Accelerating Machine Learning Inference for Satellite Component Feature Extraction Using FPGAs.

Andrew Ekblad
*Florida Institute of Technology, aekblad2019@my.fit.edu*

Follow this and additional works at: [https://repository.fit.edu/etd](https://repository.fit.edu/etd)

Part of the Artificial Intelligence and Robotics Commons, Digital Circuits Commons, Robotics Commons, Signal Processing Commons, Space Vehicles Commons, and the VLSI and Circuits, Embedded and Hardware Systems Commons

**Recommended Citation**
[https://repository.fit.edu/etd/1396](https://repository.fit.edu/etd/1396)

This Thesis is brought to you for free and open access by Scholarship Repository @ Florida Tech. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholarship Repository @ Florida Tech. For more information, please contact kheifner@fit.edu.
Accelerating Machine Learning Inference for Satellite Component Feature Extraction Using FPGAs.

by

Andrew Ekblad

A thesis submitted to the College of Engineering and Science of Florida Institute of Technology in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering

Melbourne, Florida
December, 2023
We the undersigned committee hereby approve the attached thesis,
“Accelerating Machine Learning Inference for Satellite Component Feature Extraction Using FPGAs.”
by
Andrew Ekblad

__________________________
Naveed Mahmud, Ph.D.
Assistant Professor
Electrical Engineering and Computer Science
Major Advisor

__________________________
Ryan T. White, Ph.D.
Assistant Professor
Mathematics and Systems Engineering

__________________________
Samuel P. Kozaitis, Ph.D.
Professor
Electrical Engineering and Computer Science

__________________________
Brian Lail, Ph.D.
Professor and Department Head
Electrical Engineering and Computer Science
Abstract

Accelerating Machine Learning Inference for Satellite Component Feature Extraction Using FPGAs

Author: Andrew Ekblad
Advisor: Naveed Mahmud, Ph.D.

Running computer vision algorithms requires complex devices with lots of computing power, these types of devices are not well suited for space deployment. The harsh radiation environment and limited power budgets have hindered the ability of running advanced computer vision algorithms in space. This problem makes running an on-orbit servicing detection algorithm very difficult. This work proposes using a low powered FPGA to accelerate the computer vision algorithms that enable satellite component feature extraction. This work uses AMD/Xilinx’s Zynq SoC and DPU IP to run model inference. Experiments in this work centered around improving model post processing by creating implementations in Python, C++ and FPGA Fabric. The FPGA Fabric implementation included testing a synchronous and an asynchronous implementation. The final implementation shows that a model trained for satellite component feature extraction can run at 9 frames per second while running with an average power consumption of less than 9 Watts.
Table of Contents

Abstract.................................................................................................................. iii
List of Figures......................................................................................................... vii
List of Tables........................................................................................................ viii
Acknowledgements.............................................................................................. ix
Chapter 1 Introduction............................................................................................ 1
Chapter 2 Background ........................................................................................... 3
   Raspberry Pi and NCS2....................................................................................... 3
   KV260 ................................................................................................................ 4
   Zynq UltraScale+ Architecture ......................................................................... 5
   FPGA Design Tools (RTL) ................................................................................ 6
   FPGA Design Tools (HLS) .............................................................................. 7
   YOLOv4 ............................................................................................................ 8
   VITIS AI ........................................................................................................... 8
   DPU .................................................................................................................. 9
   PYNQ ................................................................................................................ 10
   Memory Addressing Multidimensional Arrays ................................................... 11
      Memory Addressing a 2D Matrix: ................................................................. 11
      Memory Addressing a 3D Matrix: ................................................................. 12
   YOLO Post Processing ....................................................................................... 14
   Inference Processing Pipeline .......................................................................... 15
Chapter 3 Baseline Implementation .................................................................... 17
   Creating a Baseline............................................................................................ 17
Device Accuracy ................................................................................................................. 18
Device Speed ....................................................................................................................... 18
Average Power ..................................................................................................................... 19
Performance Per Watt ......................................................................................................... 20
Baseline Experimental Setup .............................................................................................. 20
Chapter 4 Improvements and Optimizations ....................................................................... 24
Filtering Using C++ ............................................................................................................. 24
Filtering Using Fabric .......................................................................................................... 24
Pipelining ............................................................................................................................. 25
Improved Model Training ................................................................................................. 26
Chapter 5 Experiments ....................................................................................................... 27
Experimental C++ Implementation ....................................................................................... 27
Experimental Fabric Implementation .................................................................................... 28
Setting up a Testbench ......................................................................................................... 28
FPGA Fabric Experimental Implementation ........................................................................ 30
Fabric Control Registers .................................................................................................... 32
Fabric Results ...................................................................................................................... 33
Comparing Experimental Fabric and C++ Filtering ............................................................. 34
Pipelining Experiments ....................................................................................................... 36
Model Pruning ...................................................................................................................... 42
Camera Implementation ...................................................................................................... 43
Chapter 6 Final Implementation and Results ..................................................................... 45
The Satellite Vision Implementation .................................................................................... 45
The Satellite Dataset ............................................................................................................ 45
Device Architecture........................................................................................................46
Model Selection and Training ..........................................................................................46
Experimental Methods ....................................................................................................47
Device Performance ........................................................................................................47
Example Satellite Results ..............................................................................................49
Summary .........................................................................................................................52
Chapter 7 Areas of Future Improvements.................................................................53
Chapter 8 Conclusions ....................................................................................................55
References ......................................................................................................................56
Appendix .........................................................................................................................61
List of Acronyms .............................................................................................................61
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>KV260 Block Diagram [10]</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>YOLOv4 Network Architecture [17]</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Mish Activation Function</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>Leaky ReLU Activation Function With a=-0.1</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>2x2 Matrix M</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>2x2 Matrix M in Memory</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>2x2x2 Matrix L</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>2x2x2 Matrix L in Memory</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>Inference Processing Piepline Block Diagram</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>Screen Capture During Baseline Testing</td>
<td>21</td>
</tr>
<tr>
<td>11</td>
<td>Conceptual Pipeline Diagram</td>
<td>26</td>
</tr>
<tr>
<td>12</td>
<td>Fabric Filtering Simulation Result</td>
<td>29</td>
</tr>
<tr>
<td>13</td>
<td>System Block Diagram with Filtering and DPU</td>
<td>30</td>
</tr>
<tr>
<td>14</td>
<td>Fabric Experiment Block Diagram</td>
<td>31</td>
</tr>
<tr>
<td>15</td>
<td>Experimental Results for Inference Speed</td>
<td>35</td>
</tr>
<tr>
<td>16</td>
<td>Experimental Results Filtering Performance Per Watt</td>
<td>36</td>
</tr>
<tr>
<td>17</td>
<td>Sigmoid Activation Function</td>
<td>37</td>
</tr>
<tr>
<td>18</td>
<td>Fabric Block Diagram with Sigmoid LUT</td>
<td>39</td>
</tr>
<tr>
<td>19</td>
<td>Final Fabric Experiment Comparison</td>
<td>41</td>
</tr>
<tr>
<td>20</td>
<td>Fabric Latency Comparisons</td>
<td>42</td>
</tr>
<tr>
<td>21</td>
<td>Camera Experiment Screen Capture</td>
<td>44</td>
</tr>
<tr>
<td>22</td>
<td>Example Image From the Testing Dataset</td>
<td>46</td>
</tr>
<tr>
<td>23</td>
<td>Example Detection on FPGA</td>
<td>50</td>
</tr>
<tr>
<td>24</td>
<td>Example Detection on Raspberry Pi</td>
<td>51</td>
</tr>
<tr>
<td>25</td>
<td>Example Detection on GPU</td>
<td>52</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1 Baseline Performance Results</td>
<td>22</td>
</tr>
<tr>
<td>Table 2 Baseline Power Results</td>
<td>22</td>
</tr>
<tr>
<td>Table 3 C++ Experimental Speed Results</td>
<td>27</td>
</tr>
<tr>
<td>Table 4 C++ Experimental Model Accuracy</td>
<td>28</td>
</tr>
<tr>
<td>Table 5 Fabric Experimental Results</td>
<td>33</td>
</tr>
<tr>
<td>Table 6 Fabric Experimental mAP</td>
<td>34</td>
</tr>
<tr>
<td>Table 7 Experimental Results Speed Comparison for Implementations</td>
<td>35</td>
</tr>
<tr>
<td>Table 8 Final Fabric Implementation mAP comparison</td>
<td>39</td>
</tr>
<tr>
<td>Table 9 Final Fabric Experiment Comparison</td>
<td>40</td>
</tr>
<tr>
<td>Table 10 Pruned Model Experimental Results</td>
<td>43</td>
</tr>
<tr>
<td>Table 11 Experimental Pruned Model mAP</td>
<td>43</td>
</tr>
<tr>
<td>Table 12 Satellite Model mAP and Speed Results</td>
<td>48</td>
</tr>
<tr>
<td>Table 13 Satellite Model Power Results</td>
<td>49</td>
</tr>
</tbody>
</table>
Acknowledgements

I want to thank Dr. White at the Florida Tech NETS Lab [1] for providing the FPGA used for this work and being available to support my work.

I also want to thank Trupti Mahendrakar and Dr. Wilde for encouraging me to proceed with this work and supporting me along the way.

Finally, I would like to express my appreciation to Professor Ellen Soash for showing me that I am capable of much more when I actually put my mind to it. I do not think I would have been in a place to write this without her help.
Chapter 1
Introduction

Developing and launching a satellite is a long and expensive process. For example, the GOES set of 4 geostationary weather satellites started design in 2005. The first of the four satellites launched in 2016. The satellites have an estimated lifetime budget of $10.8 Billion [2]. Once a functioning satellite reaches its end of life it is either moved to a decommission orbit or deorbited from space. The satellite can also become non-functional and remain in orbit where it can cause collisions with other spacecrafts.

The idea of on-orbit servicing is not new, the space shuttle program has had several missions that involved repairing and upgrading the Hubble telescope [3]. These missions were all done by flying humans to the telescope and performing maintenance. On-orbit servicing can be implemented using spacecrafts that are designed specifically for on-orbit servicing satellites. NASA is working on OSAM-1 [4] which is a satellite that would enable on-orbit servicing.

One of the issues with on-orbit servicing is docking with other spacecrafts. These spacecrafts often have different shapes and parts, to the point where most satellites can be considered unique. Adding to this problem of varying types of satellites is when they are uncooperative, meaning that they have had some sort of malfunction that prevents them from behaving in a controlled manner. A tumbling satellite with the inability to correct itself is a good example of an uncooperative satellite.

Previous work has gone into exploring how to dock with an uncooperative satellite, one method is to use a swarm of small satellites guided by an Artificial Potential Field guidance algorithm [5]. The work used a machine vision algorithm to detect satellite components that might make effective docking points. In [5] Solar panels are repulsive nodes and body is an attractive node, therefore the guidance algorithm would try to get close to the body while avoiding solar panels. The points are detected using a machine vision algorithm that classifies the components on satellites. The implementation in [5]
uses a Raspberry Pi 4 and an Intel Neural Compute Stick 2 to run the machine learning inference. The hardware in that work could only capture detections at 2 FPS [5]. In a real flight environment that capture rate would need to be significantly higher to avoid a catastrophic collision.

I have done previous work using an FPGA to accelerate the machine vision inference and have shown that even a simple FPGA implementation can produce a framerate of more than 2 FPS [6]. The work concluded that the FPGA device was capable of providing even more performance.

This paper builds on the research conducted in [6] to build a faster implementation. This work focuses on speeding up the post processing algorithm by implementing it in Python, C++ and FPGA Fabric and comparing the results between them.
Chapter 2  
Background

This section will cover any background information that is necessary to provide the reader with information they will need to understand the software, devices and methods that appear in this paper.

Raspberry Pi and NCS2

The Raspberry Pi is a well-known Single Board Computer (SBC) developed by the Raspberry Pi foundation with extensive capabilities [7]. The Raspberry Pi has an ARM Cortex-A72 (ARMv8) System on Chip (SoC) with a clock speed of 1.5Ghz with 1, 2, 4 or 8GB of LPDDR4 Memory [8]. The Raspberry Pi is a great device for prototyping and deploying embedded systems because of its low cost and comprehensive ecosystem. Software is a strong point for the Raspberry Pi, due to the number of users there is a large catalog of software that has been specifically designed and tested to work on the Raspberry Pi. One of those pieces of software enables machine learning inference.

Using the Raspberry Pi for machine learning inference is accomplished by using an Intel Neural Compute Stick 2 (NCS2) [9], which is an Application Specific Integrated Circuit (ASIC) that is designed to accelerate machine learning inference. While it is possible to run the machine learning inference on the Raspberry Pi’s CPU the performance would be considerably slower due to the low powered ARM CPU. The NCS2 was specifically designed to handle the kinds of computations that are common in machine learning, mainly matrix, vector multiplication and matrix, matrix multiplication.

Previous works used the Raspberry Pi for inference because it is able to emulate low powered space hardware [5]. In that implementation the model ran on the Intel NCS2 directly, but with several unsupported layers that needed to be offloaded to the CPU. Offloading layers to the CPU is very slow and resulted in poor performance.
The idea of this work is to use an FPGA to fix the slow inference speed. The FPGA that was selected was the AMD/Xilinx KV260 which is discussed in the next section.

**KV260**

The KV260 is a development board made by AMD/Xilinx based on their K26 SOM (However I will be referring to is a the KV260 for the rest of this paper) [10]. The KV260 development board is based around a Zynq Ultrascale+ MPSoC and has 4GB of DDR4 memory. The development kit also contains several other onboard interfaces that are accessible to the user. More detail can be seen in the block diagram in Figure 1. The KV260 development kit was chosen for this work because of its affordable price and because it is designed for machine vision applications [11].

The next section will talk about the architecture of the Zynq Ultrascale+ MPSoC.
Zynq UltraScale+ Architecture

Zynq is the name that AMD/Xilinx gives to their SoCs which consist of both a Processing System (PS) and Programmable Logic (PL) [12] [13]. The PS contains an ARM processor with one or more cores, external DDR memory and a connection to the PL. The PL is an FPGA with interconnects to the PS allowing for easy control of the PL through software as well as shared system memory for fast PS-PL communication.

The PS and PL communicate using the Advanced eXtensible Interface (AXI) which is an on-chip communication protocol [14]. AXI is part of the Advanced Microcontroller Bus
Architecture (AMBA) specification created by ARM [14]. The protocol is widely used in AMD/Xilinx devices and has 3 separate protocols (AXI4, AXI4-Lite, AXI-Stream) each with a specific purpose.

The AXI4 communication protocol is designed for high-speed low latency on-chip communication between devices and is suitable to communicate with memory controllers. The interface can become very complex but in its simplest form there is a handshake and then data transfer. This protocol allows for data bursting which can allow a single handshake to cover multiple transactions. AXI4 supports bus widths up to 1024 bits wide and bursts up to 256 words long enabling very high bandwidth transactions.

AXI4-Lite is a simpler version of the AXI4 protocol. It is limited to a 32-bit data width and does not contain any bursting functionality. This interface is generally used for control registers since they often do not require high bandwidth communication where the excess protocol complexity is unnecessary.

AXI-Stream is a protocol designed for streaming data, once a handshake occurs the stream continues indefinitely as long as there is no interruption. This protocol works very well for audio or video interfaces since audio and video streams need to run continuously without interruption.

This work makes use of AXI4 and AXI4-Lite for both data transfer and register control. PS-PL communication is implemented using the AXI4 Protocol and allows for PL logic to directly access shared system memory. The PL block is controlled by a set of AXI4-Lite registers that are controlled by the PS.

The next section will talk about classical design tools that are used to program FPGAs.

FPGA Design Tools (RTL)

Programmable logic devices such as FPGAs are very different than devices such as CPUs and GPUs. While CPUs, GPUs and FPGAs can all be “programed” the way that they are “programed” is considerably different. A CPU and GPU have a fixed set of instructions
that are used to command the behavior of the device. These instructions control different functional units and are typically programed through a programming language like C, C++, Python, etc. FPGAs have a large collection of elementary logic elements such as logic gates (Typically in the form of Look Up Tables (LUTs)) and registers that are connected with a very advanced interconnect structure. The interconnect controls the way that the gates and registers are connected. There is a fundamental difference in the functionality of these devices, and therefore a very different way of programing. FPGAs are programmed using Hardware Description Languages (HDLs) two well-known HDLs being Verilog and VHDL. VHDL for example has varying levels of abstraction. At the lowest level you describe the input and output relationships between gates using primitives such as AND, OR etc. This kind of programing is referred to as dataflow [15]. Dataflow modeling has severe limitations in describing complex circuits due to the need to explicitly describe the digital circuitry that you need, this limitation is solved by behavioral modeling. Behavioral modeling allows the designer to describe the desired behavior of the circuit but leaves the implementation of the digital logic up to the synthesis tool [15] (The synthesis tool is similar to the compiler in software). Most HDL programming that is done today uses behavioral modeling. Often programming with HDL is called Register Transfer Language (RTL), as you typically describe data transfer between registers [15].

While using RTL can aid in producing digital circuitry it is still a very low-level language and takes time and effort to create complex circuits, this is where High Level Synthesis comes in.

**FPGA Design Tools (HLS)**

High level Synthesis (HLS) is a method of programing a digital device that does not use RTL, it instead allows the device to be programmed using a higher-level language such as C and then an algorithm translates the C code into RTL. This comes at the cost of some direct control over the specific hardware implementation. However, it can allow for rapid development of complicated algorithms [16]. HLS was used in this work due to the time constraint. For this work code was written in C/C++ and there was no code directly written in a hardware description language.
The next section talks about the machine learning model that was used for this work, YOLOv4.

**YOLOv4**

![Figure 2 YOLOv4 Network Architecture [17]](image)

You Only Look Once Version 4 (YOLOv4) is a YOLO implementation introduced by [17]. It is a single stage object detector that has shown to produce good results with fewer parameters when compared to more complex two stage detectors. Figure 2 shows a basic architecture of the YOLOv4 algorithm, the figure mainly shows the feature pyramid implementation and the output from the model. Previous works have shown that YOLOv5 is capable of being used for satellite component feature extraction with acceptable results [18]. This work uses YOLOv4 because AMD/Xilinx provided an example model conversion flow for YOLOv4 [19].

**VITIS AI**

Vitis AI is AMD/Xilinx’s platform enabling quantization, compiling and running machine learning models on their FPGA platforms [20]. Vitis AI contains a set of tools enabling models to be deployed on their DPU IP. For this work it was used to quantize and convert a TensorFlow YOLOv4 model [21] for inference on their DPU IP. The steps to do so involved changing some layers to make use of the supported DPU instructions [19], quantizing the model to an INT8 representation and compiling it for the specific DPU architecture. VITIS AI also contains the necessary functions and drivers to run the DPU IP on hardware.
The next section will discuss the DPU and talk about the specific model modifications required to use the model for inference.

**DPU**

The Deep neural network Processing Unit (DPU) is an Intellectual Property (IP) that enables machine learning inference to be deployed on AMD/Xilinx FPGAs [22]. The DPU supports many different configurations with different area and performance characteristics. For minimum area utilization the DPU can shrink down all the way to an implementation with 512 operations per clock with minimal hardware resources. For maximum performance the DPU can expand up to an implementation with 4096 operations per clock which will use significantly more resources but provide significantly better performance. The IP is scalable to support up to 4 concurrent DPU cores in one design for large FPGA devices.

Due to the size of the FPGA in this work the IP was configured to use a single DPU core with 4096 operations per clock.

The DPU IP has a limited set of supported layers [22]. It is therefore important that any model running on the DPU only used the supported layers, otherwise they may be offloaded to the CPU. Offloading specific model layers to the CPU would cause a performance hit that is unacceptable. For the YOLOv4 model that is used two modifications needed to be made as suggested by [19]. First the mish activation function [23] in YOLOv4 is replaced with the Leaky ReLU function [24]. Examples of the mish and Leaky ReLU activation functions are shown in Figure 3 and Figure 4 respectively.
\[ \text{mish}(x) = x \ast \tanh(\ln(1 + \exp(x))) \]

\[ \text{Leaky ReLU}(x) = \begin{cases} 
  ax, & x < 0 \\
  x, & x \geq 0 
\end{cases} \]

Both figures are from my IEEE paper [6]

Changing the activation function from mish to Leaky ReLU will have a small impact on model accuracy [23].

The second modification is to reduce the size of the max pooling kernel to 8x8, this is because the DPU does not support max pooling kernels larger than 8x8 [22]. Changing the max pool size will also have a small impact on model accuracy.

For the best results these modifications are made to the model before it is trained, then the weights will be optimized for these two changes.

To enable the use of the DPU on hardware the PYNQ framework was used for this project and will be talked about next.

**PYNQ**

PYNQ (Pronounced “Pink”) is an environment created by AMD/Xilinx that simplifies the SoC programming process [25]. It allows for programing Zynq SoC devices using Python instead of C/C++. Using PYNQ for this work enabled a rapid design process by providing
premade drivers for the DPU and other PL related controllers. Without PYNQ a custom Linux image would need to be created with the DPU drivers [26]. Using PYNQ also accelerated driver development for custom FPGA functions as they can be written in Python. PYNQ provided the necessary functionality to make this project feasible.

The next section talks about the memory addressing scheme for multi-dimensional arrays that was required for the YOLO filtering implementations.

**Memory Addressing Multidimensional Arrays**

Images, neural networks and matrices all have a common problem when they are represented in a computer. How do you represent a multidimensional array in one-dimensional memory? There are two addressing methods to represent a multidimensional array in memory, row-major addressing and column-major addressing.

Row-major aligns the data so that rows are contiguous in memory. For example a 2D matrix A is indexed using A[i,j] is stored so that incrementing j by 1 increments the memory address by 1.

The other option is column-major where columns are contiguous in memory. For example a 2D matrix A is indexed using A[i,j] is stored so that incrementing i by 1 increments the memory address by 1.

The C programming language uses the row-major layout while Fortran and MATLAB use the column-major layout [27]. Since this work uses C/C++ and not Fortran I will go into more detail about row-major addressing and leave out column major addressing.

Memory addressing multidimensional arrays is easy to explain by example. The first example is a 2x2 matrix and the second example is a 2x2x2 matrix used to derive the memory addressing equations.

**Memory Addressing a 2D Matrix:**

Figure 5 shows a 2x2 matrix M
Using row-major indexing we take each row and put them contiguously in memory as shown in Figure 6.

To convert matrix notation \( M(i,j) \) to a linear address we use an equation that increments the address by 1 when there is a change to \( j \) and increments the address by 2 when there is a change to \( i \). The equation that satisfies these requirements is shown in Equation 1.

\[
Address\ of\ M_{2\times2}(i,j) = 2i + j \quad \text{Equation 1}
\]

To generalize the previous equation for any MxN matrix you replace 2 with the row length as shown in Equation 2.

\[
Address\ of\ M_{M\timesN}(i,j) = Ni + j \quad \text{Equation 2}
\]

Memory Addressing a 3D Matrix:
Now it is easy to extend this addressing formula type to a simple 2x2x2 Matrix \( L \) with index notation \( L[i,j,k] \). Graphically it is shown in Figure 7.
The matrix represented in memory is shown in Figure 8.

In row-major indexing the last index in an array is contiguous in memory.

\[
\text{Address of } L_{2\times2\times2}(i, j, k) = 2 \times 2i + 2j + k \quad \text{Equation 3}
\]

Generalizing this to M\times N\times O

\[
\text{Address of } L_{M\times N\times O}(i, j, k) = NOi + Oj + k \quad \text{Equation 4}
\]

The equation can be extended further, but there is no need to extend this equation further since the highest dimensional matrix that was used in this work was 3D.

The multidimensional array representation was used in creating the FPGA implementation since array can only be passed by memory address without a data type that contains shape information.
Now that all the background material has been summarized it is possible to talk about how filtering the output of YOLOv4 works.

**YOLO Post Processing**

Since the DPU has a finite number of supported operations, any operation that is not supported by the DPU must be offloaded to a device that can perform the required operations. In the case of YOLOv4 the model itself can run on the DPU with only slight modifications (See DPU section for more information about the modifications). The layers in the YOLO post processing algorithm are not supported on the DPU.

The output of the YOLO model actually contains 3 separate outputs. The size of the 3 outputs are: 52x52, 26x26 and 13x13, each output also has a 3rd dimension with size 3*(5+number of classes).

The outputs from the YOLO model are referred to as the model outputs in this work. The YOLO model output can propose up to 10,647 potential bounding boxes each potential bounding box is a vector that contains width, height, x position, y position, object probability, and class probability (there is a class probability for each class at the output of the network). The size of this output vector is dependent on the number of classes in the model, the equation for the length of the vector is 5 + number of classes. A model such as the COCO dataset [28] with 80 classes will require a vector length of 85 which leads to 904,995 results, each is represented by a 32bit float. Each output produced by the YOLO model requires 3.6MB of memory which is larger than one frame of 720p video.

The number of potential bounding boxes needs to be reduced before they can be drawn, since most of the boxes at the output do not contain any meaningful predictions. The process of reducing the number of bounding boxes is referred to as filtering in this work. Filtering eliminates the bounding boxes with a low probability and only lets boxes with a confidence higher than a set confidence threshold be used for the final prediction. The operations required for filtering are not supported on the DPU. In my previous works it was done in software [6] which resulted in slow processing speeds.
Inference Processing Pipeline

To make sure that the reader understands what steps are going to be improved in this work, I will explain the steps that occur during YOLO to get from an unlabeled image to a labeled one.

The block diagram in Figure 9 demonstrates the basic model processing pipeline.

![Figure 9 Inference Processing Pipeline Block Diagram](image.png)

The first step in the processing pipeline is capturing an image. An image can be captured from an attached camera, or it could be loaded from a file.

The next step is pre-processing. The YOLOv4 model has an input size of 416x416 so an image that is larger or smaller needs to be scaled to that size. In this case the image is scaled and letterboxed to preserve the original aspect ratio of the image.

After preprocessing the model is given to the neural network. This step is where the trained model is used.

After the model has gone through inference the output needs to be processed. The output of the YOLOv4 model contains many outputs that need to be filtered and scaled. After that the remaining boxes need to get through non-max suppression to remove any overlapping bounding boxes. Once the result has gone through non-max suppression the boxes can be drawn on the original image.

The final image can be output, this can be by either displaying the result on screen, saving the result to a file or using it as the input to a decision-making algorithm.
This work focuses mainly on improving the implementation of the filtering block which is only one small step in the entire processing pipeline.
I have done previous work exploring the deployment of machine learning models on an FPGA [6]. In that previous work I trained a machine learning model for satellite component feature extraction and deployed it on an FPGA using the AMD/Xilinx DPU IP. I found that the implementation was faster than the comparable Raspberry Pi implementation, and I concluded that an FPGA design should still be able to produce better results than the finals ones in that paper. While the foundation of the paper is good, there was a lack of quantitative testing. This paper will build off the ideas from that paper but will start by creating a solid baseline that improvements can be measured against.

Creating a Baseline

The baseline is perhaps one of the most difficult things to start, once it is set all other metrics in this paper will be measured against to determine the finals conclusions, so it is important that the metrics are meaningful, understandable and correct. To that point the metrics that were decided in no order of importance are:

- Device Accuracy
- Device Speed
  - Total Speed
  - Inference Speed
  - Filtering Speed
- Average Power
- Performance Per Watt
  - Total Performance Per Watt
  - Inference Performance Per Watt
  - Filtering Performance Per Watt
Device Accuracy

Device accuracy measures the model’s ability to successfully detect and locate the objects in an image. Since this work is not focused exclusively on training and tuning a model, I will stick with a single metric known as Mean Average Precision or mAP. The mAP was evaluated with a confidence threshold of 0.5 and an IOU of 0.5. This confidence threshold is higher than many other works use. However, the implementation from this work is intended to be used as an input to a flight control system, therefore it is very important that when the model claims that it sees something that it very confident, otherwise the consequences could be detrimental to a mission.

mAP is a measure of the average of the Average Precision (AP) metric across all classes [29]. Average precision is the waited mean of the precision achieved at each recall threshold [29]. Precision is defined as the ratio of true predictions over the total number of predictions [29]. Precisions can be thought of as: when the model makes a prediction how many of them are correct? Recall is defined as the ratio of true predictions over the predictions that should be there (true positives + false negatives) [29]. Recall can be thought of as: does the model make a prediction everywhere it should make a prediction?

Device accuracy is affected by two things, how well the model is trained, and how well was the model quantized. The first can be controlled by model training and tuning the original model. The second would be controlled by tuning the quantization process in Vitis AI.

Device Speed

Device Speed is a measure of the rate that the device processes incoming frames. This metric is broken into 3 different categories Total Speed, Inference speed (DPU speed), and Filtering Speed. All the speed metrics are measured in Frames Per Second (FPS), and the inverse can be taken to measure the time it takes to process a frame.

Total speed is the frame rate that the entire system can process incoming frames. It is a good metric to determine the overall performance of the system. This metric is measured in
FPS and is a measure of total system throughput including potential system pipelining. Total speed is programmed to exclude the time that it takes to load the images into memory since a device like this would most likely be implemented with a camera processing pipeline where frames would already be stored in memory.

Total latency is similar to total speed, but it measures the time it takes for the system to process a single frame from input to output. This metric is the same as total speed while the system does not have any pipelining. Once pipelining is implemented the latency metric will determine how long it takes to process a frame from the input to output.

Inference speed measures the throughput of the neural processor and is used to determine if the neural processor is the bottleneck of the system. The inverse of inference speed is the latency of the neural processor.

Filtering speed measures the throughput of the filtering algorithm and is used to determine if the filtering module is the bottleneck of the system. The inverse of the filtering speed is the latency of the filtering algorithm.

Device speed can be improved in any number of ways, reducing the size of the model, improving the filtering algorithms, or even increasing the clock speed (only applies to the FPGA implementation).

**Average Power**

Average power measures the average power of the system, for the FPGA and Raspberry Pi it is measured at the input to the device using a USB-C power tester that measures voltage and current [30]. The GPU power was not measured in this work due to not having the tools necessary to measure the total system power.

Power is one that would be difficult to change since it is an intrinsic metric of the hardware, so other than reducing clock speed (which is not done in this work) power remains almost constant for all devices.
Performance Per Watt

Performance per watt is used to compare the differing levels of performance on devices that have different power consumption. This metric is used to show the efficiency of each implementation.

Performance per watt changes as the speed changes, increasing the speed is the only way to really improve this metric as power is mostly considered constant.

Baseline Experimental Setup

The baseline experiment was done with the COCO 2017 validation dataset [28] which contains approximately 5000 labeled images spread across 80 classes of everyday objects. For the Raspberry Pi and the FPGA, the power was monitored using a camera pointed at the power monitor while a screen capture of the output was taken during the run. The screen capture is shown in Figure 10. The system that was used for GPU performance testing is described below.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel 5960x</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Nvidia 1080Ti</td>
</tr>
<tr>
<td>RAM</td>
<td>64 GB DDR4</td>
</tr>
<tr>
<td>OS</td>
<td>Pop!_OS 20.04</td>
</tr>
</tbody>
</table>

The implementation that was used for each testing was created using a shared code base and the same evaluation function for each implementation. The evaluation function came from a Xilinx example design [31]. The GPU implementation was derived from [21]. The FPGA implementation was derived from the [31]. The Raspberry Pi implementation was derived from [32]. The model used for each implementation was trained using a TensorFlow YOLOv4 code base [21]. The GPU, Raspberry Pi and FPGA use the same common model, but the Raspberry Ri uses OpenVINO [33] to convert the model for the NCS2 and the FPGA uses Vitis AI to convert the model for the DPU.
Table 1 shows that each device will have a different mAP, both the GPU and the Raspberry Pi have an mAP of 0.18, while the FPGA has an mAP of 0.35. This increase in accuracy is not expected as quantization typically reduces the accuracy of the model. After further investigations I believe that the increase was due to the dataset that was used for quantization. I inadvertently used the COCO validation dataset for quantization in Vitis AI and then used the same dataset for testing. Doing this possibly gave the model an opportunity to optimize the weights for the specific dataset that it would be tested on. These mAP results for the FPGA still do show that the FPGA is capable of producing good results. Further investigation into model accuracy on each device would be a potential area for future work.
Table 1 Baseline Performance Results

<table>
<thead>
<tr>
<th></th>
<th>Accuracy(mAP)</th>
<th>Inference Speed(FPS)</th>
<th>Filtering Speed(FPS)</th>
<th>Total Speed(FPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>0.18</td>
<td>24.5</td>
<td>142.3</td>
<td>20.9</td>
</tr>
<tr>
<td>RPI-Python</td>
<td>0.18</td>
<td>2.65</td>
<td>4.11</td>
<td>1.61</td>
</tr>
<tr>
<td>FPGA-Python</td>
<td>0.35</td>
<td>6.28</td>
<td>2.60</td>
<td>1.84</td>
</tr>
</tbody>
</table>

As expected, the speed of the GPU is faster for both inference and filtering than the embedded methods. While the speed of the GPU was higher, it was still lower than expected. In testing the GPU the implementation did not include batching, since the idea is to run inference on live video the buffering that is required for batching would add a considerable amount of latency that would be unacceptable for a real time guidance system.

Since this work focuses on the implementation of the embedded system, that is where the important data is. The inference of the Raspberry Pi is considerably slower than the inference of the FPGA. This could be attributed to any number of reasons, but the main difference is that Raspberry Pi and NCS2 are connected over the slower USB3 bus while the DPU and the FPGA CPU are using shared system memory with considerably more bandwidth and lower latency. The other difference is that the inference data type is different, the NCS2 uses a 16-bit floating point data type, compared to the DPU’s INT8 data type, floating point data types require more complicated hardware implementations and are generally slower than integer or fixed-point operations. The speed of the filtering is determined by the architecture and the clock speed of the CPU in the device. The Raspberry Pi most likely has a faster CPU core than the one on the FPGA.

Table 2 Baseline Power Results

<table>
<thead>
<tr>
<th></th>
<th>Avg Power (mW)</th>
<th>Inference Perf/Watt</th>
<th>Filtering Perf/Watt</th>
<th>Total Speed Perf/Watt</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPI-Python</td>
<td>5943.20</td>
<td>0.45</td>
<td>0.69</td>
<td>0.27</td>
</tr>
<tr>
<td>FPGA-Python</td>
<td>8104.49</td>
<td>0.78</td>
<td>0.32</td>
<td>0.23</td>
</tr>
</tbody>
</table>
Table 2 shows the performance per watt of the Raspberry Pi is higher than the FPGA.

Note: Due to not having the required hardware to measure the power draw of the system the GPU power and performance per watt are not considered in this work. Since this work is mostly focused on the performance of the embedded system, those power consumption numbers are more significant.

These initial results are a good starting point to indicate where the project started and what it needs to accomplish to be considered successful. The proposed improvements for this work are discussed in the next section.
Now that the baseline has been created the next step is figuring out what optimizations can be made to the FPGA device. For this work there will not be any changes to the Raspberry Pi implementation. The main algorithm that this work will attempt to improve is the implementation of the filtering algorithm. On the FPGA the filtering can be improved in two main ways first by using more efficient programing languages second implementing filtering on fabric (Fabric is used to describe an FPGA implementation in this work).

Filtering Using C++

The Python filtering is extremely slow, it is slower than the operation of the DPU. One solution to fix this would be to implement a filtering algorithm using a more efficient programing language. C++ is a compiled language instead of an interpreted language [34], this means that code can be optimized by a compiler to increase performance. A filtering algorithm written in C++ should behave exactly the same as the Python implementation. It would take the same parameters and perform the same operations, but it would be implemented as a shared library that is accessible to Python. This allows Python to run the accelerated C++ function without needing to write the entire implementation in C++.

Filtering using C++ could be faster, but the filtering would still take place on the CPU taking up valuable CPU resources that could be used for other operations. Another option would be to implement the same filtering algorithm in the FPGA fabric.

Filtering Using Fabric

The DPU is implemented in the FPGA but there is still additional space to put in an additional functional unit, therefore filtering could be implemented in the logic. Adding filtering in hardware would allow the CPU to offload filtering of the model outputs and would free the CPU to perform other tasks.
Filtering in fabric would be a considerable undertaking as it would require writing hardware that performs the operations. The fastest way to do it would be to use an HLS tool that allows programming in C to create RTL (For more information about HLS refer to the HLS Background section.).

Another complication with implementing filtering in the PL would be communication between the CPU and the filtering algorithm. Communication would need to be handled by implementing an AXI interface in the block (Information about the AXI protocol is discussed in the ZYNQ background section.). A software/hardware driver would also need to be written to handle this interaction on the CPU side.

The software driver would typically be written in C, but with the availability of the PYNQ library the driver can be written in Python to speed up development (Information about the PYNQ library is available in the PYNQ background section.).

**Pipelining**

While improvements to a single functional unit can help improve performance, there is a large problem, a functional unit sits idle for a considerable amount of time. A sequential implementation does not make use of more than one functional unit at a time. This can be fixed by submitting the data to the next functional unit as soon as it is ready and instead of sitting idle waiting for it to complete the job. Simply start the next job in the other functional unit so that both are being utilized at the same time. Using more than one function at a time to improve throughput is called pipelining. Theoretically adding pipelining to the design could increase performance up to the slowest functional unit (if there was no CPU overhead). For example, in the baseline design, pipelining the DPU so it is busy while the CPU is filtering could yield performance that approaches 2.8FPS. This idea of overlapping the functional units to create pipelining is shown in Figure 11.
Improved Model Training

For the baseline testing I have performed inference using a model that was trained on the COCO2017 Dataset [28]. More extensive model training and tuning could improve the accuracy of the model and make it better at detecting objects.
Now that there has been a discussion about the background and the areas of potential improvements, I can now discuss the implementations of the ideas and analyze the results.

Experimental C++ Implementation

Filtering in C/C++ was done by writing a filtering function in C/C++ that was callable through the main Python script. Calling C code from the python script achieved by using pybind11 which is a C++ and Python library that allows for communication between functions with shared types [35]. Pybind11 essentially enables writing specific functions in C++ without needing to rewrite the entire application in C++. Rewriting the entire application would have taken considerably more time and effort.

The C++ filtering method was implemented before the HLS code, doing it this way served two purposes, first it introduced another method for filtering, second it introduced the memory addressing scheme that would be used in the Fabric implementation without writing the full hardware implementation. Starting with C++ allowed for a more rapid introduction to the memory addressing scheme required for this work.

Just like the baseline implementation these results were gathered by running through the COCO validation dataset (more information can be found in the baseline section). The results from these tests are compared to the baseline Python filtering and are summarized in Table 3.

<table>
<thead>
<tr>
<th>Method</th>
<th>Total Inference FPS</th>
<th>DPU FPS</th>
<th>Filtering FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA-C++</td>
<td>4.46</td>
<td>6.28</td>
<td>15.4</td>
</tr>
<tr>
<td>FPGA-Python</td>
<td>1.84</td>
<td>6.28</td>
<td>2.60</td>
</tr>
</tbody>
</table>
In this implementation the DPU was running at 200MHz with a 400MHz DSP (Digital Signal Processing) clock.

In the same manner as the baseline the results were gathered by averaging the time it takes to process each frame for the entire validation set. The results of the C++ filtering show that there is a considerable increase in performance over the Python filtering. While results for filtering are about five times faster, the total inference throughput is only about twice as fast. This additional time is because the CPU has to perform both operations sequentially meaning that the time it takes to perform both steps is the sum of the times for each functional unit.

Table 4 shows the mAP from the Python and the C++ filtering methods. Changing the implementation of the filtering from Python to C++ has a minimal impact of model accuracy.

<table>
<thead>
<tr>
<th>FPGA COCO</th>
<th>Accuracy (mAP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>0.346</td>
</tr>
<tr>
<td>C++</td>
<td>0.344</td>
</tr>
</tbody>
</table>

Experimental Fabric Implementation

After implementing a function using C++ it was possible to transfer the knowledge to writing a function in HLS. Before the module was written a testbench needed to be created.

Setting up a Testbench

Creating the Programmable Logic implementation is considerably more effort than creating a C++ function. One of the biggest challenges with hardware is debugging when it is running. Debugging hardware is considerably different when compared to debugging software. Therefore, it is best to debug before it is implemented in hardware. Debugging before hardware implementations is done using simulations.
Functionally the block needs to process the DPU output and create a list of bounding boxes. The simulation acts as a wrapper around this block providing it with the necessary inputs and outputs to emulate system behavior.

The simulation for this block consists of several components. First the DPU results for an image were saved to a file. Second the results would be read by the testbench and passed to the HLS function. Once the HLS code was done processing the model output it would save the result to a file where a Python script could be used to draw the bounding boxes on an image. An example output from the testbench is shown in Figure 12.

![Figure 12 Fabric Filtering Simulation Result](image-url)
FPGA Fabric Experimental Implementation

Filtering implemented on the FPGA fabric is a complex block that needs to perform the same functions as the C++ and Python filtering algorithms. In the simplest form it reads the model output from memory, processes data and then saves the data back to memory. Figure 13 shows the block diagram for the system with both filtering and DPU implemented in the PL.

![Figure 13 System Block Diagram with Filtering and DPU](image)

The Block diagrams shows that the DPU and the Filtering block are both connected to an AXI interconnect that is implemented in the PL. The PL interconnect is then connected to an interconnect on the PS side which can connect directly to the ARM CPU as well as the memory controller. Each block connected to the AXI interconnect has a set of AXI4-Lite registers that control the modules, and each block also has a Direct Memory Access (DMA) to allow it to directly communicate with system memory.
Inside the filtering block contains several steps, Figure 14 shows the top-level block diagram for the early implementation of the filtering block.

![Figure 14 Fabric Experiment Block Diagram](image)

The block diagram in Figure 14 goes from left to right in order of operations. First the control registers are set, and the filtering is initiated. The DMA gets the model output from system memory to be used for filtering. The filtering block performs the filtering algorithm the result is then given to the bounding box scaler. The bounding box scaler scales the bounding box to ensure that it is scaled for the original image size. The processed result is stored in back in system memory using the output DMA. Finally, once filtering is complete
the control logic asserts a done bit in the control register to indicate that the results are ready for the CPU to access.

**Fabric Control Registers**

Control registers control operation of the block such as I/O mapping, filtering parameters and block control, the control register parameters are:

- Confidence Threshold
- Number of Classes
- Original Frame Size
- Scaled Frame Size
- Memory Location of Model Output
- Shape of Model Output
- Memory Location to Store the Result
- Block Control

The confidence threshold register controls the minimum confidence the model needs to have for the bounding box to not be thrown away. Confidence threshold can be controlled by software and will change how many bounding boxes are returned from the filtering.

The number of classes register controls how the memory address is traversed, as mentioned in the Memory Addressing of Multidimensional Arrays section the size of any one-dimension controls how the linear memory addresses are broken up. Without the number of classes parameter, the model would not be able to change the addressing scheme and therefore would not support models with different numbers of classes.

Original frame size and scaled frame size registers are used to scaling of the bounding box and ensures that when the bounding box is drawn back on the original image it is the correct scale and in the correct location.

The memory of the model output register indicates where in memory the output from the DPU is stored and is used as the address the DMA will access.
The yolo model shape register controls whether the block is reading a 52x52, 26x26 or a 13x13 yolo output. This register is important to make sure that the memory addressing is correct.

The memory location registers control where the output DMA will store the results, so the CPU knows where to find them.

The block control register handles starting filtering and is started by the CPU after all the parameters in the previous registers are set. The block control register also contains a flag to indicate when a block has finished filtering, and the data is ready for the CPU to access.

**Fabric Results**

The results of fabric filtering compared to Python filtering are shown in Table 5. Like the C++ experiment the values were taken from running the COCO validation set and averaging the total framerate.

<table>
<thead>
<tr>
<th>Method</th>
<th>Total Inference FPS</th>
<th>DPU FPS</th>
<th>Filtering FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA-Fabric</td>
<td>3.96</td>
<td>6.28</td>
<td>10.7</td>
</tr>
<tr>
<td>FPGA-Python</td>
<td>1.84</td>
<td>6.28</td>
<td>2.60</td>
</tr>
</tbody>
</table>

For this version of the fabric block the clock speed was at 200MHz, anything higher than this would cause timing violations, the DPU was also running at 200MHz with a 400MHz DSP clock.

Like the C++ result the total inference framerate is slower than each individual component. The filtering frame rate was greatly improved, but the total inference framerate is still relatively low.

Table 6 shows that there is a minimal change to model accuracy while using a fabric filtering implementation, this is expected because the algorithm that implements the result is similar, and the data types used are the same.
## Table 6 Fabric Experimental mAP

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy (mAP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA-Fabric</td>
<td>0.344</td>
</tr>
<tr>
<td>FPGA-Python</td>
<td>0.345</td>
</tr>
</tbody>
</table>

The small difference in mAP could be due to the way that the output from the fabric implementation is saved to system memory. Unlike software that supports dynamic memory allocation the filtering block needed to select an output buffer size that results in a fixed number of total outputs. Any additional bounding boxes are discarded resulting in a small amount of lost information. The fabric implementation is limited to 192 possible bounding boxes before non-max suppression.

### Comparing Experimental Fabric and C++ Filtering

Now that both C++ and fabric filtering implementations have been tested, it is important that their results are compared to each other. Both implementations implement the same algorithms. The C++ algorithm works on the CPU, and the fabric implementation offloads the algorithm to a functional unit implemented in the FPGA. Figure 15 shows the total frame rate for each implementation.
The results in the graph show that filtering using Fabric and C++ are considerably faster than Python filtering. The total speed of the FPGA-C++ implementation is faster than FPGA-Fabric implementation. Table 7 summarizes the speed breakdown for each functional unit. The DPU framerate is constant since the model is the same for each implementation. The filtering performance shows that the C++ implementation is about 50% faster than the fabric implementation.

**Table 7 Experimental Results Speed Comparison for Implementations**

<table>
<thead>
<tr>
<th>Method</th>
<th>Total Inference FPS</th>
<th>DPU FPS</th>
<th>Filtering FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA-Fabric</td>
<td>3.96</td>
<td>6.28</td>
<td>10.7</td>
</tr>
<tr>
<td>FPGA-C++</td>
<td>4.46</td>
<td>6.28</td>
<td>15.4</td>
</tr>
<tr>
<td>FPGA-Python</td>
<td>1.84</td>
<td>6.28</td>
<td>2.60</td>
</tr>
<tr>
<td>RPI-Python</td>
<td>1.61</td>
<td>2.65</td>
<td>4.11</td>
</tr>
</tbody>
</table>

The results for performance per watt shown in Figure 16 shows that the C++ implementation has the highest filtering performance per watt.
While the C++ filtering is faster there is still an advantage to the fabric implementation, the operations are offloaded from the CPU allowing for the CPU to perform tasks while waiting for filtering to be completed. This idea is explored next in the pipelining experiments section.

Pipelining Experiments

Many of the improvements in this work come from experiments to enable pipelining. There were a few iterations that happened while experimenting with pipelining.

The first version of pipelining that was implemented was by creating a fabric design with 3 paths in PL. This was done to reduce the number of times that the fabric function needed to be called. The implementation in the fabric experiments section needed to be called for each model output. Repeatedly calling the fabric function would prevent pipelining from being possible. The first step was to shift the three function calls to fabric and only require the CPU to initiate the filtering once. However, during testing, I found that the sigmoid function in the driver was slowing down the fabric implementation. Therefore, the sigmoid function in the driver needed to be changed.
The sigmoid function was originally implemented in the driver and is applied to the data before it is sent to the PL. The sigmoid function is a nonlinear activation function that is very common in machine learning. The sigmoid function is defined in Equation 5.

\[ \sigma(x) = \frac{1}{1 + \exp(-x)} \]  

Equation 5

The graph for the sigmoid function is shown in Figure 17.

---

**Figure 17 Sigmoid Activation Function**

The sigmoid function is bounded from (0, 1) and only changes for a small region around 0. This function behavior makes it a very suitable candidate for a Look Up Table (LUT). LUTs are very powerful tools for FPGAs, instead of calculating a result every time it is needed, the result is precalculated and stored in an array. This can significantly improve
performance in a design as it replaces complex math functions with very simple memory calculations and memory accesses. The only disadvantage to using a LUT is the loss in precision due to the reduction in resolution. The input to the sigmoid function is a 32bit float, but a reasonable LUT would be between 8 and 12 bits. An 8-bit LUT has only 256 possible entries, but with a simple workaround this lack of resolution did not turn out to be an issue for this design.

The DPU quantizes the input into an 8-bit resolution meaning that the output is only represented by 8-bits. Therefore an 8-bit LUT would not lose resolution. The only issue that exists with the DPU output is that the quantization is not consistent. During experimentation the result at the output of the DPU was in whole number increments, half number increments and quarter number increments. To accommodate this a LUT was designed to fit in 256 entries but only in a small range of values. Since the sigmoid function is constant once x is greater/less than +/- 8 there would be wasted entries in the table if a large range was used. To avoid wasted entries a piecewise function is used, the function is described in Equation 6.

$$\sigma_{lut}(x) = \begin{cases} 
1 & \text{if } x \geq 8 \\
\sigma_q(x) & \text{if } -8 < x < 8 \\
0 & \text{if } x \leq -8 
\end{cases}$$

Equation 6

Where $\sigma_q(x)$ is the sigmoid function in 0.0625 increments from -8 to 8.

Once the sigmoid LUT was created it was integrated into the design. The updated fabric design that includes the sigmoid LUT is shown in Figure 18.
Figure 18 Fabric Block Diagram with Sigmoid LUT

This new design implements the changes to enable pipelining, the sigmoid LUT as well as increasing the DPU clock speed from 200MHz to 295MHz. The fabric implementation from this section is the final fabric implementation for this work.

Table 8 Final Fabric Implementation mAP comparison

<table>
<thead>
<tr>
<th></th>
<th>mAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fabric-Async</td>
<td>0.328</td>
</tr>
<tr>
<td>Fabric</td>
<td>0.344</td>
</tr>
<tr>
<td>C++</td>
<td>0.344</td>
</tr>
<tr>
<td>Python</td>
<td>0.346</td>
</tr>
</tbody>
</table>
The mAP for the synchronous fabric is the same as the mAP for the C++ filtering indicating that implementing sigmoid in a LUT has no impact on model accuracy.

For the asynchronous result I believe that the poor mAP is due to an error in my evaluation script causing it to miss 1 frame every batch resulting in a lower total mAP. The asynchronous code is the only version to use the modified section of the evaluation script as the backend needed to be different to get the startup overhead to work.

Next is to discuss the speed results that were achieved by offloading the sigmoid function to the DPU. Speed results after implementing the sigmoid LUT into the design are shown in Figure 19 and Table 9.

Note that the filtering speed for the asynchronous implementation is derived as the time it takes to initiate the transaction and the time it takes to finish the transaction after filtering is complete.

<table>
<thead>
<tr>
<th></th>
<th>DPU Speed (FPS)</th>
<th>Filtering Speed (FPS)</th>
<th>Total Speed (FPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fabric-Async</td>
<td>8.98</td>
<td>15.46</td>
<td>5.75</td>
</tr>
<tr>
<td>Fabric-Sync</td>
<td>8.97</td>
<td>16.82</td>
<td>5.85</td>
</tr>
<tr>
<td>C++</td>
<td>8.97</td>
<td>15.63</td>
<td>5.70</td>
</tr>
<tr>
<td>Python</td>
<td>8.97</td>
<td>2.57</td>
<td>2.00</td>
</tr>
</tbody>
</table>
After moving the sigmoid LUT off the CPU onto the PL the performance of fabric filtering increased to 15.46 for the asynchronous implementation and 16.82 for the synchronous implementation. Both of the fabric implementations are on par with the C++ filtering performance. The total speed of each implementation is also very similar at 5.7-5.85 FPS. These results, while higher than the Python implementation, are still lower than desired.

**Figure 19 Final Fabric Experiment Comparison**
Figure 20 Fabric Latency Comparisons

Figure 20 shows that while the DPU still takes up most of the execution time the filtering also takes up a considerable amount of time, but when compared to the original Python implementation there is a significant performance improvement.

As the filtering implementation takes less time than the total speed will start to approach the speed of the DPU, unless the DPU is able to speed up, the implementation will be bottlenecked and will not be able to get any faster. Speeding up the DPU via model pruning is explored in the next section.

Model Pruning

Model pruning is a way to reduce the complexity of a model without a significant impact on model accuracy [11]. A pruned model was tested for this work, but due to issues with model training a model that was already trained, pruned, quantized and compiled by AMD/Xilinx and was downloaded from their Vitis AI Zoo library [31].

Like the other experiments, the performance was measured on the COCO validation dataset and the framerate was taken as an averaged over the entire set.
The results are summarized in Table 10. The results show that the performance of the pruned model is better than both the baseline and the accelerated implementations. The execution time of the DPU is reduced and therefore the entire implementation is faster.

<table>
<thead>
<tr>
<th></th>
<th>Baseline Speed (FPS)</th>
<th>COCO Final Speed (FPS)</th>
<th>Final Pruned Speed (FPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>4.46</td>
<td>5.70</td>
<td>6.85</td>
</tr>
<tr>
<td>Fabric</td>
<td>3.96</td>
<td>5.85</td>
<td>6.97</td>
</tr>
<tr>
<td>Fabric-Async</td>
<td>0.00</td>
<td>5.75</td>
<td>6.95</td>
</tr>
</tbody>
</table>

The mAP of the two models is compared, Table 11 and shows that there is a small difference between the model that I trained and the model that AMD/Xilinx trained and pruned. There is a small difference in the asynchronous mAP, but that may be down to an evaluation script error.

<table>
<thead>
<tr>
<th></th>
<th>Pruned Accuracy (mAP)</th>
<th>Original Accuracy (mAP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>0.326</td>
<td>0.344</td>
</tr>
<tr>
<td>Fabric</td>
<td>0.326</td>
<td>0.344</td>
</tr>
<tr>
<td>Fabric-Async</td>
<td>0.304</td>
<td>0.328</td>
</tr>
</tbody>
</table>

These results show that a pruned model can improve performance with minimal impact on model accuracy.

**Camera Implementation.**

The main application for this work would be applying machine learning inference in a real-time environment. This kind of environment requires a design with an image sensor instead of images that are loaded in from a file. The camera implementation was tested using a Logitech USB webcam [36], but in actual deployment it would be likely that an image processing pipeline would be built in the PL alongside the DPU and filtering logic. A screen capture of the results is shown in Figure 21. The screen capture shows an inference time of 135.4 ms which translates to 7.3 FPS. These results were captured using asynchronous fabric implementation with the pruned model trained on the COCO dataset.
Figure 21 Camera Experiment Screen Capture
Chapter 6
Final Implementation and Results

The Satellite Vision Implementation

In the previous sections the implementations tested using the COCO dataset. This results section will cover the final implementation and apply what was learned during experimentation to inferring satellite components. This section is broken into a few sections:

1. Introducing the satellite dataset
2. Discuss the final device architecture
3. Discuss how the experiment was setup
4. Analyze the and compare the performance of device implementations
5. Show example inference frames from each device
6. Summarize Results

The Satellite Dataset

The dataset that was used in this work has been used extensively in previous works and has been shown to successfully train models for satellite component feature extraction [6] [37] [5] [38] [18]. The dataset was created by gathering images available on the internet and labeling them. The images that were gathered are mainly represented by synthetic spacecrafts, but there are some pictures of real spacecrafts. The Dataset contains 1,231 Images with 7,971 annotations across 4 classes. The four classes that are in the dataset are antenna, body, solar and thruster. However, the thruster annotations were not included because thrusters generally produce poor results. The dataset was broken into three sets, training, validation and testing. The training set was used for model training and contained data augmentation to synthetically increase the size of the dataset. The validation set was used for FPGA model quantization in the Vitis AI flow. The testing set was used to test the
performance of each device. The testing set contains 40 images from the labeled dataset. Figure 22 shows an example image from the testing dataset.

![Example Image From the Testing Dataset.](image)

**Device Architecture**

The device architecture is the same as discussed in previous sections. The PS handles control of the DPU and filtering blocks which are implemented in PL. The filtering block is the same one used in previous sections and is shown Figure 18 and is described in the Pipelining Experiments section.

**Model Selection and Training**

The model that was used for satellite vision was the still the YOLOv4 [17] implementation, it was trained using [21]. This is the same model architecture that was used in the previous sections with the same model modifications mentioned in DPU background section.
Due to complications with model training the model that was used for this work is the exact same model that was used for my previous work [6] with the only change being that it was recompiled to work with Vitis AI 2.5 and the new system architecture. Due to this limitation, it is still expected that further model training could increase accuracy.

The same trained model was used for each device, for the FPGA the model went through the Vitis AI conversion flow, the NCS2 it went through Intel’s conversion flow and the GPU the model was used as is.

**Experimental Methods**

Each device was tested on the testing set and while the tests were running the performance of the device was logged and the power consumption was measured using a USB-C power tester [30]. The power results were averaged over time to get average power consumption. After the model runs inference, the model is evaluated to determine its mAP. The devices/configurations that were tested were:

- FPGA Python Filtering
- FPGA C++ Filtering
- FPGA Fabric Filtering
- FPGA Async Fabric Filtering
- Raspberry Pi Python Filtering
- GPU

For all of the tests the model was ran with a confidence threshold of 0.5 and an IOU parameter of 0.5.

**Device Performance**

The device performance results are broken in to two tables, Table 12 contains the information about model accuracy and device speed, Table 13 contains the information about power and performance per watt.
Starting with model accuracy Table 12 shows that inference on the GPU produces the highest mAP, followed by the Raspberry Pi and NCS2 then lastly the FPGA has the lowest mAP result. This order of results could be explained by the decreasing level of model precision as datatypes are changed. The GPU uses a 32-bit floating point representation, the NCS2 uses a 16-bit floating point representation, and the FPGA DPU uses an 8-bit integer representation. This result was supposed to be improved in this work, but due to model training issues it was not retrained or re-quantized, leading to poor accuracy on the FPGA. Attempting to improve the model’s accuracy would be a good area for future work.

<table>
<thead>
<tr>
<th></th>
<th>Accuracy(mAP)</th>
<th>Inference Speed(FPS)</th>
<th>Filtering Speed(FPS)</th>
<th>Total Speed(FPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>0.20</td>
<td>26.00</td>
<td>1467.00</td>
<td>25.54</td>
</tr>
<tr>
<td>RPI-Python</td>
<td>0.17</td>
<td>2.87</td>
<td>4.61</td>
<td>1.77</td>
</tr>
<tr>
<td>FPGA-Python</td>
<td>0.07</td>
<td>9.87</td>
<td>3.01</td>
<td>2.31</td>
</tr>
<tr>
<td>FPGA-C++</td>
<td>0.07</td>
<td>9.87</td>
<td>140.51</td>
<td>9.21</td>
</tr>
<tr>
<td>FPGA-Fabric</td>
<td>0.07</td>
<td>9.87</td>
<td>23.59</td>
<td>6.96</td>
</tr>
<tr>
<td>FPGA-Async-Fabric</td>
<td>0.07</td>
<td>9.87</td>
<td>19.38</td>
<td>7.00</td>
</tr>
</tbody>
</table>

The next set of data is the speed of each implementation. It is shown that the C++ implementation is considerably faster than the fabric implementation. The C++ implementation is notably faster than it was in the experimental section on the COCO dataset. The fabric implementation is the next fastest with there being little difference between the synchronous and asynchronous implementations. However, it is interesting to note that the filtering speed of the asynchronous implementation is notably lower, but the total inference speed is about the same as the synchronous version. Both Python implementations were predictably slow just like they were in the baseline implementation. All of the models saw a slight increase in inference speed when compared to the COCO model. This speedup can be attributed to the reduction in the number of classes, with fewer number of classes there are slightly fewer parameters that need to be processed. The reduction in classes also speeds up the filtering speed due to the fewer number of classes.
that need to be copied from memory. Since the size of the output is dependent on the number of classes in the model.

**Table 13 Satellite Model Power Results**

<table>
<thead>
<tr>
<th>Method</th>
<th>Avg Power(mW)</th>
<th>Inference Perf/Watt</th>
<th>Filtering Perf/Watt</th>
<th>Total Speed Perf/Watt</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPI-Python</td>
<td>5676.92</td>
<td>0.5</td>
<td>0.81</td>
<td>0.31</td>
</tr>
<tr>
<td>FPGA-Python</td>
<td>8295.65</td>
<td>1.19</td>
<td>0.36</td>
<td>0.28</td>
</tr>
<tr>
<td>FPGA-C++</td>
<td>8800.00</td>
<td>1.12</td>
<td>15.97</td>
<td>1.05</td>
</tr>
<tr>
<td>FPGA-Fabric</td>
<td>8100.00</td>
<td>1.22</td>
<td>2.91</td>
<td>0.86</td>
</tr>
<tr>
<td>FPGA-Async-Fabric</td>
<td>8700.00</td>
<td>1.13</td>
<td>2.23</td>
<td>0.80</td>
</tr>
</tbody>
</table>

Table 13 shows the power results from testing. Note that the GPU power results were not included because of the lack of tools to test total system power draw. The table shows that the C++ results were able to produce the results with the highest performance per watt of any other method with a value of 1.05 frames/watt. The second-best method is the synchronous fabric implementation followed by asynchronous fabric. All 3 enhanced filtering algorithms were able to considerably improve the performance per watt of the system. This increase in performance per watt indicates that the work was successfully able to make better use of the hardware than any implementation in of my previous works [6] [5].

**Example Satellite Results**

Figure 23 shows an example inference frame from the FPGA, Figure 24 shows an example inference frame from the Raspberry Pi and Figure 25 shows an example inference frame from the GPU.
The bounding boxes from the FPGA look to be somewhat loose in their fitting around the objects, but they are not terrible, and the confidence for each component looks good.
Figure 24 Example Detection on Raspberry Pi

The bounding boxes look similar here to the FPGA, with slightly different confidence values.
Figure 25 Example Detection on GPU

The GPU result looks similar to the other two results, just with higher confidence values.

Summary

The results show that the FPGA was able to improve performance, while it was expected that the fabric implementation would be the fastest, based on experimental results, the C++ implementation was the fastest filtering method for this dataset.

However even with the improved performance more work needs to be done to improve model accuracy as the FPGA has the worst model accuracy on the satellite dataset.
Chapter 7
Areas of Future Improvements

There are several areas of this work that could provide better performance, some of them are listed here.

The final implementation is still bottlenecked by the speed of each functional unit. It was shown in the results section that the asynchronous version was not actually faster than the synchronous version, meaning that pipelining the device was not successful as intended. If pipelining could be properly implemented, then there is a clear area for improvement.

The implementation in fabric probably has room for improvement, it is a very unoptimized implementation, but it is possible that with further optimization it could run considerably faster while using less logic resources.

The current implementation still implements Non-Max Suppression in Software, implementing this in hardware could speed up total speed.

The speed up that was observed using C++ filtering implies that rewriting the entire codebase in a compiled language could considerably improve performance. Writing the codebase in C++ could also allow for more complex software design that can use more than a single CPU core.

Due to the issue with model converting for the satellite vision model there was no chance to try to improve the model in this work. Therefore, further model training/tuning could still lead to an improvement in model usability.

Model pruning as shown in the experimental section was able to improve the speed with a minimal impact to model accuracy.
Looking at a different model backend that is not as large could yield a performance improvement. The Darknet53 backend that was used is very large and could probably be replaced with a smaller model with an acceptable drop in model accuracy.
Chapter 8
Conclusions

Using an AMD/Xilinx development kit this work shows that it is possible to run a YOLOv4 algorithm to detect satellite components at 9 frames per second while using less than an average of 9W. The work concludes that implementing some of the post processing algorithms in C++ was the best option and was able to achieve an efficiency of more than 1 frame per watt.

While these results are better than the baseline it is believed that it is possible that further improvements could enable faster inference performance, and improved model accuracy.
References


### Appendix

## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>AMBA</td>
<td>Advanced Microcontroller Bus Architecture</td>
</tr>
<tr>
<td>AP</td>
<td>Average Precision</td>
</tr>
<tr>
<td>AXI</td>
<td>Advanced eXtensible Interface</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Network</td>
</tr>
<tr>
<td>COCO</td>
<td>Common Objects in COntext</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DPU</td>
<td>Deep neural Processor Unit</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FPS</td>
<td>Frames Per Second</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HLS</td>
<td>High Level Synthesis</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>LUT</td>
<td>Look Up Table</td>
</tr>
<tr>
<td>mAP</td>
<td>Mean Average Precision</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multi-Processor System on Chip</td>
</tr>
<tr>
<td>NCS2</td>
<td>Neural Compute Stick 2</td>
</tr>
<tr>
<td>PL</td>
<td>Programable Logic</td>
</tr>
<tr>
<td>PS</td>
<td>Processing System</td>
</tr>
<tr>
<td>ReLU</td>
<td>Rectified Linear Unit</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SOC</td>
<td>System On Chip</td>
</tr>
<tr>
<td>SOM</td>
<td>System On Module</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>VHDL</td>
<td>(Very High Speed Inter-Connect)</td>
</tr>
<tr>
<td></td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>YOLO</td>
<td>You Only Look Once</td>
</tr>
</tbody>
</table>