolution ("up-convolution") that halves the number of feature channels [37]. The upsampled feature map is concatenated with the corresponding feature map from the encoder (skip connections) followed by two 3×3 convolutions and ReLU activations, restoring spatial dimensions [37]. This architecture is shown in Figure 1.5.

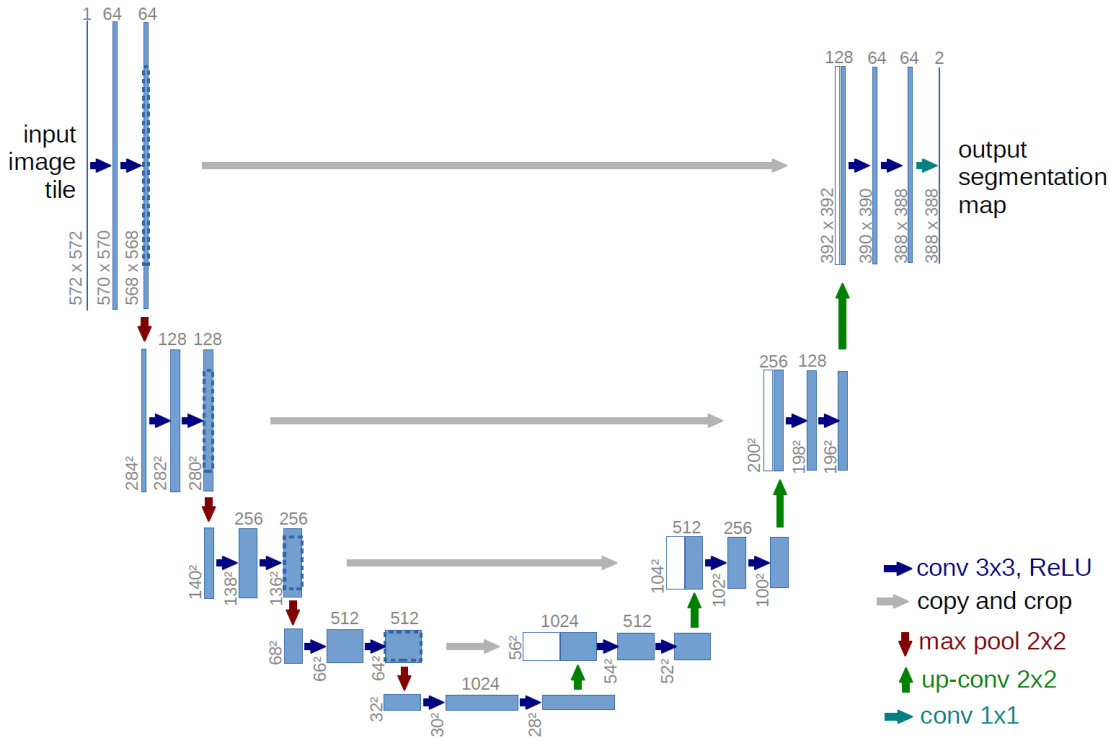

Figure 1.5. U-Net Architecture for Image Segmentation: Downsampling and Upsampling Pathways with Convolutional, Pooling, and Upsampling Layers. Source: [37].

The UNet architecture was pivotal in the field of image segmentation and has been applied to various applications since [28]. The advent of skip connections allowed the model to retain detailed information lost during the encoding phase, which before had been a large issue in the field [37]. Despite its capabilities, the architecture also allows relatively high computational efficiency. Since the seminal UNet architecture was developed, there have been a variety of architectures that seek to improve upon it, while maintaining its general structure [33, 51].

For example, UNet++ incorporates nested dense skip connections that incrementally refine the features before they are transferred to the decoder [52]. This modification helps in reducing the semantic gap between encoder and decoder features, improving segmentation accuracy. Another modification of the original UNet is TransUNet, which incorporates a transformer in the bottleneck to capture more long-range dependencies [7]. A third modification is the use of Squeeze and Excitation blocks [38]. The squeeze portion of these blocks uses global average pooling to compress the spatial dimensions of feature maps, producing a single value for each channel. The excitation portion then uses fully connected layers to learn channel-wise dependencies and generate a set of weights.

These blocks are integrated into the network to improve its ability to focus on important features while suppressing less useful ones. Combinations of these additions, as well as added attention mechanisms, have also been used to great effect [2].

1.5 Evaluation of Models

Various metrics have been developed to measure the accuracy of image segmentation tasks. Below, we describe and provide the mathematical formulations of the most commonly used ones.

1.5.1 Dice Similarity Coefficient (DSC):

The Dice Similarity Coefficient (DSC), also known as the F1 Score in binary segmentation, quantifies the overlap between the predicted mask P and the ground truth mask G. It was introduced by Dice in 1945 [10] and is defined as:

$$\text{DSC} = \frac{2|P \cap G|}{|P| + |G|} = \frac{2TP}{2TP + FP + FN}$$

where TP, FP, and FN are the number of true positives, false positives, and false negatives, respectively.

1.5.2 Intersection over Union (IoU):

The Intersection over Union (IoU), also known as the Jaccard Index, measures the ratio between the intersection and the union of the predicted and ground truth masks. It was proposed by Jaccard in 1912 [20]:

$$\text{IoU} = \frac{|P \cap G|}{|P \cup G|} = \frac{TP}{TP + FP + FN}$$

IoU is more stringent than DSC as it penalizes both false positives and false negatives equally.

1.5.3 Average Hausdorff Distance (HD):

The Hausdorff Distance is a boundary-based metric that measures the greatest of all the distances from a point in one set to the closest point in another set. A widely used implementation for comparing shapes in images was proposed by Huttenlocher et al. [18].

The average Hausdorff Distance (aHD) considers the mean of these distances and is more robust to outliers. Given the boundary point sets ∂P and ∂G of the predicted and ground truth masks respectively, the directed Hausdorff distance from ∂P to ∂G is:

$$h(\partial P, \partial G) = \max_{p \in \partial P} \min_{g \in \partial G} \|p - g\|$$

The symmetric Hausdorff Distance is:

$$H(\partial P, \partial G) = \max \{h(\partial P, \partial G), h(\partial G, \partial P)\}$$

The average version is defined as:

$$\text{aHD}(\partial P, \partial G) = \frac{1}{|\partial P|} \sum_{p \in \partial P} \min_{g \in \partial G} \|p - g\| + \frac{1}{|\partial G|} \sum_{g \in \partial G} \min_{p \in \partial P} \|g - p\|$$

This metric is particularly relevant in high-precision applications, such as medical image segmentation, where accurate boundary delineation is critical [2].

1.6 Societal and Ethical Implications

The acceleration of deep learning models on FPGA hardware for image segmentation has significant implications beyond technical performance. This section evaluates the economic impact of such technologies, their relation to public welfare, potential risk and safety issues, and the ethical and professional responsibilities they entail.

1.6.1 Economic Analysis

The adoption of FPGAs in deep learning pipelines presents an attractive economic proposition for edge computing and embedded systems due to their high computational efficiency and low power consumption. Unlike GPUs and CPUs, FPGAs can be optimized for specific tasks, enabling tailored execution paths that reduce both latency and energy consumption per inference. However, this efficiency comes with an upfront trade-off: FPGAs tend to have a higher cost per raw computation when compared directly to GPUs in terms of floating-point operations per second (FLOPs) [49]. Additionally, development costs are elevated due to the need for specialized hardware design, longer development cycles, and lower developer familiarity relative to more common platforms.

Despite these factors, FPGAs often deliver superior performance-per-watt, making them especially appealing in environments where power and thermal budgets are con-

strained—such as medical devices, remote sensors, or mobile robots. In many such use cases, the total cost of ownership (TCO) becomes more favorable over time, particularly when factoring in long-term energy savings and increased hardware longevity [32]. Moreover, when used for highly specialized, repetitive inference tasks, the deterministic nature of FPGA execution can further reduce hidden performance costs such as unpredictable latency or memory bottlenecks that may occur in CPU/GPU systems.

Over time, these benefits can lead to more scalable and sustainable deployments in both industrial and research contexts, especially where operational efficiency outweighs upfront capital expenditure.

1.6.2 Public Welfare Considerations

Applications such as medical diagnostics and autonomous vehicles, which rely on image segmentation, are inherently tied to public welfare. FPGA-accelerated models can enable real-time decision making, contributing to faster diagnoses, more effective treatments, and enhanced safety in critical systems [43]. In addition, their low power consumption could potentially lead to more sustainable processes overtime, especially as high energy consumption and exhaustion of raw materials becomes more of a critical issue in the future [49]. However, the societal benefits must be weighed against the risk of unequal access. Without proper policy and deployment strategies, advanced systems may become available only in wealthier regions, exacerbating existing disparities in healthcare and infrastructure [47]. Ensuring equitable access to these technologies is an important goal moving forward.

1.6.3 Risk and Safety Considerations

Deploying FPGA-accelerated models in safety-critical systems introduces notable risks. In medical imaging, an inaccurate segmentation could lead to misdiagnosis or incorrect treatments. In autonomous systems, segmentation errors could result in failures with potentially life-threatening consequences. It is therefore essential to rigorously test these models under diverse conditions and incorporate safety mechanisms. It is also important to slowly roll these technologies out into real-life systems, and have proper feedback to evaluate their efficacy. Given the static nature of FPGA designs post-deployment, updates or retraining must undergo thorough verification to ensure continued safety and performance [3].

1.6.4 Ethical and Professional Responsibilities

Professionals working in this space bear ethical responsibilities both to individual users and society as a whole. These responsibilities include maintaining transparency about model performance and limitations, mitigating bias in training data, and ensuring fairness in deployment. Developers should follow ethical standards outlined by organizations such as the ACM and IEEE, which emphasize accountability, inclusivity, and public good [4, 19]. Additionally, practitioners should communicate clearly with stakeholders to foster trust and understanding of the technology's capabilities and constraints. As gatekeepers of powerful tools, engineers must strive not only for innovation but for the responsible and equitable application of their work.

Chapter 2
Methods and Materials

The following section details the workflow implored when collecting data for this thesis. Section 2.1 describes the particular UNet architecture used for hardware acceleration. Section 2.2 presents the three datasets that were used to train the model. Section 2.3 details the specific hyperparameters and hls4ml configuration used when training and synthesizing the network onto an FPGA. Lastly, section 2.4 describes how the latency shown in the FPGA synthesis was compared to traditional hardware, utilizing resources available on Google Colab.

2.1 UNet Architecture

Designing a UNet for FPGA synthesis is quite challenging, mainly due to the resource constraints that come with FPGA hardware. The original UNet has over 30 million parameters, which is currently infeasible for acceleration on a single FPGA, especially when using HLS tools such as hls4ml. The primary issue that these resource constraints cause is designing a network that is still able to give a high image segmentation accuracy while being much smaller than a traditional UNet [11]. The architecture used in this paper is a simplified variant of the original U-Net, implemented using Keras. While the original U-Net consists of four encoder blocks, a bottleneck, and four decoder blocks with corresponding skip connections [37], the architecture presented here includes only two encoder blocks, a bottleneck, and two decoder blocks. The number of filters in each convolution layer has also been significantly reduced, with the first and last encoder/decoder blocks having 16 filters instead of 64. The doubling of the number of filters in each encoder block and subsequent halving of the number of filters in each decoder block was kept from the original architecture. Keras UpSampling2D layers (as opposed to ConvTranspose2D layers) are used in decoder blocks due to their availability

in hls4ml. Skip connections are still employed to preserve spatial information in the form of Keras Merge layers. Batch normalization, dropout, and l2 regularization were added to boost performance and prevent overfitting. The outputs of CAMUS and BraTS were softmaxes containing 4 classes while the output of Oxford Pets was a sigmoid since it involved binary segmentation. The network reduces total parameters from around 31,000,000 to around 100,000. The network was designed using both research from previous studies as well as individual experimentation. All three datasets were trained with the same network (with the input/output sizes changing depending on the dataset). This was primarily done due to the difficulty of finding a high-accuracy image segmentation network that was small enough and compatible with hls4ml. Below is a more succinct summary of the model and its components.

Input

- **Input shape:** $(128, 128, 3)$ for Oxford Pets, $(384, 384, 1)$ for CAMUS, and $(128, 128, 4)$ for BraTS.

Encoder (Downsampling Path)

Each encoder block contains:

1. Two convolutional layers (Conv2D) using "same" padding with ReLU activation and Batch Normalization.
2. Dropout for regularization.
3. MaxPooling2D to reduce spatial dimensions.

- **Block 1:** 16 filters.
- **Block 2:** 32 filters.

Bottleneck

- Two Conv2D layers using "same" padding with 64 filters, BatchNorm, and ReLU.
- No further downsampling is applied here.

Decoder (Upsampling Path)

Each decoder block contains:

1. UpSampling2D to increase resolution.
2. Concatenation with the corresponding encoder feature map (skip connection).
3. Two Conv2D layers using "same" padding with BatchNorm and ReLU.
4. Dropout for regularization.

- **Block 1:** Upsamples and concatenates with encoder Block 2 (32 filters).
- **Block 2:** Upsamples and concatenates with encoder Block 1 (16 filters).

Output

- For the Oxford Pets Dataset, a final Conv2D layer with a sigmoid activation outputs a binary segmentation map of shape $(128, 128, 1)$. For the CAMUS Dataset, a final Conv2D layer with a softmax activation of 4 classes outputs a segmentation map of shape $(384, 384, 4)$. For the BraTS Dataset, a final Conv2D layer with a softmax activation of 4 classes outputs a segmentation map of shape $(128, 128, 4)$.

Figure 2.1 displays a visual representation of the unet architecture, removing batch normalization, relu, and dropout layers.

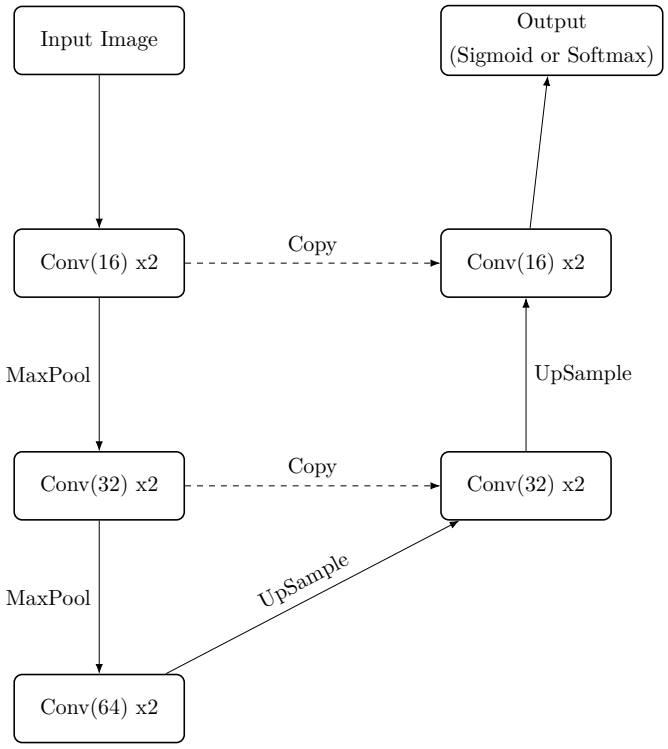

Figure 2.1. Reduced UNet architecture for image segmentation. The network consists of a contracting path (left) with max pooling operations, and an expanding path (right) with upsampling operations. Skip connections (dashed arrows) help preserve spatial information by copying features from the contracting path to the corresponding level in the expanding path.

Table 2.1, Table 2.2, and Table 2.3 each give a more detailed look into each layer the reduced UNet architectures used in the Oxford Pets, CAMUS, and BraTS datasets, respectively, showing both layer shapes and parameter counts.

Table 2.1: Model Summary for Oxford Pets

Layer (type)	**Output Shape**	**Param #**
input_layer (InputLayer)	(None, 128, 128, 3)	0
conv1_1 (Conv2D)	(None, 128, 128, 16)	448
bn1_1 (BatchNormalization)	(None, 128, 128, 16)	64
relu1_1 (ReLU)	(None, 128, 128, 16)	0
conv1_2 (Conv2D)	(None, 128, 128, 16)	2320
bn1_2 (BatchNormalization)	(None, 128, 128, 16)	64
relu1_2 (ReLU)	(None, 128, 128, 16)	0
dropout1 (Dropout)	(None, 128, 128, 16)	0
pool1 (MaxPooling2D)	(None, 64, 64, 16)	0
conv2_1 (Conv2D)	(None, 64, 64, 32)	4640
bn2_1 (BatchNormalization)	(None, 64, 64, 32)	128
relu2_1 (ReLU)	(None, 64, 64, 32)	0
conv2_2 (Conv2D)	(None, 64, 64, 32)	9248
bn2_2 (BatchNormalization)	(None, 64, 64, 32)	128
relu2_2 (ReLU)	(None, 64, 64, 32)	0
dropout2 (Dropout)	(None, 64, 64, 32)	0
pool2 (MaxPooling2D)	(None, 32, 32, 32)	0
conv3_1 (Conv2D)	(None, 32, 32, 64)	18496
bn3_1 (BatchNormalization)	(None, 32, 32, 64)	256
relu3_1 (ReLU)	(None, 32, 32, 64)	0
conv3_2 (Conv2D)	(None, 32, 32, 64)	36928
bn3_2 (BatchNormalization)	(None, 32, 32, 64)	256
relu3_2 (ReLU)	(None, 32, 32, 64)	0
dropout3 (Dropout)	(None, 32, 32, 64)	0
upsample1 (UpSampling2D)	(None, 64, 64, 64)	0
conv4_1 (Conv2D)	(None, 64, 64, 32)	8224
bn4_1 (BatchNormalization)	(None, 64, 64, 32)	128
relu4_1 (ReLU)	(None, 64, 64, 32)	0
merge1 (Concatenate)	(None, 64, 64, 64)	0
conv4_2 (Conv2D)	(None, 64, 64, 32)	18464
bn4_2 (BatchNormalization)	(None, 64, 64, 32)	128

Continued on next page

Table 2.1 – continued from previous page

Layer (type)	Output Shape	Param #
relu4_2 (ReLU)	(None, 64, 64, 32)	0
conv4_3 (Conv2D)	(None, 64, 64, 32)	9248
bn4_3 (BatchNormalization)	(None, 64, 64, 32)	128
relu4_3 (ReLU)	(None, 64, 64, 32)	0
dropout4 (Dropout)	(None, 64, 64, 32)	0
upsample2 (UpSampling2D)	(None, 128, 128, 32)	0
conv5_1 (Conv2D)	(None, 128, 128, 16)	2064
bn5_1 (BatchNormalization)	(None, 128, 128, 16)	64
relu5_1 (ReLU)	(None, 128, 128, 16)	0
merge2 (Concatenate)	(None, 128, 128, 32)	0
conv5_2 (Conv2D)	(None, 128, 128, 16)	4624
bn5_2 (BatchNormalization)	(None, 128, 128, 16)	64
relu5_2 (ReLU)	(None, 128, 128, 16)	0
conv5_3 (Conv2D)	(None, 128, 128, 16)	2320
bn5_3 (BatchNormalization)	(None, 128, 128, 16)	64
relu5_3 (ReLU)	(None, 128, 128, 16)	0
dropout5 (Dropout)	(None, 128, 128, 16)	0
output_conv (Conv2D)	(None, 128, 128, 1)	17
Total params: 118,513 (462.94 KB)		
Trainable params: 117,777 (460.07 KB)		

Table 2.2: Model Summary for CAMUS dataset

Layer (type)	**Output Shape**	**Param #**
input_layer (InputLayer)	(None, 384, 384, 1)	0
conv1_1 (Conv2D)	(None, 384, 384, 16)	160
bn1_1 (BatchNormalization)	(None, 384, 384, 16)	64
relu1_1 (ReLU)	(None, 384, 384, 16)	0
conv1_2 (Conv2D)	(None, 384, 384, 16)	2320
bn1_2 (BatchNormalization)	(None, 384, 384, 16)	64
relu1_2 (ReLU)	(None, 384, 384, 16)	0
dropout1 (Dropout)	(None, 384, 384, 16)	0
pool1 (MaxPooling2D)	(None, 192, 192, 16)	0
conv2_1 (Conv2D)	(None, 192, 192, 32)	4640
bn2_1 (BatchNormalization)	(None, 192, 192, 32)	128
relu2_1 (ReLU)	(None, 192, 192, 32)	0
conv2_2 (Conv2D)	(None, 192, 192, 32)	9248
bn2_2 (BatchNormalization)	(None, 192, 192, 32)	128
relu2_2 (ReLU)	(None, 192, 192, 32)	0
dropout2 (Dropout)	(None, 192, 192, 32)	0
pool2 (MaxPooling2D)	(None, 96, 96, 32)	0
conv3_1 (Conv2D)	(None, 96, 96, 64)	18496
bn3_1 (BatchNormalization)	(None, 96, 96, 64)	256
relu3_1 (ReLU)	(None, 96, 96, 64)	0
conv3_2 (Conv2D)	(None, 96, 96, 64)	36928
bn3_2 (BatchNormalization)	(None, 96, 96, 64)	256
relu3_2 (ReLU)	(None, 96, 96, 64)	0
dropout3 (Dropout)	(None, 96, 96, 64)	0
upsample1 (UpSampling2D)	(None, 192, 192, 64)	0
conv4_1 (Conv2D)	(None, 192, 192, 32)	8224
bn4_1 (BatchNormalization)	(None, 192, 192, 32)	128
relu4_1 (ReLU)	(None, 192, 192, 32)	0
merge1 (Concatenate)	(None, 192, 192, 64)	0
conv4_2 (Conv2D)	(None, 192, 192, 32)	18464
bn4_2 (BatchNormalization)	(None, 192, 192, 32)	128

Continued on next page

Table 2.2 – continued from previous page

Layer (type)	Output Shape	Param #
relu4_2 (ReLU)	(None, 192, 192, 32)	0
conv4_3 (Conv2D)	(None, 192, 192, 32)	9248
bn4_3 (BatchNormalization)	(None, 192, 192, 32)	128
relu4_3 (ReLU)	(None, 192, 192, 32)	0
dropout4 (Dropout)	(None, 192, 192, 32)	0
upsample2 (UpSampling2D)	(None, 384, 384, 32)	0
conv5_1 (Conv2D)	(None, 384, 384, 16)	2064
bn5_1 (BatchNormalization)	(None, 384, 384, 16)	64
relu5_1 (ReLU)	(None, 384, 384, 16)	0
merge2 (Concatenate)	(None, 384, 384, 32)	0
conv5_2 (Conv2D)	(None, 384, 384, 16)	4624
bn5_2 (BatchNormalization)	(None, 384, 384, 16)	64
relu5_2 (ReLU)	(None, 384, 384, 16)	0
conv5_3 (Conv2D)	(None, 384, 384, 16)	2320
bn5_3 (BatchNormalization)	(None, 384, 384, 16)	64
relu5_3 (ReLU)	(None, 384, 384, 16)	0
dropout5 (Dropout)	(None, 384, 384, 16)	0
output_conv (Conv2D)	(None, 384, 384, 4)	68
softmax_output (Activation)	(None, 384, 384, 4)	0
Total params: 118,276 (462.02 KB)		
Trainable params: 117,540 (459.14 KB)		

Table 2.3: Model Summary for BraTS dataset

Layer (type)	**Output Shape**	**Param #**
input_layer (InputLayer)	(None, 128, 128, 4)	0
conv1_1 (Conv2D)	(None, 128, 128, 16)	592
bn1_1 (BatchNormalization)	(None, 128, 128, 16)	64
relu1_1 (ReLU)	(None, 128, 128, 16)	0
conv1_2 (Conv2D)	(None, 128, 128, 16)	2320
bn1_2 (BatchNormalization)	(None, 128, 128, 16)	64
relu1_2 (ReLU)	(None, 128, 128, 16)	0
dropout1 (Dropout)	(None, 128, 128, 16)	0
pool1 (MaxPooling2D)	(None, 64, 64, 16)	0
conv2_1 (Conv2D)	(None, 64, 64, 32)	4640
bn2_1 (BatchNormalization)	(None, 64, 64, 32)	128
relu2_1 (ReLU)	(None, 64, 64, 32)	0
conv2_2 (Conv2D)	(None, 64, 64, 32)	9248
bn2_2 (BatchNormalization)	(None, 64, 64, 32)	128
relu2_2 (ReLU)	(None, 64, 64, 32)	0
dropout2 (Dropout)	(None, 64, 64, 32)	0
pool2 (MaxPooling2D)	(None, 32, 32, 32)	0
conv3_1 (Conv2D)	(None, 32, 32, 64)	18496
bn3_1 (BatchNormalization)	(None, 32, 32, 64)	256
relu3_1 (ReLU)	(None, 32, 32, 64)	0
conv3_2 (Conv2D)	(None, 32, 32, 64)	36928
bn3_2 (BatchNormalization)	(None, 32, 32, 64)	256
relu3_2 (ReLU)	(None, 32, 32, 64)	0
dropout3 (Dropout)	(None, 32, 32, 64)	0
upsample1 (UpSampling2D)	(None, 64, 64, 64)	0
conv4_1 (Conv2D)	(None, 64, 64, 32)	8224
bn4_1 (BatchNormalization)	(None, 64, 64, 32)	128
relu4_1 (ReLU)	(None, 64, 64, 32)	0
merge1 (Concatenate)	(None, 64, 64, 64)	0
conv4_2 (Conv2D)	(None, 64, 64, 32)	18464
bn4_2 (BatchNormalization)	(None, 64, 64, 32)	128

Continued on next page

Table 2.3 – continued from previous page

Layer (type)	Output Shape	Param #
relu4_2 (ReLU)	(None, 64, 64, 32)	0
conv4_3 (Conv2D)	(None, 64, 64, 32)	9248
bn4_3 (BatchNormalization)	(None, 64, 64, 32)	128
relu4_3 (ReLU)	(None, 64, 64, 32)	0
dropout4 (Dropout)	(None, 64, 64, 32)	0
upsample2 (UpSampling2D)	(None, 128, 128, 32)	0
conv5_1 (Conv2D)	(None, 128, 128, 16)	2064
bn5_1 (BatchNormalization)	(None, 128, 128, 16)	64
relu5_1 (ReLU)	(None, 128, 128, 16)	0
merge2 (Concatenate)	(None, 128, 128, 32)	0
conv5_2 (Conv2D)	(None, 128, 128, 16)	4624
bn5_2 (BatchNormalization)	(None, 128, 128, 16)	64
relu5_2 (ReLU)	(None, 128, 128, 16)	0
conv5_3 (Conv2D)	(None, 128, 128, 16)	2320
bn5_3 (BatchNormalization)	(None, 128, 128, 16)	64
relu5_3 (ReLU)	(None, 128, 128, 16)	0
dropout5 (Dropout)	(None, 128, 128, 16)	0
output_conv (Conv2D)	(None, 128, 128, 4)	68
softmax_output (Activation)	(None, 128, 128, 4)	0
Total params: 118,708 (463.70 KB)		
Trainable params: 117,972 (460.83 KB)		

2.2 Datasets

2.2.1 Oxford Pets Dataset

The Oxford-IIIT Pet dataset [34] consists of 7,349 images of 37 different cat and dog breeds. Each image is annotated with a pixel-level trimap segmentation mask that classifies pixels as belonging to the pet, the background, or the outline. This dataset is widely used for semantic segmentation tasks and provides a balanced mix of classes with considerable intra-class variation in pose, scale, and lighting conditions. The relatively small image sizes and well-defined object boundaries make it a good benchmark for evaluating lightweight segmentation models. The dataset includes RGB images and corresponding segmentation masks in PNG format.

2.2.1.1 Pre-processing

We split the dataset into training and validation sets using an 80-20 ratio via stratified random sampling to maintain class balance. All RGB images were resized to a fixed resolution of 128×128 using bilinear interpolation. Pixel intensities were normalized to the $[0, 1]$ range by dividing by 255. Each annotation mask originally encodes three classes: background (label 2), pet (label 1), and border (label 3). To convert this into a pure binary segmentation problem, we binarized the masks to distinguish foreground (class 1, mapped to 1.0) from background and border (classes 2 and 3, mapped to 0.0).

2.2.1.2 Data Augmentation

To reduce overfitting and increase generalization, we applied random data augmentation during training, including:

- Random horizontal and vertical flips with 50% probability
- Random 90-degree rotations (0, 90, 180, or 270 degrees)

Augmentations were applied identically to both images and corresponding masks to maintain alignment. This approach is consistent with standard practices in semantic segmentation [41].

Examples of images and their ground truth segmentation are shown in Figure 2.2.

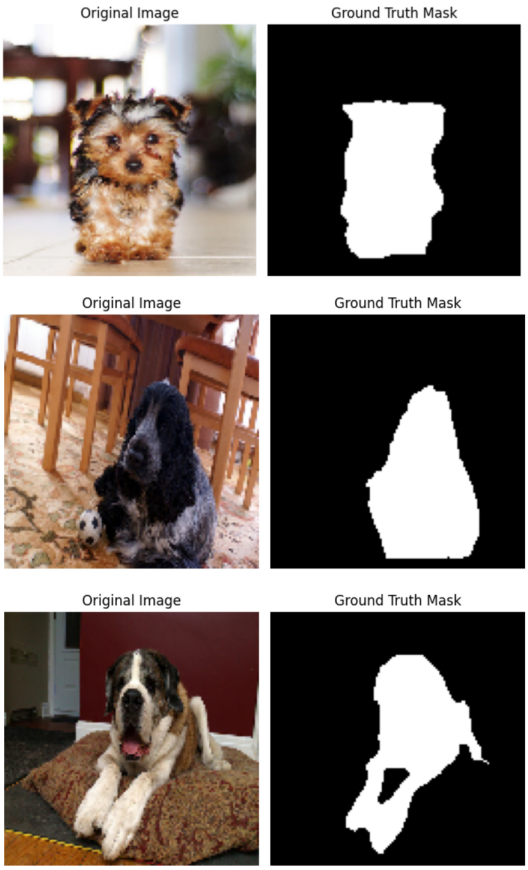

Figure 2.2. Examples of Image Segmentation on The Oxford Pets dataset.

2.2.2 CAMUS Dataset

We also used an ultrasound dataset acquired at the University Hospital of St Etienne (France) and included the regulations set by the local ethical committee of the hospital [24]. The dataset was collected using a GE M5S probe from the GE Vivid E95 ultrasound scanner. The EchoPAC analysis software exported the 2D apical four-chamber and two-chamber view sequences for each patient. The acquisitions were optimized to perform Left Ventricle Ejection Fraction (LVEF). Images from 450 patients were used for training and testing in this study. The dataset contains images of End-Diastole (ED) and End-Systole (ES) for each patient [24], which encapsulates different moments in a heartbeat cycle. We only used the 2D apical two-chamber view sequences for the purposes of this study.

2.2.2.1 Pre-processing

We split the dataset into training and validation sets using a 80-20 ratio. The images were normalized and resized to 384*384. No augmentation was performed, following [24].

Examples of images and their ground truth segmentation are shown in Figure 2.3.

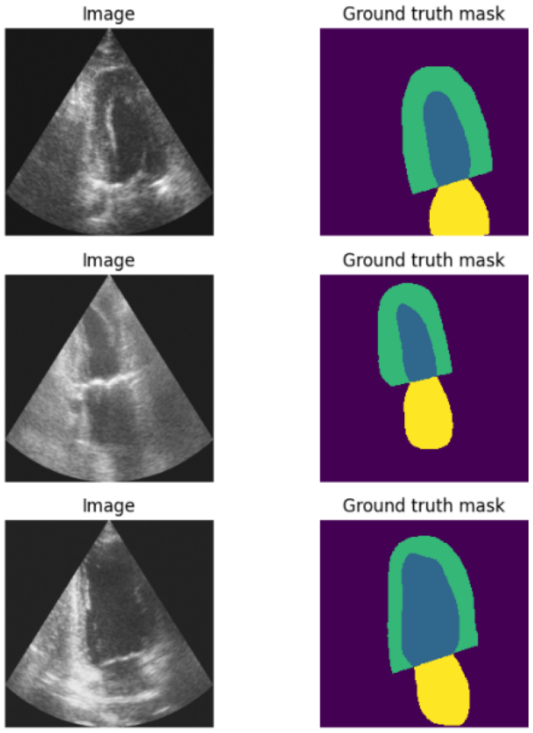

Figure 2.3. An Example of Image Segmentation on The CAMUS Dataset.

2.2.3 BraTS Dataset

The Brain Tumor Segmentation (BraTS) dataset [6] is a large-scale medical imaging benchmark focused on the segmentation of brain tumors from multi-modal MRI scans. It includes high-grade gliomas (HGG) and low-grade gliomas (LGG), annotated with detailed labels for tumor subregions including enhancing tumor, tumor core, and whole tumor. Each case includes four MRI modalities: T1, T1-contrast, T2, and FLAIR.

2.2.3.1 Pre-processing

Preprocessing of this dataset was taken primarily from [30]. The 3D-MRI scans were resized from 240*240*155 to 128*128*128 dimension to reduce the unnecessary background data and for uniformity. The images were scaled using the min-max scaler for a mean

of 0 and max value of 1. Since the model was defined for 2D convolution kernels, each image was sliced to from 3D to 2D slices for all the modalities and masks. The modalities for each sample were converted to numpy arrays and combined or stacked into one image, resulting in an input shape of 128*128*4 for each image (one 128*128 2D image for each of the 4 MRI modalities stacked on one another). No augmentation was done on the BraTS dataset.

Examples of images and their ground truth segmentation are shown in Figure 2.4.

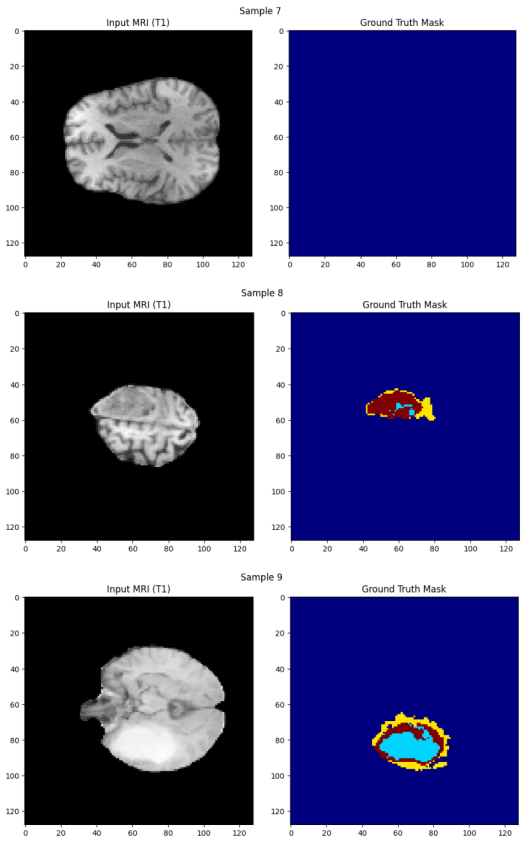

Figure 2.4. Examples of Image Segmentation on The BraTS Dataset.

2.3 Training and Synthesis Details

2.3.1 Hyperparameter Tuning

When training networks for high accuracy, hyperparameter tuning is a crucial step for optimal performance [25]. The following hyperparameters were chosen based on both previous UNet papers and experimentation.

All models were trained using the `Adam` optimizer with an initial learning rate of `1e-3`. The `dice coefficient` on the validation sets was used as the evaluation metric across all datasets. Stopping training early when no more progress can be made and reducing the learning rate upon an accuracy plateau were callbacks used in the training of the models. Early stopping was employed with a patience of 5 epochs, monitoring the validation dice score with `mode="max"`. The setting `restore_best_weights=True` was set so the best weights would be chosen in the event the model begins to start overfitting. Additionally, the `ReduceLROnPlateau` scheduler was used with a reduction factor of 0.1 and a minimum learning rate of `1e-6`, to further boost results. Model pruning was applied uniformly with a target sparsity of 50% and a constant pruning schedule in all three datasets, utilizing the TensorFlow Model Optimization Toolkit library.

Notable training differences between the datasets include the loss function and batch size. The Oxford Pets model used a custom binary `dice_loss`, while CAMUS used a multiclass *generalized_dice_loss*, and BraTS employed the standard *sparse categorical crossentropy*. The batch size for Oxford Pets and BraTS was set to 16, whereas a smaller batch size of 4 was used for CAMUS due to the larger image dimensions occasionally causing resource exhaustion issues at larger batch sizes. Furthermore, the number of training epochs was only set to 30 for BraTS, due to the large amount of training time per epoch, while both Oxford Pets and CAMUS were trained for 150 epochs (though early stopping caused the actual number of epochs to be around 50-60). For this paper, the Dice Similarity Coefficient was used to evaluate the accuracy of image segmentation for all three datasets. Multi-class Dice was calculated for CAMUS and BraTS (ignoring the background), and binary Dice was calculated for the Oxford Pets dataset.

The hyperparameters and callbacks used for the Oxford Pets, CAMUS, and BraTS datasets are summarized in Table 2.4, Table 2.5, Table 2.6, respectively.

Hyperparameter	**Value**
Training Setup	
Optimizer	Adam
Learning Rate	1e-3
Loss	`dice_loss`
Metrics	`[binary_dice, "accuracy"]`
Batch Size	16
Epochs	150
Early Stopping	
`monitor`	`"val_multiclass_dice"`
`min_delta`	0
`patience`	5
`mode`	`"max"`
`restore_best_weights`	`True`
ReduceLROnPlateau	
`monitor`	`"val_multiclass_dice"`
`factor`	0.1
`min_lr`	1e-6
Pruning	
Sparsity	50%
Schedule	Constant

Table 2.4. Hyperparameters and pruning configuration for Oxford Pets

Hyperparameter	**Value**
Training Setup	
Optimizer	Adam
Learning Rate	1e-3
Loss	`generalized_dice_loss`
Metrics	`[multiclass_dice, "accuracy"]`
Batch Size	4
Epochs	150
Early Stopping	
`monitor`	`"val_multiclass_dice"`
`min_delta`	0
`patience`	5
`mode`	`"max"`
`restore_best_weights`	`True`
ReduceLROnPlateau	
`monitor`	`"val_multiclass_dice"`
`factor`	0.1
`min_lr`	1e-6
Pruning	
Sparsity	50%
Schedule	Constant

Table 2.5. Hyperparameters and pruning configuration for CAMUS

Hyperparameter	Value
Training Setup	
Optimizer	Adam
Learning Rate	1e-3
Loss	`sparse_categorical_crossentropy`
Metrics	`[multiclass_dice, "accuracy"]`
Batch Size	16
Epochs	25
Early Stopping	
`monitor`	`"val_multiclass_dice"`
`min_delta`	0
`patience`	5
`mode`	`"max"`
`restore_best_weights`	`True`
ReduceLROnPlateau	
`monitor`	`"val_multiclass_dice"`
`factor`	0.1
`min_lr`	1e-6
Pruning	
Sparsity	50%
Schedule	Constant

Table 2.6. Hyperparameters and pruning configuration for BraTS

2.3.2 Hls4ml Configuration

When using tools such as hls4ml and HLS synthesis tools from Xilinx, the specific configurations used are crucial for allowing for successful synthesis and balance of latency vs. resource consumption [1]. The following hls4ml configurations were chosen from experimentation and other papers that used hls4ml.

The backend used for HLS synthesis was `Vitis`, which leverages the more recent and optimized `Vitis HLS` compiler, an evolution of `Vivado HLS`, providing improved performance and broader support for modern FPGA features. The numerical precision throughout the network is set to `ap_fixed<16,6>`, ensuring a balanced trade-off between accuracy and resource usage. The model strategy is configured as `Resource`, optimizing for reduced hardware utilization over latency. Currently, this setting is necessary for

successful synthesis in a network this large (even though it is still reduced significantly from standard UNet architectures). A model reuse factor of 1 is used to minimize latency by fully unrolling computations where possible. The softmax output is generated using the `Stable` strategy to preserve numerical stability during inference (in the models that contained softmax) as suggested by [11]. Importantly, the I/O type is set to `io_stream`, which is a requirement when targeting designs that include convolutional layers, as it allows for streaming data between layers and reduces on-chip memory bottlenecks. The HDL language that was used during synthesis was Verilog. Finally, synthesis is targeted to the Xilinx Alveo U250 FPGA (`xcu250-figd2104-2L-e`), a high-performance platform suitable for low-latency inference workloads. A summary of the hls4ml configurations used for all three datasets is shown in Table 2.7.

Table 2.7. hls4ml Configuration Summary

Configuration Item	**Value**
Backend	`Vitis`
Precision	`ap_fixed<16,6>`
Model Strategy	`Resource`
Model Reuse Factor	`1`
Softmax Output Strategy	`Stable`
IO Type	`io_stream`
HDL	`Verilog`
Xilinx Part	`xcu250-figd2104-2L-e`

2.4 Latency Comparison to Traditional Hardware

To assess the real-time performance of our pruned deep learning model, we conducted inference latency measurements across three hardware backends available in Google Colab: CPU, NVIDIA T4 GPU, and TPU v2-8. The models from previous training were loaded into Google Colab. The datasets were stored on Google Drive and mounted into Colab for access.

We selected 100 individual samples from the test set for latency measurement in each dataset. Each sample was passed through the model one at a time (batch size of 1), with a warm-up prediction performed beforehand to mitigate initialization overhead. For each inference, we measured the elapsed wall-clock time using Python's `time.time()` before and after the call to `model.predict()`. This was repeated 100 times, and the resulting latencies were recorded in milliseconds. We then calculated and reported the

mean latencies for each hardware configuration on each dataset. For GPU and TPU measurements, we selected the "GPU T4" and "v2-8 TPU" runtimes from the Google Colab environment menu. All code was executed in eager mode.

While by no means a direct comparison, this benchmarking procedure provides a decent estimation of inference latency across high-end traditional hardware. Current access to elite physical hardware for direct comparisons was infeasible due to resource limitations.

Chapter 3
Results

The following section details the results obtained from all three datasets.

3.1 BraTS Dataset

On the BraTS dataset, the Keras model achieved a multiclass dice score of **0.8485**, while the HLS model reached **0.8396**. This minor drop in performance demonstrates that the HLS-converted model retains much of the accuracy of the original Keras model, despite the hardware constraints and quantization involved in the conversion.

To visually assess the quality of segmentation, we compare a randomly sampled predicted mask from both models against the ground truth. Figure 3.1 illustrates this comparison. Both the Keras predicted and HLS predicted mask shown great similarity with the ground truth, indicating a high quality segmentation.

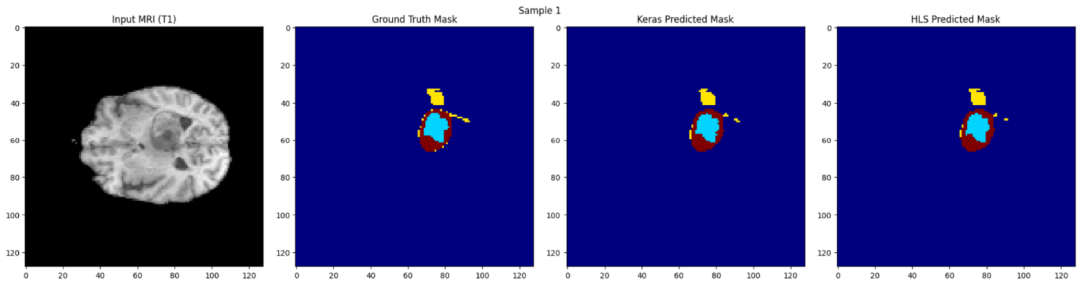

Figure 3.1. Comparison of the original image, ground-truth mask, and predictions from the Keras and HLS models on the BraTS dataset.

Next, we evaluate the inference latency across different hardware platforms. The latency on the FPGA, obtained via synthesis report, is remarkably low—just 0.930 ms.

This is over 100 times faster than both the v2-8 TPU and the T4 GPU available on Google Colab, and over 200 times faster than the CPU. These results are summarized in Table 3.1.

CPU	T4 GPU	v2-8 TPU	FPGA Sim
201.505 ms	142.719 ms	136.211 ms	0.930 ms

Table 3.1. Inference latency comparison of FPGA synthesis report compared to simulation of traditional hardware on Google Colab for the BraTS dataset.

To better understand the FPGA performance in terms of timing, Table 3.2 provides the estimated timing results. The achieved clock period of 3.762 ns is well within the target 5.00 ns, ensuring that the design meets timing requirements for stable operation.

Clock	Target	Estimated	Uncertainty
ap_clk	5.00 ns	3.762 ns	1.35 ns

Table 3.2. Performance timing summary for the BraTS dataset.

Further breakdown of the latency characteristics is shown in Table 3.3. The fixed latency of 185,927 cycles (0.930 ms absolute) and a consistent dataflow pipeline interval indicate that the design is highly deterministic and optimized for throughput.

Latency (cycles)		Latency (absolute)		Interval (cycles)		Pipeline Type
min	max	min	max	min	max	
185927	185927	0.930 ms	0.930 ms	185904	185904	dataflow

Table 3.3. Performance latency summary for the BraTS dataset.

Finally, we assess the FPGA resource utilization in Table 3.4. Notably, the design exceeds 100% DSP utilization within a single SLR, suggesting that additional partitioning or optimization might be necessary for deployment on this particular device. While URAM, BRAM, LUT, and FF usage remains within acceptable limits, the DSP demand highlights the computational intensity of the network, particularly in the number of multiplications used. A potential solution would be to increase the reuse factor in the hls4ml configuration. This would reduce the resource usage at the cost of latency.

Name	BRAM_18K	DSP	FF	LUT	URAM
Total Used	882	39726	1634383	1610542	0
Available SLR	1344	3072	864000	432000	320
SLR Util. (%)	65	1293	189	372	0
Total Available	5376	12288	3456000	1728000	1280
Overall Util. (%)	16	323	47	93	0

Table 3.4. Utilization estimates summary for the BraTS dataset. DSP usage exceeding 100% per SLR suggests the design requires additional partitioning or optimization.

3.2 CAMUS Dataset

On the CAMUS dataset, the Keras model achieved a multiclass dice score of **0.7478**, while the HLS model reached **0.7352**. This slight decrease in performance highlights that the HLS model effectively preserves segmentation accuracy, even after quantization and hardware-specific constraints.

Figure 3.2 presents a visual comparison between the predicted masks from both models and the ground truth. Both the Keras and HLS predictions demonstrate strong agreement with the actual segmentation, with only minor deviations near boundary regions.

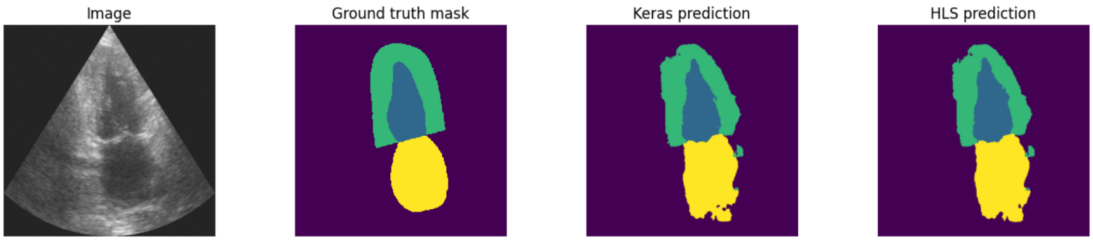

Figure 3.2. Comparison of the original ultrasound image, ground-truth segmentation mask, and the predictions from the Keras and HLS models on the CAMUS dataset.

Inference latency results, shown in Table 3.5, reveal a significant speedup on the FPGA compared to traditional hardware platforms. The FPGA latency of 8.201 ms is over 13 times faster than the T4 GPU and v2-8 TPU and nearly 50 times faster than the CPU.

CPU	T4 GPU	v2-8 TPU	FPGA Sim
413.433 ms	107.727 ms	110.820 ms	8.201 ms

Table 3.5. Inference latency comparison of FPGA synthesis report compared to simulation of traditional hardware on Google Colab for the CAMUS dataset.

The timing summary in Table 3.6 confirms that the achieved clock period of 3.762 ns meets the 5.00 ns target, indicating a valid and stable design.

Clock	Target	Estimated	Uncertainty
ap_clk	5.00 ns	3.762 ns	1.35 ns

Table 3.6. Performance timing summary for the CAMUS dataset.

Table 3.7 further illustrates the latency characteristics. With a fixed latency of 1,640,292 cycles (equating to 8.201 ms) and a constant pipeline interval, the design exhibits strong determinism and high throughput.

Latency (cycles)		Latency (absolute)		Interval (cycles)		Pipeline Type
min	max	min	max	min	max	
1640292	1640292	8.201 ms	8.201 ms	1638960	1638960	dataflow

Table 3.7. Performance latency summary for the CAMUS dataset.

Lastly, Table 3.8 provides FPGA resource utilization data. While the BRAM, URAM, LUT, and FF usages are within reasonable margins, DSP utilization again exceeds 100% per SLR, implying that optimization strategies such as increasing reuse factor or distributing the workload across multiple SLRs are necessary to enable successful deployment on this particular part.

Name	BRAM_18K	DSP	FF	LUT	URAM
Total Used	881	37217	1715812	1682339	0
Available SLR	1344	3072	864000	432000	320
SLR Util. (%)	65	1211	198	389	0
Total Available	5376	12288	3456000	1728000	1280
Overall Util. (%)	16	302	49	97	0

Table 3.8. Utilization estimates summary for the CAMUS dataset. DSP usage exceeding 100% per SLR suggests the design requires additional partitioning or optimization.

3.3 Oxford Pets Dataset

On the Oxford Pets dataset, the Keras model achieved a multiclass Dice score of **0.7665**, while the HLS model achieved **0.7615**. This small reduction in accuracy demonstrates that the HLS model maintains strong segmentation performance after quantization and hardware-specific transformations.

Figure 3.3 shows an example of both model predictions alongside the ground truth. While visually not as impressive as the other two datasets, both predictions still show high overlap with the ground truth mask, with some deviations at the boundaries. Perhaps including the boundary class given in the original dataset would have helped mitigate this issue.

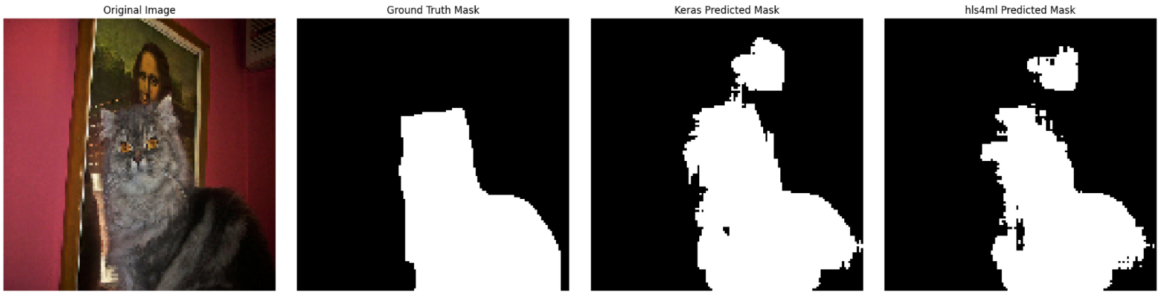

Figure 3.3. Example of the original image, ground-truth segmentation mask, Keras model prediction, and HLS model prediction for the Oxford Pets dataset.

To evaluate inference performance across platforms, Table 3.9 summarizes the latency measurements. The FPGA achieves a latency of 0.932 ms—over 150 times faster than CPU

inference, and significantly faster than both GPU and TPU counterparts, highlighting its potential for low-latency applications.

Platform	**CPU**	**T4 GPU**	**v2-8 TPU**	**FPGA**
Latency (ms)	140.885	91.000	126.142	0.932

Table 3.9. Inference latency comparison of FPGA synthesis report compared to simulation of traditional hardware on Google Colab for the Oxford Pets dataset.

The timing summary shown in Table 3.10 confirms that the design comfortably meets the timing requirements, with an estimated clock period of 3.646 ns against a 5.00 ns target.

Clock	**Target**	**Estimated**	**Uncertainty**
ap_clk	5.00 ns	3.646 ns	1.35 ns

Table 3.10. Performance timing summary for the Oxford Pets dataset.

Further insights into latency are provided in Table 3.11. The latency remains constant at 186,478 cycles (equivalent to 0.932 ms), and the pipeline interval is also consistent, indicating high determinism and throughput.

Latency (cycles)		**Latency (absolute)**		**Interval (cycles)**		**Pipeline Type**
min	max	min	max	min	max	
186478	186478	0.932 ms	0.932 ms	185904	185904	dataflow

Table 3.11. Performance latency summary for the Oxford Pets dataset.

Table 3.12 presents the FPGA resource utilization. While URAM, LUT, FF, and BRAM usage remain within manageable limits, DSP usage exceeds the available resources, suggesting that further partitioning or optimization is needed for full deployment on this particular FPGA part.

Name	BRAM_18K	DSP	FF	LUT	URAM
Total Used	869	36227	1561053	1703503	0
Available SLR	1344	3072	864000	432000	320
SLR Util. (%)	64	1179	180	394	0
Total Available	5376	12288	3456000	1728000	1280
Overall Util. (%)	16	294	45	98	0

Table 3.12. Utilization estimates summary for the Oxford Pets dataset. High DSP usage exceeding 100% indicates a need for a higher reuse factor or further optimization.

Chapter 4
Conclusion

4.1 Discussion of Results

Across one general and two medical imaging datasets, our experiments demonstrated that a modified UNet network can be synthesized into FPGA hardware while maintaining a relatively high accuracy. While the reduced UNet was significantly smaller in size than traditional UNet architectures, the Keras implementation was still able to achieve relatively high accuracy through techniques such as preprocessing, pruning, and hyperparameter tuning. The HLS-model generated by hls4ml was also able to closely match the segmentation performance of a reference Keras model, achieving multiclass Dice scores within approximately 1–2% of its baseline. While the Keras model generally provided slightly higher accuracy, the HLS implementation remained competitive despite having fixed-point arithmetic of 16 integer bits and 6 decimal bits, compared to 32-bit floating point arithmetic used in Keras.

FPGA synthesis on each dataset consistently achieved lower latencies compared to Google Colab's CPU, GPU, and TPU platforms. For instance, on Oxford Pets and BraTS, inference latency dropped to less than 1 ms on the FPGA—over a hundredfold speedup relative to CPU/GPU/TPU inference. The CAMUS dataset also exhibited a major reduction in latency of 8.201 ms, over 13 times faster than the TPU and GPU. While these results are not a direct comparison to the latencies shown in the FPGA synthesis report, they still provide a decent estimation for how well FPGAs can perform when compared to traditional hardware. These results show that deploying deep-learning-based segmentation models on FPGA hardware can offer near real-time performance in resource-constrained or edge settings.

The resource utilization data highlighted both the opportunities and constraints of FPGA implementations. While the BRAM_18K, FF, LUT, and URAMs all stayed under

utilized, the DSP utilizations exceeded 100% when mapped to this particular FPGA part. To remedy this, a part with more DSPs available could be used, or a higher reuse factor could be used during HLS synthesis, which would significantly reduce resource consumption at the cost of latency. Future designs also may need more aggressive optimization strategies (e.g., pruning, quantization, or partitioning across multiple SLRs) in order to have a design that can be deployed on actual FPGA hardware. Nonetheless, the results indicate that it is feasible to synthesize a high-performing UNet onto an FPGA using open-source tools, underscoring the hardware's capability for efficient, low-latency inference in imaging tasks.

4.2 Future Work

While the current results are promising, there are several directions for improvement.

- **Optimizing hls4ml Configuration:** In this paper, we used the "Resource" strategy with a reuse factor of 1 for the FPGA synthesis. Further experimentation with different strategies and reuse factors could strike an optimal balance between resource usage and latency. The high DSP utilization observed on some networks indicates a potential need for higher reuse factor to reduce resource consumption at the cost of latency for this particula FPGA part.

- **Pruning, Quantization, and Advanced Optimization Strategies:** To further reduce resource utilization and improve throughput, advanced techniques such as weight pruning, mixed-precision quantization, and multi-SLR partitioning may be necessary—especially given the high DSP usage observed in this paper. More aggressive pruning can be explored to eliminate redundant weights without significantly impacting accuracy. Likewise, while this work employed a $\langle 16, 6 \rangle$ fixed-point format for FPGA deployment, experimenting with lower bitwidths (e.g., $\langle 8, 4 \rangle$ or even $\langle 4, 2 \rangle$) may yield additional performance gains. Tools like `QKeras` can help maintain model accuracy by introducing quantization during the model training process.

- **Advancing Model Architectures:** As model compression techniques continue to mature, it may become feasible to train and deploy more complex base models without incurring prohibitive resource costs. This opens the door to higher accuracy—especially valuable for challenging segmentation tasks—while still benefiting

from the low-latency inference offered by FPGAs. Current limitations also exist within tools such as hls4ml in the type and size of networks that are able to be accelerated. An example is combining multi-headed attention layers with convolution layers, which is currently not feasible in hls4ml. While these layers can be synthesized on their own, the `IO_stream` setting that is required for convolution layers is currently not supported in multi-headed attention layers [21]. Once these features are implemented, exploring advanced UNet variants such as TransUNet, could yield further improvements without drastically increasing parameter counts.

- **Enhancing Evaluation Metrics:** Beyond the Dice coefficient, future studies could incorporate additional metrics such as the Hausdorff distance and Intersection over Union dicussed in the introduction section to provide a more comprehensive and nuanced assessment of segmentation quality. These complementary metrics can offer deeper insights into model performance, especially for boundary-sensitive or imbalanced datasets.

- **More Direct Evaluation with Physical Hardware:** Although the current comparisons were performed using simulation (FPGA synthesis reports vs. Colab metrics), a direct, head-to-head test with physical hardware (CPU, GPU, TPU, and FPGA) would yield a more precise understanding of real-world throughput and latency. Access to on-premise hardware is needed for these tests, but such an evaluation would be invaluable for clinical or commercial deployment considerations.

- **Real-Time Clinical Integration:** Incorporating the FPGA-based pipeline into clinical workflows would require a full systems-level evaluation, including data transfer overheads, reliability testing, and regulatory considerations. Demonstrating robust, end-to-end performance in real clinical environments will be a pivotal step for broader adoption.

- **Expanding to Additional Imaging Domains:** While this study targeted segmentation tasks across three datasets, future work could incorporate more diverse imaging modalities (such as CT scans or 3D volumes) and pathologies. Evaluating the FPGA approach on larger and more varied datasets would help validate its generalizability and potential impact.

Collectively, these avenues offer opportunities to further enhance the balance of accuracy, latency, and resource usage that FPGA-based designs provide, as well as allow for better generalization to real-life applications. By leveraging emerging U-Net

variants, additional compression techniques, new hls4ml features, and physical hardware, future work can continue to push the boundaries of low-latency, high-accuracy image segmentation on FPGA platforms.

Bibliography

[1] Thea Aarrestad, Vladimir Loncar, Nicolò Ghielmetti, Maurizio Pierini, Sioni Summers, Jennifer Ngadiuba, Christoffer Petersson, Hampus Linander, Yutaro Iiyama, Giuseppe Di Guglielmo, Javier Duarte, Philip Harris, Dylan Rankin, Sergo Jindariani, Kevin Pedro, Nhan Tran, Mia Liu, Edward Kreinar, Zhenbin Wu, and Duc Hoang. Fast convolutional neural networks on FPGAs with hls4ml. *Machine Learning: Science and Technology*, 2(4):045015, July 2021.

[2] Ahmed AL Qurri and Mohamed Almekkawy. Improved UNet with Attention for Medical Image Segmentation. *Sensors*, 23(20):8589, January 2023.

[3] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

[4] Association for Computing Machinery. ACM Code of Ethics and Professional Conduct. https://www.acm.org/code-of-ethics, 2018.

[5] Riadh Ayachi, Yahia Said, and Abdessalem Ben Abdelali. Optimizing Neural Networks for Efficient FPGA Implementation: A Survey. *Archives of Computational Methods in Engineering*, 28(7):4537–4547, December 2021.

[6] Spyridon Bakas, Mauricio Reyes, Andras Jakab, Stefan Bauer, Markus Rempfler, Alessandro Crimi, Russell T Shinohara, Christian Berger, Sungmin Ha, Matthew Rozycki, et al. Identifying the best machine learning algorithms for brain tumor segmentation, progression assessment, and overall survival prediction in the brats challenge. *arXiv preprint arXiv:1811.02629*, 2018.

[7] Jieneng Chen, Yongyi Lu, Qihang Yu, Xiangde Luo, Ehsan Adeli, Yan Wang, Le Lu, Alan L. Yuille, and Yuyin Zhou. TransUNet: Transformers Make Strong Encoders for Medical Image Segmentation, February 2021.

[8] Ben Cohen, Sergey Gleyzer, Cory Hauck, Jim Kowalkowski, Yen-Chen Lin, Thong Papadopoulou, Archana Radhakrishnan, Eric Rauch, Yash Sehgal, Aristeidis Tsaris, et al. Qkeras: Quantization-aware training for deep neural networks. https://github.com/google/qkeras, 2019. Accessed: 2025-04-14.

[9] Jason Cong and Zhiru Wang. Fpga-based accelerator design with high-level synthesis. In *Design Automation Conference*, pages 1–6. IEEE, 2018.

[10] Lee R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[11] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, and Zhenbin Wu. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13(07):P07027–P07027, July 2018.

[12] Abdelrahman Elabd, Vesal Razavimaleki, Shi-Yu Huang, Javier Duarte, Markus Atkinson, Gage DeZoort, Peter Elmer, Scott Hauck, Jin-Xuan Hu, Shih-Chieh Hsu, Bo-Cheng Lai, Mark Neubauer, Isobel Ojalvo, Savannah Thais, and Matthew Trahms. Graph Neural Networks for Charged Particle Tracking on FPGAs. *Frontiers in Big Data*, 5:828666, March 2022.

[13] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P. Carloni, Giuseppe Di Guglielmo, Philip Harris, Jeffrey Krupa, Dylan Rankin, Manuel Blanco Valentin, Josiah Hester, Yingyi Luo, John Mamish, Seda Orgrenci-Memik, Thea Aarrestad, Hamza Javed, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, Sioni Summers, Javier Duarte, Scott Hauck, Shih-Chieh Hsu, Jennifer Ngadiuba, Mia Liu, Duc Hoang, Edward Kreinar, and Zhenbin Wu. Hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices, March 2021.

[14] Nicolò Ghielmetti, Vladimir Loncar, Maurizio Pierini, Marcel Roed, Sioni Summers, Thea Aarrestad, Christoffer Petersson, Hampus Linander, Jennifer Ngadiuba, Kelvin Lin, and Philip Harris. Real-time semantic segmentation on FPGAs for autonomous vehicles with hls4ml. *Machine Learning: Science and Technology*, 3(4):045011, November 2022.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, Cambridge, MA, 2016.

[16] Yulan Guo, Yuxuan Liu, Thanassis Georgiou, and Michael S Lew. A review of semantic segmentation using deep neural networks. *International Journal of Multimedia Information Retrieval*, 11(1):1–21, 2022.

[17] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

[18] Daniel P Huttenlocher, Gregory A Klanderman, and William J Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, 1993.

[19] Institute of Electrical and Electronics Engineers. IEEE Code of Ethics. https://www.ieee.org/about/corporate/governance/p7-8.html, 2020.

[20] Paul Jaccard. The distribution of the flora in the alpine zone. *New phytologist*, 11(2):37–50, 1912.

[21] Zhixing Jiang, Dennis Yin, Yihui Chen, Elham E. Khoda, Scott Hauck, Shih-Chieh Hsu, Ekaterina Govorkova, Philip Harris, Vladimir Loncar, and Eric A. Moreno. Low Latency Transformer Inference on FPGAs for Physics Applications with hls4ml, September 2024.

[22] Zhixing Jiang, Dennis Yin, Elham E. Khoda, Vladimir Loncar, Ekaterina Govorkova, Eric Moreno, Philip Harris, Scott Hauck, and Shih-Chieh Hsu. Ultra Fast Transformers on FPGAs for Particle Physics Experiments, February 2024.

[23] Elham E. Khoda, Dylan Rankin, Rafael Teixeira de Lima, Philip Harris, Scott Hauck, Shih-Chieh Hsu, Michael Kagan, Vladimir Loncar, Chaitanya Paikara, Richa Rao, Sioni Summers, Caterina Vernieri, and Aaron Wang. Ultra-low latency recurrent neural network inference on FPGAs for physics applications with hls4ml. *Machine Learning: Science and Technology*, 4(2):025004, April 2023.

[24] Sarah Leclerc, Erik Smistad, João Pedrosa, Andreas Østvik, Frederic Cervenansky, Florian Espinosa, Torvald Espeland, Erik Andreas Rye Berg, Pierre-Marc Jodoin, Thomas Grenier, Carole Lartizien, Jan D'hooge, Lasse Lovstakken, and Olivier Bernard. Deep Learning for Segmentation Using an Open Large-Scale Dataset in 2D Echocardiography. *IEEE Transactions on Medical Imaging*, 38(9):2198–2210, September 2019.

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[26] Logic Fruit Technologies. Fpga design: Architecture and applications. https://www.logic-fruit.com/blog/fpga/fpga-design-architecture-and-applications/, 2023. Accessed: 2025-03-26.

[27] Matthieu Martel. Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics. *Formal Methods in System Design*, 35:265–278, 12 2009.

[28] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(7):3523–3542, 2021.

[29] Anouar Nechi, Lukas Groth, Saleh Mulhem, Farhad Merchant, Rainer Buchty, and Mladen Berekovic. FPGA-based Deep Learning Inference Accelerators: Where Are We Standing? *ACM Transactions on Reconfigurable Technology and Systems*, 16(4):1–32, December 2023.

[30] Modise Kagiso Neiso, Nicasio Maguu Muchuka, and Shadrack Maina Mambo. FPGA-based Implementation of a Resource-Efficient UNET Model for Brain Tumour Segmentation. *International Journal of Advanced Computer Science and Applications*, 15(1), 2024.

[31] Jennifer Ngadiuba, Vladimir Loncar, Maurizio Pierini, Sioni Summers, Giuseppe Di Guglielmo, Javier Duarte, Philip Harris, Dylan Rankin, Sergo Jindariani, Mia Liu, Kevin Pedro, Nhan Tran, Edward Kreinar, Sheila Sagear, Zhenbin Wu, and Duc Hoang. Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml. *Machine Learning: Science and Technology*, 2(1):015001, December 2020.

[32] Eriko Nurvitadhi, Gokul Venkatesh, Debbie Marr, and Russell Huang. Can fpgas beat gpus in accelerating next-generation deep neural networks? *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*, pages 5–14, 2017.

[33] Ozan Oktay, Jo Schlemper, Loic Le Folgoc, Matthew Lee, Mattias Heinrich, Kazunari Misawa, Kensaku Mori, Steven McDonagh, Nils Y Hammerla, Bernhard Kainz, et al. Attention u-net: Learning where to look for the pancreas. In *International Conference on Medical Imaging with Deep Learning (MIDL)*, pages 1–10, 2018.

[34] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and CV Jawahar. Cats and dogs. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3498–3505. IEEE, 2012.

[35] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kimon Constantinides, John Demme, Hadi Esmaeilzadeh, Jason Fowers, Kalin Gopal, J Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[36] Benjamin Ramhorst, Vladimir Loncar, and George A. Constantinides. FPGA Resource-aware Structured Pruning for Real-Time Neural Networks, December 2023.

[37] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation, May 2015.

[38] Leonardo Rundo, Changhee Han, Yudai Nagano, Jin Zhang, Ryuichiro Hataya, Carmelo Militello, Andrea Tangherloni, Marco S. Nobile, Claudio Ferretti, Daniela Besozzi, Maria Carla Gilardi, Salvatore Vitabile, Giancarlo Mauri, Hideki Nakayama, and Paolo Cazzaniga. USE-Net: Incorporating Squeeze-and-Excitation blocks into U-Net for prostate zonal segmentation of multi-institutional MRI datasets, July 2019.

[39] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[40] Ahmad Shawahna, Sadiq M. Sait, and Aiman El-Maleh. FPGA-based Accelerators of Deep Learning Networks for Learning and Classification: A Review. *IEEE Access*, 7:7823–7859, 2019.

[41] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):1–48, 2019.

[42] Suraj Subramanian. Pruning neural networks. https://towardsdatascience.com/pruning-neural-networks-1bb3ab5791f9, 2019. Accessed: 2025-03-26.

[43] Eric Topol. High-performance medicine: the convergence of human and artificial intelligence. *Nature Medicine*, 25:44–56, 2019.

[44] Yaman Umuroglu, Nicholas Fraser, Giulio Gambardella, Michaela Blott, Philip H.W. Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74. ACM, 2017.

[45] Stylianos I Venieris and Christos-Savvas Bouganis. Fpga convnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47. IEEE, 2016.

[46] Stylianos I Venieris and Christos-Savvas Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[47] World Health Organization. Ethics and governance of artificial intelligence for health. https://www.who.int/publications/i/item/9789240029200, 2021.

[48] Xilinx. Ultraram: High density, ultra-fast embedded memory. https://www.xilinx.com/products/technology/ultramram.html. Accessed: 2025-04-14.

[49] Xilinx Inc. Vivado design suite user guide: High-level synthesis. https://docs.xilinx.com, 2023. UG902 (v2023.1).

[50] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[51] Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. Unet++: A nested u-net architecture for medical image segmentation. *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*, pages 3–11, 2018.

[52] Zongwei Zhou, Md Mahfuzur Rahman Siddiquee, Nima Tajbakhsh, and Jianming Liang. UNet++: A Nested U-Net Architecture for Medical Image Segmentation, July 2018.