BX-4 CubeSat Software Architecture: Data Collection & Computer Vision

Taha Teke*
University of Michigan, Ann Arbor, MI 48109

BX-4 is a nanosattelite being developed at the University of Michigan as part of the BLUE Program. For its mission, the spacecraft will have electronics on board to capture flight data and process video information to detect rockets. This paper details the software and hardware architectures designed to perform the aforementioned tasks.

I. Introduction

Moore's Law observes that the number of transistors in a dense integrated circuit approximately doubling about every two years, meaning microprocessor speeds increase exponentially at a rate of two within that time. Technologies that previously needed more physical space can fit in smaller sizes now. In our case, this is especially relevant for the electronics of a nanosatellite, which has the primary purpose of shrinking a satellite in size while keeping as much functionality as possible.

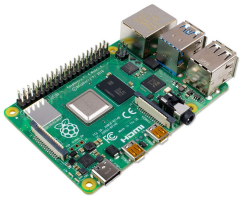
Making use of recent technological advancements in both hardware and software, an integrated hardware-software system has been developed to achieve the BX-4 mission's goals. A central processor is used to control mission execution, an IMU is used to capture flight data, a camera is used to capture video footage and feed it to a deep learning object detection model, and a hardware accelerator is used by the deep learning model to speed up inference of detecting rockets.

II. Hardware

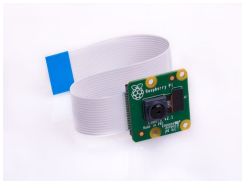
For BX-4, the hardware was chosen considering physical size and computational speed. The size vs speed trade-off is important because functional hardware that can also fit inside a small space are desired. Overall, the hardware is composed of a central processor, an IMU, a camera, and a hardware accelerator.

Overview:

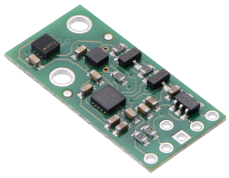
- **Processor** - Raspberryi Pi 4B is the highest model Raspberry Pi with a Quad core ARM Cortex-A72 CPU and 8GB of RAM.
- **Camera** - Pi Camera Module 2 is compatible with all models of Raspberry Pi and supports up to 1080p30 or 720p60 video modes.
- **IMU** - Pololu AltIMU-10 v5 is a 10-DoF that features a LSM6DS33 gyroscope and accelerometer, LIS3MDL magnetometer, and LPS25H barometer.
- **Hardware Accelerator** - Google Coral USB Accelerator is a USB device that provides an Edge TPU to speed up deep learning inferencing on existing systems.



(a) Raspberry Pi 4b



(b) Pi Camera Module 2



(c) Pololu AltIMU-10 v5

(d) Google Coral USB Accelerator

Fig. 1 Hardware

*BS student, Dept. of Electrical Engineering and Computer Science

III. Software

The software system is designed with Python to run on the Linux-based Raspbian operating system. Overall, the software encompasses everything related to flight data capture, dataset pre-processing, mission execution control, and rocket detection.

A. Flight Data Capture

Flight data capture is done with a Pololu AltIMU-10 v5 accessed through the Raspberry Pi's GPIO interface. The data is saved in a CSV file with the filename formatted the date and time of the mission start: *Year-Month-Day_Hour-Minute-Second.csv*. From mission start to end, the x, y, and z gyroscope, x, y, and z accelerometer, x, y, and z magnetometer, pressure, altitude, and temperature information for the payload is saved with a corresponding timestamp as fast as the IMU can capture the data and communicate it to the data read/write software.

```
Time,X-Gyro,Y-Gyro,Z-Gyro,X-Accel,Y-Accel,Z-Accel,X-Mag,Y-Mag,Z-Mag,Pressure,Altitude,Temperature
2021-07-15 18:51:41.577120,3.5,-5.145,-3.15,-3.8979080153999996,0.2739781877,-8.903692894599999,-1026,-2500,2656,987.9,212.99,23.9
2021-07-15 18:51:41.611218,3.255,-5.425,-3.01,-3.8991044266999997,0.2703889538,-9.1776710823,-1026,-2500,2656,987.9,212.99,23.9
2021-07-15 18:51:41.645571,3.08,-4.935,-2.905,-3.9469608786999997,0.22013967919999997,-8.8163548697,-1115,-2590,2659,987.9,213.25,23.9
2021-07-15 18:51:41.679527,3.535,-5.46,-2.38,-3.8787654345999996,0.24526431649999997,-9.175278259699999,-1115,-2590,2659,987.9,213.25,23.9
2021-07-15 18:51:41.713518,3.29,-5.355,-3.15,-3.8931223702,0.26919254249999996,-8.8833539025,-1115,-2590,2659,987.9,213.33,23.9
```

Fig. 2 Flight information in a CSV format.

B. Data Pre-processing

About 950 JPEG rocket images were collected from Google Images for the training, validation, and testing datasets. After data collection, the LabelImg tool was used to manually annotate each rocket in every image with bounding boxes. LabelImg creates new files for each image and saves the rocket bounding box information in an XML format in those files. However, Tensorflow does not accept the XML format, so an additional conversion has to be done to convert the information in these files into a format that Tensorflow accepts. For this case, Tensorflow's custom TFRecord format is acceptable. To generate TFRecord files for an entire dataset (train, validation, etc.), a supplementary program was developed that takes every JPEG image in that dataset and their corresponding XML files to create a new TFRecord file that contains all image and bounding box information.

C. Mission Controller

To control the mission flow execution for the various tasks that need to be completed, a *MissionController* class has been developed as part of a *controller* package. In addition, a *CVDetect* class has been developed as part of a *cv* package to perform the rocket detection and a *DataRW* class has been developed as part of a *data* package to perform the flight data capture. The *MissionController* has instances of *DataRW* and *CVDetect* to control their corresponding tasks. Using Multiprocessing, the *MissionController* makes use of the *DataRW* and *CVDetect* to run the flight data capture and rocket detection tasks in parallel. Parallel execution of tasks is beneficial because it utilizes more computing resources to overcome potential execution bottlenecks. This way, faster clock speeds can be reached for each task than would be possible with sequential execution, where the bottleneck would be caused by each task waiting in a queue for every other task to complete and getting back in the queue again after its execution turn has passed.

Moreover, the software was designed using modular programming and object oriented design principles. Architecting the software this way enables it to more easily be extended in the future to perform additional tasks. An example is potentially adding telemetry software in the future. Once the task is finalized, the same software design pattern can be followed where a *Telemetry* class can be developed as part of a *telemtry* package. Then, an instance of the *Telemetry* class can be added to *MissionController*, which can run then run the telemetry in parallel along with the other mission tasks. Additionally, this structure allows for data sharing between tasks as well, like the bounding box prediction for a rocket detection from *CVDetect* is shared with the *ControlsSystem*, so these coordinates can be used to pivot the payload appropriately when the controls algorithm is finalized and tested.

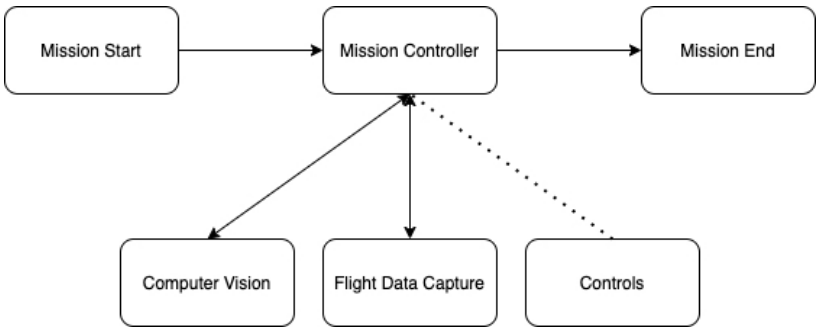


Fig. 3 Mission execution flow graph.

D. Computer Vision

Rocket detection is performed each frame of the video stream from the Pi Camera Module 2 using an EfficientDet-Lite2 object detection model running on the Google Coral USB Accelerator.

1. Hardware

Deep learning models require extensive memory and computational power to make inferences. Because of this, dedicated external hardware accelerators to speed up on-device computation for embedded systems using deep learning models have gained traction recently. Some examples are the Intel Neural Compute Stick 2 and the Google Coral USB Accelerator. Additionally, the NVIDIA Jetson Nano Developer Kit is a microprocessor that has a GPU onboard. When building the computer vision system, we decided to use a hardware accelerator for better models and faster computations.

After researching [1][2][3] to compare and contrast the aforementioned devices, it made most sense to use the Google Coral USB Accelerator. Out of the choices, the NVIDIA Jetson Nano Developer Kit was the most powerful, but it was not possible to use. That's because it would replace the Raspberry Pi as a microprocessor, but the BX-4 build had already been designed considering the Raspberry Pi, so the build would have had to been redesigned, which was not feasible at that moment. Among the remaining choices, the Google Coral USB Accelerator was more powerful than the Intel Neural Compute Stick 2. At the same time, Google maintains the open-source deep learning library Tensorflow and has started to more and more support Tensorflow models on their Edge TPUs [4]. Because of these reasons, the decision was made to use the Google Coral USB Accelerator.

2. Model Selection

Among object detection models, some of the most appropriate for running inferences on embedded devices are the MobileNet family, MobileDet, and the EfficientDet-Lite family because they're specifically designed to be run on mobile systems. From the options, the MobileNet family is the fastest with the lowest accuracy, while the EfficientDet-Lite family is the slowest with the highest accuracy. MobileDet is in the middle in regards to speed and accuracy. Based on the speed of the initially proposed controls system by the controls team, it was agreed that the speed of the EfficientDet-Lite2 model would suffice. Even though EfficientDet-Lite2 is the slowest, it is also the most accurate, so the decision was made to use the EfficientDet-Lite2 model. In the future, if mission requirements change, it'll be possible to pivot to other models, though.

Object Detection			
Models	Input Size	Latency	mAP
SSD MobileNet V1	300x300x3	6.5ms	21.5%
SSD MobileNet V2	300x300x3	7.3ms	25.6%
SSDLite MobileDet	320x320x3	9.1ms	32.9%
EfficientDet-Lite0	320x320x3	37.4ms	30.4%
EfficientDet-Lite1	384x384x3	56.3ms	34.3%
EfficientDet-Lite2	448x448x3	104.6ms	36.0%

Fig. 4 Model object detection stats running on a Google Edge TPU.

3. Training

Training was done with a Jupyter Notebook on Google Colab using a GPU. Overall, the entire training session took about 12 hours. On a sidenote, it's highly discouraged to train on a CPU, as it will take significantly longer, probably a couple days at least. A TensorFlow EfficientDet-Lite2 model pre-trained on the COCO 2017 Object Detection dataset was used. The custom rocket dataset described in section III.B Data Pre-processing is used to retrain the EfficientDet-Lite2 model with transfer learning, making use of features the model previously learned on the larger COCO 2017 dataset. Training is done with a batch size of 32 for 150 epochs. The best observed validation performance was at epoch 125, so the model weights at that epoch were saved. After saving the best model, full integer post-training quantization is performed to export the model to the TensorFlow Lite format for compatibility with the Google Coral USB Accelerator. Using integers instead of floats shrinks the model size and lowers accuracy, but decreases inference time. After exporting to TensorFlow Lite, the model is then compiled for the Edge TPU, which is the internal hardware of the Google Coral USB Accelerator.

4. Inference

The Pi Camera Module 2 is used to get a live video stream. Since the EfficientDet-Lite2 model works with still images, a single frame from the video is grabbed to analyze. That frame is resized to 448x448 pixel dimensions, which is the input size for the EfficientDet-Lite2 mode, and input into the EfficientDet-Lite2 model using a score threshold of 25 out of 100. The score threshold is customizable when running the EfficientDet-Lite2 model, so higher or lower thresholds can be used to eliminate or include lower probability predictions. In our case, a lower threshold is used due to the EfficientDet-Lite2 model being a mobile model, meaning weaker detection capabilities than larger models, and only needing to detect a single rocket, which the model might sometimes detect but give a low score for.

After being input an image, the EfficientDet-Lite2 model then outputs bounding box predictions with scores greater than 25 for potential rockets. It takes about 100-150 ms from the image input into the EfficientDet-Lite2 model to it processing the image and outputting all bounding box predictions. Each bounding box prediction includes a score, along with (x_{min}, y_{min}) and (x_{max}, y_{max}) pairs for the bottom-left and top-right coordinates of the bounding box. Because the scope of BX-4's mission is to detect a single rocket, the predictions are iterated over to find the one with the highest score to use as the best prediction. Using the (x_{min}, y_{min}) and (x_{max}, y_{max}) from that best prediction, the center of the best prediction can be found with the following formulas:

$$x_{center} = \frac{x_{min} + x_{max}}{2} \quad (1) \qquad y_{center} = \frac{y_{min} + y_{max}}{2} \quad (2)$$

For each video frame analyzed, the center of the best rocket prediction is calculated. Once the controls system is developed, it can use the centers of these predictions (x_{center}, y_{center}) to track the rocket by pivoting the payload to align the center of the camera's point of view with the center of the prediction.

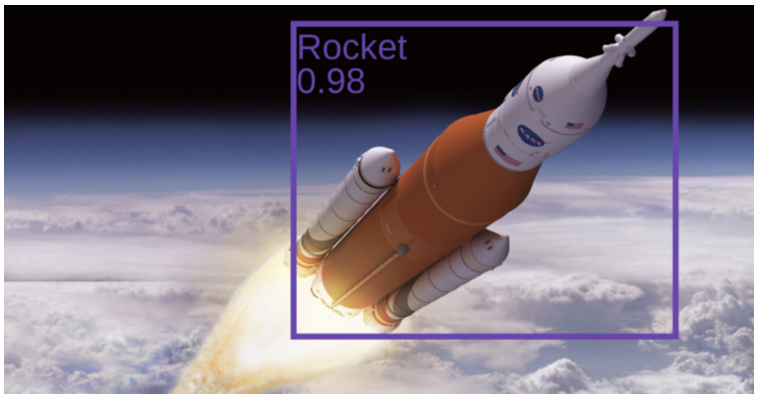


Fig. 5 Model prediction with bounding box and score.

5. Difficulties

For the computer vision, there was about a 3-4 week period of attempting to segment the EfficientDet-Lite2 model and pipeline it across multiple Google Coral USB Accelerators. However, after working on the code and doing more research into the problem online, this goal proved more difficult than initially expected. Currently, the PyCoral API supports making pipelined models for classification tasks, but not for detection tasks. In its current state, one inference can be made on only a single image with a pipelined model for detections, not multiple inferences on a continuous stream of images. This is due to how images are queued across multiple Google Coral USB Accelerators. With the queue, images have to be popped after performing detections on them. However, the pipelined models for detection don't have consistent pop operations for the queue, which can cause memory overflow and being stuck on the same image without looking at new ones.

There are discussions on Google's official PyCoral Github repository on this topic. In October 2021, extending the queue pop functionality was discussed, but it was mentioned that due to other priorities that request won't be possible in the near future. [5] However, after checking the discussions again recently, it seems like active work has been done on this in November 2021 to fix this broken functionality. [5][6]

IV. Controls System

A skeleton package was added for the controls system that can filled out with functionality once the mechanics of it and the algorithm to control it are finalized. A *ControlsSystem* class was developed as part of a *controls* package. An instance of the *ControlsSystem* class was added to *MissionController*, which can run then run the controls in parallel along with the other mission tasks. Additionally, the prediction from the computer vision task is being data between tasks as well, as in the bounding box prediction for the rocket detection from *CVDetect* is being shared with the *ControlsSystem*, so in the future code can be added to pivot the payload appropriately using this information.

V. Integration Testing

Functionality has now been added to do practice mission runs. The purpose of practice runs is to have GUIs for the mission tasks to manually verify that overall code functionality is working as expected. For a practice run, the computer vision task opens a window with the camera's view with bounding boxes for the best rocket prediction and the flight data capture task prints the IMU data to the terminal window. In a real mission run, neither of these visuals are displayed due to I/O negatively impacting performance and tasks are run purely as background processes. To do a practice run, a command line argument is added when running the program: *–run practice*.

VI. Documentation

For future team members, the code has been thoroughly commented throughout the codebase, including the *controller*, *cv*, and *data* packages. Additionally, README documentations are being developed for the aforementioned packages and should be completed by the end of 2021. A README for the overall codebase will be added as well. With these documentations, a new team member should be able to understand the code and contribute to the codebase significantly more quickly, especially if there may not be anyone else there to explain how the code works or why it's structured in the way it is. Additionally the Jupyter Notebook developed for training the detection model has also been commented, so a future member can use it for retraining if required, possibly even for a different task with a different model or a different dataset.

VII. Future Work

With the recent advancements of Google's support for pipelining object detection models across multiple Google Coral USB Accelerators, it may be worthwhile to revisit implementing segmenting the model to make use of additional memory and computing power. However, it would be advised to approach this with caution because the Google Coral USB Accelerator (released in 2019) and PyCoral API (released in 2020) are fairly new technologies and can have bugs or incomplete functionalities, as experienced in this paper.

Moreover, an appropriate direction to strongly consider is substituting the current hardware and software with alternatives. Replacing the hardware with more powerful alternatives will not only allow better performance of the current software, but will also allow to use better performing and more compute-intensive software as well. In place of the Raspberry Pi 4B and Google Coral USB Accelerator, NVIDIA hardware with onboard GPUs can be used. Some

alternatives to consider are the medium-end Jetson Xavier NX Developer Kit or the high-end Jetson AGX Xavier Developer Kit. Each of these devices can replace both the general purpose functionality of the Raspberry PI 4B and the specialized functionality of the Google Coral USB Accelerator with significantly higher performances. [7] As an example, consider that the NVIDIA hardware can run state of the art object detection models due to NVIDIA specific and Jetson series specific software optimizations, like the Jetson AGX Xavier Developer Kit can run a YOLOv4 model optimized with TensorRT and tkDNN at about 30 frames per second. [8]

Finally, the controls system software has not been completed yet. In the future, it's recommended to prioritize finalizing an algorithm and performing the physical testing of the proposed controls system, iterating on and improving the algorithm. The codebase has been built in a such way that adding the functionality mainly requires the aforementioned work, as a *controls* package has already been added with a *ControlsSystem* class that has access to the prediction from the computer vision task.

References

[1] Bangash, I., "NVIDIA Jetson Nano vs Google Coral vs Intel NCS: A Comparison," ???? URL https://towardsdatascience.com/nvidia-jetson-nano-vs-google-coral-vs-intel-ncs-a-comparison-9f950ee88f0d.

[2] Cook, J. S., "Google Coral Edge TPU Accelerator vs. Intel Neural Compute Stick 2," ???? URL https://www.arrow.com/en/research-and-events/articles/google-coral-edge-tpu-accelerator-vs-intel-neural-compute-stick-2.

[3] Weiss, S., "JETSON NANO AND GOOGLE CORAL EDGE TPU - A COMPARISON," ???? URL https://3dvisionlabs.com/2019/09/18/jetson-nano-and-google-coral-edge-tpu-a-comparison/.

[4] Google, "TensorFlow models on the Edge TPU," ???? URL https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview.

[5] "Error at PipelinedModelRunner with detection models." , ????. URL https://github.com/google-coral/pycoral/issues/36.

[6] "model pipelining error for object detection," , ????. URL https://github.com/google-coral/tutorials/issues/17.

[7] "model pipelining error for object detection," , ???? URL https://developer.nvidia.com/embedded/jetson-benchmarks.

[8] Bochkovskiy, A., "YOLOv4 — the most accurate real-time neural network on MS COCO dataset." ????