A TRANSFER LEARNING APPROACH TO OBJECT DETECTION ACCELERATION FOR EMBEDDED APPLICATIONS

by

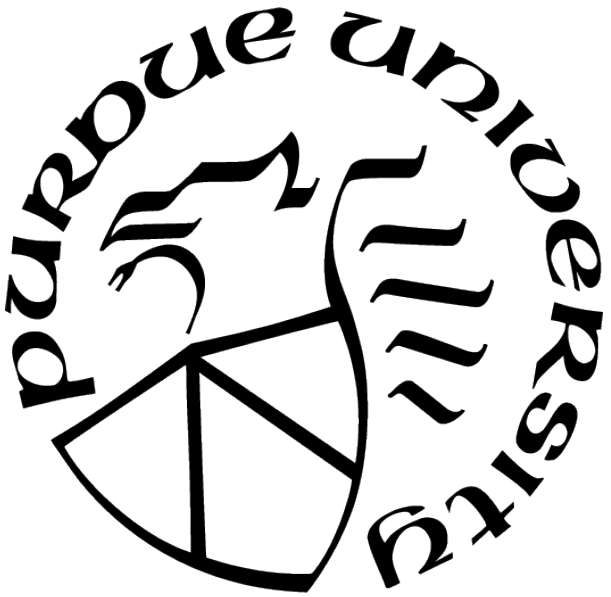
Lauren M. Vance

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science



Department of Electrical and Computer Engineering

Indianapolis, Indiana

August 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Lauren Christopher, Chair

Department of Electrical and Computer Engineering

Dr. Brian King

Department of Electrical and Computer Engineering

Dr. Maher Rizkalla

Department of Electrical and Computer Engineering

Approved by:

Dr. Brian King

This work is dedicated to my father, Tim Vance, for being both my role model and my biggest fan. Thank you for everything.

ACKNOWLEDGMENTS

My deepest thanks to my advisor Dr. Lauren Christopher for being an inspiring mentor, a great teacher, and for much encouragement during the constant coding struggles. I learned a lot from you during this process and hope to have the opportunity to do the same for others one day.

I would also like to thank the rest of our research group members for being wonderful to work and interact with. Special thanks to Ashley Dale and Gregory Vaughn for lending your knowledge when I needed it and for being so helpful and kind along the way.

I am also grateful to Sherrie Tucker for all of her hard work ensuring that ECE graduate students have all the information they need to succeed. I have met few people more kind and helpful than Sherrie, and her guidance made this entire process much easier.

Thank you to Dr. Maher Rizkalla for serving on my thesis committee. I greatly enjoyed learning from you during my time at IUPUI and appreciate the energy and enthusiasm you bring to the classroom.

I am greatly indebted in particular to Dr. Brian King, ECE Department Chair at IUPUI, for your encouragement and support in applying to the IUPUI University Fellowship which funded this work. I would have never considered myself capable of being awarded such an honor, and it was a gentle reminder to never underestimate myself. Thank you for your guidance and advice during my time here.

Finally, I extend my deepest gratitude to all of my friends and family for the support and the joyful company during a whirlwind of a collegiate career. To my parents, thank you for always being there for me in ways big and small. To my partner Luke, I truly could not have done any of this without your support; through all of the ups and downs in this process (including a pandemic and an untimely broken leg) you are a constant beam of light.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AI	Artificial Intelligence
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DNN	Deep Neural Network
DPU	Deep-learning Processor Unit
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
GPU	Graphics Processing Unit
HDL	Hardware Descriptive Language
ILSVRC	Imagenet Large Scale Visual Recognition Challenge
IoT	Internet of Things
IP	Intellectual Property
PL	Programmable Logic
PS	Processor System
mAP	Mean Average Precision
ML	Machine Learning
MPSoC	Multi-Processor System on Chip
MS-COCO	Microsoft Common Objects in Context
YOLO	You Only Look Once

ABSTRACT

Deep learning solutions to computer vision tasks have revolutionized many industries in recent years, but embedded systems have too many restrictions to take advantage of current state-of-the-art configurations. Typical embedded processor hardware configurations must meet very low power and memory constraints to maintain small and lightweight packaging, and the architectures of the current best deep learning models are too computationally-intensive for these hardware configurations. Current research shows that convolutional neural networks (CNNs) can be deployed with a few architectural modifications on Field-Programmable Gate Arrays (FPGAs) resulting in minimal loss of accuracy, similar or decreased processing speeds, and lower power consumption when compared to general-purpose Central Processing Units (CPUs) and Graphics Processing Units (GPUs). This research contributes further to these findings with the FPGA implementation of a YOLOv4 object detection model that was developed with the use of transfer learning. The transfer-learned model uses the weights of a model pre-trained on the MS-COCO dataset as a starting point then fine-tunes only the output layers for detection on more specific objects of five classes. The model architecture was then modified slightly for compatibility with the FPGA hardware using techniques such as weight quantization and replacing unsupported activation layer types. The model was deployed on three different hardware setups (CPU, GPU, FPGA) for inference on a test set of 100 images. It was found that the FPGA was able to achieve real-time inference speeds of 33.77 frames-per-second, a speedup of 7.74 frames-per-second when compared to GPU deployment. The model also consumed 96% less power than a GPU configuration with only approximately 4% average loss in accuracy across all 5 classes. The results are even more striking when compared to CPU deployment, with 131.7-times speedup in inference throughput. CPUs have long since been outperformed by GPUs for deep learning applications but are used in most embedded systems. These results further illustrate the advantages of FPGAs for deep learning inference on embedded systems even when transfer learning is used for an efficient end-to-end deployment process. This work advances current state-of-the-art with the implementation of a YOLOv4 object detection model developed with transfer learning for FPGA deployment.

1. INTRODUCTION

Machine learning has proven to be an effective solution to a variety of intriguing computer vision tasks in recent years. The full self-driving capabilities of autonomous vehicles are closer than ever to revolutionizing the roadways [1]–[3] and medical imaging tasks can predict and diagnose disease with a precision unmatched in humans [4], [5]. Convolutional neural networks (CNN) have been the computer vision solution of choice since approximately 2012 [6], and they have shown to provide great accuracy when applied to computer vision tasks. However, with great accuracy comes great power: improvements in model accuracy have proven to be inversely related to other performance metrics such as speed and power consumption, and their computational complexity is to blame for this [7]. The current state-of-the-art (SOTA) hardware configuration for CNN deployment (performing inference on an input image or video) is the general-purpose graphics processing unit (GPU). Current available GPUs are highly specialized to handle CNN architecture processing to reduce speed and increase accuracy during deployment, but at the cost of high power consumption.

Due to this contrasting nature of performance to power, there are many limitations in extending recent CNN advancements to potential embedded computer vision tasks. Embedded systems differ from traditional general-purpose computer systems by their compact size, which lends to major restrictions in memory and power. For example, unmanned aerial vehicles (UAV) could be useful for implementing computer vision tasks at a birds'-eye-view, but they have very low power and weight requirements to maintain flight that high-performance machine learning models cannot meet with an embedded general-purpose processor. There is a growing market for embedded and Internet of Things (IoT) devices with unique needs that are not suitable for current machine learning deployment hardware configurations. High performance GPUs are too power-hungry and not portable enough for these applications, but smaller embedded processors do not have the necessary resources to host and deploy machine learning models. Eventually, accuracy will no longer be the ideal performance metric for CNNs if the system hardware cannot keep up with the increasing computational demand and decreasing inference speed during deployment.

Currently available strategies of resource optimization for embedded CNN deployment include strategic model design, network compression, and hardware choice [8]. Resource-efficient model design takes optimization to the source by reducing the size and complexity of CNN models for compatibility with the limitations of embedded devices. One such example, called You Only Look Once (YOLO), performs inference very quickly even on embedded devices by streamlining the prediction network to reduce computation time [9].

Field-Programmable Gate Arrays (FPGAs) are a resource with great potential for embedded deep learning tasks. FPGAs are low-power programmable devices that can replace GPUs in deployment. In theory, this approach can maintain or improve inference speeds of GPU deployment with minimal loss in accuracy while satisfying the low power, weight, and size requirements of embedded systems.

In this work, transfer learning was used to re-purpose a pre-trained YOLO object detection model to be used for detection and classification of images from a dataset of 5 classes. Transfer learning is a model development method that has shown recent success in reducing training time and overall computational resource requirements without sacrificing accuracy [10]–[12]. If the desired object classes to be detected during deployment are present in the object classes of the pre-trained model, then it is advantageous to utilize the weights from the pre-trained model and perform inference on the data immediately, skipping the training process altogether. However, it is more often the case that a pre-trained model has been trained on very similar information to the data intended to be used for inference but needs a little more fine-tuning to create more specific output classifiers. In this case, transfer learning can help lessen the impact of training while allowing for greater customization to the desired task. The knowledge previously obtained by the model is used as a starting point for learning new tasks, which greatly reduces training time and can result in greater accuracy. This is done by ”freezing” (can no longer be updated during training) the weights on all layers of the pre-trained model except for the output classification layer. The frozen layers then become the convolutional base and the new classifier is trained from scratch to output custom classes.

The custom model was then deployed for inference using GPU, CPU, and FPGA hardware to compare performance metrics such as accuracy, speed, and power. The model performance was also compared to that of the pre-trained model. The primary contribution of this research

shows that transfer learning can be used to develop networks that maintain the accuracy of traditional deep CNNs in FPGAs.

The following sections of this chapter discuss the differences in deploying a model on different hardware configurations and an overview of state-of-the-art object detection model architectures. The subsequent chapter describes the underlying FPGA hardware through the lens of CNN architectures. This work also provides an in-depth comparison of the advantages and disadvantages of using FPGAs for deep learning inference in contrast to GPUs. The remaining chapters will describe the experimental setup of this work and present and discuss the performance results for each deployment configuration.

1.1 The Role of Hardware in Deep Learning

Regarding inference with CNNs, there are two main sources of speed degradation: multiply-accumulate (MAC) operations and large memory requirements [13]. MAC operations form the basis of convolution, but are costly to implement in hardware. Furthermore, deep CNNs achieve great accuracy but at the cost of a large number of parameters, which greatly increase the number of required matrix multiplications. Additionally, more parameters require more memory storage. Deep CNNs are so large that associated memory must be stored in multiple places on the hardware architecture, leading frequent memory accesses that also contribute to latency during inference [14]. There have been many recent advancements to optimize CNN processing speeds by designing specialized hardware for handling these specific operations and requirements, motivated by the need to increase inference speeds for practical computer vision solutions.

1.1.1 Current SOTA for CNN Deployment Hardware

In 2006, Chellapilla et al. achieved the first CNN implementation on a GPU, and found that speeds were up to 4 times faster than central processing units (CPU) [15]. The success of GPUs in deep learning applications is primarily due to the ability to perform the convolutional layer (MAC) operations in parallel, rather than sequentially like CPUs. The difference lies in the design strategy of the underlying hardware architecture. CPUs are divided into several

cores, which can each take on a single task at a time. Combined with fast cache memory for retrieval, this makes them very effective for serial processing tasks. In contrast, GPUs are composed of hundreds of cores, all of which are dedicated to processing the same single task. This narrowed focus allows the GPU to dedicate all of its resources to convolutional operations, increasing inference throughput dramatically. These design features have led the GPU to become the current hardware platform of choice for most machine learning deployment applications

The success of GPUs over CPUs introduced the need for specialized hardware to handle the computational complexity of current CNNs and allow development of better, faster networks in the future. However, the advantages of the GPU come at a cost that may be disadvantageous for some applications: they require a lot of power to run and often cost into the thousands of dollars for a quality unit, which is insurmountable for tasks that require low power and have limited resources to dedicate. In response, other specialized hardware is available for computer vision tasks that improves upon some aspects where the GPU lacks. Among these choices is the FPGA, which has several advantages over the GPU to be discussed in Chapter 2. In short, the FPGA can be designed to parallelize CNN operations for optimal deployment speed, while providing other advantages that are impossible with GPUs.

1.2 Neural Networks for Object Detection

Object detection algorithms have two main tasks: detect and classify. To achieve this, most object detection model architectures are comprised of two main components: a CNN "backbone" for feature extraction on the input image and a "head" to output the detection/classification prediction. In practice, there are two dominant categories for how to implement the head of an object detection model [16]. The first is two-stage detection, in which the tasks of detection and classification are divided into two separate phases. The first phase compiles a set of proposed regions of interest (RoI) within the image, then passes each of these proposals to a CNN which performs and outputs classification for each detected object. Two-stage object detectors typically boast high localization and classification accuracy,

but are slow during inference. One example is Mask R-CNN [17], which is a progression of earlier iterations R-CNN [18], Fast R-CNN [19], and Faster R-CNN [20]. In [17], it is mentioned that Mask R-CNN achieves SOTA accuracy results but at the expense of inference speeds that are not considered to be real-time.

In contrast, single-stage object detectors are designed for speed. A single CNN is applied to the entire image at once to output bounding boxes and class predictions simultaneously. This approach results in much faster outputs for real-time inference with minor sacrifices in accuracy. As mentioned previously, this feature of single-stage detector model design is the reason they are favored for embedded deep learning applications with limited resources to improve speeds for real-time inference [8]. The following sections will focus on the YOLO single-stage object detection architecture. A brief overview of the YOLO development history is first presented, followed by a detailed explanation of the YOLOv4 architecture to be deployed in this work.

1.2.1 You Only Look Once: Real-Time Object Detection

YOLO was first introduced in May 2016 by Redmon et al. as a "new approach to object detection" [9]. It proposed an improvement upon existing object detection architectures by simplifying the task with single-stage detection.

The YOLO model was inspired by the ability of humans to quickly and accurately come to several conclusions about the nature of an image with a single glance. Likewise, YOLO was designed to receive an input image and use a single CNN to simultaneously detect and classify objects with a single glance. The CNN backbone architecture of YOLO was inspired by GoogLeNet [23] and contains 24 convolutional layers plus two fully connected layers which output the prediction. Its performance did not match that of the human visual system of course, but it showed to make great strides toward achieving similar real-time inference on visual input.

The main results of YOLO's debut were two-fold. First, inference speeds were dramatically increased when compared to other SOTA object detection algorithms (earning the real-time designation by boasting less than 25 ms latency during video streaming). Quantitatively,

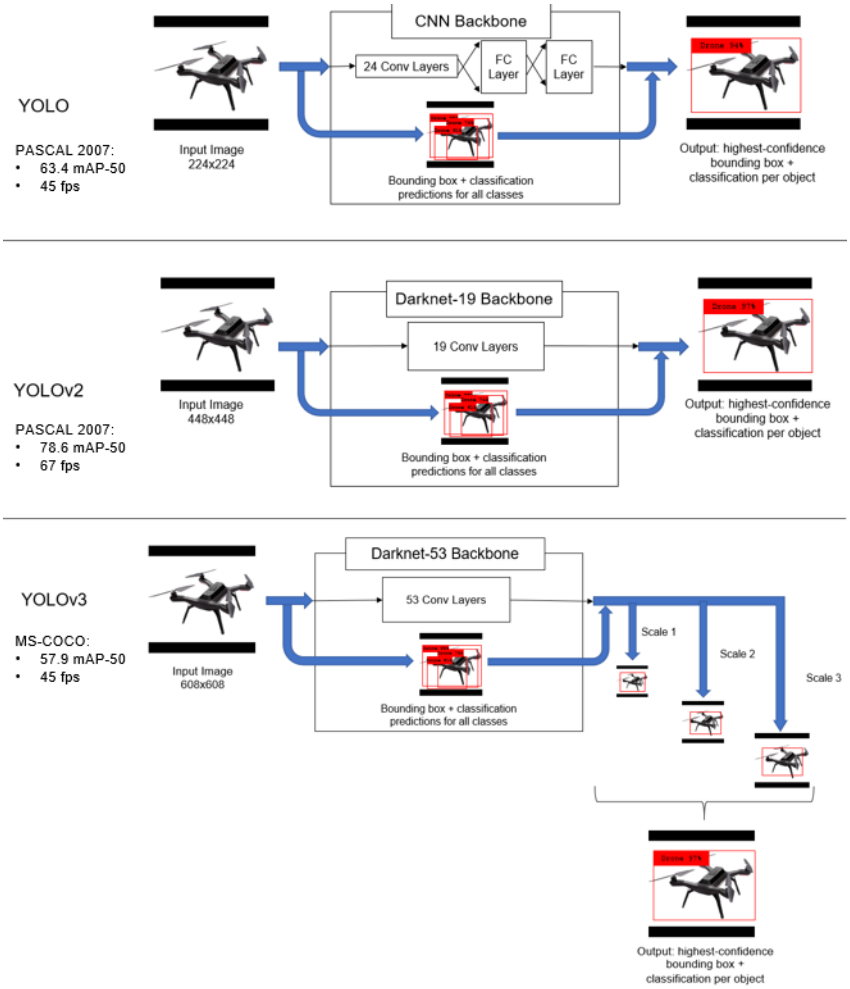


Figure 1.1. Model Architecture for YOLO [9], YOLOv2 [21], and YOLOv3 [22]. Each model is a single-shot object detector, but underlying changes to the CNN backbone allow for improvements in speed and accuracy with each iteration.

YOLO reported a throughput of 45 frames-per-second (FPS) for detection on the PASCAL VOC 2007 dataset [24], a vast improvement on the lethargic R-CNN family of architectures at 0.5-7 FPS. Second, these increased speeds did not come at a major expense to accuracy. In fact, YOLO boasted a mean average precision (mAP) of 63.4 on the PASCAL VOC 2007 dataset in contrast to the 16.0-21.1 from other real-time architectures available at the time. This was still slightly behind the less-than-real-time R-CNN family (at approximately 70 mAP), but a vast improvement that promoted YOLO to fast *and* practical. The YOLO approach also gained an additional advantage of lower false positives on backgrounds, since

the single-stage approach (whole image) allows it to add contextual information to each object's identification. It showed an improvement on background error in comparison to Fast R-CNN by 8.85%.

There were a few drawbacks to this first YOLO approach: while it achieved the goal of being very fast, it struggled with precise object localization compared to other models. For example, detection on the PASCAL VOC 2007 dataset with YOLO reported a impressive 10.4% increase in localization error compared to Fast R-CNN. It also was still lacking in overall detection accuracy compared to SOTA object detection models of the time. Subsequent redesigns improved upon these drawbacks to design models that increased both speed and accuracy with each iteration.

In December 2016, Redmon et al. introduced YOLOv2 [21] as an improvement to YOLO's shortcomings and a faster, more accurate model overall. The first set of major changes were aimed at improving localization accuracy while maintaining the classification accuracy achieved with YOLO. Rather than aiming for a larger, deeper network (as is typical with computer vision network improvements [25], [26]), other fine-tuning strategies were used to improve learning while maintaining a simple network to ensure the detection is still incredibly fast. A few example strategies used were adding batch normalization to all of the convolutional layers and using higher input image resolution for classification (448x448 vs. 224x224). The second set of major changes aimed for increased speed, and resulted in the design of a new CNN backbone called Darknet-19 with 19 convolutional layers and 5 maxpooling layers. The new network decreases the number of convolutional layers from the original YOLO model, which resulted in fewer MAC operations for even better performance. It was found that YOLOv2 now boasted 78.6 mAP on the PASCAL VOC 2007 dataset while maintaining real-time inference speed, a vast improvement upon its predecessor and now a contender against two-stage models in both speed and accuracy. However, YOLOv2 still struggled with detection of small objects.

YOLOv3 [22] premiered from Redmon et al. in 2018 with some changes from YOLOv2, including the development of the Darknet-53 CNN backbone for feature extraction and the addition of 3 prediction heads. Darknet53 contains 53 convolutional layers as opposed to 19, but achieved better accuracy than Darknet-19. Darknet-53 was also designed to maximize

the number of floating-point operations per second (FLOPs), which GPUs are specialized to handle at the hardware level (see section 1.1.1). This results in better utilization of the hardware for improved deployment speed despite its larger size. The 3 prediction heads predict bounding boxes for the input image at three different scales using a method similar to feature pyramid networks (FPN) [27], which lead to improvements in small object detection to curb the main weakness of YOLOv2. Overall, these changes resulted in major improvements in inference speeds, but still trailed just slightly behind other models in accuracy. Figure 1.1 illustrates a comparison of the model architectures of the first three YOLO iterations.

1.2.2 YOLOv4

After the release of YOLOv3, the YOLO project was taken over by Bochkovskiy et al., who developed YOLOv4 in early 2020 [28]. This newest iteration was shown to yet again improve upon its predecessor, increasing mAP by 10% and speed by 12% to become a model that is fully optimized for real-time detection and parallel computations. As of this writing, YOLOv5 has been recently released; however, YOLOv4 [28] is the object detector of choice for this work since it is more stable to work with and still maintains SOTA performance.

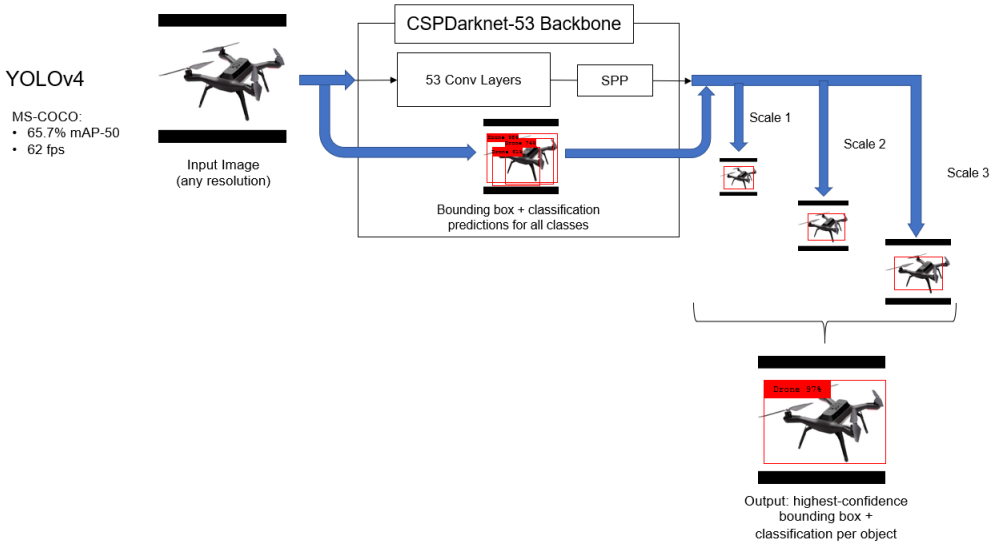


Figure 1.2. Model Architecture for YOLOv4 [28]. It consists of the CSPDarknet53 CNN backbone, an SPP module for fixed-length outputs, and the same head used in YOLOv3.

YOLOv4 improved upon YOLOv3 in several ways (see Figure 1.2). First, several new features were added to the CNN backbone for accuracy improvements. The backbone was renamed CSPDarknet53, named for the Darknet53 backbone architecture used in YOLOv3 and the addition of Cross-Stage-Partial-connections (CSP) [29], one of the features used to improve CNN accuracy. Another feature used for improvement was the addition of a Spatial Pyramid Pooling (SPP) [30] module over the CSPDarknet53 backbone. When added to the top of the last convolutional layer of a CNN, SPP is used to generate fixed-length outputs to feed into the head of the model, which eliminates the need for fixed-size input images and improves CNN performance. Finally, the YOLOv4 model uses the same YOLOv3 head described above following the SPP module for outputting predictions.

The Darknet53 backbone makes YOLOv4 a very favorable choice for real-time inference on platforms that are specialized for parallel processing without much sacrifice in accuracy. This work will focus on utilizing transfer learning with YOLOv4 by using weights from a different pre-trained model as a starting point and re-training only the head to output predictions for a new model with a different set of classes. This approach will allow for significantly decreased training time and computational resources to produce a model of slightly lower, but comparable, accuracy compared to training from scratch. This decreases the overall burden that training has on the entire model development lifecycle and is a great tool for embedded computer vision tasks.

2. OBJECT DETECTION DEPLOYMENT ON FPGAs

Today, there are many computer vision tasks that benefit from CNNs, but there are also many options to consider for the best deployment setup. Developing a machine learning model has two phases: training and inference (see Figure 2.1). Training is on the back-end of the development process and can usually be completed on a typical desktop or cloud computer with the use of a GPU as an effective setup. In this environment, the main goal is to achieve high accuracy to ensure deployment of the model on new images is successful, so speed is not of much concern and the setup does not need to be portable. However, for the inference phase of deployment, unique considerations for each use-case often create a trade-off between the most important considerations: speed, latency, power, and accuracy [31]. Today, the most common hardware to utilize for inference are CPUs and GPUs, with a major trade-off of speed versus cost and power. However, neither of these options are very portable for use in embedded (on-the-field) system applications, as more compact embedded hardware generally lacks the resources to employ these complex architectures.

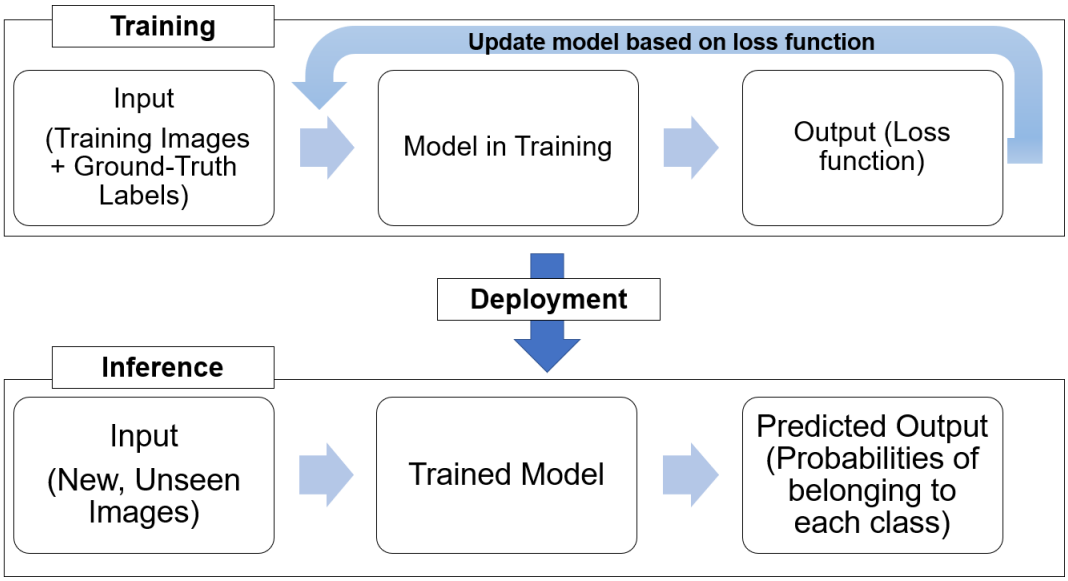


Figure 2.1. Relationship Between Training, Inference, and Deployment in CNN Development

This is where FPGAs come in: they are smaller than traditional CPU/GPU setups so they can be utilized for embedded applications, can utilize parallel processing and high-bandwidth

memory for real-time inferencing, are reprogrammable to ensure flexibility as needs change, and are more energy-efficient than GPUs [31]. The main problem with FPGA utilization is due to the complexity of the hardware environment preparation. Despite their flexibility, they are very complicated devices and there are not many programmers with the knowledge to take advantage of this. For FPGA utilization to be realistic, there needs to be a way for software developers to utilize the power of FPGAs without needing to understand the underlying hardware details.

Xilinx introduced an answer to this problem in 2019 called Vitis AI, which is a development platform that was specifically designed to provide a high-level way to port machine learning programs in software to utilize the hardware on FPGA. Vitis AI can be used to deploy custom machine learning models or ready-to-use models from their library onto supported FPGAs. The goal of the platform is to abstract the details of the underlying hardware for software engineers and programmers with little to no FPGA experience to make FPGAs more attractive for deep learning inference applications. Therefore, embedded computer vision tasks may become much simpler to develop without sacrificing performance. Vitis AI is also free to download and use, can be easily set up for use with supported Xilinx boards or configured for use with a custom board, and most of the software is available as open source.

There are many published works deploying pre-trained models for FPGAs and evaluating their performance, but work is lacking in custom models that achieve higher accuracy for specific tasks. Transfer learning helps bring these models to fruition without the need to dedicate hours or even days to training on large amounts of data while preserving adequate accuracy. The combination of transfer learning and FPGA deployment could be a relatively quick and low-power solution for embedded computer vision tasks from end-to-end.

2.1 How Acceleration Works in FPGAs

The digital-signal processor (DSP) block is used on FPGAs for MAC operations. The DSP block is only able to operate with fixed-point numerical representations for greater computational efficiency [31], whereas typical CNN weights are stored in 32-bit floating-point (FP32) format for higher precision. Deploying a CNN on FPGA therefore requires a

pre-processing step called quantization to compress the model size by converting the FP32 weights to fixed-point integer weights, typically at a width of 8 bits (INT8). Quantization is accomplished by mapping the minimum/maximum value of FP32 weights to the minimum/maximum of the INT8 range. It has been demonstrated that FP32 precision is not required to maintain the same level of accuracy [32], [33], and this gives FPGAs the ability to show speed and power improvement over GPUs in CNN deployment. Reducing the precision of the weights to INT8 makes addition and multiplication operations less computationally costly, increasing inference speeds and decreasing network size and power consumption [31].

The board used in this work is the Zynq UltraScale+ MPSoC ZCU102 from Xilinx [34]. It is a general-purpose evaluation board, which features the Zynq UltraScale+ XCZU9EG MPSoC (multiprocessor system-on-chip). This MPSoC ontains (i) a processing system (PS), with the Arm Cortex-A53 64-bit quad-core PCU and the Arm Cortex-R5 dual-core processor, and (ii) a programmable logic (PL), where the Deep-learning Processor Unit (DPU) is implemented with direct connections to the PS [35]. The DPU is an intellectual property (IP) block that handles typical deep learning operations, such as pooling and convolutions, while the PS handles all other program execution aspects, such as pre-/post-processing instructions. In a way, the relationship of the DPU to the PS is analogous to that of the GPU and CPU: the former is specialized for parallelization and dedicating many resources to a single task, and the latter is better at serialization and quickly handling tasks that do not require extensive resources.

2.2 Why FPGAs for CNNs?

As briefly introduced, the use of CNNs in computer vision has provided many breakthroughs but at a great cost to time and computing resources. In this section, the cause of these challenges in context of CNN architectures are explored in more detail. These details will then be used to illustrate why the use of FPGAs can be more beneficial for CNN deployment than typical general-purpose processor setups in the context of embedded computer vision tasks.

2.2.1 Challenges of CNN Deployment

In Alexnet [6], only 8 layers were used to achieve groundbreaking results at the time. With further development came deeper and deeper networks: in 2015, the Visual Geometry Group (VGG) model was developed and achieved a top-5 test error rate of 6.8 percent with their 19-layer network [26], greatly improving upon the results in [6]. Unfortunately, this increase in accuracy with model depth did not continue indefinitely and it was found that accuracy eventually degrades after a certain number of layers [17]. The ResNet family of CNNs was established in 2016 to solve this issue by introducing "shortcut connections" to the CNN model [25]. It was shown in [25] that the ResNet architecture was able to circumvent performance degradation and allow for continued increase in accuracy with increased depth for models containing 34, 50, 101, and 152 layers, and outperformed the current SOTA models to win the ILSVRC in 2015. Most popular CNNs used in image classification and object detection backbones today utilize 50-100 layers.

The increase in SOTA network size leads to the creation of very large models that require additional speed and power considerations during inference. Both GPUs and FPGAs can create better speed efficiency during inference with parallelization of MAC operations in each layer, and model architectures such as YOLOv4 have focused on streamlining the process for real-time detection on such hardware. However, these models can also require a lot of power to process during inference, as they require frequent memory accesses and utilize high-precision arithmetic [36]. Additionally, SOTA CNN models are rapidly evolving, and the hardware used for deployment may not be flexible enough to accommodate the types of layers and structures that may appear in better networks of the future. Some deep CNN applications may require or benefit from continued training in the field (known as Deep Reinforcement Learning) which creates additional demand for resources and the need for flexibility and portability in a CNN deployment system [37]. The challenges above are simply expensive and impractical for GPUs to handle.

2.2.2 FPGA Solutions in Comparison to General-Purpose Processors

There are many FPGA features that are suitable for handling the challenges described above. There is a wide variety available to suit specific needs, with low-cost options for simple CNNs or higher-cost options for more complex CNNs which require more processing power. FPGAs are also more flexible than general-purpose processors; they can be designed and programmed at the hardware level to be optimal for specific networks and can be quickly adapted for future CNN structures that are yet to be achieved. FPGAs are often used for military applications because they typically have a longer lifespan than general-purpose processors, providing another vector for future stability.

System design with FPGAs also allows for a smaller footprint than general-purpose processors, because the same piece of silicon can host several IP blocks to integrate much, if not all, of the required hardware for the entire deep learning application. This is particularly useful in image processing applications, which can process input in real-time with the use of cameras or other image capture hardware connected to the same system as the CNN accelerator and post-processing operations, minimizing latency (see Figure 2.2). In comparison, CPUs and GPUs are more limited in this regard, as they do not possess the same ability to integrate other IP into their applications for seamless functionality.

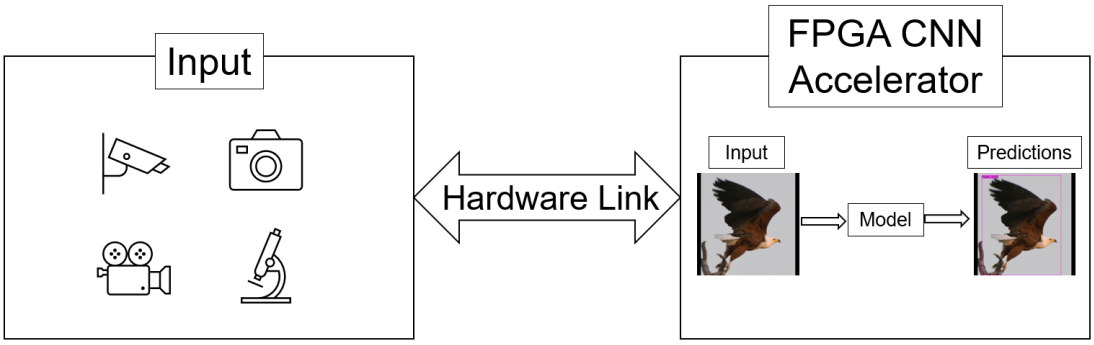


Figure 2.2. Footprint of Possible FPGA Deep Learning Applications

Finally, FPGAs provide better speed and power profiles than typical general-purpose processors. It is well-known that GPUs outperform CPUs for inference because they excel

at performing the data operations required by the convolutional layers of CNNs; FPGAs also excel at this, so they outperform CPUs as well. Additionally, hardware programming on the FPGA allows for parallelism; this can accelerate CNN processing, particularly in the convolutional layers where each convolution operation can be done in parallel. FPGAs are also well-known for their power efficiency, while GPUs are not. Several studies have shown that FPGAs have proven to be capable of comparable or better speedup and performance than GPUs while consuming 10-28 percent less power [38]–[40].

2.3 Literature Review

There has been much recent work done to implement a variety of deep learning algorithms on FPGAs and evaluate the performance. Most of the work done so far uses hardware description languages (HDLs) and design tools (such as Vivado by Xilinx) to program the FPGAs for deep learning inference and subsequently deploy the pre-trained models for inference and performance evaluation. Some research focuses on the hardware and makes conclusions on the relative performance of FPGAs to GPUs, and some focuses on the software to propose algorithm adjustments to speedup FPGA inference or compare performance between existing deep learning models. However, there is not much work illustrating an end-to-end development of a custom model utilizing strategies such as transfer learning for subsequent deployment and evaluation on relatively small datasets. This approach is more realistic for embedded applications and is worth investigating further.

An example of relatively early work in 2017, [41] implements an LSTM model for speech recognition on an FPGA using pruning and quantization for compression. They noted that they used LSTM in their work because it is "the most complex" recurrent neural network model for speech recognition due to high computational complexity, memory footprint, and power consumption. The focus of their work was in reducing the high memory footprint to speed up inference. Their results reported that their hardware architecture, titled Efficient Speech Recognition Engine (ESE), performed 43-times faster than a CPU and 3-times faster than a GPU while maintaining 40-times and 11.5-times higher energy efficiency (power consumption) than the CPU and GPU, respectively.

A similar approach was proposed in [42] to make the case for quantizing model weights from floating-point to fixed-point to speed up inference in neural networks. The inspiration for their work is neural network design for mobile devices with limited resources, and although there are mobile neural networks available for use, they still use floating-point weights and calculations. The authors suggest quantizing these pre-trained mobile neural networks and moving the computations to hardware (FPGA) and methods for how to design neural network for improved performance with fixed-point calculations. They also present their hardware design and implementation, reporting a processing speed of 150 frames per second (FPS); they do not compare their performance to any other architecture.

The work presented in [43] explores how to modify an existing object detection algorithm (tinyYOLO) to run more efficiently on an FPGA. This was done by (i) making several modifications to the network architecture and (ii) again using quantization to compress the model. They compared the performance of their model for inference on FPGA, CPU, and GPU for speed, accuracy, and power, and found that the GPU was still faster than the FPGA (while the FPGA was 44.9-times faster than the CPU), but the GPU consumed 18.9-times more power than the FPGA, an important consideration for embedded systems. Finally, the model was slightly less accurate than both the CPU and GPU, but only by approximately 3-4 percent, respectively. This work illustrates that there can still be a variety of outcomes when deploying a deep learning model on FPGA, but the results are still comparable and the great decrease in power consumption is very favorable for embedded systems applications.

There are also many examples comparing the performance of several SOTA object detection algorithms on FPGA implementations. The results in these studies can serve as a baseline for future performance evaluations. In [44], an early open-source tool from Xilinx called Python Productivity for ZYNQ (PYNQ), which is intended to ease the hardware design process of Xilinx ZYNQ SoCs, was used to deploy several object detection models on a ZCU104 FPGA. The models evaluated were SSD with MobileNetv1 backbone, SSD with Inceptionv2 backbone, Faster-RCNN with Inceptionv2 backbone, and SSD with MobileNetv1 and FPN backbone. The performance of each model was evaluated for latency, accuracy, and ease of implementation. They found that across all evaluation metrics, the SSD with Inceptionv2 model was the most suitable for their setup.

Another study [45] took this a step further and compared performance of object detection algorithms SSD, Faster R-CNN, and YOLO when deployed for inference on FPGA on two different hardware setups. Their strategy to improve performance was to add an Intel Neural Compute Stick (NCS) to the hardware setup alongside the FPGA, which is a co-processor used to implement and accelerate DNNs. The performance of each of the three object detection algorithms (FPS, runtime, and accuracy) was evaluated for both hardware setups: FPGA with and without NCS. They found that the addition of the NCS to the FPGA improved results in all three categories across all three models. Across the three algorithms on the NCS configuration, the fastest runtime went to YOLO, the highest FPS went to SSD, and the highest accuracy went to Faster-RCNN, illustrating each selection's strengths and weaknesses.

Finally, there are a few examples of published works which utilize software development stacks like Vitis AI to aid in the deployment of existing SOTA computer vision models on FPGAs. One such example in [46] illustrates an end-to-end implementation of an autonomous driving system on FPGA. These tools allow for integration of the DPU for deep learning deployment on the FPGA without the need for hardware design knowledge. There are currently still many constraints for using these, such as limited model and device support, but overall results have been promising. Another good example is given in [47], which used these development tools to implement object detection and semantic segmentation on FPGA for future use in an Advanced Driver Assistance System (ADAS). Their system was deployed successfully, and the algorithms were evaluated against CPU and GPU, and they found that both speed and power consumption were improved with only a slight decrease in accuracy. These findings use models trained on large, popular datasets to show that FPGA usage in embedded machine learning development can be practical with little sacrifice with the aid of software development tools. This work takes that a step further by using these pre-trained models as a starting point, then using transfer learning to re-purpose that knowledge for use on smaller, custom datasets for FPGA deployment with the aid of the development tools. The result is a practical machine learning solution that is suitable for embedded device applications without sacrificing the positives of GPU deployment.

3. METHODOLOGY

This chapter begins by describing the experimental setup of this work. The details of the software and hardware configurations used to prepare and deploy the models and the image dataset used during training and testing will be discussed. Lastly, the performance metrics that are used to evaluate the test results will be defined, along with a detailed explanation of how they were obtained for each configuration.

3.1 Experimental Setup

Two tests were performed to analyze and compare the performance of a YOLOv4 object detection model that was developed using transfer learning on a custom 5-class dataset. The first test deployed the model for inference on a GPU with the use of Google Colaboratory [48] and a laptop CPU (Intel Core i7). The second test deployed the model for inference on an FPGA with the aid of the Xilinx Vitis AI development environment. For each deployment test setup, inference was performed on 100 images and the performance metrics of each configuration were recorded and tabulated for comparison. An extra 150 images (30 per class) were added to the test set to be used for the accuracy tests to get a better evaluation of the average classification accuracy.

The model was developed with the use of the Darknet repository [49], which hosts the code used to develop and deploy YOLOv4 models. The configuration file used to create the neural network had to be modified slightly from the base YOLOv4 network to (i) modify the output layers for a 5-class dataset and (ii) be compatible for deployment on the DPU. The DPU does not support the activation functions used in the base model, so they were replaced by the Leaky Rectified Linear Unit (ReLU) activation function. Leaky ReLU is a modification of the ReLU activation function (Equation 3.1), in which the function defines negative inputs as a very small linear component of the input rather than zero (Equation 3.2). This modification solves a common issue with ReLU called ”dying ReLU” [50] by giving

an output (albeit very small, but not zero) for negative values so that the associated neurons are not deactivated.

$$f(x) = max(0, x) \tag{3.1}$$

$$f(x) = max(0.01 * x, x) \tag{3.2}$$

Another required modification was the number of kernels used in two of the max pooling layers of the base model. The maximum kernel size supported by the DPU is 8, but the base model used 9 and 13, so these were changed to 6 and 8 to be compatible with the DPU. The code used to prepare the model and configuration file for FPGA deployment was modified from [51].

After configuration, the model was trained for approximately 6800 iterations on the data described in 3.1.2 below. According to [49], the general recommendation for number of training iterations can be found with Equation 3.3.

$$\text{iterations} = 2000 * (\text{number of classes}) \tag{3.3}$$

For the five classes in this case, the model would be expected to train for 10000 iterations. However, due to transfer learning the model achieves an mAP-50 of 92.64% at around 6800 iterations. Figure 3.1 shows the mAP-50 and loss values (using the Focal Loss function [52]) recorded for each iteration of the training process; the process was stopped at around 7500 iterations as it was clear the mAP was leveling off.

The deployment process for each test setup is illustrated in Figure 3.2. After training completes, the model can be immediately deployed on the GPU and CPU with the use of [49] deployment code and the test images. For FPGA use, the deployment process requires 3 main steps after training (see the orange block in Figure 3.2):

1) Model Conversion: quantization requires a Tensorflow graphdef file that has had its weights frozen (no longer trainable or modifiable). This requires the Darknet weights file produced during training to first be converted to a

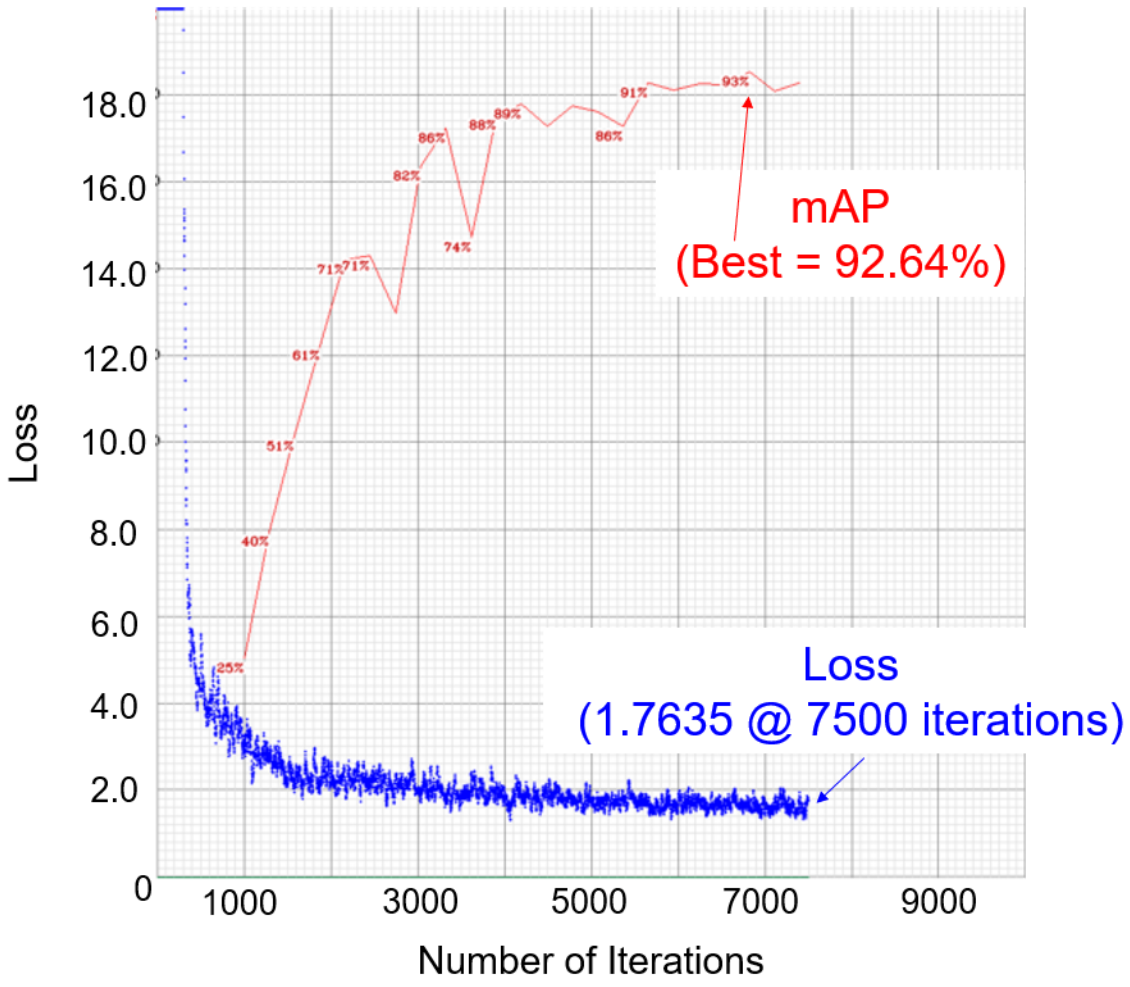


Figure 3.1. Model mAP-50 and Loss versus Number of Training Iterations

Keras-style `h5` file, then converted into a protobuf (`pb`) frozen graph.

2) Quantization: the frozen graph is used as input to the quantizer along with a calibration dataset. The calibration dataset is a subset of the images used to train the model, used to calibrate the conversion process from FP32 weights to INT8 weights. The output is a `pb` file that represents the model with INT8 weights that is compatible for DPU deployment.

3) Compilation: the quantized model is used as input to produce an `xmodel` file, which contains the necessary hardware information for mapping the model to the FPGA for deployment. The `xmodel` file generated during compilation is used in tandem with the deployment software to deploy inference on the desired data.

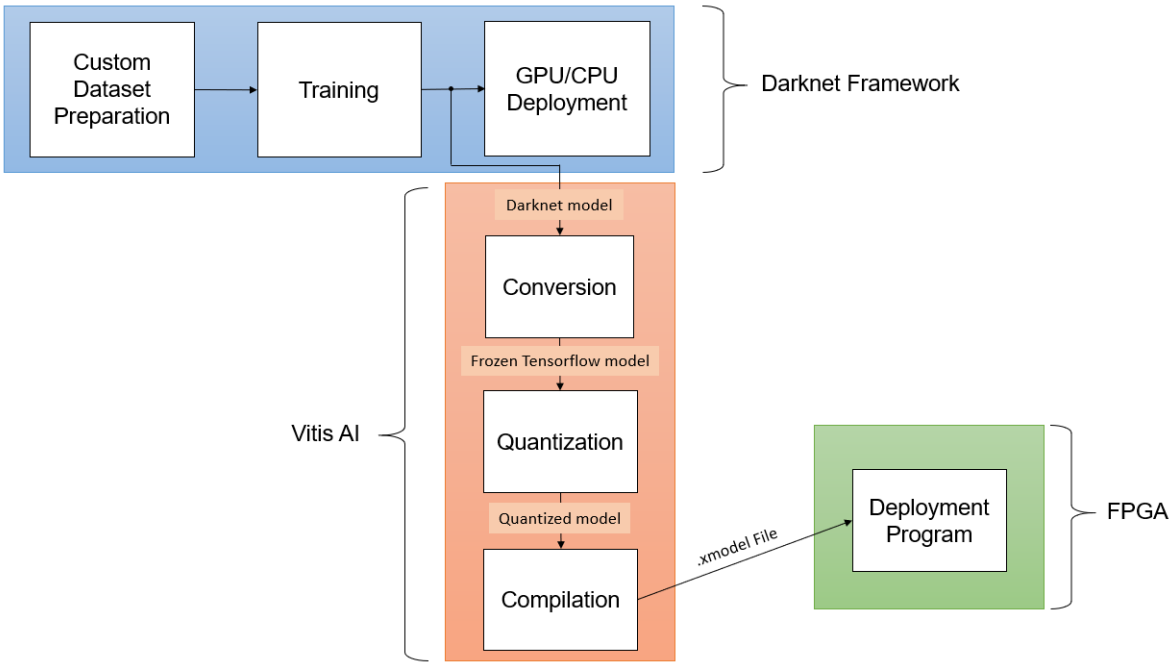


Figure 3.2. Experimental Workflow

Once the `xmodel` file has been generated, it is copied onto the board and utilized by the deployment program to perform inference (from [53]). The deployment code is a C++ program that performs image pre-processing, inputs the image to the model for detection and classification, then performs post-processing to output bounding boxes, class predictions, and probabilities.

3.1.1 Hardware

The GPU used for inference in Google Colaboratory was the NVIDIA Tesla T4. This GPU was designed for high-performance inference, with 320 Tensor cores and 2560 CUDA cores for a total of 2880 cores to utilize for parallel processing operations. Tensor Cores operate concurrently alongside CUDA cores to enable mixed-precision processing designed to further increased inference speeds on neural networks [54]. This GPU also lists a single precision (FP32) performance of 8.1 tera-FLOPs (floating-point operations per second). With a relatively small form-factor and a maximum power rating of 70 Watts, the Tesla T4 boasts significantly lower power consumption than other typical GPUs employed for computer vision

tasks [55]. The model developed for this work can be deployed for inference on the GPU directly after training with the use of the Darknet deployment code.

The ZCU102 was chosen for this work because (i) it was already available for use and (ii) it is explicitly supported by Vitis AI for the most efficient deployment process that does not require any additional hardware programming. Booting the ZCU102 from the system image allows for interfacing via either the UART (Universal Asynchronous Receiver/Transmitter) serial port or via the Ethernet interface/SSH service of the host computer. Deployment on the FPGA requires extra processing steps compared to the GPU. The host computer is responsible for all steps illustrated in Figure 3.2. After the model is compiled, the output files can be transferred to the FPGA for deployment; the ZCU102 utilizes three DPU cores during the deployment process. The resulting FPGA hardware test setup can be seen in Figure 3.3.

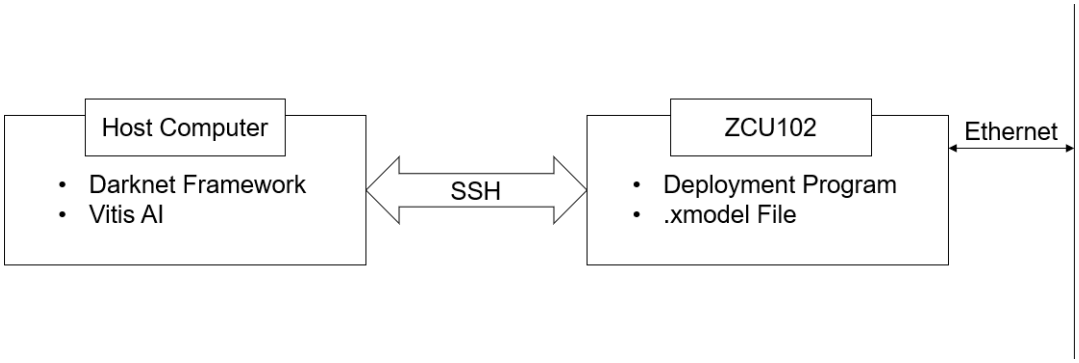


Figure 3.3. FPGA Test Setup Hardware Block Diagram

3.1.2 Data

A custom dataset was used to analyze the performance of each of the models described above. It consists of 5 classes: eagle, glider, kite, plane, and quadcopter. The dataset contains a mix of real-world photographs and virtual-world (computer-generated) images of each object; more information on this dataset can be found in [56]. The real-world images were used for testing because that is a more practical application of object detection, but the virtual-world images were used to train the YOLOv4 model during transfer learning.

Table 3.1 illustrates in detail how the dataset was used for this work, resulting in a split of approximately 82% train and 21% validation.

Table 3.1. Train and Test Images per Class

Class	Train	Test
Eagle	1629	200
Glider	1015	200
Kite	289	200
Plane	432	200
Quadcopter	1323	200
Total	4688	1000

The YOLOv4 model used in this test setup was pre-trained on the Microsoft Common Objects in Context (MS-COCO) dataset [57], with 80 different object classes. Released in 2015, the goal was to provide image training data that showcased less "iconic" view of objects to provide more robust data for computer vision tasks. They argue that most available object images are too perfectly arranged in profile at or near the center of the image, with no occlusion or competing background noise. The pre-trained model is useful for utilizing transfer learning as described in Chapter 1: the features learned from several of the 80 object classes in the MS-COCO dataset are similar enough to the objects in the five-class dataset that they can be frozen and re-purposed. The Darknet backbone of the pre-trained model was configured with the same considerations described in Section 3.1 for compatibility with the ZCU102 FPGA.

3.2 Performance Metrics

When the YOLOv4 model runs inference on an input image, it returns a class label (from Table 3.1), a confidence score (between 0 and 1) corresponding to a probability that the detected object belongs to the predicted class, and a set of coordinates which correspond to a rectangular bounding box for all detected objects. The model assigns confidence scores (0 to 1) to all available class labels on which it has been trained, and the prediction assigned to each object is the one which has the highest confidence score. The inference process requires

the input image to traverse through the model, and for reasons previously discussed this step is generally the bottleneck of the deployment process.

Three performance metrics were used to evaluate the models: accuracy, inference speed, and power consumption. The rest of this chapter will discuss how these metrics were evaluated, what they mean, and how each of them were used for comparison.

3.2.1 Accuracy

There are several methods for evaluating model accuracy in deep learning. Some examples are classification accuracy, mAP, and F1-score. Classification accuracy refers to the simple percentage of correct predictions over the total number of predictions, where prediction refers to the classification of the object detected during inference. This metric has been criticized as being misleading for certain applications due to an imbalance in class occurrences during inference. Due to this, other metrics were developed for a more precise evaluation of model accuracy.

mAP is used to quantify the accuracy of the predicted bounding box by calculating the Intersection over Union (IoU) of the ground-truth box and the predicted box. IoU is found by Equation 3.4 below, where $A_{overlap}$ is the area of overlap between the two boxes and A_{total} is the total area occupied by the two boxes (see Figure 3.4). It returns a value between 0 and 1 which corresponds to the percentage of overlap between the two boxes. IoU is also paired with a threshold, which is used to determine whether the prediction is a True Positive, True Negative, False Positive, or False Negative; for example, mAP-50 uses a IoU threshold of 0.5 (50% overlap) to classify a prediction as a True Positive. These assignments are used to find two values called Precision and Recall, which are shown in Equations 3.5 and 3.6. These two values are plotted as a Precision-Recall curve and the area beneath the curve corresponds to the Average Precision (AP) of each class; the mean AP over all classes results in the mAP (from 0 to 1). This is a useful method to use during training because the ground-truth of each object's bounding box has already been given in order to train the model. However, for

testing accuracy on new images, the ground-truth information is not necessarily given and would have to be manually embedded for each test image to calculate the mAP.

$$IoU = \frac{A_{overlap}}{A_{total}} \tag{3.4}$$

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \tag{3.5}$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \tag{3.6}$$

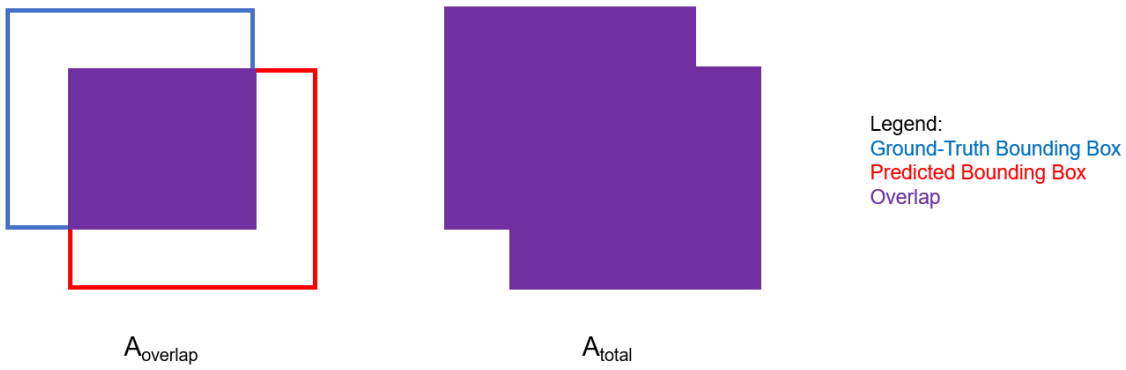


Figure 3.4. Illustration of IoU for Detection Accuracy

The F1-score method also utilizes Precision and Recall by the equation shown in 3.7. However, unlike mAP the number of True/False Positives/Negatives in Equations 3.5 and 3.6 are obtained from the classification outputs of the detected objects (like Classification Accuracy) rather than from IoU. The F1-score was developed to create a balance between precision and recall in cases where there is an uneven class distribution, to solve the major issue with Classification Accuracy.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.7}$$

The number of class occurrences in the test set used for deployment has already been balanced with an equal number of images per class, so Classification Accuracy is not going to

be a misleading metric to use in this case. If those numbers were off-balance (for example, if dramatically more eagles are witnessed in a test environment than quadcopters), one of the other two metrics may be more suitable for measuring model accuracy. For all tests performed in this setup, the top-1 (greatest confidence score) classification of each test image was recorded and the Classification Accuracy was averaged across all classes for each of the three hardware configurations. Note that the main objective for this work was not to explicitly find the best accuracy of each, but rather to observe how well accuracy is retained from the model that uses floating-point weights (GPU and CPU) to the model that has had its weights quantized to fixed-point (FPGA). This process is expected to result in a slight loss in accuracy [51] due to the nature of the process as discussed in Chapter 2.

3.2.2 Speed

The speed of the models was evaluated as the average (mean) time in milliseconds (ms) it takes to perform inference on a single image. This value can also be converted to FPS by Equation 3.8, where the number of frames is equal to one for a single image.

$$\text{FPS} = \frac{\text{Number of Frames}}{\text{Processing Time in Seconds}} \tag{3.8}$$

The GPU and CPU deployment code outputs a run-time for inference on each image by default (prediction only, excluding any image pre-/post-processing), and this time was averaged across all 100 test images to find the mean inference time. The FPGA deployment code can evaluate and output the mean inference run-time for all 100 test images as both end-to-end (E2E) run-time (pre-processing, DPU, and post-processing; see Section 3.1) and DPU run-time. The number of DPU threads (1-8) to use during deployment can also be selected for desired FPGA performance. First, a thorough comparison of inference speeds observed with each number of threads on the FPGA was performed. Then, one of the results was selected for comparison to the CPU run-time, GPU run-time, and the COCO pre-trained model run-time on the FPGA for the same number of threads.

3.2.3 Power

The final evaluation metric for this test setup was the power consumption of the hardware used for each test setup while performing inference on the 100-image test set. The purpose of this metric is to compare power consumption during model deployment in consideration of the strict power constraints for embedded systems. Power in this case is defined by Equation 3.9, where P is the power in Watts, V is the voltage in Volts, and I is the current in Amps used by the hardware during inference.

$$P = V * I \tag{3.9}$$

The GPU power consumption from the Google Colaboratory test setup was recorded with the aid of a tool called Weights and Balances [58] which integrates with the deployment environment to output GPU power consumption over the time the program is running. The evaluation program was initiated prior to starting inference on the 100-image test set and ended after the last image was evaluated. The power consumption is then reported as the maximum value recorded during this time period and the idle power recorded when inference was not running. Likewise, the power consumption of the FPGA was recorded during the entire inference period for the 100-image test set with the use of [59]. The program outputs the power consumed by the PL, PS, and total power (PL + PS) at one-second intervals during the test. The PL consumes the most power during this process as it is the source of MAC operations during inference. The power consumption was also evaluated while varying the number of DPU threads used. DPU multi-threading can be used for higher performance (FPS) but greater power consumption [60], so there is a trade-off for each of these values depending on the application. The power consumption was then reported as the maximum value recorded during this time period (for each number of threads) and the idle power, and compared to that of the GPU deployment.

4. RESULTS

In this chapter, the results of the experimental setup described in Chapter 3 are presented. First, a sample of the most accurate outputs for each configuration are shown, including the bounding boxes overlaid on the images and the object classifications. Then, the performance will be evaluated according to the three performance metrics discussed in Section 3.2. Finally, the chapter concludes with a discussion that analyzes the results in the context of embedded model deployment and highlights the advantages of FPGAs for such tasks in comparison to the other hardware considered.

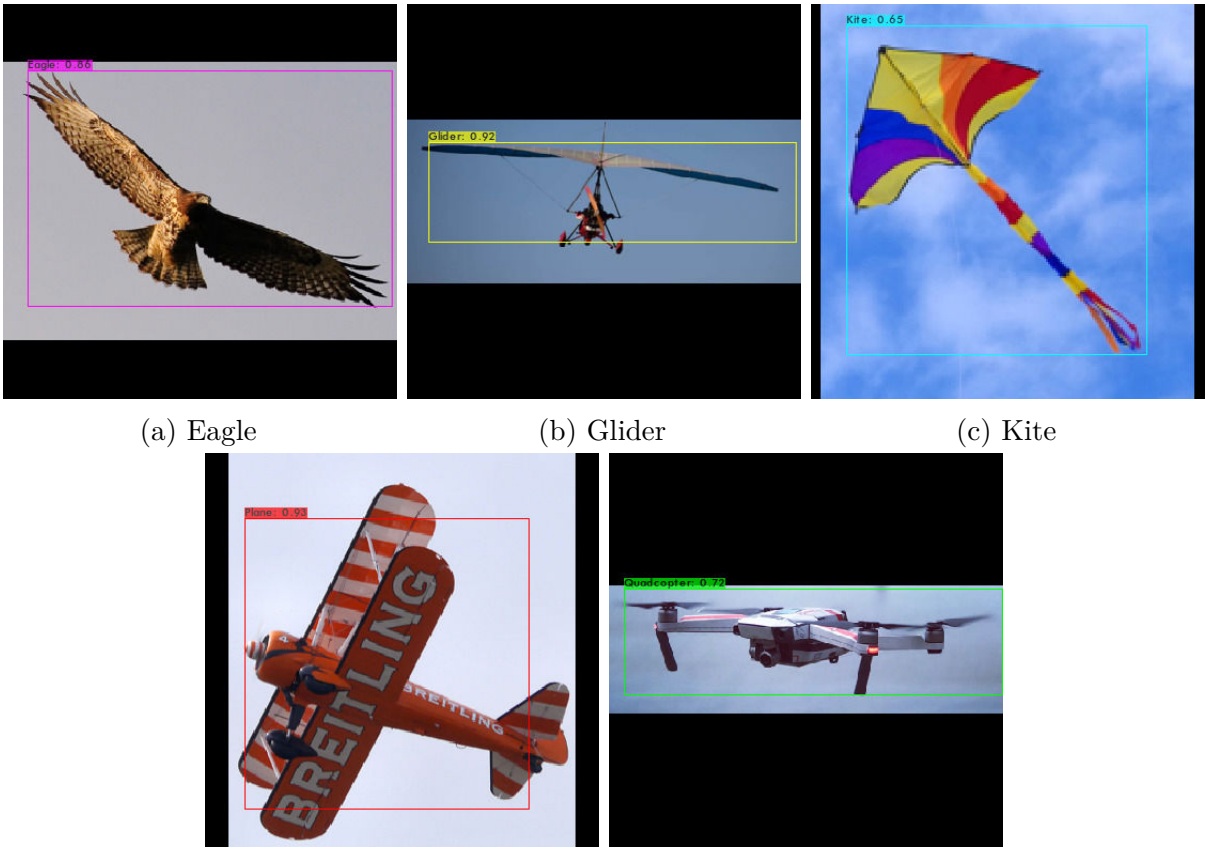
4.1 Output

Because the CPU and the GPU both utilize the same model and deployment setup, the outputs for each are identical, so they are presented together in this section (the main difference between the two is deployment speed, and this will be discussed in the following section). Figure 4.1 shows a sample of outputs (one per class) obtained after inference with the transfer-learned model. The top-1 classification and its corresponding confidence score are included with the bounding box that outlines the detected object. Additionally, the color of the bounding box corresponds to the class of the detected object.

Similar to the GPU/CPU, the FPGA outputs a bounding box that is colored according to the predicted classification of the detected object (as shown in Figure 4.2). However, the deployment code does not embed the classification and confidence score into the bounding box on the output image, so this information is located in Table 4.1.

Table 4.1. Sample of FPGA YOLOv4 Outputs: Classification and Confidence Scores for Figure 4.2

Image	Classification	Confidence
a	Eagle	0.84
b	Glider	0.68
c	Kite	0.52
d	Plane	0.87
e	Quadcopter	0.44



(a) Eagle (b) Glider (c) Kite

(d) Plane (e) Quadcopter

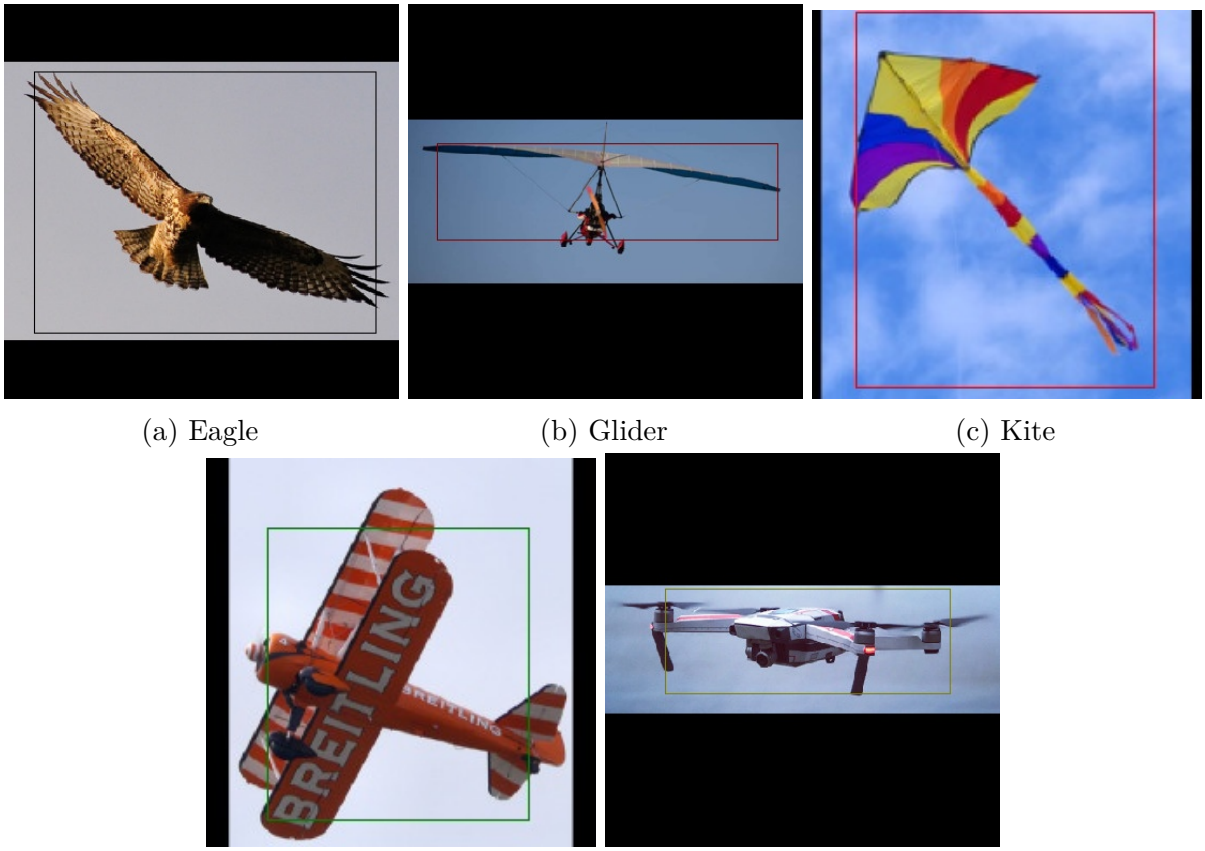
Figure 4.1. Sample of GPU/CPU YOLOv4 Outputs

The bounding boxes for both outputs do fairly well at object localization, but the GPU seems to do slightly better at this than the FPGA. The GPU also tends to return slightly higher confidence scores than the FPGA, which may be the result of weight quantization.

4.2 Performance

4.2.1 Accuracy

The classification results for each model is summarized in Table 4.2. For the purposes of computing Classification Accuracy, the data in Table 4.2 is reduced to two prediction categories per class: Correct or Incorrect. This information was then used to find the Classification Accuracy for each of the model deployment tests by dividing the number of "Correct" classifications by the total number of images tested for all five classes. These five



(a) Eagle (b) Glider (c) Kite

(d) Plane (e) Quadcopter

Figure 4.2. Sample of FPGA YOLOv4 Outputs: Bounding Boxes

values were then used to compute an average Classification Accuracy across all five classes. The results are shown in Table 4.3. It is observed that the Classification Accuracy does drop during conversion from the floating-point weights (GPU/CPU) to the fixed-point weights (FPGA), although the drop in average Classification Accuracy is not enough to be significant.

4.2.2 Speed

First, model inference speed was evaluated on the FPGA with varying number of threads to analyze the change in performance. These values were cross-referenced with the DPU utilization (%) to select the thread count with the best relative performance for further comparison. The DPU utilization is an important metric because it contributes to the bulk of the inference run-time. To illustrate this, Table 4.4 shows the breakdown of the model

Table 4.2. Predicted Classification by Class for Each Test Setup

Ground Truth	Prediction	CPU	GPU	FPGA
Eagle	Eagle	50	50	49
	Not Eagle	0	0	1
Glider	Glider	49	49	43
	Not Glider	1	1	7
Kite	Kite	16	16	11
	Not Kite	34	34	39
Plane	Plane	44	44	41
	Not Plane	6	6	9
Quadcopter	Quadcopter	31	31	37
	Not Quadcopter	19	19	13

Table 4.3. Average Classification Accuracy on 100-Image Test Deployment

Class	CPU	GPU	FPGA
Eagle	1.00	1.00	0.98
Glider	0.98	0.98	0.86
Kite	0.32	0.32	0.22
Plane	0.88	0.88	0.82
Quadcopter	0.62	0.62	0.74
All	0.76	0.76	0.72

average inference speed in ms/image when one thread is used. E2E speed refers to how long it takes for the entire inference process to run from pre-processing to post-processing in all locations on the board. The DPU speed shows the run-time of the steps that are handled by the DPU, and everything else from E2E is handled by the CPU. As 96.7% of the E2E speed, the DPU is clearly contributing to the bulk of the deployment latency.

Table 4.4. FPGA Deployment Speed Breakdown

Location	Speed (ms/image)
E2E_MEAN	73.220
DPU_MEAN	70.794
CPU_MEAN	2.426

The relationship of thread count to speed and DPU utilization is shown in Figure 4.3. The graph shows the speed leveling off rather quickly once three threads are used, and this observation corresponds with the number of DPU cores available for utilization. The utilization was only recorded for 1-5 threads in Figure 4.3 after observing the performance leveling off. From these observations, the 3-thread deployment option was chosen for further comparison to the other hardware setups.

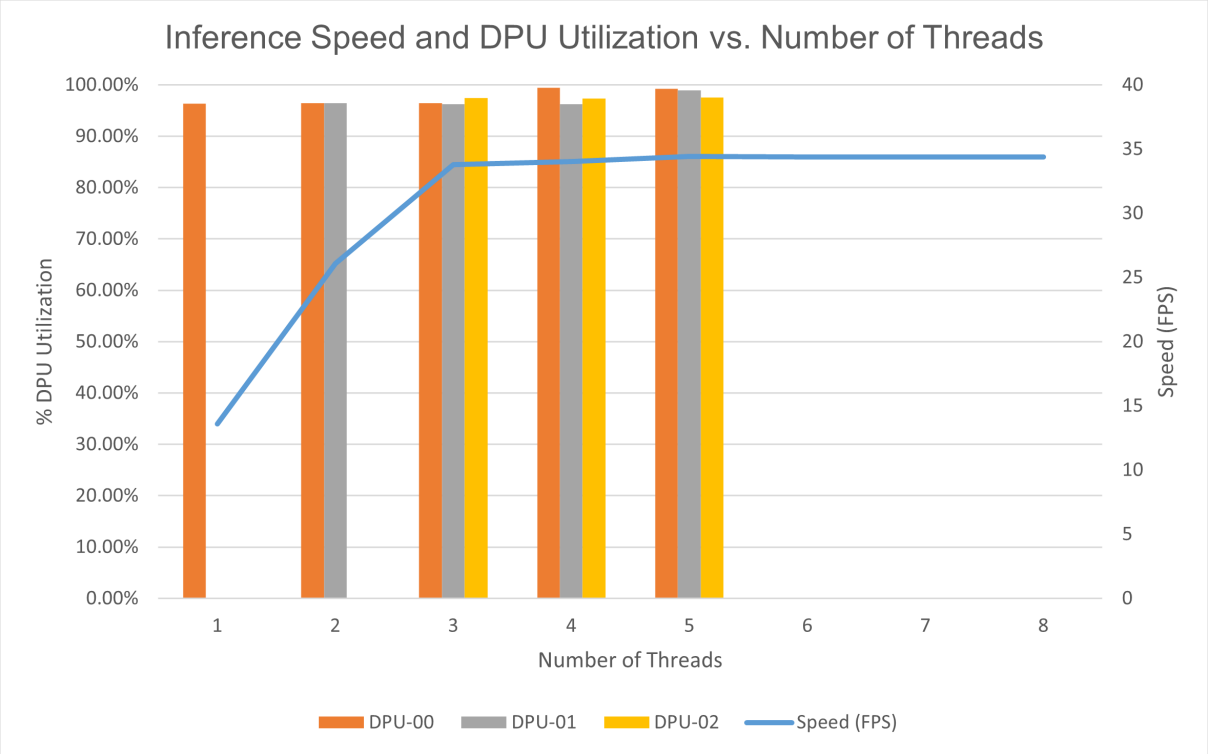


Figure 4.3. Comparison of Speed in FPS and DPU % Utilization to Number of DPU Threads varied from 1 to 8

The average inference speed per image is shown in Table 4.5. Upon observing these values, it is clear that using the CPU for inference is simply impractical for any application that aims for real-time performance, at over ten seconds per image. The other three results are above or near real-time inference speeds of ≥30 FPS. The FPGA model with transfer learning resulted in the best inference run-time of all configurations, at 7.74 FPS faster than the GPU. Further, when comparing the model pre-trained on 80 COCO classes and the transfer-learned model trained on 5 classes, the speedup is 0.3 ms/image. This result illustrates a slight advantage of transfer learning during deployment in which there are fewer model output layers for more specific detection tasks and a slightly faster run-time per image as a result.

Table 4.5. Average Inference Speed

Unit	CPU	GPU	FPGA (TL, 3 Threads)	FPGA (COCO, 3 Threads)
ms/image	10502.30	38.41	29.61	29.91
FPS	0.095	26.03	33.77	33.43

4.2.3 Power

The power consumption during GPU deployment of the 100-image test set is shown in Figure 4.4. Numerical values for the idle and maximum power during this period are also given in Table 4.6. The idle power of the GPU was found at the beginning of the recorded outputs. The maximum power was found at the highest peak of the recorded outputs. The power values seem to average between 25-30 W during the entire run-time.

Table 4.6. Comparison of Power Consumption of GPU and FPGA During Deployment of Transfer-Learned Model on 100-Image Test Set

Total Power (W)	**GPU**	**FPGA**
Idle	9.96	0.142
Maximum	36.3	1.43

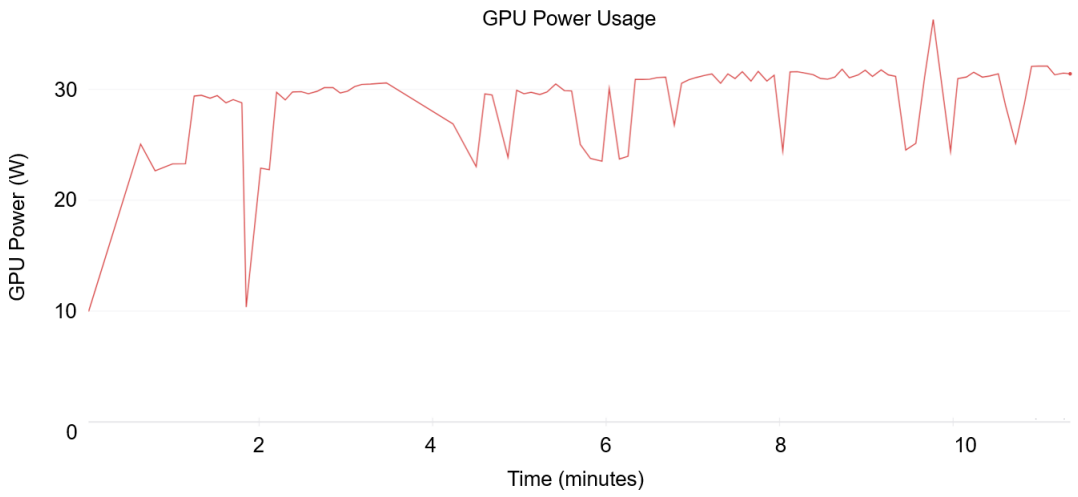


Figure 4.4. GPU Power Usage During Inference for 100-Image Test Set

In contrast, the FPGA power consumption during deployment of the 100-image test set is shown in Figure 4.5 and summarized in Table 4.6. As discussed in Chapter 3, the graph also includes the power output for varying DPU thread count (from 1 to 5) versus the resulting change in FPS. As expected, the power output and FPS are both proportional to the amount of processing required by increased thread count, leveling off around 5 threads. This shows that a real-time deployment ($\geq$30 FPS) can be achieved by this model on the FPGA when greater than 2 DPU threads are used, in exchange for a small but increased power consumption. Finally, it is obvious from Table 4.6 that the power consumption is greatly reduced during deployment on the FPGA in comparison to the GPU, even with increased DPU thread count.

4.3 Conclusion

The results discussed in this chapter solidify three main takeaways: (i) accuracy does not significantly degrade during quantization of a transfer-learned model, (ii) transfer learning allows for real-time inference speeds and contributes to a speedup when a more specific detection task is utilized versus a general pre-trained model, and (iii) the power consumption of FPGA deployment is dramatically less than that of the GPU deployment.

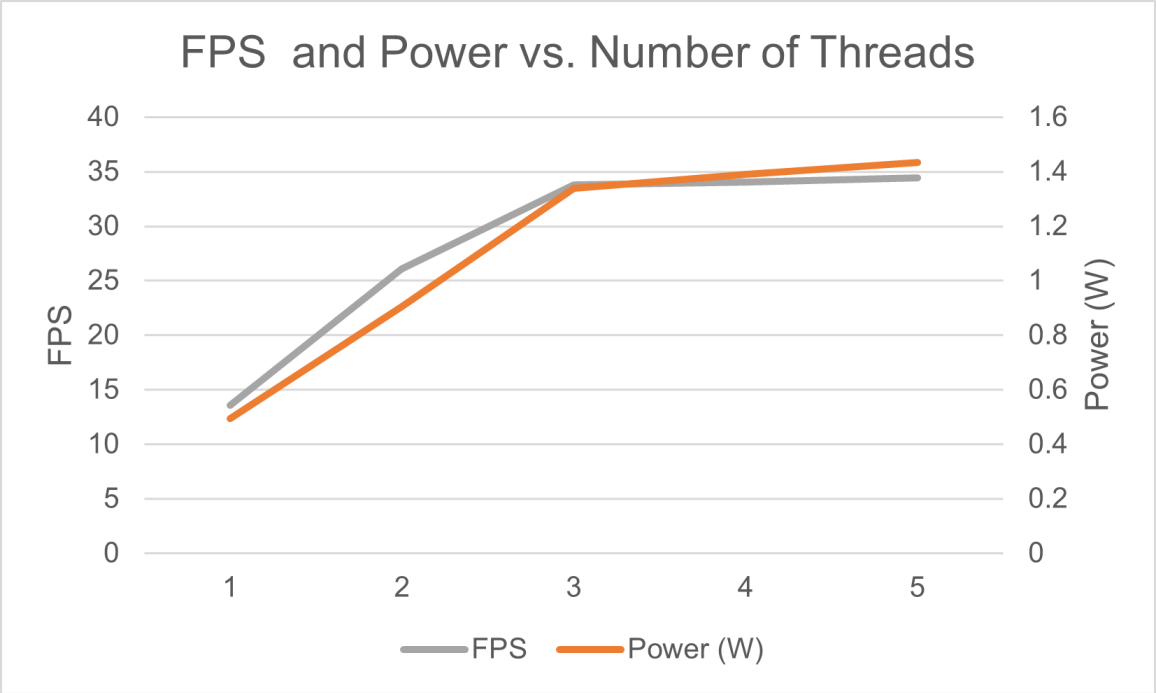


Figure 4.5. Relationship of FPGA Power Consumption to FPS as the Number of Threads is Increased During Deployment on FPGA

These findings can be used to illustrate the advantages of FPGA versus general-purpose processors for embedded object detection tasks. For those tasks that require real-time inference, this model can be deployed with 3 or more DPU threads to achieve this without excessive power consumption during the process. Additionally, CPU inference speeds at over 10 ms/image further illustrate that typical embedded CPUs would not be capable of real-time inference for object detection tasks. Accuracy can be maintained for these applications from the required model quantization, although the model itself could use additional work in evaluating how to increase overall accuracy. Power constraints may be the most important consideration for embedded systems to maintain small size and portability, and the FPGA results show superior advantages during model deployment.

5. SUMMARY

A YOLOv4 object detection model developed with the use of transfer learning was successfully deployed and analyzed on three different hardware configurations. The model utilized a different model pre-trained on the MS-COCO dataset (80 object classes) to redirect the knowledge to a new, more application-specific set of object classifications (5 object classes). The model was then evaluated over a test set of 100 images equally distributed to each of the five classes, and evaluated using accuracy, speed, and power metrics for CPU, GPU, and FPGA deployment.

It was found that the FPGA provided favorable deployment metrics with the use of the transfer-learned model. Accuracy did degrade slightly during the required weight quantization process by an average of 4%, but this was expected as a byproduct of reducing the precision of the model weights. FPGA deployment speed was 7.74 FPS faster than the GPU, and the average inference run-time slightly increased from the COCO pre-trained model to the transfer-learned model for the same test set of images on the FPGA. Finally, power consumption during the 100-image inference test was clearly a major advantage for FPGA deployment, with a 96% improvement over the GPU.

In conclusion, these findings illustrate the effectiveness of FPGA deployment in consideration of embedded applications even with the use of transfer learning. The CPU, which is typically used in embedded systems, does not have the means to deploy deep networks due to the lack of parallelization capability at the hardware level. In contrast, the GPU is built specifically for parallelization and as a result is powerful and fast for deep CNN deployment. However, this power literally becomes its critical disadvantage in embedded deployment. Filling in the gap between the two general-purpose processors is the FPGA, which provides major design flexibility with a low power, real-time deployment configuration without sacrificing much in accuracy. Further, recent development in tools such as Vitis AI abstract the details of the FPGA model deployment process to eliminate the need to understand the underlying hardware design or learn HDL to implement it. This work therefore advances the SOTA in this area by successfully deploying a transfer-learned YOLOv4 object detection model on an FPGA.

Future work should include further evaluation of the training setup for this model to achieve better results in overall accuracy across all classes. The results showed that the model performed well when detecting the Eagle, Glider, and Plane classes, but had very poor Classification Accuracy for the Kite and Quadcopter classes across all three configurations. This may be due to several factors, including a large variability in kite and quadcopter features that could be improved by introducing greater variety during training. Further investigation would be beneficial for overall model performance. Finally, these results could be extended by developing a physical embedded object detection test setup using this model and an FPGA to analyzing how well it performs in the field to identify the five flying object classes and maintain real-time speeds and low-power requirements.

REFERENCES

[1] Z. V. Ilyichenkova, S. M. Ivanova, A. I. Volkov, and A. Y. Ermakova, "The usage of neural networks for motion prediction of autonomous objects," in *2019 Systems of Signals Generating and Processing in the Field of on Board Communications*, 2019, pp. 1–5. DOI: 10.1109/SOSG.2019.8706752.

[2] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, and G. S. Sachdev, "Compute solution for Tesla's full self-driving computer," *IEEE Micro*, vol. 40, no. 2, pp. 25–35, 2020. DOI: 10.1109/MM.2020.2975764.

[3] Y. Li, L. Ma, Z. Zhong, F. Liu, M. A. Chapman, D. Cao, and J. Li, "Deep learning for lidar point clouds in autonomous driving: A review," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2020. DOI: 10.1109/TNNLS.2020.3015992.

[4] S. Roy, W. Menapace, S. Oei, B. Luijten, E. Fini, C. Saltori, I. Huijben, N. Chennakeshava, F. Mento, A. Sentelli, E. Peschiera, R. Trevisan, G. Maschietto, E. Torri, R. Inchingolo, A. Smargiassi, G. Soldati, P. Rota, A. Passerini, R. J. G. van Sloun, E. Ricci, and L. Demi, "Deep learning for classification and localization of COVID-19 markers in point-of-care lung ultrasound," *IEEE Transactions on Medical Imaging*, vol. 39, no. 8, pp. 2676–2687, 2020. DOI: 10.1109/TMI.2020.2994459.

[5] L. Zhang, L. Lu, X. Wang, R. M. Zhu, M. Bagheri, R. M. Summers, and J. Yao, "Spatio-temporal convolutional LSTMs for tumor growth prediction by learning 4D longitudinal patient data," *IEEE Transactions on Medical Imaging*, vol. 39, no. 4, pp. 1114–1126, 2020. DOI: 10.1109/TMI.2019.2943841.

[6] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol. 25, Jan. 2012. DOI: 10.1145/3065386.

[7] K. He and J. Sun, "Convolutional neural networks at constrained time cost," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 5353–5360. DOI: 10.1109/CVPR.2015.7299173.

[8] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019. DOI: 10.1109/JPROC.2019.2921977.

[9] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," Jun. 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[10] H.-C. Shin, H. R. Roth, M. Gao, L. Lu, Z. Xu, I. Nogues, J. Yao, D. Mollura, and R. M. Summers, "Deep convolutional neural networks for computer-aided detection: Cnn architectures, dataset characteristics and transfer learning," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1285–1298, 2016. DOI: 10.1109/TMI.2016.2528162.

[11] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, *DeCAF: A deep convolutional activation feature for generic visual recognition*, 2013. arXiv: 1310.1531 [cs.CV].

[12] S. J. Pan and Q. Yang, "A survey on transfer learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 10, pp. 1345–1359, 2010. DOI: 10.1109/TKDE.2009.191.

[13] J. Chang, Y. Choi, T. Lee, and J. Cho, "Reducing MAC operation in convolutional neural network with sign prediction," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, 2018, pp. 177–182. DOI: 10.1109/ICTC.2018.8539530.

[14] A. Stoutchinin, F. Conti, and L. Benini, *Optimally scheduling CNN convolutions for efficient memory access*, 2019. arXiv: 1902.01492 [cs.NE].

[15] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," Oct. 2006.

[16] L. Jiao, F. Zhang, F. Liu, S. Yang, L. Li, Z. Feng, and R. Qu, "A survey of deep learning-based object detection," *IEEE Access*, vol. 7, pp. 128 837–128 868, 2019. DOI: 10.1109/ACCESS.2019.2939201.

[17] K. He and J. Sun, "Convolutional neural networks at constrained time cost," Dec. 2014.

[18] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Region-based convolutional networks for accurate object detection and segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 1, pp. 142–158, 2016. DOI: 10.1109/TPAMI.2015.2437384.

[19] R. B. Girshick, "Fast R-CNN," *CoRR*, vol. abs/1504.08083, 2015. arXiv: 1504.08083. [Online]. Available: http://arxiv.org/abs/1504.08083.

[20] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015. arXiv: 1506.01497. [Online]. Available: http://arxiv.org/abs/1506.01497.

[21] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," Jul. 2017, pp. 6517–6525. DOI: 10.1109/CVPR.2017.690.

[22] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," *CoRR*, 2018. arXiv: 1804.02767. [Online]. Available: http://arxiv.org/abs/1804.02767.

[23] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *CoRR*, vol. abs/1409.4842, 2014. arXiv: 1409.4842. [Online]. Available: http://arxiv.org/abs/1409.4842.

[24] M. Everingham, S. M. Eslami, L. Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes challenge: A retrospective," *Int. J. Comput. Vision*, vol. 111, no. 1, pp. 98–136, Jan. 2015, ISSN: 0920-5691. DOI: 10.1007/s11263-014-0733-5. [Online]. Available: https://doi.org/10.1007/s11263-014-0733-5.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. arXiv: 1512.03385. [Online]. Available: http://arxiv.org/abs/1512.03385.

[26] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2015.

[27] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, "Feature pyramid networks for object detection," *CoRR*, vol. abs/1612.03144, 2016. arXiv: 1612.03144. [Online]. Available: http://arxiv.org/abs/1612.03144.

[28] A. Bochkovskiy, C. Wang, and H. M. Liao, "YOLOv4: Optimal speed and accuracy of object detection," *CoRR*, vol. abs/2004.10934, 2020. arXiv: 2004.10934. [Online]. Available: https://arxiv.org/abs/2004.10934.

[29] C. Wang, H. M. Liao, I. Yeh, Y. Wu, P. Chen, and J. Hsieh, "CSPnet: A new backbone that can enhance learning capability of CNN," *CoRR*, vol. abs/1911.11929, 2019. arXiv: 1911.11929. [Online]. Available: http://arxiv.org/abs/1911.11929.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *CoRR*, vol. abs/1406.4729, 2014. arXiv: 1406.4729. [Online]. Available: http://arxiv.org/abs/1406.4729.

[31] E. Delaye, A. Sirasao, C. Dudha, and S. Das, "Deep learning challenges and solutions with Xilinx FPGAs," Nov. 2017, pp. 908–913. DOI: 10.1109/ICCAD.2017.8203877.

[32] T. Dettmers, "8-bit approximations for parallelism in deep learning," *CoRR*, 2016.

[33] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *CoRR*, vol. abs/1604.03168, 2016. arXiv: 1604.03168. [Online]. Available: http://arxiv.org/abs/1604.03168.

[34] *ZCU102 evaluation board user guide*, English, version Version 1.6, Xilinx, Jun. 12, 2020, 125 pp.

[35] *Zynq DPU v3.2 product guide*, English, version Version 3.2, Xilinx, Jul. 7, 2020, 57 pp.

[36] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights," *CoRR*, vol. abs/1606.05487, 2016. arXiv: 1606.05487. [Online]. Available: http://arxiv.org/abs/1606.05487.

[37] G. Dulac-Arnold, D. J. Mankowitz, and T. Hester, "Challenges of real-world reinforcement learning," *CoRR*, vol. abs/1904.12901, 2019. arXiv: 1904.12901. [Online]. Available: http://arxiv.org/abs/1904.12901.

[38] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. Chung, "Accelerating deep convolutional neural networks using specialized hardware," Microsoft Research, Feb. 2015. [Online]. Available: https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/.

[39] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of FPGAs and GPUs," *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 93–96, 2018. DOI: 10.1145/3174243.3174970.

[40] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. O. G. Hock, Y. T. Liew, K. Srivatsan, D. J. M. Moss, S. Subhaschandra, and G. Boudoukh, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[41] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, "ESE: efficient speech recognition engine with compressed LSTM on FPGA," *CoRR*, vol. abs/1612.00694, 2016. arXiv: 1612.00694. [Online]. Available: http://arxiv.org/abs/1612.00694.

[42] R. Solovyev, A. Kustov, D. Telpukhov, V. Rukhlov, and A. Kalinin, "Fixed-point convolutional neural network for real-time video processing in FPGA," *2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, Jan. 2019. DOI: 10.1109/eiconrus.2019.8656778. [Online]. Available: http://dx.doi.org/10.1109/EIConRus.2019.8656778.

[43] Z. Li and J. Wang, "An improved algorithm for deep learning YOLO network based on Xilinx Zynq FPGA," in *2020 International Conference on Culture-oriented Science Technology (ICCST)*, 2020, pp. 447–451. DOI: 10.1109/ICCST50977.2020.00092.

[44] A. Sharma, V. Singh, and A. Rani, "Implementation of CNN on Zynq based FPGA for real-time object detection," in *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2019, pp. 1–7. DOI: 10.1109/ICCCNT45670.2019.8944792.

[45] S. P. Kaarmukilan, S. Poddar, and A. Thomas K., "FPGA based deep learning models for object detection and recognition comparison of object detection comparison of object detection models using FPGA," in *2020 Fourth International Conference on Computing Methodologies and Communication (ICCMC)*, 2020, pp. 471–474. DOI: 10.1109/ICCMC48092.2020.ICCMC-00088.

[46] T. Wu, W. Liu, and Y. Jin, "An end-to-end solution to autonomous driving based on Xilinx FPGA," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 427–430. DOI: 10.1109/ICFPT47387.2019.00084.

[47] S. Fang, L. Tian, J. Wang, S. Liang, D. Xie, Z. Chen, L. Sui, Q. Yu, X. Sun, Y. Shan, and Y. Wang, "Real-time object detection and semantic segmentation hardware system with deep learning networks," in *2018 International Conference on Field-Programmable Technology (FPT)*, 2018, pp. 389–392. DOI: 10.1109/FPT.2018.00081.

[48] T. Carneiro, R. V. Medeiros Da NóBrega, T. Nepomuceno, G. Bian, V. H. C. De Albuquerque, and P. P. R. Filho, "Performance analysis of Google Colaboratory as a tool for accelerating deep learning applications," *IEEE Access*, vol. 6, pp. 61 677–61 685, 2018. DOI: 10.1109/ACCESS.2018.2874767.

[49] Alexey, J. Redmon, S. Sinigardi, cyy, T. Hager, V. Zhang, M. Maaz, IlyaOvodov, P. Kahn, J. Veitch-Michaelis, A. Dujardin, duohappy, acxz, J. Aughey, E. Özipek, J. White, D. Smith, Aven, T. K. C. Shibata, M. Giordano, G. Daras, HagegeR, B. Gąsiorzewski, A. Babaei, H. Vhavle, E. Arends, D. Cho, C.-H. Lin, A. Baranski, and 7FM, *Alexeyab/darknet: YOLOv4 pre-release*, version darknet_yolo_v4_pre, May 2020. DOI: 10.5281/zenodo.3829035. [Online]. Available: https://doi.org/10.5281/zenodo.3829035.

[50] L. Lu, "Dying relu and initialization: Theory and numerical examples," *Communications in Computational Physics*, vol. 28, no. 5, pp. 1671–1706, Jun. 2020, Global Science Press, ISSN: 1991-7120. DOI: 10.4208/cicp.oa-2020-0165. [Online]. Available: http://dx.doi.org/10.4208/cicp.OA-2020-0165.

[51] Xilinx, *YOLOv4 tutorial*, version 1.3.0, Apr. 2021. [Online]. Available: https://github.com/Xilinx/Vitis-Tutorials/tree/master/Machine_Learning/Design_Tutorials/07-yolov4-tutorial (visited on 05/31/2021).

[52] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, "Focal loss for dense object detection," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017.

[53] Xilinx, *Vitis AI*, version 1.3.0, 2020. [Online]. Available: https://github.com/Xilinx/Vitis-AI (visited on 05/31/2021).

[54] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091.

[55] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the NVIDIA turing T4 GPU via microbenchmarking," *CoRR*, vol. abs/1903.07486, 2019. arXiv: 1903.07486. [Online]. Available: http://arxiv.org/abs/1903.07486.

[56] A. Dale, "3D object detection using virtual environment assisted deep network training," Master's thesis, Indiana University-Purdue University Indianapolis, 2020.

[57] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, *Microsoft COCO: Common objects in context*, 2015. arXiv: 1405.0312 [cs.CV].

[58] L. Biewald, *Experiment tracking with weights and biases*, Weights and Biases, 2020. [Online]. Available: https://www.wandb.com/ (visited on 05/31/2021).

[59] Parker-Xilinx, *Xilinx linux power utility*, 2019. [Online]. Available: https://github.com/parker-xilinx/xilinx-linux-power-utility (visited on 05/31/2021).

[60] *Vitis AI user guide*, English, version 1.3, Xilinx, Dec. 17, 2020, 231 pp.