Towards On-Orbit Machine Learning in Picosatellites

Jack Spreckley (2383831S)

May 23, 2023

ABSTRACT

The current method of downlinking the data collected by satellites to Earth causes high latency and power consumption and therefore is unsuitable for Picosatellites with limited system resources. On-Orbit Machine Learning (on-orbit ML) offers a solution by decreasing the amount of data to transmit. However, while ML methods exist for use in Picosatellites, they lack suitable recordings of system measurements such as power, memory, and processing time, which hinders their implementation. We address this issue by testing various ML models with compression methods and lightweight architectures on edge setups, using a mix of hardware and software optimisations, while focusing on different processing architectures, accelerators, and delegate improvements. We found that different combinations of models and edge setups for various goals and limitations that Picosatellites can have, providing valuable insights for practical applications.

1 Introduction

Over the last half-century artificial satellites have found many applications that have benefited large segments of the world population. For example, telecommunications, weather forecasting, navigation, and crop forecast, to name a few [21]. These are all technologies that modern life depends on. These traditional satellites have limited scalability and adaptability due to their high costs and outdated components compared to a new collection of satellites called *Smallsats* [7, 30, 4].

Smallsats have been developed using the recent miniaturisation of computers and hardware, this collection includes: Minisatellites, Microsatellites, Nanosatellites, Picosatellites and Femtosatellites [15, 2]. These satellites have the advantages of small size, low power consumption and short development cycle. Smallsats have caused a record number of satellites to be launched into space in 2021 at a greatly reduced cost over previous years [2, 3]. Smallsats also have a reduced cost of entry into space due to their small size and cheaper components [31, 3]. For example, the small size allows many small satellites to be fitted onto a single launch, the cost of the launch is greatly reduced for each individual satellite [31]. This reduced cost could allow many more small groups or individuals to take advantage of and benefit society with this technology.

For the above reasons, Smallsats could be starting a new age of satellites which will be even more impactful than the last age, allowing for numerous new applications and more specialisation with *greater experimental risk-taking* due to the reduced cost [31, 7, 30, 3]. Satellites create a vast amount of data from recordings, be they images or radar information. This data is then transferred back to ground stations on Earth to be analysed. This supply of data has been ideal for exploiting the development of image processing with *computer vision* (CV), *machine learning* (ML) and its ever-improving pattern recognition capability enabled by *deep learning* (DL) [9, 32, 28].

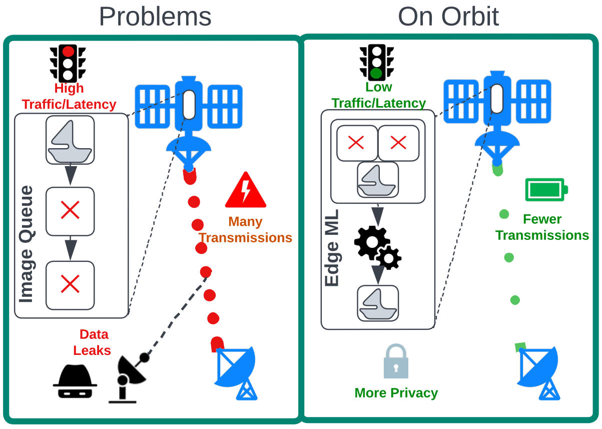


Figure 1: On the left the current problems with satellite data collection. On the right the benefits of On-Orbit processing.

The current method of satellites sending data back to Earth is limiting as vast amount of this data is not relevant for the individual satellite application and can cause harm [5, 22, 27]. This method also increases latency, causing delays in critical information being returned due to the bandwidth constraints, so a traffic jam of unuseful data occurs, see Figure 1 (left) showing these current problems (the image queue with the red traffic light). Moreover, the figure shows this large amount of unneeded data requires a greater use of the transmitter which consumes power and requires processing that could have been allocated to other tasks [17, 18].In addition, the figure demonstrates that privacy could be breached through unauthorised recipients intercepting personal information.

The above issues calls for *on-board* or *on-orbit processing techniques* that execute data processing algorithms such as ML and DL on a computing chip in the satellite allowing the unnecessary data to be filtered out before the costly transfer. This follows the edge computing paradigm, which refers to performing computations close to the source of the data, usually with respect to mobile or IoT devices [10, 19]. On-orbit processing refers to edge processing onboard satellites in the space environment. This approach reduces latency by filtering out unnecessary data, reduces power consumption as data processing is more energy-efficient than transmitting all the data [18, 17] and provides increased data security by filtering out personal information, such as people's locations, that satellites may unintentionally capture or the use of encryption [18, 19, 28, 27]. Figure 1 (right) shows the benefits of on-orbit processing with ML and DL.

The miniaturisation of satellites has caused deployable computing platforms to be even more volume and power constrained, resulting in such platforms having lower computing resources [4, 35, 18]. This, compounded with the high computational workload of ML and DL algorithms, causes a significant challenge when deploying on-orbit processing [10, 36]. However, currently they cannot be applied to many real-world applications as they lack or have ineffectively approximate resource usage measurements such as power and memory consumption. This works primarily focuses on enhancing the on-orbit ML literature in the following ways.

1. Addressing the existing literature gap in system measurements, such as power, memory, and CPU usage.
2. The development of lightweight edge models using a variety of neural architectures and compression techniques.
3. Setting up a broad range of edge boards, with variations in CPU architectures, hardware accelerators and operation delegates.
4. Using established system measurements with some variations from other edge computing areas for testing the suitability of ML models on different setups, which we have created for edge environments that minimise power and hardware resource consumption.
5. Identifying the best combination of model and accelerator by testing on a range of boards and accelerators over diverse Picosatellite applications based on specific requirements. Then evaluating these setups' performance on various detection models. Both in terms of precision and recall with the system evaluation metrics of power, memory and CPU usage.

An example of this is through Alba Orbital who were unsure how to apply ML to their resource-constrained Picosatellites due to a lack of such measurements. With this in mind we specifically focus on Picosatellites.

2 Related Work

While there is a larger body of work on edge ML and processing, this research focuses on the emerging niche of on-orbit ML. Due to its rapid development, no literature reviews are up to date and to our knowledge the most recent is [28]. Therefore, to start this work we conducted an integrative search [29]. This integrative search looked at the current literature on on-orbit ML tasks, see Figure 2. We considered object detection as there was a lack of suitable evaluations and the potential of its use. A more detailed explanation of the different techniques and why object detection was chosen is in Appendix A. Additionally, Table 7 in the Appendix shows the measurements, and techniques used by the Literature surveyed.

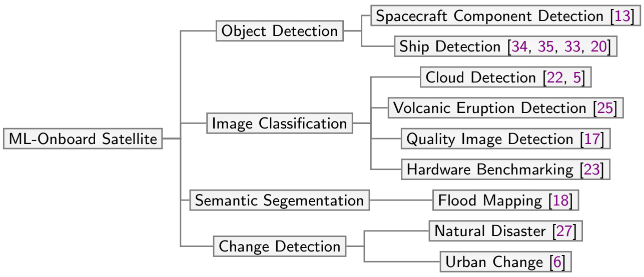


Figure 2: Taxonomy of the integrative search in on-orbit ML in satellites.

2.1 Hardware and Acceleration

To find accurate system metrics for comparing on-orbit ML approaches one must consider the available underlying hardware. This section reviews hardware used to simulate the constrained environment of satellites for on-orbit ML and its applicability to Picosatellites. A significant component is *acceleration*, which is the process of using specialised hardware to process tasks more effectively, usually by means of the parallelisation of processes.

System on a Chip Hardware. System on a Chip hardware is widely used and cost-effective making this class an appropriate candidate for Picosatellites. The Raspberry Pi 3b and 3b+ have been used to simulate on-orbit for flood mapping [18] and volcanic eruption detection [25]. Additionally, the Alba Orbital Picosatellites contain Raspberry Pi's, however it does have low memory access speed [10] making tasks needed for ML inefficient. The Odroid XU4 board is one of the most cost-effective similar boards, though not used yet in the literature.

Hardware Acceleration. The literature shows the advantages of hardware acceleration [33, 22, 13, 18, 5, 20, 23] which can be backed up with appropriate recording in [33, 22] and partially in [20]. Hardware acceleration in FPGAs shows the most comprehensive results for memory and time for system measurements for on-orbit image classification [23]. In addition the best fitting paper for my task and requirements SAR ship-detection [33] is FPGA accelerated but its implementation of FPGAs uses older ML models and software optimisation, this is a trend throughout hardware acceleration [20].

FPGAs has been used by [20, 34, 22] for ship detection. In particular, [34] presents SAR ship detection using a state-of-the-art model running on a Xilinx Virtex 5QV FPGA only consuming 1.2W ideal for Picosatellites. This space-grade chip is however expensive and may not be suitable for Picosatellites which have around a 2-year lifespan and are usually low cost. Hence, a cheaper FPGA for Picosatellites is a better option. New developments have made FPGAs more accessible such as the PYNQ development board. These small FPGAs meet the requirements for Picosatellites, but further work is needed to evaluate if and how well object detection models can perform on such chips especially as there is little software tooling available. A similar approach is followed by efforts deployed on ASICs boards, e.g., TPU and VPU. These promise similar performance benefits as FPGAs, but in a static non-reprogrammable approach.

GPU Hardware. For object detection, the Jetson TX2 module is used to simulate the capabilities of nanosatellites [35], and satellites [34]. This follows the general trend of ML acceleration at the edge. This module is known for effective GPU acceleration in terms of power consumption and floating point operations per second (FLOPs) [16]. However, it is unsuitable for modelling Nanosatellites and Picosatellites, such as the Unicorn-2, as the modules payload dimensions. are too large to fit within this Picosatellite's dimensions

GPU acceleration is used in [13] for spacecraft components and demonstrates its success. However the lack of any direct metrics makes drawing concrete conclusions from this risky, as it was not simulated in a constrained device. The reconfigurability of FPGAs allows for in-field updates and modifications, which can be crucial in dynamic space environments. Furthermore, FPGAs typically exhibit higher radiation tolerance than GPUs, making the former better suited for the harsh conditions encountered in space.

2.2 Machine Learning Models and Techniques

In this section, we explain the requirements of an ML model for it to be suitable for a Picosatellite, and what object detection models are currently used in the literature.

ML Model Criteria. A ML model for Picosatellites must consume a small amount of virtual and real memory, as well as limited power and processing, while still being accurate. Due to the literature [33, 1, 22] showing the effectiveness of hardware acceleration, especially with FPGA, the model must take advantage of this. Another requirement is a short runtime for the process from start to finish so the Picosatellite can complete its other tasks.

Object Detectors. Object detection is locating and classifying objects in images, usually in rectangular bounding boxes. Modern Object detection uses quick and computationally effective ways of finding objects

The main object detectors used are *Faster R-CNN* [24] and *YOLO* [34, 20]. Faster R-CNN (Faster Regions with Convolutional Neural Networks) is a two-staged detector meaning it first proposes regions in an image and then classify object and refines bounding boxes resulting in high accuracy at the cost of speed. YOLO (You Only Look Once) is a real-time one-stage detector that performs object localisation and classification without a separate region processing step known for speed accuracy. This model traditionally works by dividing the input image into a grid and predicts bounding boxes and class probabilities for each grid cell.

Faster R-CNN [24] uses ResNet50 [35], and therefore not suitable for Picosatellites as the model is too large. YOLOv2, now considered an older ML model, has lower resource usage, and better accuracy and detection speed so it is a good choice for object detection on Picosatellites. A hardware-accelerated YOLOv2 model on an FPGA for a space environment is implemented in [33], with results that are more effective than anything else found in the literature. Therefore, the aim of this work will use YOLO object detection over two-stage methods such as R-CNN, due to their lower memory and processing requirements shown effectiveness in resource-constrained environments [13].

In [34] the results of YOLOv5s are compared with their method lite-YOLOv5. The method lite-YOLOv5 was found to be a slight improvement over YOLOv5s, but is complex to implement for only a small gain in detection performance and FLOPs. YOLOv5s [11] is considered an improved version in terms of detection performance since the release of the paper and it fulfils the requirements mentioned in the model criteria, as well as the lighter version of YOLOv5n which is now available. To test a more recent ML model, YOLOv8, released in January 10th, 2023, is also considered. The main differences are an anchorless architecture and activation functions, there is not enough space to discuss the YOLO models, see Ultralytics for more details.

Finally, we mention [22] which demonstrates the effectiveness of *pruning* and *quantisation* in CV approaches. These are general techniques that still apply to parts of object detection so can still be applied to the object detection models.

Compression techniques. Quantisation [28] maps high-precision weights to lower-precision ones by rounding, e.g., floating point 32-bit (FP32) to 16-bit (FP16), reducing computation, memory, and power usage while trying to retain maximum prediction accuracy. This makes it appropriate for constrained devices, including Picosatellites, with their restraints on computation memory and power. However, to achieve practical speedups, the target hardware and framework must support lower-precision operations.

Edge ML Frameworks. ML frameworks are software libraries, interfaces or tools that make building and developing ML models faster and easier, example libraries are PyTorch and TensorFlow.

To deploy ML effectively on constrained devices they should work with a compatible edge framework supported by the target device. These specialised edge frameworks are optimised for low-latency-power and memory consumption on resource-constrained devices, examples include Caffe2 [5] (Convolutional Architecture for Fast Feature Embedding 2) and MX-Net [35].

We focus on TensorFlow Lite (TF-Lite) due to its support of a range of devices and documentation that makes utilising it simpler. It includes an interpreter for executing models and can compression techniques we discussed. Additionally, the interpreter can be set up to work with different hardware accelerators that we use and their corresponding libraries which would make testing multiple accelerators easier. A performance evaluation also showed TF-Lite outperforming other frameworks in terms of latency, memory requirement, and energy consumption across various edge devices [28].

2.3 Metrics

We now discuss the metrics used by the literature to report on the effectiveness of their on-orbit ML implementation and critically evaluate their suitability.

Floating-Point Operations Per Second. The literature currently relies on imprecise metrics [13, 35, 34, 18, 20] known as Floating-Point Operations Per Second (FLOPs). This is noted as important in [28] due to the need to understand performance on satellites which is true. While helpful at showing the difference between the performance of different processes or models, quickly and simply, it is an approximation of direct metrics such as CPU usage, latency, and power consumption which uses FLOPs per watt. These direct metrics are what is key to effective deployment. This approximation has been shown to be inaccurate and lead to suboptimal designs [14]. In the constrained environment of Picosatellites, this imprecision would cause fatal errors.

Power. Power is not recorded frequently in the sample literature, which is a gap in itself [18, 27, 34, 25, 35, 6, 13, 20]. However, it is important also to note that when power is recorded, most of the time, only a single number is given [33, 5], while both idle and inference power states are key. It is also important to know what is consumed when idle to be confident in the choices so other parts of the satellite can be designed around this or their impact eliminated, e.g., see [22].

Time Taken. The time taken in the literature can refer to different things, e.g., it can correspond to the time for just processing of the models or it can mean the time to run the whole process, including tasks such as loading data to and from memory before and after the model inference. The latter is important as these tasks can be computationally significant and for synchronisation-dependent processes in a constrained environment. Sometimes instead in the literature time taken will refer to the model instead of the full sets of processes.

Noise. Many models do not consider the interference of illumination noise and the earth's background. The available datasets do not reflect this reality and are not representative of the image quality of Smallsats. For example, datasets used in Sentinel 1 imagery are from a traditional satellite with advanced features not reflective of the image quality of Smallsats.

Memory. Memory is recorded frequently, however it is dependent on the hardware, the accelerator used and it has an impact on power so it is best to present it together with CPU utilisation and power recordings for completeness.

2.4 Limitations in the Literature

The literature is limited in scope and there is no benchmarking of hardware with object detection models. In particular, no current on-orbit ML papers focus on Picosatellite deployment. This compounded with weak metrics limits the practical use of existing papers. Any benchmarking of hardware with more appropriate metrics with object detection models would enhance the literature.

The implementation of many hardware accelerators uses older ML models and software optimisation, this is a trend throughout hardware acceleration [20]. The approaches in the literature with more current ML models for object detection [13, 34] somewhat demonstrate the potential strength of software acceleration but are held back with partially relevant metrics [33, 22, 13, 5, 17]. By taking advantage of the Hardware acceleration shown to be effective with software acceleration and new ML models and techniques, a better-combined solution could be achieved.

3 Methodology

This section describes how the models are created and modified to work with software optimisation techniques and hardware accelerators, to produce a range of models with different optimisations. Next, we explain how the boards are set up and the test cases created, then the test process to address the lack of power, memory and CPU utilisation recordings in on-orbit ML, that each test case and its models are tested on. Finally, the methods of collecting and analysing the data are explained.

3.1 Edge Model Implementations

This work focuses on YOLO methods a one-stage object detector that has been shown to perform well in resource constraints settings, see Section 2.2.

Dataset. The DIOR dataset [12] is used for the training and testing of the object detection models. It was chosen as it is suitable for evaluating different types of Picosatellites where image quality may be unknown or unreliable due to specific characteristics. This is a large-scale dataset, containing 23,463 remote sensing images of 800×800 pixels with 192,472 object instances across 20 object categories, an example image with its bounding boxes can be found in Figure 3. The large size of the images allows for a detailed evaluation of different detection models. The dataset includes various object sizes due to different spatial resolutions and size variations within and between object categories. This aspect ensures that the detection models are tested in diverse conditions, similar to what they might encounter in Picosatellite applications. It offers diverse image variations, covering over 80 countries and includes images taken under various weather, season, and imaging conditions. This allows for robust testing of the object detection systems in handling diverse image variations.

To use the DIOR dataset the labels are converted from XML format to YOLOv5-compatible text boxes, the horizontal bounding boxes are used, and then the data is divided into test, validation, and training sets following the split suggested by the datasets ImageSets. Finally, a second dataset is made which includes only the SHIP class, which is all bounding boxes labelled ship so the effects of having one class can also be evaluated. The full and restricted datasets are referred to *ALL* and *SHIP*, respectively.

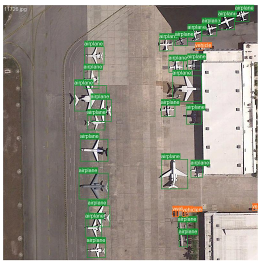


Figure 3: An example image from the DOTA Dataset.

Models. To find the right combination of object detection model and hardware a range of specialised lightweight YOLO models were used, including YOLOv5n, YOLOv5s, YOLOv8s, and YOLOv8n.

Training the Object Detection Models. The high-level steps we took in order to set up each object detection model are presented in Figure 4. From the raw dataset, the labels are transformed to fit your chosen object detection model and depending on the ML training framework the effort for each step will differ. For example, some frameworks provide up-loaders, while others will require you to build your own. With the models, we used the PyTorch framework to train the model with the respective YOLOv5 and YOLOV8 repositories. Once the model is trained it is converted to the edge framework. These models were trained for 100 epochs in batches of four, on a Titan XP GPU. Models are made for the two datasets, ALL and SHIP.

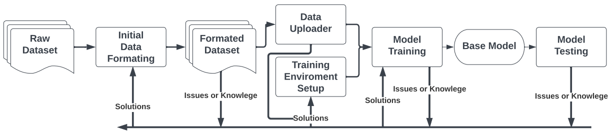


Figure 4: The high-level steps to setup object detection models.

Edge ML conversion. This project converts the trained Pytorch model to the Tflite edge model, which allows it to be run on the edge framework, this involves FP16 (floating point 16) quantisation and static INT8 (integer 8) quantisation, to try different quantisation approaches and see how this affects the different hardware setups.

TPU Conversion. For models to run on the Edge TPU, the operations the model performs must be mapped to their equivalent coral TPU operations. This can be achieved with the Google Coral API, When doing this it is important that every operation is mapped to the TPU as even if only one is mapped to the CPU can cause speed reductions by order of magnitudes. This reduction is due to the synchronisation overhead that occurs when the TPU has to wait for the CPU to complete its task, leading to TPU underutilisation. This waiting period disrupts the TPU's inherent instruction-level parallelism, reducing performance. Additionally, communication overhead between the CPU and TPU introduces latency, further slowing down the overall performance. Figure 5 presents the conversion steps described above.

3.2 Boards and Test Setup

To achieve our third aim, which is to find the best combination of model and accelerator for different Picosatellite applications, we set up and tested a variety of single-board computers. Table 1 gives an overview of the three boards used in our experiments. The process followed for setting up the boards is presented in Figure 6. Each board has its own SoC, CPU architecture, RAM, hardware accelerators, and software environment. We conducted

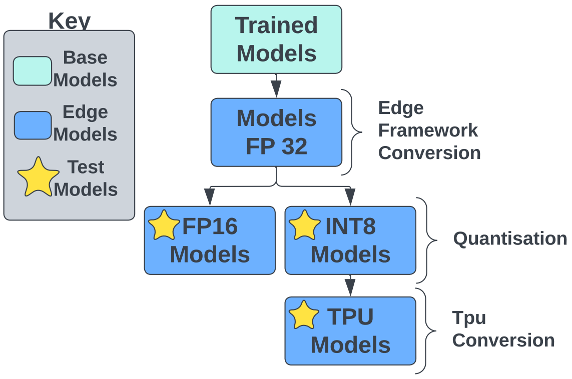


Figure 5: A high-level overview of the process from the stage of the trained models to the final models to be run on the test cases. These represent any of the trained YOLOv5n, YOLOv5s, YOLOv8n and YOLOv8s models. The trained models are shown to be converted to TF-Lite format and then quantised in different ways, next the INT8 quantised models are transformed to TPU instructions. The Stars show which of these models are used in the setups. Unless explicitly stated that this model did not support this transformation.

experiments on these boards and assessed the performance of various detection models to find the most suitable combination, considering precision, recall, power, memory, and CPU usage. Setting up the software on these boards can be difficult due to the interdependence of different software and hardware, see Figure 7. Therefore we focus on the combinations that worked well to make it easier to reproduce our work.

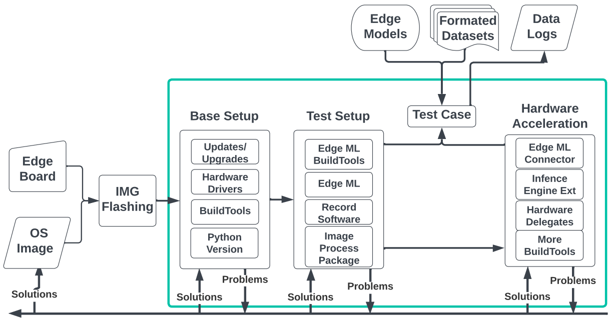


Figure 6: Overview of the process for setting up an edge board, OS, tools and packages, the ML framework, recordings software, and hardware acceleration.

To use GPU acceleration the Arm NN TF Lite delegate is used which has delegates for MALI GPUs, this is compatible with TF Lite models by adding it to the framework so no conversion is needed, this needs to be installed onto the board.

For each board given in Table 1 separate test cases were made. For the Odroid XU4 the test case (denoted XU4) uses the CPU and for the Odroid N2+ there are two test cases: the first (denoted N2+ CPU) uses the CPU with ARMNN Inference engine and the second (denoted N2+ GPU) uses the GPU with the ARMNN Inference Engine. The RPI3b+ tests case (denoted PI3b+) uses the TPU PyCoral delegate for TPU models otherwise the delegate is not enabled and the CPU is used with the XNN Pack.

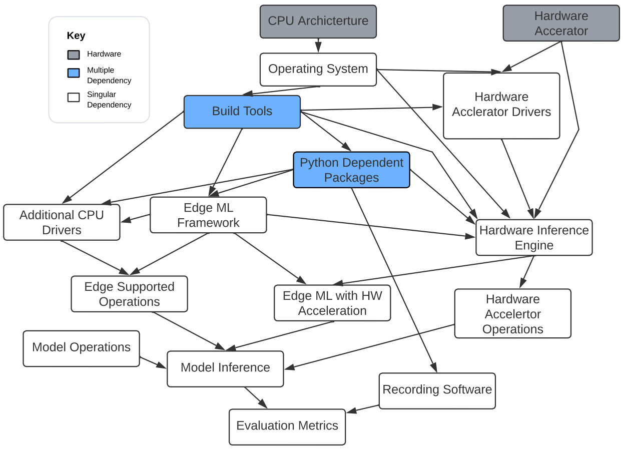


Figure 7: An example of the inter-dependencies when setting up hardware acceleration on an edge device; the white boxes represent one dependency, the blue boxes represent many dependencies of a category, the grey boxes represent hardware which is a hard constraint for example a 32bit CPU architecture means a 32bit operating system must be used.

3.3 Description of Test Process

In order to demonstrate the application of on-orbit ML, a simulation process is created. The process involves reading 100 images, adding bounding boxes to them, and saving the modified images. Figure 8 provides an overview of this process, outlining the steps and sub-tasks involved. The entire process can be divided into the following four stages.

1. **Initialization.** The ML model is loaded and memory is allocated for it. This is performed only once at the beginning of the script.
2. **Pre-processing.** Quantization variables are loaded, if applicable. The image is then loaded, resized if necessary, and converted into a tensor.
3. **Inference.** The input tensor is set for the model, the model is invoked, and the output tensor is retrieved.
4. **Post-processing.** If needed, the output is then dequantized. Bounding boxes are extracted from the output tensor and added to the image, then saved to an output folder.

The green-colored boxes in Figure 8 represent steps that are required when using the TensorFlow Lite (TF Lite) framework, as well as when using the ArmNN inference engine or Coral TPU extensions. The blue-colored boxes, on the other hand, can be customized, e.g., images can be loaded using the OpenCV library.

3.4 Collection of Measurements and Metrics

In this section, we explain the methods used to test various object detection models on different embedded devices. The goal is to get a clear picture of how well each model performs in terms of power use, processing speed, memory needs, and ability to detect objects accurately. These results are all collected separately to stop the contamination of different measurement tools.

Power. To record power, see Figure 9, the Odroid Smart Power 3 power supply is used as it can record high-precision results for embedded devices and can deal with high fluctuations. It has a USB-UART interface that can record more than 10 measurements per second, so any spikes or drops in power in the infer-

Table 1: Specification of Raspberry Pi 3 Model B+, Odroid XU4, and Odroid N2+ with Test Cases and TF Lite Delegate.

Feature	**Raspberry Pi 3B+**	**Odroid XU4**	**Odroid N2+**
SoC	BCM2837B0	Exynos 5422	S922X Rev. C
SoC Mfg. Process	40 nm	28 nm	12 nm
CPU (part of SoC)	4x ARM A53	4x ARM A15 + 4x A7	4x ARM A73 + 2x A53
CPU Arch.	ARMv8-A (64-bit)	ARMv7-A (32-bit)	ARMv8-A (64-bit)
CPU Clock Speed	1.4 GHz	A15: 2.0 GHz	A73: 2.2 GHz
		A7: 1.4 GHz	A53: 1.8 GHz
RAM	1 GB LPDDR2	2 GB LPDDR3	4 GB DDR4
HW Accelerators	Google Coral TPU	None	ARM Mali-G52 MP4 (Part of SoC)
Accelerator Clock Speed	500 MHz	-	800 MHz
Dimensions (mm)	85 x 56 x 17	83 x 58 x 20	90 x 90 x 17
Power Supply	5V/2.5A (micro USB)	5V/4A (barrel)	12V/2A (barrel)
OS	Debian 11 Raspbian	Ubuntu 20.04.6	Ubuntu 20.04.6
Python Ver.	3.9.2	3.8.10	3.8.10
TF Lite Runtime Ver.	2.5.0.post1	2.11	2.7
TF Lite Delegate	PyCoral	None	ARMNN
Test Cases	1) TPU(PyCoral)	1) CPU	1) GPU(ARMNN)
	2) CPU(No Pycoral)		2) CPU(ARMNN)

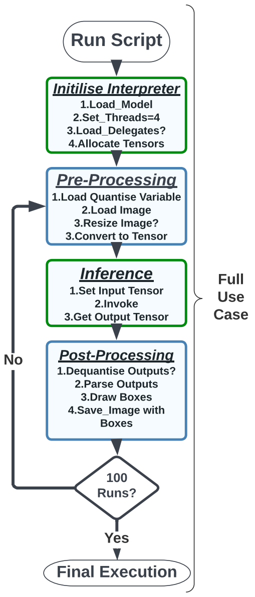


Figure 8: Flowchart of the simulation process.

ence can be identified. However, recording 10 measurements per second is a computationally heavy task, and therefore it cannot be done on the edge device itself as it would add noise to the results. Instead the edge device sends a TCP signal to an external computer that records the data from the USB-UART interface. Additionally, in the power calculation the average standby power is subtracted from the power reading so that we can focus on the increase in power.

During the recording, only the power supply and Ethernet cable are attached and the WiFi antenna and Bluetooth should be deactivated as even when not connected to a network they draw current, and therefore consume power. This is done for the full use case and the inference steps. Once this is recorded, we chose the two most important statistics to be:

- *Peak Power*: the highest Watt consumption in order to prevent any power surges.
- *Efficiency*: what can be completed with a Watt of power.

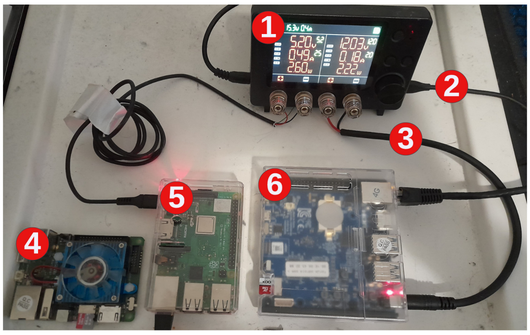


Figure 9: 1) Is the Smart Power 3 used in the power recording 2) USB-Uart cable for transferring power data to the main computer 3) Split cable power cable split leads in to red and black so it can draw power from Smart Power 3 4)Odroid Xu4 board 5) Raspberry Pi3b+ board 6) Odroid N2+ Board

Peak power is calculated by averaging the top ten values, and the efficiency is calculated through the average power and elapsed time. This is done using Frames-Per-Seconds Per-Watt (FPS-PW) in the full use case and Output Tensor Per-Seconds Per-Watt (OTPS-PW) for inference.

Measuring Time for Different Stages. Different measurements were taken to cast a wide net as the bottlenecks may not be immediately apparent. The package used here to record time was the Python Time package, both Wallclock time (the actual elapsed time from start to end of a task, including all waiting and idle times) and CPU time (the total time the CPU spends processing a specific task, excluding idle or waiting times).

- *Total Process Time* is that required to run the Python script to the completion of the last child process. This is collected separately from the rest of time recordings, as not to record the time it takes to record. From this the average time is calculated from three cases.
- *Sub-task Time* is that required just for the steps of an individual sub-task, see Figure 8. This is recorded 1000 times from ten runs and the average is used, except initialisation where the average of ten cases is used.

Memory. To record memory consumed, psutil is used as this al-

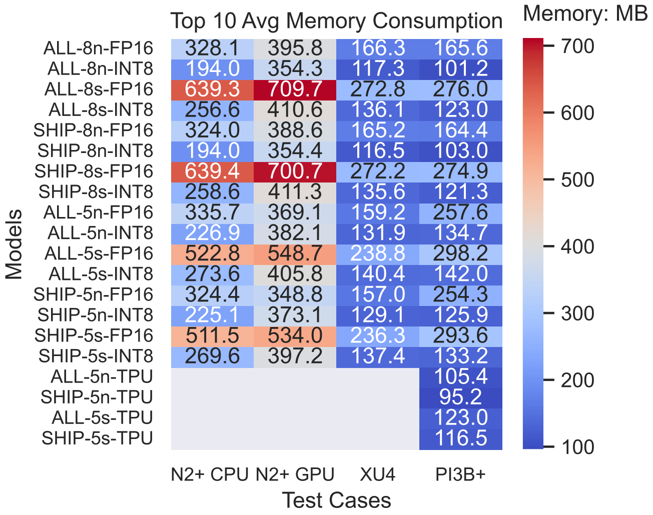


Figure 10: Heatmap showing the average of the top ten real memory records from three full use cases, measured in MB (Megabytes), these values are rounded to one decimal place.

lows us to record the memory from the creation of the first parent process until the last child process. This is important for concurrent task deployment on constrained devices.

Recording memory usage is important for devices with limited resources because it can help to make better use of the available hardware, keep the device responsive, save energy, balance workloads, and improve performance. By tracking memory usage from start to end, developers can plan how and when tasks should run. For example, if a device has 2GB of RAM and one task needs 500MB while another task needs 2GB, developers can figure out if these tasks can run simultaneously, making the overall system run more smoothly and avoiding slowdowns.

CPU. CPU utilisation is recorded throughout the full process using PSUTILS. Additionally, CPU time is recorded for each subsection and for the full process.

Object Detection Performance Metrics. *Precision*, *Recall*, and *mAP50* are evaluation metrics for object detection. Precision measures the proportion of correct detections, while Recall measures the proportion of true objects detected. mAP50 combines precision and recall by considering the model's performance across varying confidence score thresholds. By plotting precision against recall for these varying confidence thresholds, a precision-recall curve can be created for each class. The *Average Precision* (*AP*) is then calculated as the area under the precision-recall curve, providing a single value representing the model's performance for that class. mAP50 is the mean of the *AP* values for all classes. This provides a single metric that accounts for both precision and recall. The “50" in mAP50 refers to the intersection over union (*IoU*) threshold, which measures the overlap between two bounding boxes (predicted and ground truth). An IoU of 50% means the predicted bounding box has at least 50% overlap with the ground truth bounding box to be considered a correct detection. To gather these detection metrics the official YOLOv5 and YOLOv8 testing scripts are used.

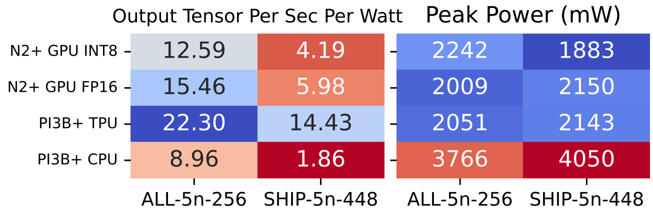


Figure 11: The left heatmap displays the average of three hundred inferences from three full use cases, measured in WallClock time in seconds, this is rounded to two decimal places. The heatmap on the right displays the Peak Power consumption in terms of milliwatts (mW) which is the top ten Average Power rounded to a whole number.

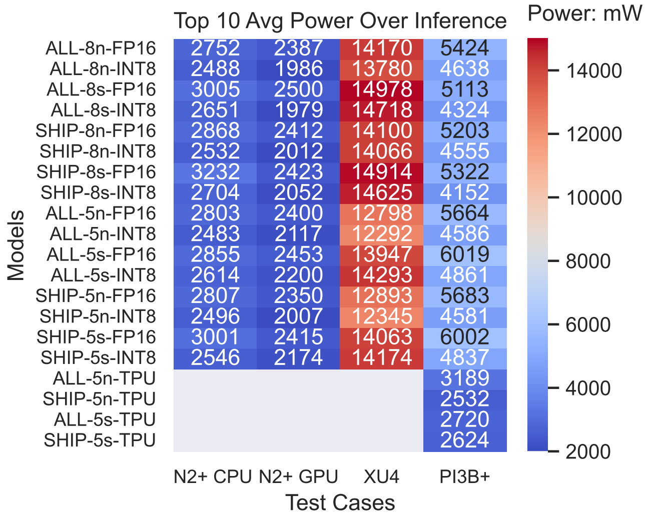


Figure 12: Heatmap showing the average of the top ten power recordings from three full processes, measured in Mw (Mili Watts), these values rounded to an integer.

4 Results

In this section, we will discuss the performance of the models and the recorded metrics of the models on the test cases described in Section 3.2. The results for CPU Utilisation and time can be found in the Appendix 21.

4.1 Power

This section looks at the power efficiency and peak power of different models and test case combinations. For a more extensive breakdown of power as a boxplot see Figure 20 in the appendix. This analysis focuses on the YOLOv5n and YOLOv8n models.

Inference Power Efficiency. Here we consider the OT-PS-PW to demonstrate the efficiency from the start of inference to measure how many output tensors you get per watt of power. Figure 18 demonstrates that YOLOv5n INT8 for CPU (single class) and YOLOv5nFP16 for GPU (single class) provide the highest power efficiency for generating output tensors, as evidenced by the peak OT-PS-PW values of 1.896 and 1.901 for the GPU and CPU approaches in the case of YOLOv5n. Figure 18 demonstrates that INT8 Quantisation is more power efficient on all CPU test cases and FP16 is more power efficient on the GPU test cases. The CPU methods that require INT8 quantisation cause worse detection

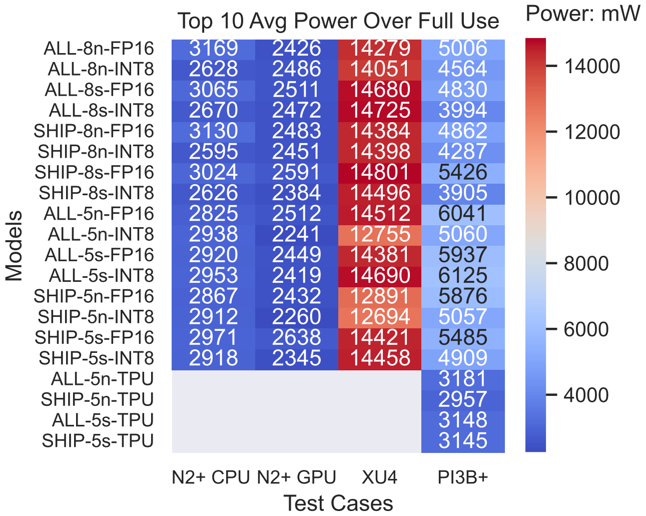


Figure 13: Heatmap showing the average of the top ten power records from three hundred inferences, measured in Mw (Mili Watts), rounded to an integer.

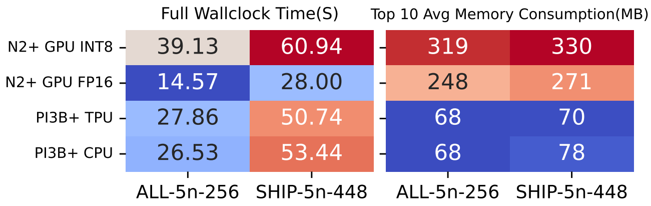


Figure 14: The left heatmap displays the average WallClock time in seconds, rounded to two decimals, from three hundred inferences over 3 full use cases. The right heatmap denotes Peak Power consumption in mW rounded to an integer.

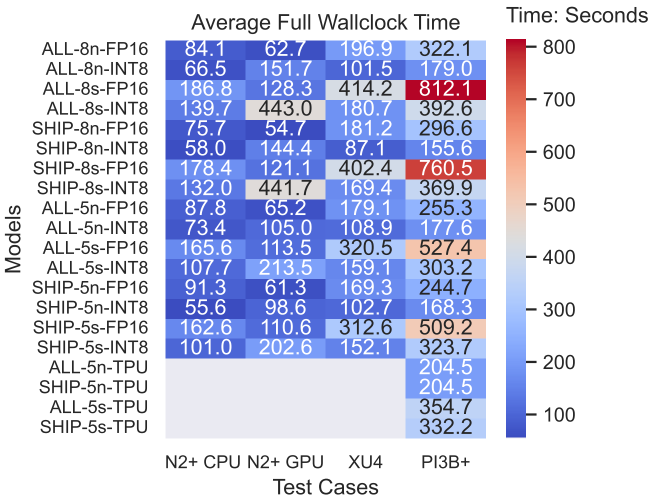


Figure 15: Heatmap showing the average of the full use case from at least three full use cases, in wallclock time measured in Seconds, these values are rounded to one decimal places.

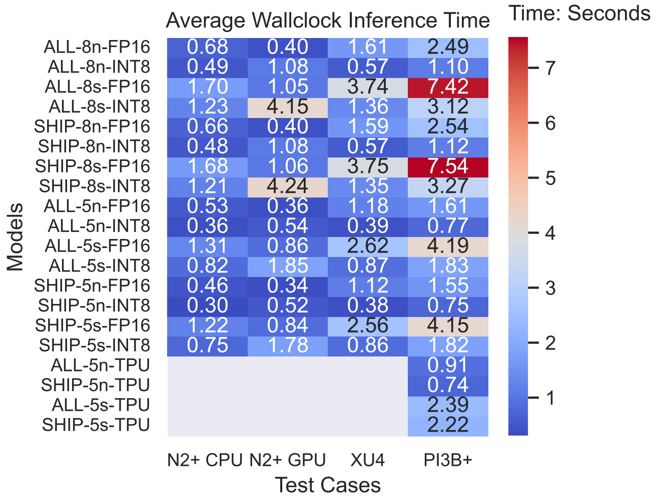


Figure 16: Heatmap showing the average time from three hundred inferences in wallClock time measured in Seconds, these values are rounded to two decimal places.

performance compared to FP16 as seen in Tables 2 and 3.

When all classes are considered in YOLOv5n, both CPU and GPU models experience a decrease in efficiency, with the CPU decrease being larger. The GPU efficiency surpasses the CPU due to the enhanced parallelisation required for multiple classes, with the GPU showing 1.764 OT-PS-PW compared to 1.534 OT-PS-PW for the CPU. In contrast, the efficiency differences between SHIP and ALL YOLOv8n models are minimal, with only a 0.024 OT-PS-PW difference. Despite this better handling of multiple classes when comparing YOLOv5n and YOLOv8n ALL models, we still find that YOLOv5n demonstrates greater power efficiency, with a 1.764 OT-PS-PW compared to 1.471 OT-PS-PW for YOLOv8n. Additionally, the high CPU OT-PS-PW efficiency observed in YOLOv5n INT8 (single class), which matches the GPU, does not apply to YOLOv8n INT8 (single class).

YOLOv5n (single class) seems highly optimised for CPU but at the expense of poorer use of GPU Acceleration. The benefits of parallelisation from GPU acceleration are more apparent in the YOLOv8 models or larger and more complex YOLOv5s.

The TPU models in the PI3B+ test case are shown to be more power efficient than the RPI3b+ CPU, but as they can not be fully mapped onto the TPU, their efficiency is bottlenecked by the RPI3b+ board causing them to be worse than the N2+ Test Cases.

Full Use Power Efficiency. Figure 18 demonstrates that, while YOLOv5n models (INT8 for CPU and FP16 for GPU) provide the highest power efficiency for generating output tensors, as shown in above, with peak OT-PS-PW of 1.896 and 1.901 for both GPU and CPU approaches in the case of YOLOv5n on just ships dataset, However, though YOLOv5n does seem to be most efficient for OT-PS-PW when looking at the inference, we need to consider how this affects the use cases power efficiency as the number of boxes produced from the output tensor the YOLOv8 models produces 13,125 boxes contrasting to 39,375 boxes for YOLOv5 at the default setting of three anchor boxes per cell. From this, the number of boxes to be processed is tripled, compared to the anchorless architecture of YOLOV8 which has one box per grid.

YOLOv8 models become relatively more power-efficient in the full use cases, see Figure 17, while not being as efficient as YOLOv5n it becomes closer especially when post-processing is performed on the CPU. This is due to the anchorless head in YOLOv8, which

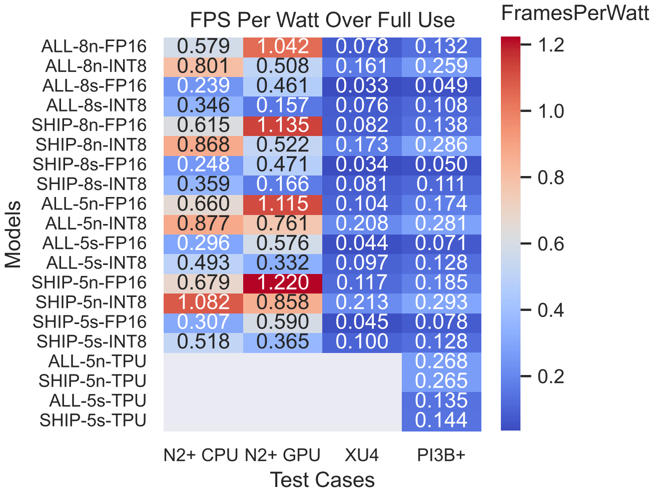


Figure 17: Heatmap showing the FPS-PW which represents the power efficiency of a full use case, which is the total time divided by a hundred representing each image processed, this is then divided by the average power consumption. These values are rounded to three decimal places.

as explained above reduces the number of boxes calculated by a third, thereby requiring fewer checks and fewer fetches to apply the boxes. Additional tests with NMS (Non-Maximun-Suppression) were conducted and are presented in Figure 19. Due to time constraint tests with NMS and image cutting were limited to the N2+ tests, we see that YOLOv8n models become significantly more power efficient over YOLOv5n, additionally showing that YOLOv5n SHIP on CPU is efficient but it is not enough to make up for the extra post-processing time of NMS, caused by the extra boxes from the output tensor. These tests were based off the previous conclusions that INT8 should be used with a CPU and FP16 with a GPU, see Figure 19.

The choice between YOLOv5 and YOLOv8 models, in the case of GPU-accelerated ML models, would suggest YOLOv8n models when using NMS. However, we do not use GPU acceleration in the post-processing which is the current bottleneck for YOLOv5 models due to their higher box count. If GPU-accelerated post-processing is available, then YOLOv5n models may become more efficient, this will need to look at the trade-off between the higher inference time of YOLOv8 and the higher post-processing time of YOLOv5. In cases without a GPU, even in the best-case scenario for the YOLOV5n on SHIP (single class) it is shown to be the best for CPU inference efficiency, the YOLOv8n models will be more efficient due to the lower post-processing time.

PeakPower GPU. Figure 12 shows that, although the INT8 quantised models have a reduced top ten average peak in inference for the GPU this reduction in peak power in inference, is not relevant due to bottlenecks in booting-up or post-processing that cause higher peak powers when looking at the full use, see Figure 13. For YOLOv5, although peak power from inference is reduced for INT8 GPU models and is not for F16 GPU models, the overall peak power for both GPU models is very similar. On the other hand, for YOLOv8, both the peak power from inference and overall peak power is lower for INT8 GPU models than F16 GPU models, see Figure 13.

PeakPower CPU. Regarding CPU-based test cases, we focus on the INT8 model as they are the most power efficient. From Figure 13 we see that the peak power for quantisation with YOLOv5n

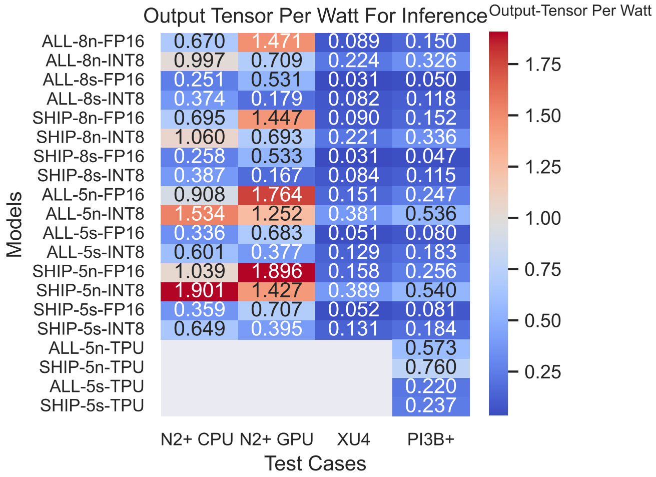


Figure 18: Heatmap showing the OT-PS-PW which represents the power efficiency of inference, which is the average inference time divided by a hundred representing each image processed, this is then divided by the average power consumption. These values are rounded to three decimal places. It is important to note the output tensor for YOLOv8 produces a third of the boxes making it less costly to use in post-processing.(X-Axis Figure Guide, the x-axis names are split into three parts split by "-", the first part references the dataset trained on either SHIP or ALL, the next part the YOLO models used, to save space only the suffix is used. For example 8n= YOLOv8n and 5s = YOLOv5s. Finally, the last part represents the Quantisation or Model Transformation for TPU models notated by TPU.)

and YOLOv5s is in the range 2912–2953mW, while YOLOv8n to YOLOv8s INT8 has the lowest peak power is in the range 2595–2670mW. So if only a CPU is available and peak power is an important constraint this indicates that either YOLOv8n or YOLOv8s are good choices. This is due to initialisation which causes an increase in the peak power for all and only the YOLOv5 models. Additionally, this is also the trend with RPI3b+ test case with the YOLOv8 models having lower peak powers than their YOLOv5 counterparts. On the other hand, there are several Odroid Xu4 occurrences with the FP16 YOLOv5n models that have lower peak power demonstrating that the YOLOv5n model benefit from the 32-bit architectures, which will be discussed more in Section 4.3. Figure 24 in the Appendix gives the relationship matrix between median memory and peak power and how it contrasts this relationship in the other boards.

4.2 Time

Figure 15 demonstrates that in the full use case evaluation, YOLOv8n FP16 GPU models exhibit the lowest average total run time, primarily due to their anchorless architecture, which predicts a third of the bounding boxes compared to YOLOv5 models. This results in reduced post-processing times, making YOLOv8 quicker overall. Despite having a longer inference time see Figure 16 for output tensor generation, YOLO-8n-FP16 on SHIP with the Odroid n2+ GPU is the fastest across various image sizes. It outperforms other models on both the SHIP and ALL datasets, with the difference in average time between these datasets being more significant for YOLOv8 than YOLOv5, mainly due to their output tensor processing methods.

In terms of CPU-only processing, YOLOv5 models are faster as when run on a CPU the extra inference time for YOLOv8 is longer

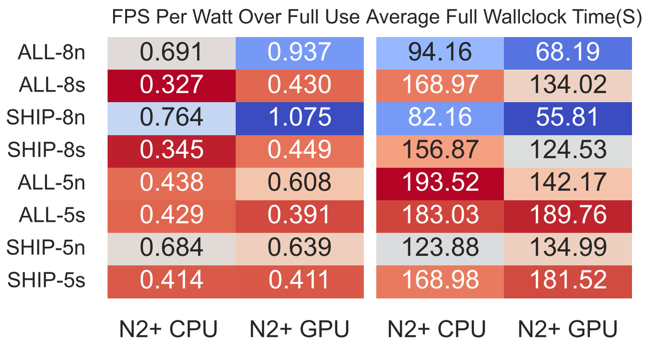


Figure 19: These heatmaps show the full use case with NMS. The heat map on the left represents the FPS Per Watt which shows the power efficiency of a full use case with NMS, which is the total time divided by a hundred representing each image processed, this is then divided by the average power consumption. These values are rounded to three decimal places, the heatmap on the right shows the average of the full use case from at least three full use cases, in wallclock time measured in seconds, these values are rounded to three decimal places.

than the extra post-processing time from more boxes in YOLOv5, leading YOLOv5 to be faster on CPU, and therefore also more efficient. This holds true for all CPU test cases as can be seen in Figure 15. This further shows that the YOLOv8 architecture is better designed for taking advantage of GPU parallelism.

YOLOv8 is better suited to handle bottlenecks of CPU post-processing on boards with an accelerator, but slower CPUs. In contrast, YOLOv5n models are more effective when only using a CPU, as the CPU is slow at performing inference with YOLOv8, benefiting less from parallelisation due to the architecture.

GPU models perform best with FP16 quantisation, while INT8 quantisation performs poorly on GPUs for time, likely due to the Armnn inference engine's suboptimal optimisation for INT8 quantisation with GPU.

In summary, future work should focus on investigating the trade-off between YOLOv8 and YOLOv5 models, particularly for systems with a GPU, and the impact of employing a GPU accelerated post-processing strategy with YOLOv5 models and their more efficient OT-PS-PW but less efficient FPS-PS-PW tensor, with a larger number of prediction boxes 39,375 compared to 13,125 for YOLOv8. This will help determine the most suitable model depending on the specific system requirements and constraints.

4.3 Memory

We focus on peak memory as a hard and safe constraint when designing task synchronization. Median memory information is available in Figure 26 of the Appendix. From Figure 10 we see that the Armnn inference engines for CPU and GPU provide fast and power-efficient processing for 800×800 images, but require significant memory, with ALL-5s-FP16 models doubling real memory consumption. The YOLOv5 FP16 GPU models consume 1.8–2.6 times more memory than the Raspberry Pi 3B+ FP16 counterparts, which reduces for the smaller YOLOv5n models to 1.4–2.3 times more memory. This leads to higher memory usage due to the need for additional software libraries. If memory constraints are a concern, the Armnn CPU can be used, requiring 1.7 times more memory (225MB peak) than the power-efficient GPU (369MB). The YOLOv8n models consume less memory with INT8 quantisation, while the YOLOv5s models consume less memory with FP16 quantisation. Alternatively, a TPU with smaller images on one class (448×448) can be used, consuming only 70MB, which is an improvement upon the 78MB used when tested without the TPU. When a smaller image (i.e., 256×256) is used for all classes, it results in no improvement upon the CPU RPI3b+ approach, but uses only 68MB.

Memory Efficiency. We also see in Figure 10 that Odroid XU4 demonstrates significantly lower peak memory for YOLOv5 FP16 models compared to the same model on different test cases. This is due to a 32-bit architecture reducing memory consumption, as memory addresses and data representations are smaller. Contrastly, Odroid XU4 has similar memory requirements for YOLOv8 FP16 models relative to the other test cases. This is due to YOLOv5 memory consumption relying on more anchor boxes, which benefit from the smaller memory footprint of 32-bit architectures. However, as YOLOv8 uses a different approach with its anchor-less architecture, it does not benefit as much from the smaller footprint using fewer boxes. Interestingly, the 64-bit Raspberry Pi 3B+ consumes less memory than the 32-bit Odroid XU4 board for INT8 models.

Although YOLOv8 INT8 models demonstrate inferior detection performance compared to YOLOv5 INT8 models, see Tables 2 and 3s they perform best in terms of main memory consumption across all devices, which is promising for future TPU usage where the memory is a bottleneck for their implementation.

4.4 Detection Performance

The best performing detection model was YOLOv8n FP16 for both data sets, see Tables 2 and 3. Additionally, the performance detection of the models trained on the ALL dataset were in general better than for the models trained on the SHIP dataset when considering the SHIP class, the exception is the YOLOv5n FP16 models. We also see that when quantising from FP16 to INT8, the models trained on the ALL dataset are better at retaining their mAP50 performance, than those trained on the SHIP dataset and YOLOv5n is no longer an exception to the above improved performance.

4.5 Efficiency of the TPU

We discuss TPU separately due to the difference in image size used to evaluate the TPU at its maximum efficiency. A secondary set of tests were conducted, examining the largest image size for TPU-accelerated models when all inference operations are offloaded to the TPU. In this scenario, we considered the GPU at optimal efficiency using FP16 and the TPU at INT8, as it only supports INT8 quantised models.

Figure 11 compares how the change in image size affected the GPU, and TPU with a CPU example for context. Under these conditions, the largest image size that the TPU can support while solely utilising the TPU is 448×448 pixels for one class in SHIP dataset and 256×256 pixels for all classes in ALL dataset within the TPU. When the TPU is fully utilised on SHIP, it results in an 18.9 times improvement in OT-PS-PW compared to the TPU on the larger images, see Figures 11 and 18. The same reduction in image size the GPU's on the same size only has a 3.1 times improvement in OT-PS-PW. This makes the TPU 6.1 times more efficient compared to GPU, when changing from 800×800 to 448×448 images. For 256×256 images, the GPU demonstrates an 8.8–fold improvement and the TPU a 39–fold improvement. However, this is relatively only 4.4 times more efficient than the 6.1 times for 448×448. Now when looking at Figure 11 and not comparing it to the 800x800 image tests, the TPU is 2.41 times greater than OT-PS-PW for 448x488 images, and 1.44 greater for 256x256 images. Thus, maximising the image size is crucial, resulting in a two-fold improvement in efficiency. When using TPU models there is a drop in mAP50 from their 800×800 counterparts, with a 0.478 mAP50 for 448x448 ships model, in contrast to the YOLOv5n 800×800 TPU of 0.615. This is greater for all classes, see Tables 4 and 2.

With the TPU accelerator on the smaller image sizes allowing for high OT-PS-PW, a new bottleneck in the computation time of the full use case is caused by the post-processing time. The post-processing calculating the bounding boxes and writing these images out takes far longer due to the less powerful RPI3b+ CPU than the N2+. So, despite the greater efficiency of the inference, the total time is greater due to the slow processing of the RASP PI 3b+, see Figure 14. Thus, when using specialised hardware like TPUs, it is crucial to consider the CPU's impact on post-processing, as it can become the limiting factor in overall performance. This weak CPU post-processing bottleneck could be reduced with the more efficient lower box output tensor models like YOLOv8. The TPU conversion of YOLOv8 models was not done as it has known bugs, additionally the YOLOv8 INT8 models demonstrate inferior detection performance compared to the YOLOv5 INT8 models, see Tables 2 and 3. As the former experiences a more significant loss in performance. These will both likely be implemented and improved upon with future iterations of YOLOv8 making it a promising future solution for this, due to its recent release.

Table 2: Model performance of models trained with ALL dataset: the columns represents the precision (P), recall (R), and mAP50 across ALL classes and the SHIP class. The models without quantisation suffix are the PyTorch models before any quantisation.

Model	All Classes			SHIP Class		
	P	R	mAP50	P	R	mAP50
v8s	0.829	0.667	0.734	0.936	0.874	0.922
v8s-FP16	0.829	0.638	0.705	0.948	0.861	0.917
v8s-INT8	0.732	0.558	0.605	0.626	0.677	0.597
v8n	0.837	0.626	0.708	0.942	0.843	0.907
v8n-FP16	0.814	0.608	0.673	0.948	0.838	0.905
v8n-INT8	0.708	0.548	0.582	0.611	0.728	0.635
v5s	0.805	0.661	0.711	0.873	0.877	0.91
v5s-FP16	0.792	0.636	0.676	0.89	0.872	0.902
v5s-INT8	0.757	0.615	0.657	0.666	0.837	0.807
v5s-TPU	0.749	0.619	0.652	0.672	0.839	0.81
v5n	0.772	0.624	0.668	0.850	0.863	0.896
v5n-FP16	0.761	0.611	0.648	0.863	0.858	0.893
v5n-INT8	0.683	0.596	0.612	0.572	0.813	0.747
v5n-TPU	0.676	0.596	0.606	0.568	0.813	0.743

5 Conclusions

This research investigated and compared the performance of various YOLOv5 and YOLOv8 object detection models at different sizes, classes, and quantisation, across diverse board setups. These setups included Raspberry Pi 3B+ with and without TPU, Odroid XU4, and Odroid N2+ using the ARM TF-Lite delegate with and without GPU. The study evaluated the models based on power efficiency, peak power, timings, and memory usage, highlighting the strengths and weaknesses of each model and the applied techniques, providing insights for their optimal utilisation in real-world small satellite applications across a range of test setups and Model variations. Tables 5 and 6 present an overview of the results from Section 4 identifying the best combination of model and accelerators. The most important conclusions drawn from the results are as follows.

Table 3: Model performance of models trained with the SHIP dataset, the columns representing precision (P), recall (R), and mAP50. The models without quantisation suffix are the PyTorch models before any quantisation.

Model	P	R	mAP50
v8s	0.947	0.843	0.917
v8s-FP16	0.946	0.842	0.912
v8s-INT8	0.585	0.637	0.489
v8n	0.934	0.822	0.899
v8n-FP16	0.936	0.818	0.896
v8n-INT8	0.594	0.665	0.583
v5s	0.951	0.858	0.911
v5s-FP16	0.936	0.858	0.896
v5s-INT8	0.586	0.68	0.637
v5s-TPU	0.576	0.681	0.634
v5n	0.934	0.838	0.9
v5n-FP16	0.931	0.837	0.897
v5n-INT8	0.592	0.661	0.62
v5n-TPU	0.591	0.66	0.615

Table 4: Detection Performance of Small Models used in the optimal TPU comparison, the columns representing precision (P), recall (R) and mAP50. SHIP in ALL refers to performance on SHIP class with models trained on the ALL dataset.

Model	P	R	mAP50
TPU (ALL)	0.632	0.311	0.341
TPU (SHIP in ALL)	0.53	0.32	0.333
TPU (SHIP)	0.552	0.484	0.478
CPU SM (ALL)	0.658	0.306	0.344
CPU SM (SHIP in ALL)	0.57	0.303	0.336
CPU SM (SHIP)	0.515	0.517	0.481

High mAP50. If you need the satellite to detect objects as accurately as possible, GPU acceleration should be used with it performing most effectively in terms of power on the higher accuracy FP16 models and with low peak powers. A YOLOv8s model should be used with their better Map50, see Table 2. These suggestions remain consistent for both SHIP and ALL classes, with no significant increase in any metric when using all classes. The constraint of this approach is memory Table 5.

High mAP50 and Low Memory. When using the GPU on ARM delegates memory is the main constraint, the 32-bit architecture GPU approach mentioned in the memory Section 4.3 could be a good approach. However, this approach would have the downside of less map50 but still accurate as YOLOV8 can not take advantage of the decrease in memory from 32bit architecture, so the YOLOv5 should be used that has a large decrease in memory for FP16 models but has worse detection performance. These suggestions remain consistent for both single and multiple classes, if you want to save memory with a GPU only using one class will only give a marginal benefit.

Memory Bottleneck for a Single Class. Using hardware acceleration with a TPU is recommended for a low-memory solution, see Figure 14, if you are only looking for a single class. It is advised to use only one class and the largest possible image size, as

Table 5: Model performance on ALL dataset: Columns represent the Use Case, Model, Test Setup, Dataset, Peak Power, OT-PS-PW, time, and mAP50 for all classes with the ALL models, YOLOv8 Boxes in output 13,125 YOLOV5 Boxes in output 39,375. In the mAP50 when there are two values, the first represents the average on ALL classes and the second represents the performance on the SHIP class.

Requirement	Model	Test Setup	Dataset	Peak Power (mW)	OT-PS-PW	Peak Memory (MB)	Fulltime (S)	mAP50
mAP50	v8s-FP16	N2+ GPU	ALL	2511	0.531	709.7	128.3	**0.705,0.917**
mAP50	v8s-FP16	N2+ GPU	SHIP	2591	0.533	700.7	121.1	**0.912**
Memory or Power Efficiency	v5n-FP16	N2+ GPU	SHIP	2432	**1.896**	**348.8**	61.3	0.897
Time	v8n-FP16	N2+ GPU	SHIP	2483	1.447	388.6	**54.7**	0.896
Memory Multiclass	v5n-FP16	N2+ GPU	ALL	2419	1.764	369.1	65.2	0.648,0.893
Memory and Time Multiclass	v8n-FP16	N2+ GPU	ALL	2426	1.471	395.8	62.7	0.673,0.905
Memory	TPUv5n	Pi3b+ TPU	SHIP	2143	14.43	**70**	50.74	0.478

Table 6: Model performance on SHIP dataset: Columns represent the Use Case, Model, Test Setup, Dataset, Peak Power, OT-PS-PW, time, and mAP50 for all classes with the SHIP models.

Use Case	Model	Test Setup	Dataset	Peak Power(mW)	FPS-PW	Peak Memory(MB)	Fulltime(S)	mAP50
CPU	v5n INT8	pi3b+ CPU	SHIP	5057	0.293	125.9	168.3	**0.62**
CPU	v5n INT8	pi3b+ CPU	ALL	5060	0.281	134.7	177.6	**0.612,0.747**
CPU	v8n INT8	pi3b+ CPU	SHIP	**4287**	0.286	**103.0**	155.6	0.583
CPU	v8n INT8	pi3b+ CPU	ALL	**4564**	0.259	**101.2**	179.0	0.582,0.635

the main constraint of this approach of a low mAP50 is caused by the small images used. To further optimise this approach, pruning the YOLOv5n model could potentially allow for an even larger image size by reducing the space the model takes up.

Memory Bottleneck for Multiple Classes. The TPU approach may also work for multiple classes with sufficient pruning, but this is uncertain. Instead, perhaps an accelerated CPU could be used with an INT8 model. YOLOv8n would be the better model here with its lower peak memory and shorter post-processing time which doubles up as you probably do not have a GPU for accelerating the post-processing. Additionally, CPU have low initialisation times, allowing them to be used in short bursts.

Peak Power. Using the N2+ Mali GPU or TPU would allow for the lowest peak power and these results are too close to say one is better than the other decisively. Here the TPU would be more efficient in terms of power for the inference. A GPU would have the advantage of higher accuracy using FP16 but at the cost of higher memory. Therefore, if additional memory is not a problem, then the larger YOLOv8s or YOLOv5s models do not significantly affect peak power with the GPU and so could be used. Also, INT8 could be used, but it's a slight decrease in peak power for a much longer processing time. This is true both for the All and SHIP classes. If no hardware acceleration is used, then the YOLOv8n or YOLOv8s INT8 should be used again for both of these classes.

Power efficency. At 800x800 image size the YOLOv5n models on the N2+ GPU have the highest power efficiency in generating output tensors, but YOLOv8 models become more power-efficient in full use cases due to their anchorless architecture reducing the number of boxes to be processed. The choice between these models depends on whether post-processing is GPU accelerated, the specific use case, and the trade-off between higher inference time (YOLOv8) and higher post-processing time (YOLOv5). Additionally, at smaller sizes, TPU becomes more power efficient at inference.

MultiClass Summary. In terms of processors, there is little overhead for multiclass on GPU, and a moderate amount on CPU and for TPU it's a severe bottleneck making it unsuitable.

CPU. INT8 quantisation offers improvements in power efficiency, full use time and memory which make the loss in detection performance justified. When choosing the exact model YOLOv5n shows better detection at INT8, with higher OT-PS-PW. In terms of CPU-only processing, YOLOv5n models are faster in the majority of full-use case tests however this is without NMS so it is likely that YOLOv8n models are faster which can be partially shown in the NMS tests. Additionally, YOLOv8 models have lower peak power at 64bit architecture and YOLOv5 have lower peak power at 32bit architecture. These conclusions are broad and the differences in test setups cause it to be different across setups.

5.1 Limitations

This is a complex topic, as such this work only scratches the surface of a much larger research area. Several limitations were imposed either by the availability of hardware, hardware specification, software compatibility, and/or dependency availability and compatibility. Additionally, the scope focuses on inference as it is the most relevant stage for the use case.

To use FPGAs the PYNQ-Z2 was selected. This is a development board which combines the Xilinx Zynq-7000 series System-on-Chip (SoC) which includes a programmable logic FPGA and ARM Cortex-A9 processor. The flexibility of Python programming makes this a good choice for testing FPGA-accelerated YOLO models. Despite this, the framework required for effective ML implementation is VITIS AI which has dropped support for the SoC 7000 series. Additionally, a new TF-Lite Delegate approach called Secda-Tflite which supports Pynq-Z2 has shown promise but does not yet support object detection models [8]. Although it would have been possible to build a modern YOLO from the ground up, it would be prohibitively time-consuming.

Odroid-XU4 has a GPU in its SoC and faces constraints in using it as the ARMNN TF-Lite delegate no longer supports 32-bit architectures. Attempts to use older versions did not work. Alternative approaches like ONE Neural Engine were successful but required the making of custom operations which was prohibitively time-consuming and TF Lite-GPU delegate faced issues, including OS restrictions to Android which would limit the other functions of a Picosatellite so this was dismissed.

The Coral USB TPU Accelerator would be too large to add to the board of a Picosatellite. However, if this could be used, then you could use the coral TPU accelerator module which is 15.0 ×10.0×1.5mm. We did not use it for this project as engineering skills are needed to apply it to a board. The USB TPU accelerator has increased data transfer overhead and power consumption, while the TPU accelerator module directly integrates with the host system, potentially improving performance and efficiency. Additionally, the Google Coral TPU requires tflite-runtime 2.5.0.post1 to work, and therefore different tflite-runtime versions are used between boards.

There are various points of failure when setting up hardware acceleration for ML on edge boards. For example, a Raspberry Pi 3B+ with its Videocore GPU cannot work due to the lack of proper drivers or support. Similarly, while the Odroid XU4's Mali GPU is a supported type, the XU4's 32-bit ARM CPU architecture is no longer supported and has been removed in the latest software version. In summary, compatibility issues between hardware and software stop the effective implementation of hardware acceleration, with these cases being just one example.

Figure 7 shows how the different properties of the software and hardware rely on each other when setting up hardware acceleration and edge ML on a SOC board.

Another possible use could have been the ARMNN CPU library with the cortex a53, as the 3b+ is 64-bit architecture this could have been developed, this was not done due to time constraints and other tests being more important, but this could still be useful in seeing the effects of the CPU ARMNN delegates.

5.2 Future Work

Future work could include investigating the trade-off between YOLOv8 and YOLOv5 models with GPU-accelerated post-processing strategy with YOLOv5 models and their more efficient OT-PS-PW, but with a greater amount of prediction boxes and YOLOv8 with higher average time and power consumption in inference but fewer boxes to post-process. This will help determine the most suitable model depending on GPU hardware acceleration.

As memory is the limiting factor for ASIC TPU, future work to optimise the model's size with pruning reducing the model size or PCA to reduce the size of the image, allowing the ASIC to work with larger image size which is the current bottleneck on getting higher ML evaluation metrics which holds this approach back.

Future work implementing a 32-bit architecture with a GPU FP16 YOLOv5n or YOLOv5s model could provide a more memory-efficient GPU approach, though an alternative board to the XU4 might be preferable due to its high power consumption, see Figures 10.

Another potential future direction is the Odroid N2L, a 64bit board, which currently does not have the supported OS the ARMNN TF-Lite delegate, being one version ahead, it is likely that the ARMNN will be updated to work with a newer OS at some point making it suitable. Additionally, as this work used Ubuntu and Raspibian OS to focus on other elements of the process so specialised lightweight OS like Archlinx or Linux lite could be used in future work where the bottlenecks and overheads of each operating system are evaluated. Also, how does this affect accuracy, recall, precision, and MAP50, which is a limit of this work as it should not significantly change performance and the detection serves its purpose of contextualising the detection purpose, but should be tested to confirm.

From the analysis of the power and memory results comparing the YOLO models results from a range of setups. Green Computing solutions could be developed to be more power efficient and reduce memory consumption model architecture designs could be considered. It also could be used to inform how HW-NAS for object detection is designed, showing a need to understand how the output tensor could affect the model designs in it or how the board's architecture would effect memory consumption or how each HW-Accelerator is best utilised. These are just some examples that this work shows and could help with optimising.

The ML evaluation looks at mAP50 which is a good evaluation of detection, however more granular detection results to check the performance on different image sizes would greater reveal the best model for the use case.

This work looked at RGB images with the DIOR dataset, further validity from another dataset would help. However, focusing on different image formats, such as SAR and Multispectral to further contextualise the model's performance on commonly used datatype in satellites should also be done.

This work started with 800×800 images, in many satellites large images are collected that sliced down into smaller check which is done on the satellite, so this works need to be used in context and future work that slices the images will need to be evaluated similarly in terms of memory and power.

Acknowledgments. Thank my Supervisor Dr Lito Michala and also Dr Gethin Norman without their help this would not have been possible. Additionally, Alba Orbital for giving me the opportunity of doing this research and Dr Jose Cano Reyes for his guidance.

6 References

[1] P. Babu and E. Parthasarathy. Hardware acceleration for object detection using YOLOv4 algorithm on Xilinx Zynq platform. *Journal of Real-Time Image Processing*, 19:931–940, 2022.

[2] brycetech. Smallsats by the number 2022.

[3] C. Davenport. The revolution in satellite technology means there are swarms of spacecraft no bigger than a loaf of bread in orbit, Apr 2021.

[4] M. Ghiglione and V. Serra. Opportunities and challenges of ai on satellite processing units. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, pages 221–224. Association for Computing Machinery, 2022.

[5] G. Giuffrida, L. Diana, F. de Gioia, G. Benelli, G. Meoni, M. Donati, and L. Fanucci. Cloudscout: A deep neural network for on-board cloud detection on hyperspectral images. *Remote Sensing*, 12, 2020.

[6] G. Guerrisi, F. D. Frate, and G. Schiavon. Satellite on-board change detection via auto-associative neural networks. *Remote Sensing*, 14, 2022.

[7] A. Guzman. Miniature satellites with massive benefits, Jun 2022.

[8] J. Haris, P. Gibson, J. Cano, N. Bohm Agostini, and D. Kaeli. SECDA-TFLite: A toolkit for efficient development of FPGA-based DNN accelerators for edge inference. *Journal of Parallel and Distributed Computing*, 173:140–151, 2023.

[9] T. Hoeser, F. Bachofer, and C. Kuenzer. Object detection and

image segmentation with deep learning on earth observation data: A review—part II: Applications. *Remote Sensing*, 12, 2020.

[10] Z. Huang, S. Yang, M. Zhou, Z. Gong, A. Abusorrah, C. Lin, and Z. Huang. Making accurate object detection at the edge: review and new approach. *Artificial Intelligence Review*, 55:2245–2274, 2022.

[11] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, NanoCode012, Y. Kwon, K. Michael, TaoXie, J. Fang, imyhxy, Lorna, Z. Yifu, C. Wong, A. V, D. Montes, Z. Wang, C. Fati, J. Nadar, Laughing, UnglvKitDe, V. Sonck, tkianai, yxNONG, P. Skalski, A. Hogan, D. Nair, M. Strobel, and M. Jain. ultralytics/yolov5: v7.0 - YOLOv5 SOTA Real-time Instance Segmentation, Nov. 2022.

[12] K. Li, G. Wan, G. Cheng, L. Meng, and J. Han. Object detection in optical remote sensing images: A survey and A new benchmark. *CoRR*, abs/1909.00133, 2019.

[13] Y. Liu, X. Zhou, and H. Han. Lightweight CNN-based method for spacecraft component detection. *Aerospace*, 9, 2022.

[14] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. Shufflenet v2: Practical guidelines for efficient CNN architecture design. *CoRR*, abs/1807.11164, 2018.

[15] E. Mabrouk. What are smallsats and CubeSats?, Mar 2015.

[16] S. C. Magalhães, F. N. dos Santos, P. Machado, A. P. Moreira, and J. Dias. Benchmarking edge computing devices for grape bunches and trunks detection using accelerated object detection single shot multibox deep learning models. *Engineering Applications of Artificial Intelligence*, 117:105604, 2023.

[17] A. Maskey and M. Cho. CubeSatNet: Ultralight convolutional neural network designed for on-orbit binary image classification on a 1U CubeSat. *Engineering Applications of Artificial Intelligence*, 96:103952, 2020.

[18] G. Mateo-Garcia, J. Veitch-Michaelis, L. Smith, S. V. Oprea, G. Schumann, Y. Gal, A. G. Baydin, and D. Backes. Towards global flood mapping onboard low cost satellites with machine learning. *Scientific Reports*, 11:7249, 2021.

[19] M. Merenda, C. Porcaro, and D. Iero. Edge machine learning for AI-enabled IoT devices: A review. *Sensors*, 20:2533, 10 2020.

[20] R. Neris, A. Rodríguez, R. Guerra, S. López, and R. Sarmiento. FPGA-based implementation of a CNN architecture for the on-board processing of very high-resolution remote sensing images. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 15:3740–3750, 2022.

[21] J. N. Pelton, S. Madry, and S. Camacho-Lara. *Handbook of satellite applications*. Springer, 2017.

[22] R. Pitonak, J. Mucha, L. Dobis, M. Javorka, and M. Marusin. CloudSatNet-1: FPGA-based hardware-accelerated quantized CNN for satellite on-board cloud coverage classification. *Remote Sensing*, 14, 2022.

[23] E. Rapuano, G. Meoni, T. Pacini, G. Dinelli, G. Furano, G. Giuffrida, and L. Fanucci. An FPGA-based hardware accelerator for CNNs inference on board satellites: Benchmarking with myriad 2-based solution for the cloudscout case study. *Remote Sensing*, 13, 2021.

[24] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[25] M. P. D. Rosso, A. Sebastianelli, D. Spiller, and P. P. M. andd S. Ullo. On-board volcanic eruption detection through cnns and satellite multispectral imagery. *Remote Sensing*, 13, 2021.

[26] A. Russo and G. Lax. Using artificial intelligence for space challenges: A survey. *Applied Sciences*, 12:5106, 12 2022. has diagram of applcaiton.

[27] V. Růžička, A. Vaughan, D. D. Martini, J. Fulton, V. Salvatelli, C. Bridges, G. Mateo-Garcia, and V. Zantedeschi. RavÆn: unsupervised change detection of extreme events using ML on-board satellites. *Scientific Reports*, 12:16939, 2022.

[28] Machine learning onboard satellites, 2021.

[29] H. Snyder. Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, 104:333–339, 2019.

[30] A. Space. 10 advantages of cubesats vs. conventional satellites, May 2020.

[31] M. Sweeting. Modern small satellites - changing the economics of space. *Proceedings of the IEEE*, 106(3):343 – 361, 2018.

[32] A. Vali, S. Comai, and M. Matteucci. Deep learning for land use and land cover classification based on hyperspectral and multispectral earth observation data: A review. *Remote Sensing*, 12, 2020.

[33] M. Xu, L. Chen, H. Shi, Z. Yang, J. Li, and T. Long. FPGA-based implementation of ship detection for satellite on-board processing. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 15:9733–9745, 2022.

[34] X. Xu, X. Zhang, and T. Zhang. Lite-YOLOv5: A lightweight deep learning detector for on-board ship detection in large-scene sentinel-1 SAR images. *Remote Sensing*, 14, 2022.

[35] Y. Yao, Z. Jiang, H. Zhang, and Y. Zhou. On-board ship detection in micro-nano satellite based on deep learning and COTS component. *Remote Sensing*, 11, 2019.

[36] S. S. A. Zaidi, M. S. Ansari, A. Aslam, N. Kanwal, M. Asghar, and B. Lee. A survey of modern deep learning based object detection models. *Digital Signal Processing*, 126:103514, 2022.

[37] Z. Zhang, C. Wang, J. Song, and Y. Xu. Object tracking based on satellite videos: A literature review. *Remote Sensing*, 14, 2022.

APPENDIX

A Computer Vision Techniques

A.1 Image Classification

Image classification is the process of using ML to apply usually one, but can be more than one category to an image. Currently, on-orbit ML Image classification is mainly used to filter out low-quality images [22, 5, 17]. Image classification is limited as it can only give categorisation to the whole image, which means it is less suitable for a range of tasks, particularly if one wishes to take advantage of complex images.

A.2 Object Detection

Object detection is locating and classifying objects in images, usually in rectangular bounding boxes. Modern Object detection uses quick and computationally effective ways of finding objects/targets within an image compared to other image recognition tasks discussed. There is a lack of implementation of new object detection models for ML on-orbit small satellites.((fast rate of progress of ML development)

on-orbit Object detection is used mainly for the task of ship detection [34, 35, 20] due to its economic and military applications [37]. It also has a large supply of quality publicly available datasets[1], which makes it testable by using benchmarking/reliable field. If object detection performance is recorded well on a range of hardware options for picosats, it will help develop ml-on-orbit such satellites further. Such results can be used as a launchpad for other key tasks which benefit from faster analysis, like crop detection or fire detection where latency is key to effective responses.

A.3 Semantic Segmentation

Semantic Segmentation is the task of classifying pixels of an image. To the best of our knowledge, it has only been used once for on-orbit ML [18]. It faces the same challenges as Object Detection, as well as some extra challenges, such as struggling with low-quality images [28]. So, proof of effective Object detection is a more suitable first step for picosats.

A.4 Change Detection

Change detection is detecting change over a period of time by comparing past images to new images and seeing if they differ beyond a certain threshold. This approach has been investigated in [27, 6] looking at Urban change and Natural disaster detection respectively. It has several merits. However, lightweight change detection models have fewer/weaker proofs of concept while object detection is able to complete many of the same tasks. Datasets used for change detection on-orbit ML are also relatively untested and limited. More research is expected in this area in the future.

[1] github.com/jasonmanesis/Satellite-Imagery-Datasets-Containing-Ships

B Figures and Tables

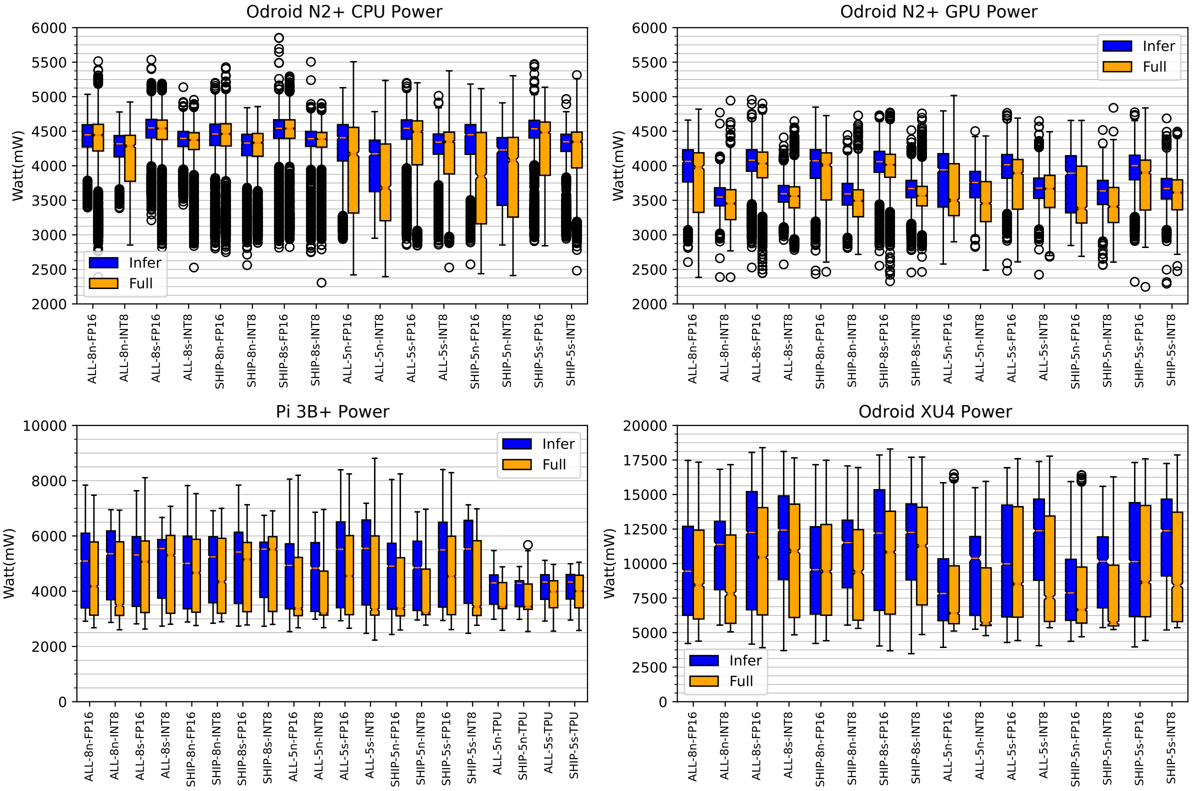


Figure 20: Boxplots of power in mW (Milli Watts), the Box represents 25 to 75 Percentiles of the data being 50 per cent of the data in total and this is known as the IQR. The median is represented by the yellow line, the whiskers represent the spread of data outside the interquartile range (IQR) but within the 1.5 times range. The dots showing the outside are the value that doesn't fall within the whiskers. This has been performed for every test setup, N2+ with CPU, GPU, RPi3B+ with the 800x800 tpu acceleration as additions and the odroid XU4 board. The blue colour shows the boxplot of power in inference and the orange box shows the box plot during a full use case.

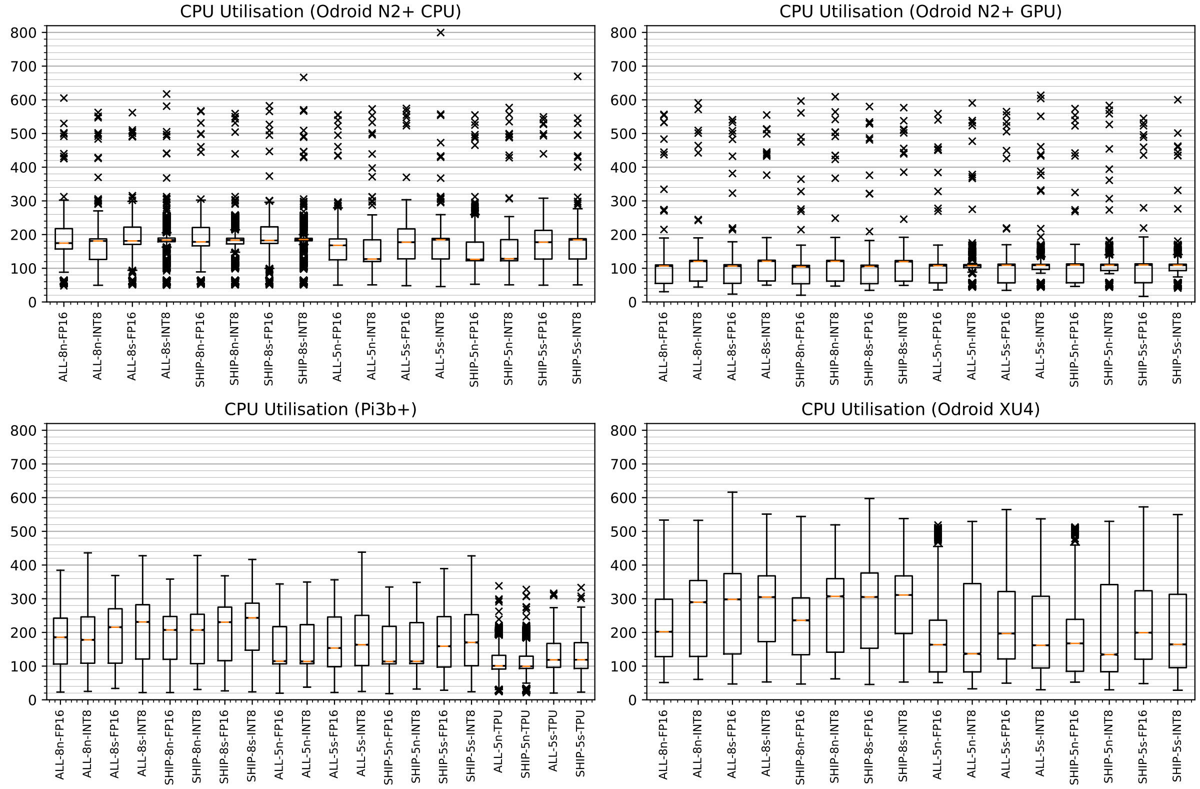


Figure 21: Boxplots of CPU Utilisation as a percentage of the CPU used, the Box represents 25 to 75 Percentiles of the data being 50 per cent of the data in total and this is known as the IQR. The median is represented by the yellow line, the whiskers represent the spread of data outside the interquartile range (IQR) but within the 1.5 times range. The dots showing the outside are the value that doesn't fall within the whiskers. This is done for every test setup, N2+ with CPU, GPU, RPi3B+ with the 800x800 tpu acceleration as additions and the odroid XU4 board.

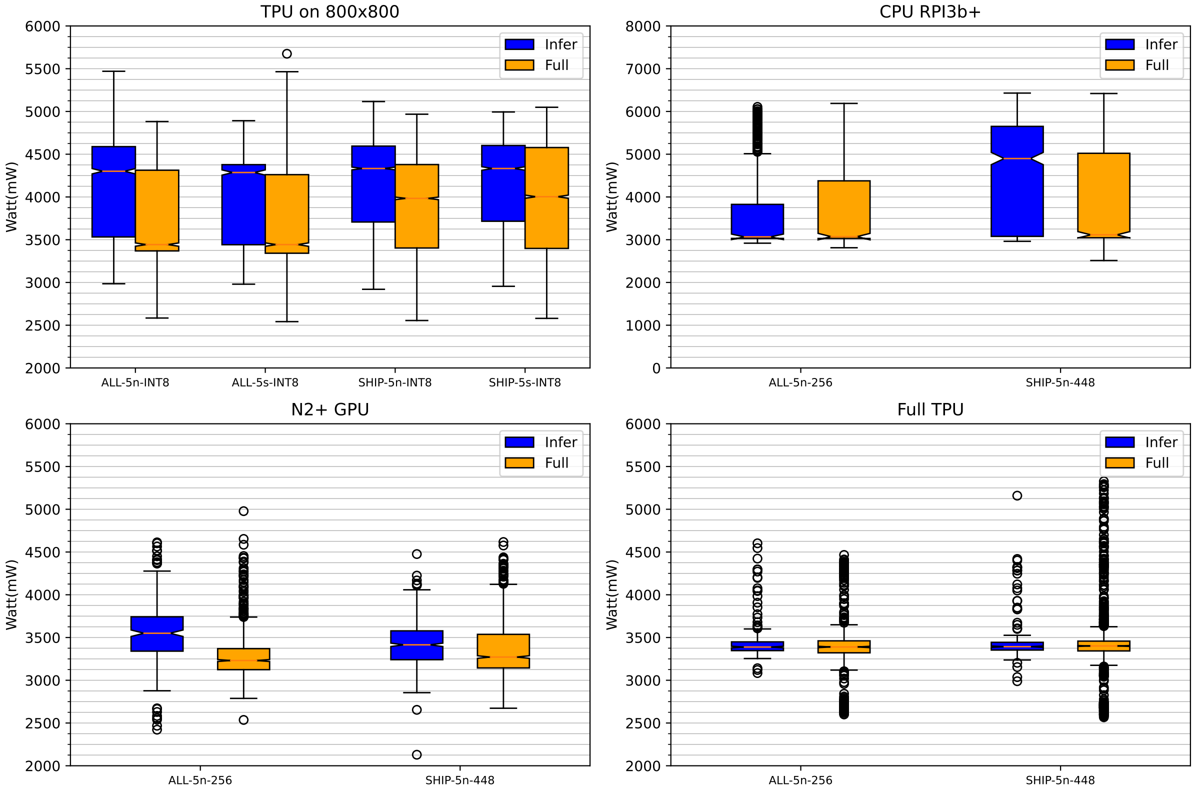


Figure 22: Boxplots of power in mW(Milli Watts), the Box represents 25 to 75 Percentiles of the data being 50 per cent of the data in total and this is known as the IQR. The median is represented by the yellow line, the whiskers represent the spread of data outside the interquartile range (IQR) but within the 1.5 times range. The dots showing the outside are the value that doesn't fall within the whiskers. This is done for every test setup, N2+ with CPU, GPU, RPi3B+ with the 800x800 tpu acceleration as additions and the odroid XU4 board. The blue colour shows the boxplot of power in inference and the orange box shows the box plot during a full use case.

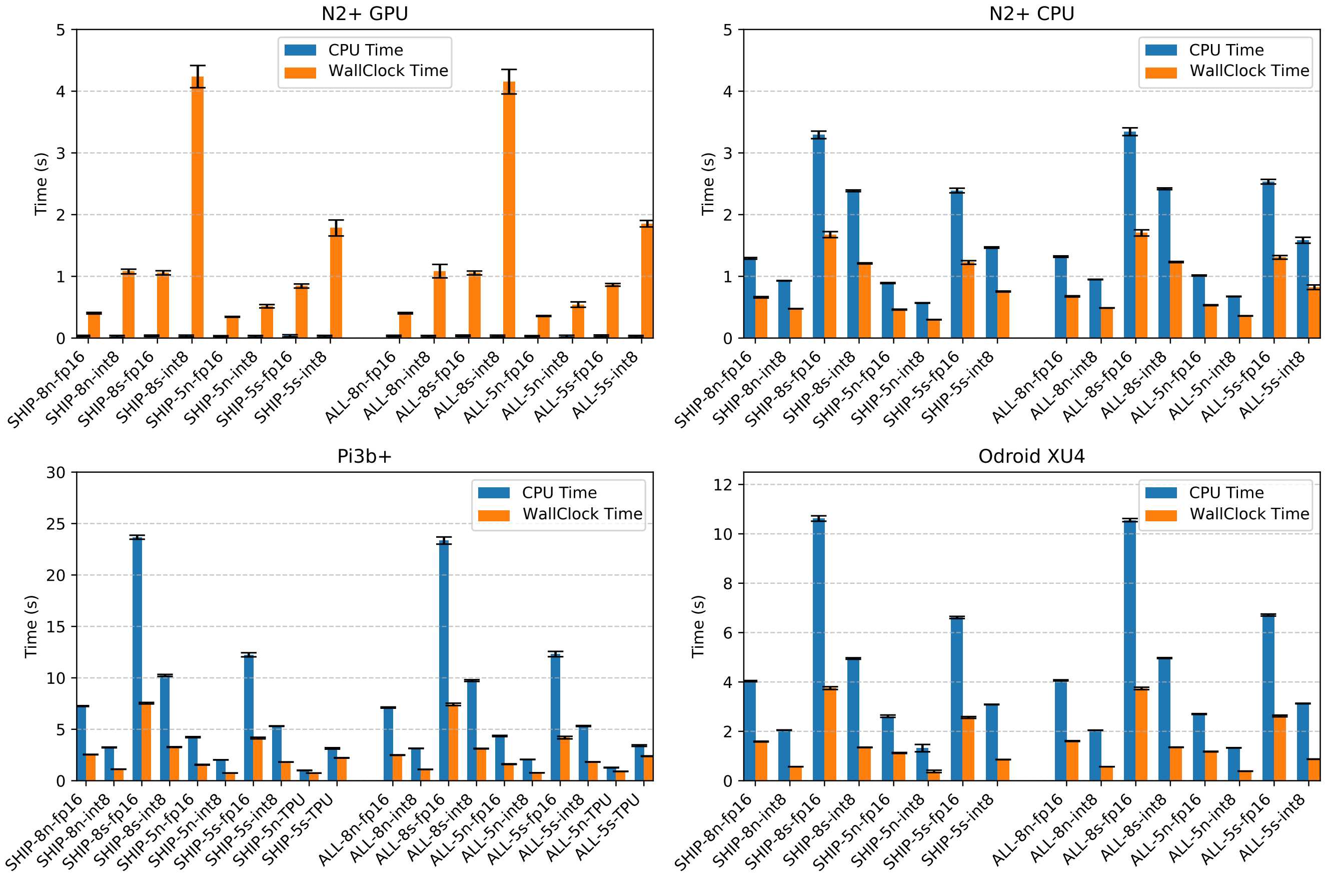


Figure 23: Average Inference time bar charts, for each test case in 800x800 images. In blue CPU time and orange Wall-Clock time, error bars show the standard deviation.

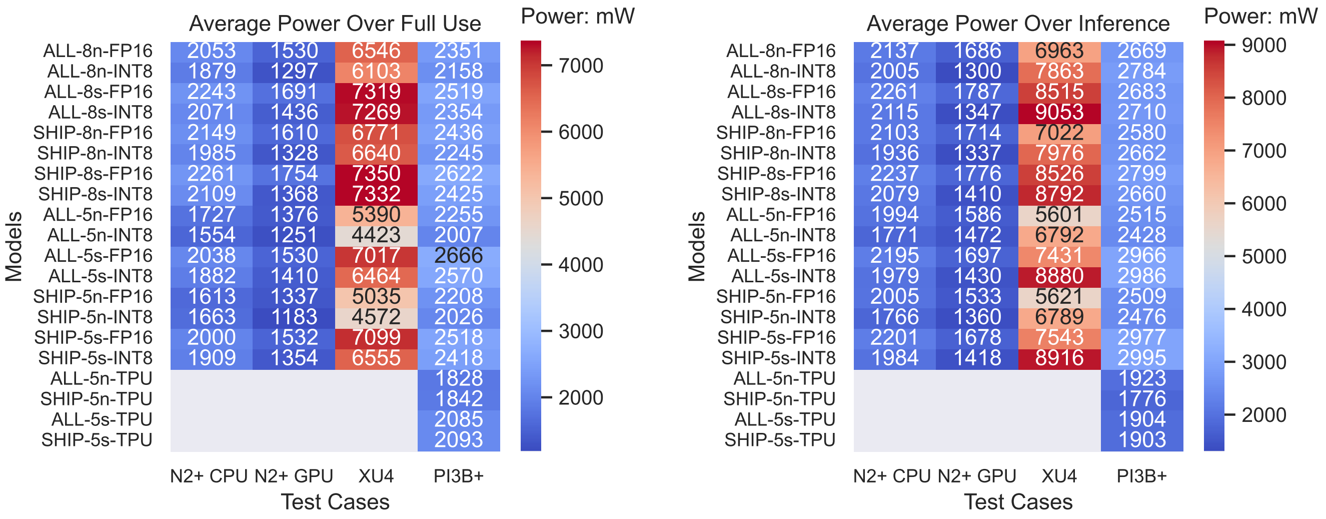


Figure 24: This heatmap shows the average power consumption from three hundred inferences measured in mW(milliwatts), these values are rounded to full numbers.

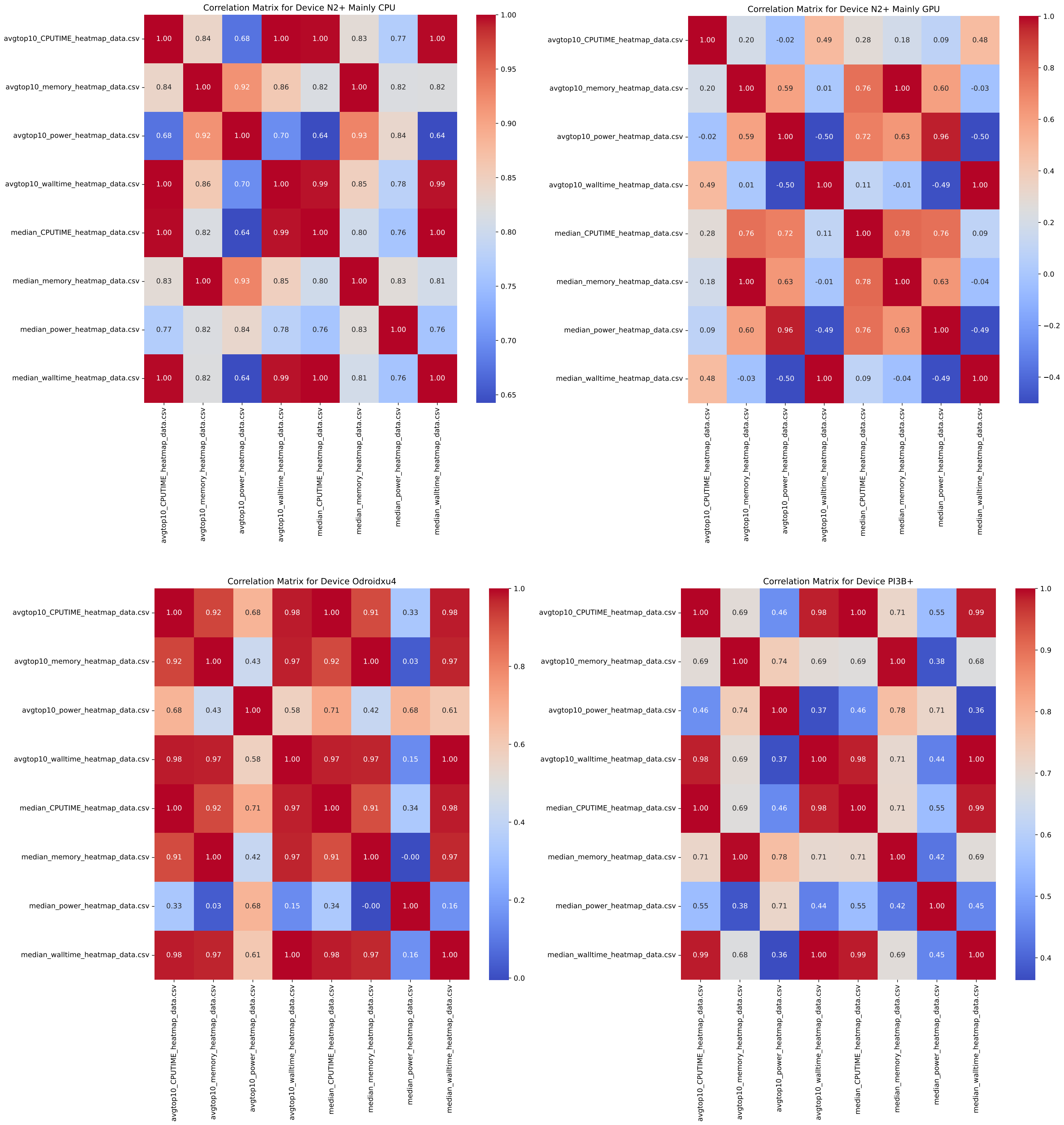
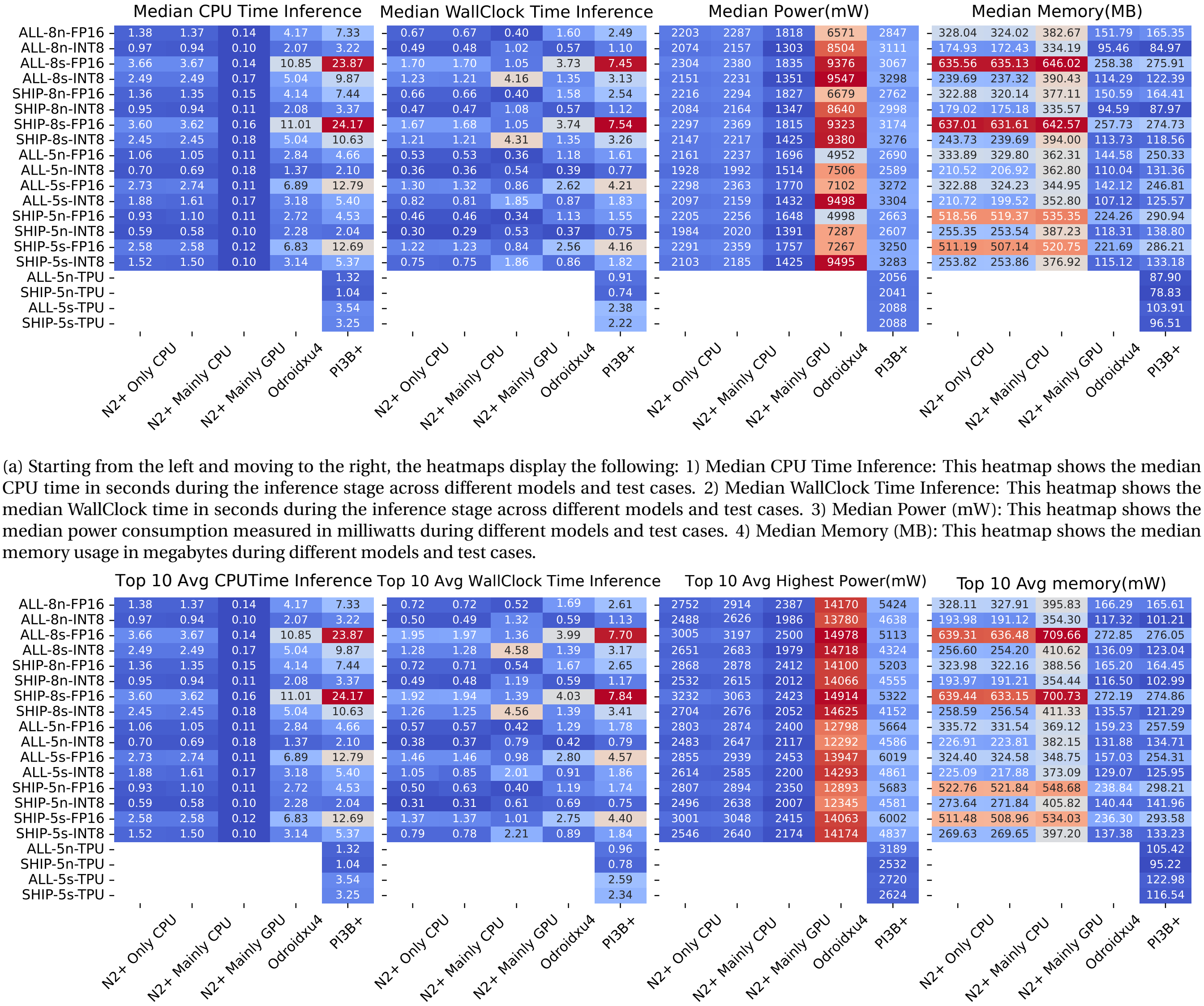


Figure 25: Correlation matrices showing the correlation of different models on the test setups using pandas with Pearson Correlation Coefficient.



(a) Starting from the left and moving to the right, the heatmaps display the following: 1) Median CPU Time Inference: This heatmap shows the median CPU time in seconds during the inference stage across different models and test cases. 2) Median WallClock Time Inference: This heatmap shows the median WallClock time in seconds during the inference stage across different models and test cases. 3) Median Power (mW): This heatmap shows the median power consumption measured in milliwatts during different models and test cases. 4) Median Memory (MB): This heatmap shows the median memory usage in megabytes during different models and test cases.

(b) Starting from the left and moving to the right, the heatmaps display the following. 1) Top ten Average CPU Time Inference: This heatmap shows the median CPU time in seconds during the inference stage across different models and test cases. 2) Top ten Average WallClock Time Inference: This heatmap shows the median WallClock time in seconds during the inference stage across different models and test cases.3) Top ten Average Power (mW): This heatmap shows the median power consumption measured in milliwatts during different models and test cases. 4) Top ten Average Memory (MB): This heatmap shows the median memory usage in megabytes during different models and test cases.

Figure 26: Extra inference heatmaps from the experiments.

Table 7: Simplified On-orbit ML literature table from Integrative Search

	Year	CV Technique	Acceleration	Power	Acceleration Device	Simulate Satellite board	Model Size(VM)	Real Memory	Model Type	Processer Usage	Inference Time	model input size
[5]	2020	Image Classification	Pruning Quantization Hardware Acceleration	1.8W	VPU Movidius Myriad 2	EoT board	Not Given	2.1Mb Memory foot print	CNN	Not Given	325ms	512x512px
[22]	2022	Image Classification	Quantization Hardware Acceleration	2.32 W Idle 2.5W Inference	FPGA SoC Xilinx Zynq Z7020	zturn development board	Not Given	1.43 Mb to 3.06 Mb BRAM	CNN	LUT =46.27% FF = 31.41% BRAM = 29.29%, DSP = 0.45%	Not Given	512x512px
[18]	2021	Image Segmentation	Hardware Acceleration	Not Given	VPU Movidius Myriad 2	Raspberry Pi3B+	Not Given	Not Given	FCNN	Not Given	Not Given	Not Given
[27]	2022	Change Detection	None	Not Given		Xilinx PYNQ board	Not Given	67% of ram	VAE	Not Given	2.06/4.86/13.98s	574 × 509 px
[34]	2022	Object Detection	Pruning	Not Given		Jetson TX2	Not Given	Not Given	YOLO	Not Given	37.51 s	24,000px× 16,000px
[26]	2021	Image Classification	Pruning Hardware Acceleration	Not Given	VPU Movidius Myriad 2	Raspberry Pi 3B	1.9MB small model+AM4	Not Given	CNN	Not Given	1 or 0.143s	512×512px
[35]	2019	Object Detection	None	3-5W Standby 15W Peak		Jetson TX 2	108.03MB	Not Given	Faster R-CNN	Not Given	1.25 s	1kx1kpx
[6]	2022	Change Detection	None	Not Given	None	None	less than 3MB	no edge board	AANN	No Hardware sim	No Hardware sim	Not Given
[33]	2022	Object Detection	Hardware Acceleration	1.32W	FPGA Xilinx XQR5VFX130 -1CN1752B	FPGA Xilinx XQR5VFX130 -1CN1752B	Not Given	LUTs 89% LUTRAM 2% BRAM 31% DSP48E 9% IO 43%	YOLO	LUTs 89% LUTRAM 2% BRAM 31% DSP48E 9% IO 43%	>1 s	4kx4k image
[17]	2020	Image Classification	Quantization	Not Given	STM32 MCUs	STM32 MCUs	104KB	Not Given	CNN	Not Given	0.25 s	100x100px
[13]	2022	Object Detection	Pruning Quantization Hardware Acceleration	not given	not simulated	not simulated	Not Given	not given	YOLO	Not Given	Not Given	640x640px
[20]	2022	Object Detection	Quantization Hardware Accelerated	Not Given	Ultrascale XCKU040- 2FFVA1156E	Ultrascale XCKU040- 2FFVA1156E	Not Given	CLBs 41% LUTs 25% BRAMs 39% DSPs 31% FFs 15%	YOLO	CLBs 41% LUTs 25% BRAMs 39% DSPs 31% FFs 15%	Not Given	Not Given