2017-06-01

CUDA-Accelerated ORB-SLAM for UAVs

Donald Bourque
Worcester Polytechnic Institute

CUDA-Accelerated Visual SLAM For UAVs

by

Donald Bourque

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Robotics Engineering

May 2017

APPROVED:

____

Professor Michael A. Gennert, Major Advisor, Head of Program

Abstract:

This thesis explores ORB-SLAM, a popular Visual SLAM method appropriate for UAVs. Visual SLAM is computationally expensive and normally offloaded to computers in research environments. However, large UAVs with greater payload capacity may carry the necessary hardware for performing the algorithms at sufficient framerates. The inclusion of general-purpose GPUs on many of the newer single board computers allows for the potential of GPU-accelerated computation within a small board profile. The NVidia Jetson TX2, a single board computer contain an NVidia Pascal GPU, was selected for this research. CUDA, NVidia's parallel computing platform, was used to accelerate monocular ORB-SLAM to perform onboard Visual SLAM on a small UAV.

Table of Contents:

Table of Figures

1. Introduction

The use of cameras and computer vision algorithms to provide state estimation for robotic systems has become increasingly popular for small mobile robots, self-driving cars, and unmanned aerial vehicles (UAVs) [1]. This popularity stems from the availability and decreasing cost of technology as well as recent advancements in the field. Computer vision algorithms extract information from the camera images and perform simultaneous localization and mapping (SLAM) for path planning, obstacle avoidance, or 3D reconstruction of the environment.

There are several types of sensors that can be used with SLAM algorithms. However, camera's are becoming more widely used. This is due to higher resolution cameras have become inexpensive and are a lightweight and smaller alternative to other sensing hardware, such as laser scanners. Laser scanners are most commonly used on ground vehicles where there are greater payload capabilities and battery life. Instead, UAVs often have monocular camera or stereo camera setups since payload and size impose the greatest restrictions on their flight time and maneuverability.

UAVs that perform off-board processing require constant communication with the system that is performing the SLAM computation. While off-boarding the processing can decrease the size of the UAV and increase flight time, the range can severely be affected as the UAV would need to stay within wireless communication range of the processing system. To eliminate this impact on the range, SLAM processing could be performed onboard the drone. Unfortunately, this would normally increase the size of the drone and decrease flight time due to the addition of the processing hardware. Decreasing the size, power consumption and computational time of the onboard processing system would be beneficial for a mobile robot, especially UAVs.

GPU-accelerated processing could grant these benefits. GPU's typically are in larger devices such as laptops, but many of the newer single board computers being released contain general-purpose GPUs, all within a small board profile. This thesis explores the use of GPU-acceleration for ORB-SLAM, a popular Visual SLAM method appropriate for UAVs. The parallelized algorithm will be then by implemented on a small UAV. A small UAV capable of onboard computation of Visual SLAM would be useful in applications such as search and rescue, disaster response, and surveillance in GPS-denied, unstructured environments with unreliable communication. In one such scenario, a first responder to a scene would be capable of having a small UAV map a building to provide firefighters with greater awareness of the state of the building and the location of individuals inside.

2. Literature Review

This section provides a background and a basis of knowledge for the rest of the paper. Topics that are discussed are Visual SLAM, Visual SLAM methods such as PTAM, ORB-SLAM, LSD-SLAM and DSO, GPU-acceleration and CUDA programming.

2.1 Visual SLAM

Simultaneous localization and mapping (SLAM) is a method to solve the problem of mapping an unknown environment while localizing oneself in the environment at the same time [28,29]. SLAM provides a means to autonomously navigate an area which has many applications, including self-driving cars and mapping the topology or inspecting buildings with UAVs [30,31] . Due to sensor errors or model inaccuracies, relying on IMUs or wheel odometry for robot localization can be very inaccurate over time. If the robot's pose cannot accurately be determined, then it is not possible to produce an accurate map of the environment. Other systems like GPS can assist with this problem but they are not always available. SLAM can use various sensors to accomplish mapping and robot localization: cameras, 2D laser range finders, LiDar or sonar sensors.

Visual SLAM is the method that specifically uses cameras to map the environment. Image processing is used to extract features from the images that become landmarks in the developing map.Using landmarks in the mapped environment allows the robots’ position in the environment to be determined. Below is a diagram (Figure 1) of SLAM pose estimation for a robot navigating an environment.

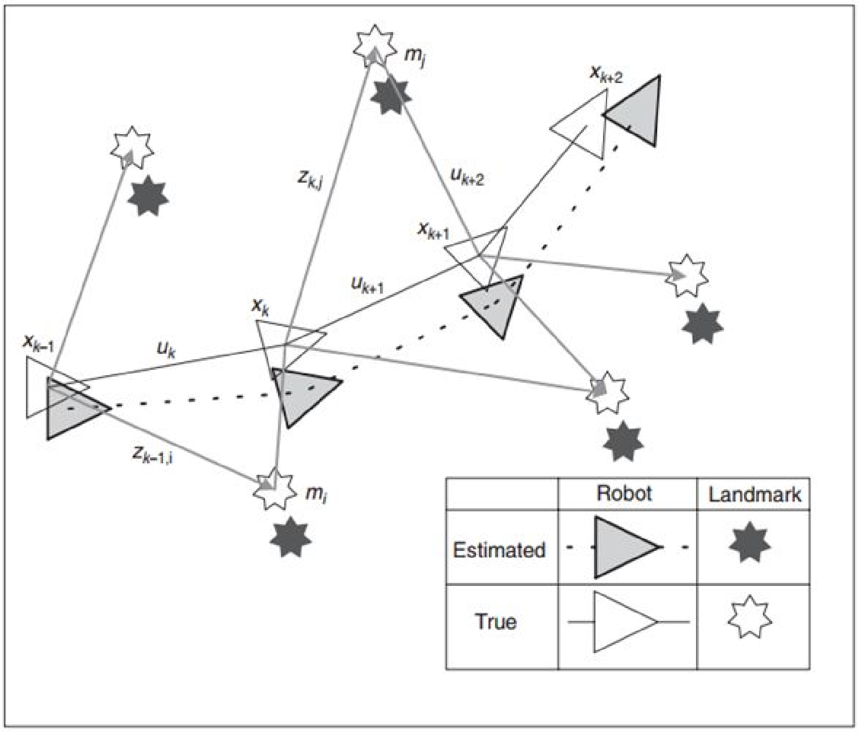


Figure 1: The estimation problem of SLAM [16].

However mapping a large area can be difficult as the feature points take up memory and increase computational complexity, and that drift can accumulate over time. These problems are frequent in SLAM algorithms. In the fields of research, monocular systems are very popular because cameras are inexpensive but offer lots of valuable information about the environment. Small UAVs and small robots typically have a monocular camera system, but the absolute scale of the environment can not be known using just a monocular camera for SLAM.

2.2 Visual SLAM Methods

There are several important characteristics that Visual SLAM methods can have. Illumination-Invariance is the algorithm's ability to cope with different illuminations of landmarks. Methods based on the brightness constancy assumption are not illumination invariant. Loop closure is when the algorithm can recognize a past location after encountering new terrain and adjust the map accordingly [32]. Relocalization is a valuable feature of an algorithm because if tracking were to ever fail due to an overly saturated image or heavy motion blur the algorithm would be capable of relocalizing. The generated map density is also important to consider. Robots due not require dense maps to localize themselves in an environment, but sparse maps can be very difficult for human viewers to interpret. A comparison of LSD-SLAM semi-dense map reconstruction against ORB-SLAM's sparse reconstruction shown below make this point clear. And lastly, computation complexity is an important algorithm characteristic to consider when selecting a Visual SLAM algorithm. Densely reconstructed maps are very pleasing to look at, but require GPU-acceleration to achieve an amount of satisfactory results.

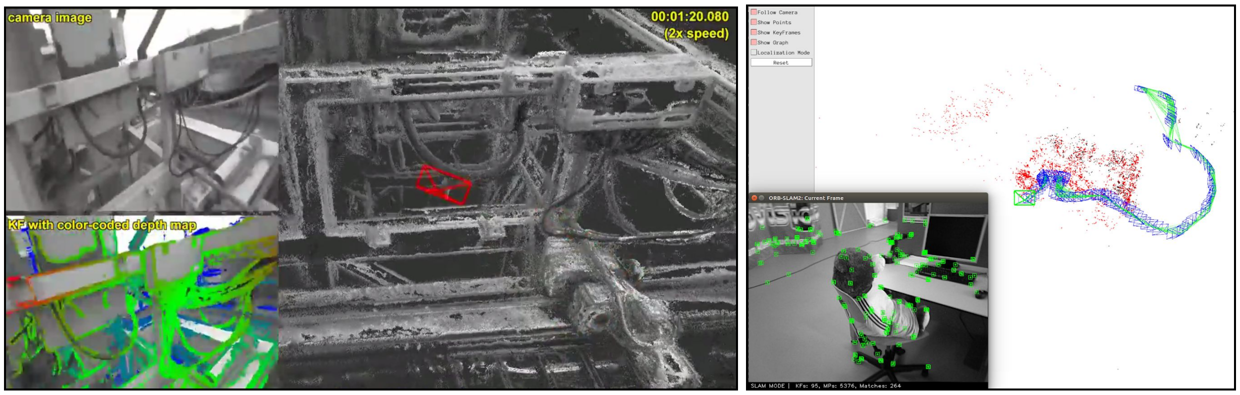
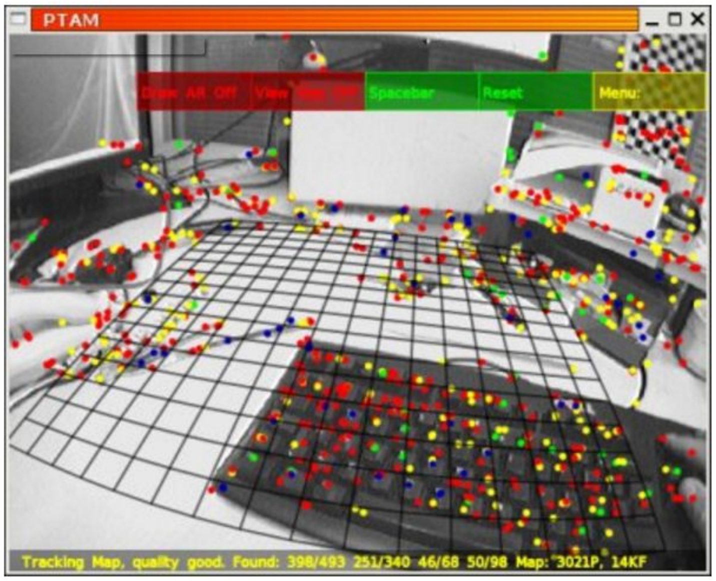


Figure 2: Reconstruction density of Visual SLAM systems. Left: LSD-SLAM semi-dense reconstruction [18]. Right: ORB-SLAM sparse reconstruction [17].

2.2.1 Parallel Tracking and Mapping

Parallel Tracking and Mapping (PTAM) was developed by two Augmented Reality (AR) researchers, Georg Klein and David Murray, in 2007. PTAM is a feature-based SLAM algorithm that produces a sparse reconstruction of landmarks. It is also notably the first algorithm to parallelize the motion estimation and mapping tasks. It uses keyframe-based Bundle Adjustment (BA) for pose and map refinement. Most modern feature-based Visual SLAM systems are based on this parallelization of tracking and mapping, and use keyframe-based map refinement with BA. PTAM's keyframe generation is linear over time.



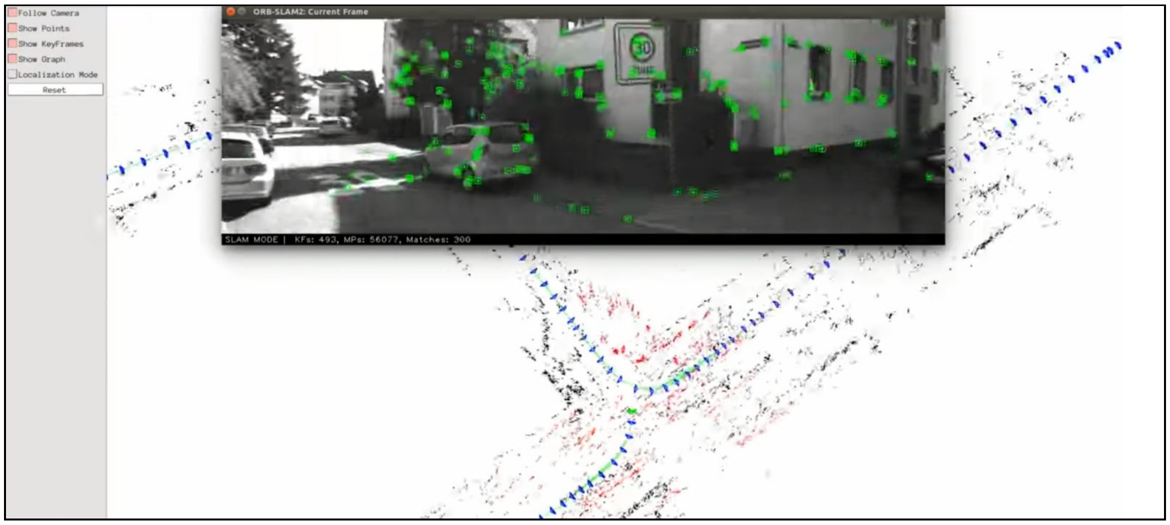
Tracking Map, quality good. Found: 398/493 251/340 46/68 50/98 Map: 3021P, 14KF

Figure 3: PTAM landmark tracking [14].

Prior to PTAM, Extended Kalman Filter (EKF) based Visual SLAM was the standard. PTAM outperforms these EKF Visual SLAM algorithms, like Mono-SLAM, in efficiency and precision [7]. Frames are tracked and mapped in a relatively constant amount of time with PTAM, but the amount of time for each frame scales quadratically for EKF-SLAM with the number of landmarks. PTAM uses keyframe-based map refinement with BA, which is considered a Graph SLAM technique where keyframes and map points are graph nodes and are optimized to minimize their measurement errors [12]. The sparsity of the PTAM's graph leads to greater computational efficiency when compared to EKF-SLAM's approach of maintaining a sparse feature map within a state vector and non-sparse large covariance matrix [13].

2.2.2 ORB-SLAM

ORB-SLAM is a feature-based system, like PTAM, but has many improvements. It consists of three main parallel threads for tracking, mapping, and loop closing. It also makes use of ORB feature detector and descriptor which improves robustness to scale and orientation changes during image tracking and feature matching [27]. Oriented FAST and Rotated BRIEF, termed ORB, is based upon FAST corner detection, computing orientations of those corners, and generating binary descriptors for the features to provide rotation invariance and some robustness to illumination changes. Image pyramids yield scale invariance, and ORB-SLAM also has automatic map initialization where PTAM requires manual operation to initialize. Loop closure, pruning of redundant keyframes, and scene recognition using Bag of Words are all attractive features of ORB-SLAM. ORB-SLAM maps sparse landmarks, which can be seen below in Figure 4, a car driving sequence.



```
SLAM MODE |  KFs: 493, MPs: 56077, Matches: 300
```

Figure 4: ORB-SLAM (stereo) in long driving sequence [17].

2.2.3 LSD-SLAM

Large-Scale Direct (LSD) SLAM is a popular direct Visual SLAM algorithm for monocular camera setups that reconstructs a semi-dense depth map online (2). It generates highly accurate pose estimations of the camera based on direct image alignment. 3D reconstruction is done in real-time as a pose-graph of keyframes along with the semi-dense depth maps. LSD-SLAM offers loop closure and map optimization using pose graph optimization, but relies on the brightness constancy assumption. LSD-SLAM performing loop closure is shown below in Figure 5. A major drawback of choosing LSD-SLAM for the Visual SLAM algorithm is that if tracking fails, the algorithm can not recover.

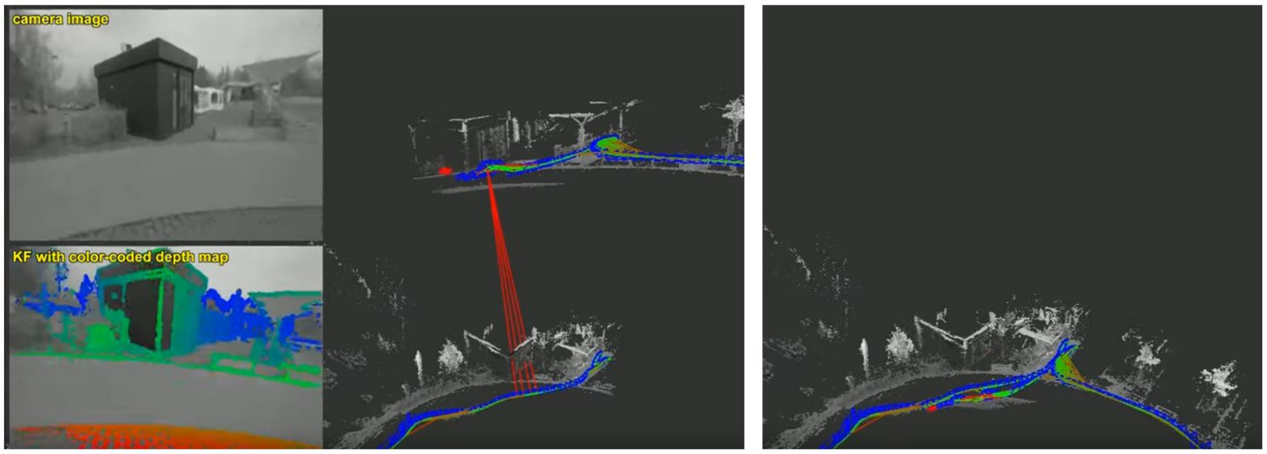


Figure 5: LSD-SLAM loop closure [18]. Left: loop closure is being detected. Right: Corrected semi-dense reconstruction after loop closure.

2.2.4 Direct Sparse Odometry

Direct Sparse Odometry (DSO) is a direct sparse Visual odometry method for monocular camera setups. This method combines the benefits of direct methods with flexibility and efficiency of sparse approaches. This method does not depend on keypoint detectors or descriptors and omits the smoothness prior used in other direct methods, instead, by sampling across all image regions that have intensity gradient.

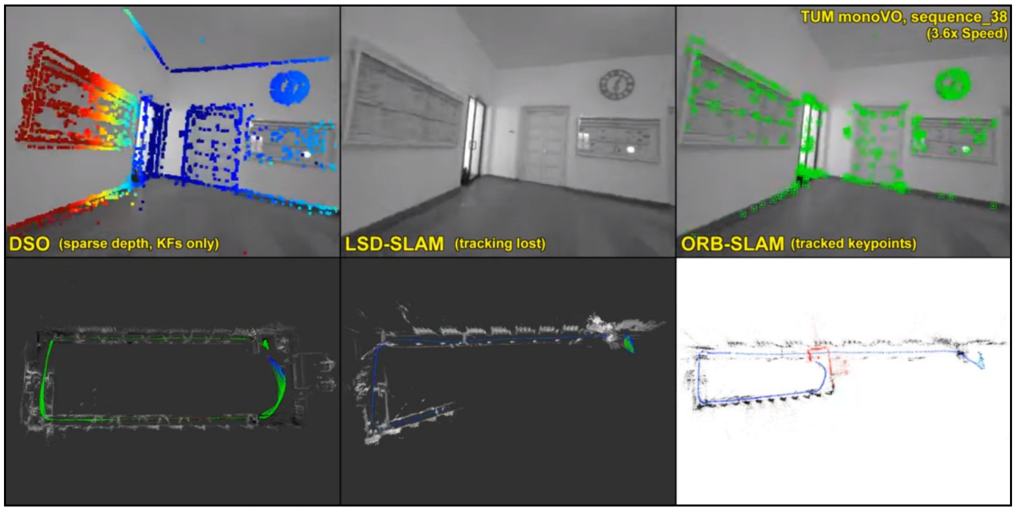


Figure 6: Comparison of DSO, LSD-SLAM, and ORB-SLAM systems [19]. Left: DSO. Middle: LSD-SLAM with tracking failure during a sequence. Right: ORB-SLAM with disabled loop closure.

DSO incorporates photometric camera calibration through the use of an image formation model which accounts for a non-linear response function as well as lens attenuation. This model corrects each image frame making DSO robust to large changes in illumination. The number of points chosen in each frame is adjustable, allowing flexibility under real-time constraints. Engel et. al. have shown that an increased number of points does not increase tracking accuracy, but does make the 3D model denser (5). It is also shown that using all points instead of only corners does provide increased accuracy and robustness over indirect methods such as ORB-SLAM (8).

2.3 GPU-Acceleration and Parallel Programming

GPU-acceleration is the term used when a GPU is used alongside a CPU to accelerate programs. Sections of code can be offloaded to a GPU while the remainder of the code is still run on the CPU. Below is a graphic from NVidia showing how GPU acceleration works (Figure 7).

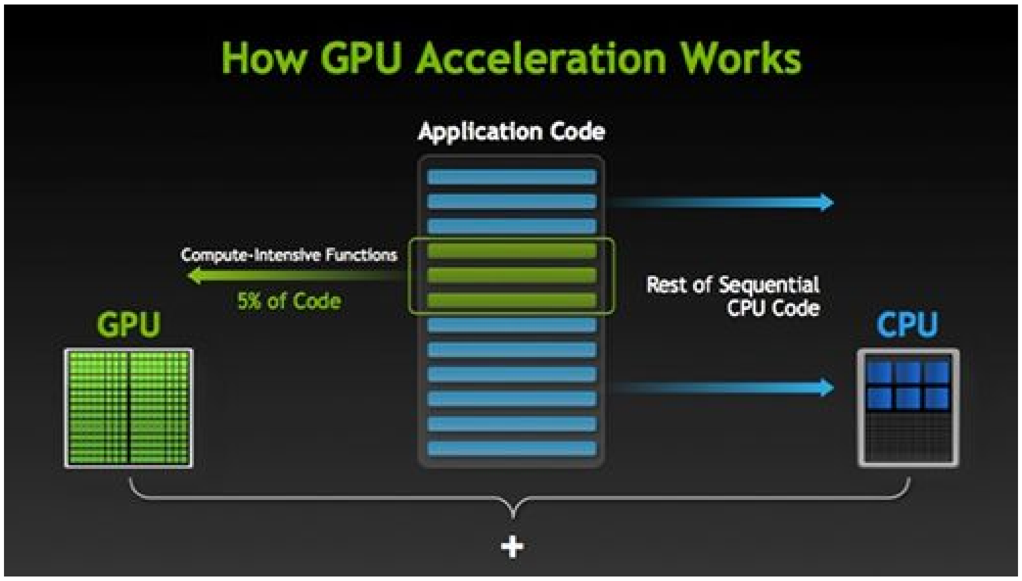

How GPU Acceleration Works
Application Code
Compute-Intensive Functions
5% of Code
GPU
Rest of Sequential CPU Code
CPU
+

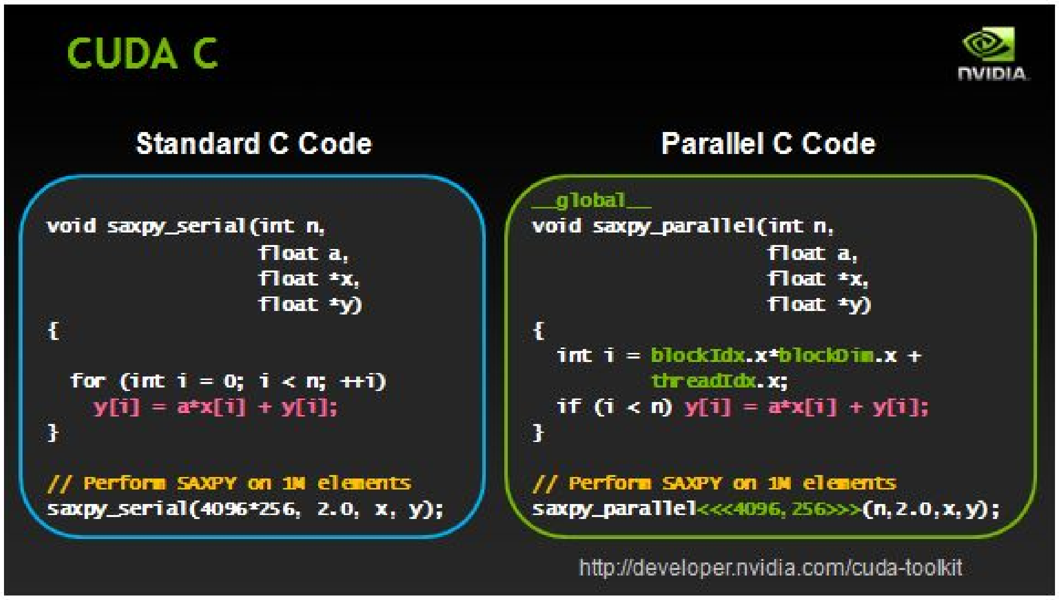
Figure 7: How GPU Acceleration works by NVidia [23].

Despite only a small portion of the code being processed by the GPU, if that code executes 100's of times, the greater throughput the GPU has compared to the CPU can greatly decrease the overall run time.

This acceleration is possible due to the different architectures of the CPU and GPU. CPU's contain a few small processing cores that are optimized for sequential, serial processing. CPU's also contain a lot of cache memory which are able to handle only a few software threads at a time. GPU's, however, are optimized for processing in parallel and consists of thousands of smaller, more efficient cores. GPU's can run thousands of threads simultaneously, allowing for faster processing than a CPU while being more power and cost efficient [33, 34].

GPU's were originally designed for game rendering, however their processing is now being used to accelerate the computational workload of various processes and algorithms. Many processes that involve image, video or signal processing are now performed on a GPU for the increased efficiency.

One of the major developers parallel computing platforms is NVidia. NVidia developed it's own parallel computing platform API called CUDA [35]. CUDA allows for other developers to easily parallelize their code on NVidia GPU's by adding a few lines of CUDA code. Below in Figure 8 shows a basic example of CUDA code .



```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

```
__global__
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);
```

Figure 8: CUDA C example of parallelization of serial code [23].

In the parallelized code, block IDs and thread IDs must be tracked for CUDA. To keep track of threads, CUDA has a number of threads in the same block, with each thread having its own unique identifier compared to the other threads in the same block. Each block then has its own unique id so that the threads within each block will not be accidentally seen as threads from another block. The block dimensions which tell the size of each block, which can show how many threads are in each block. Threads are placed into blocks to store thread tracking information without using a lot of memory. For example, if there are 2 million threads that are running, it's much easier to store the numbers (2,3) and (0,0) corresponding to the block and thread IDs compared to 2 million, a 7-digit number. Figure 9 below is a simple graphic that shows the blocks and threads in CUDA.

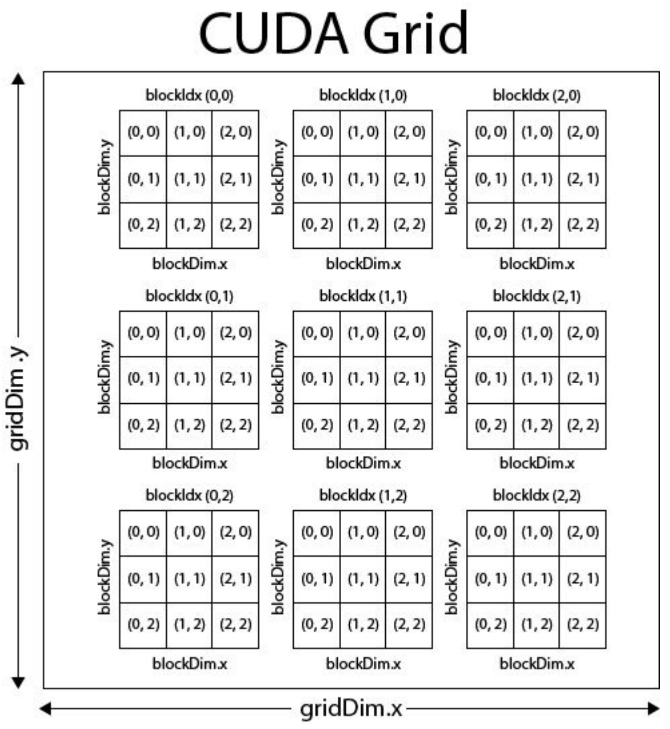
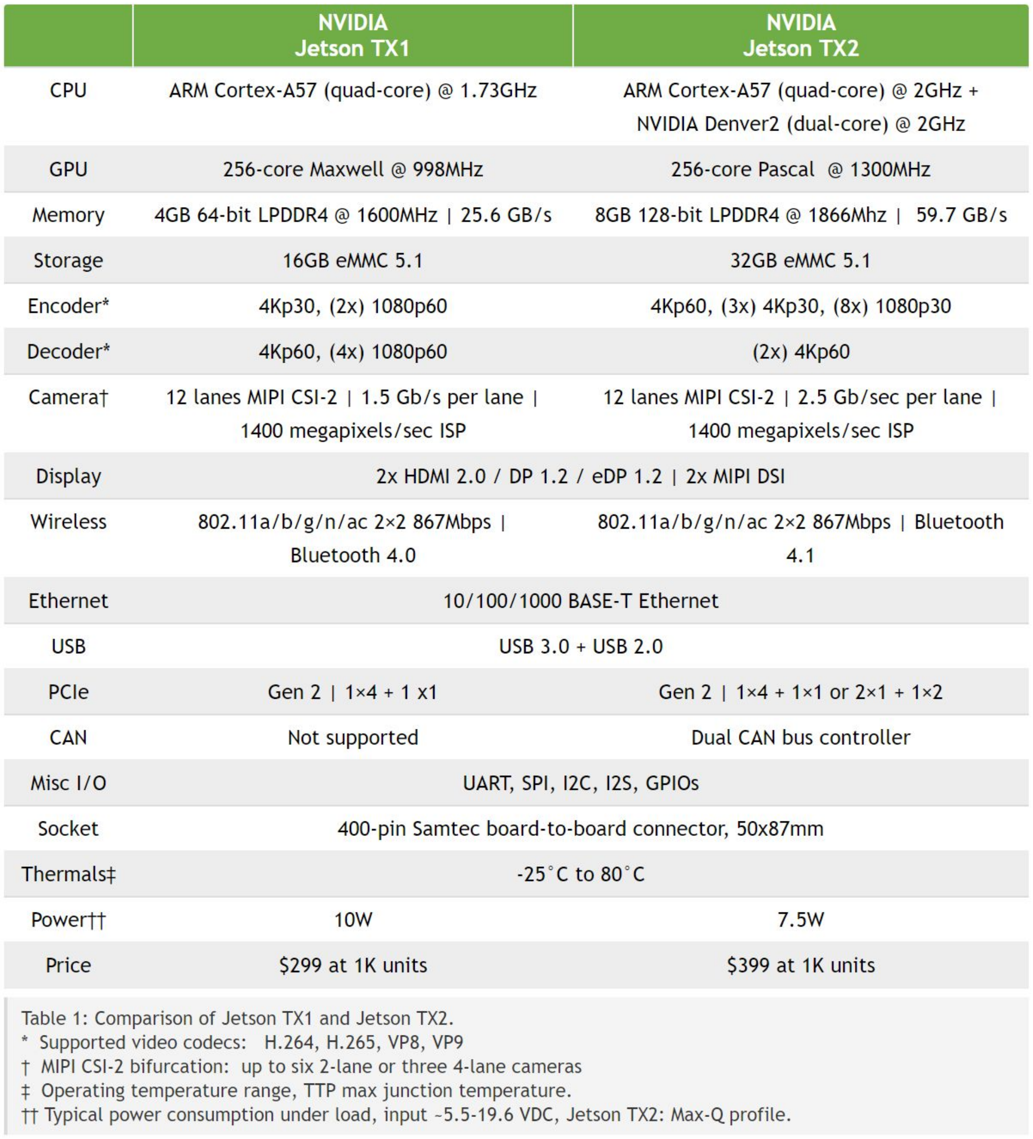


Figure 9: CUDA block and thread diagram [36]

2.3.1 NVidia Jetson TX2

One of NVidia's newest boards, with an onboard GPU and CUDA capabilities for accelerated programming, is the Jetson TX series. The Jetson TX series is known for its small form factor and high computational output. The Jetson TX series has two models, the TX1 and TX2 which are compared in the figure below (Figure 10). The TX2 has an NVidia's Pascal GPU that is 30% more powerful than the TX1's already ~1TFLOPS Maxwell GPU. The TX2 is also equipped with a more power quad-core CPU, along with a dual-core NVidia Denver2 CPU, all while using 25% less power overall. [37]



	NVIDIA Jetson TX1	NVIDIA Jetson TX2
CPU	ARM Cortex-A57 (quad-core) @ 1.73GHz	ARM Cortex-A57 (quad-core) @ 2GHz + NVIDIA Denver2 (dual-core) @ 2GHz
GPU	256-core Maxwell @ 998MHz	256-core Pascal @ 1300MHz
Memory	4GB 64-bit LPDDR4 @ 1600MHz | 25.6 GB/s	8GB 128-bit LPDDR4 @ 1866Mhz | 59.7 GB/s
Storage	16GB eMMC 5.1	32GB eMMC 5.1
Encoder*	4Kp30, (2x) 1080p60	4Kp60, (3x) 4Kp30, (8x) 1080p30
Decoder*	4Kp60, (4x) 1080p60	(2x) 4Kp60
Camera†	12 lanes MIPI CSI-2 | 1.5 Gb/s per lane | 1400 megapixels/sec ISP	12 lanes MIPI CSI-2 | 2.5 Gb/sec per lane | 1400 megapixels/sec ISP
Display	2x HDMI 2.0 / DP 1.2 / eDP 1.2 | 2x MIPI DSI	
Wireless	802.11a/b/g/n/ac 2×2 867Mbps | Bluetooth 4.0	802.11a/b/g/n/ac 2×2 867Mbps | Bluetooth 4.1
Ethernet	10/100/1000 BASE-T Ethernet	
USB	USB 3.0 + USB 2.0	
PCIe	Gen 2 | 1×4 + 1 x1	Gen 2 | 1×4 + 1×1 or 2×1 + 1×2
CAN	Not supported	Dual CAN bus controller
Misc I/O	UART, SPI, I2C, I2S, GPIOs	
Socket	400-pin Samtec board-to-board connector, 50x87mm	
Thermals‡	-25°C to 80°C	
Power††	10W	7.5W
Price	$299 at 1K units	$399 at 1K units

Table 1: Comparison of Jetson TX1 and Jetson TX2.
* Supported video codecs: H.264, H.265, VP8, VP9
† MIPI CSI-2 bifurcation: up to six 2-lane or three 4-lane cameras
‡ Operating temperature range, TTP max junction temperature.
†† Typical power consumption under load, input ~5.5-19.6 VDC, Jetson TX2: Max-Q profile.

Figure 10: Comparison of Jetson TX2 to previous Jetson TX1 [22].

In a comparison to other boards with a similar form size, Figure 11 shows the total runtime of the C-Ray benchmark program with the Jetson TX1. There were no TX2 comparisons at this time, due to the recent release of the TX2.

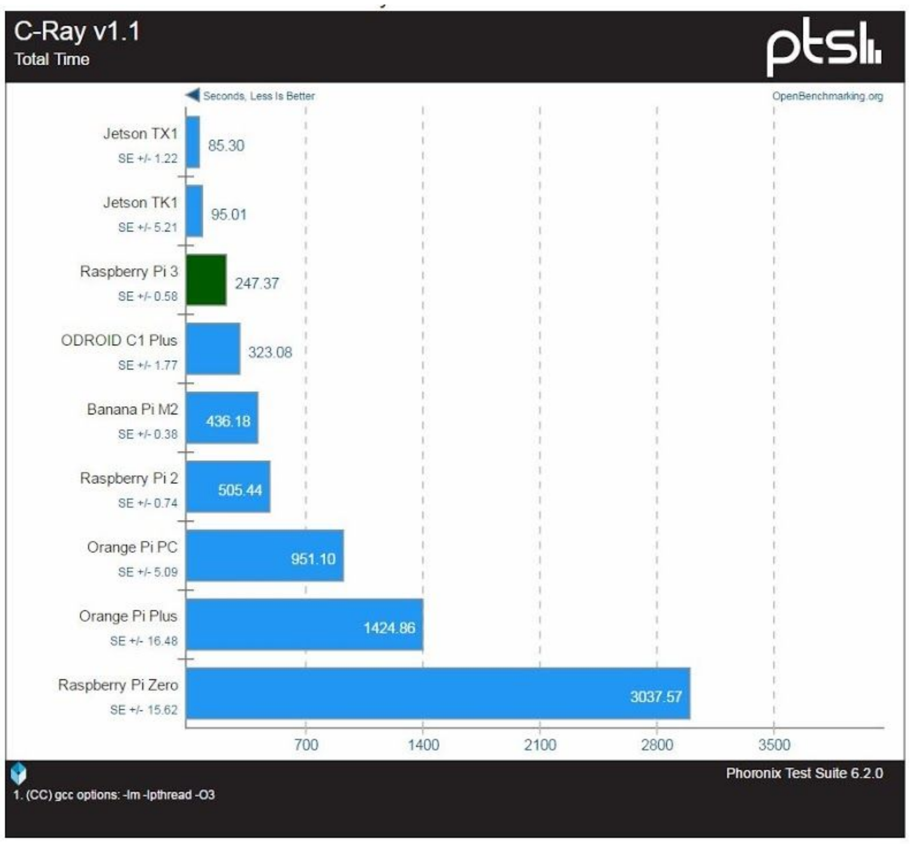


Figure 11: Comparison of Raspberry Pi 3 with other ARM Linux boards with C-Ray multi-threaded ray-tracer [24].

3. Methodology

3.1 Visual SLAM Software

ORB-SLAM was ultimately chosen as the Visual SLAM method to use for this research due to its loop closure, ability to recover from lost tracking, and its pruning of redundant keyframes. PTAM is best-suited for small workspaces and does not scale well to larger maps since it does not have loop closing ability. LSD-SLAM does have loop closing ability, but it is incapable of recovering from lost tracking and it seemed to lose tracking too frequently during image sequences. DSO is quite impressive and it could be argued that it does not require loop closure to produce satisfactory results, but this method is pure visual odometry. DSO only locally tracks the motion of the camera and does not build a consistent, global map of the environment. For these reasons, ORB-SLAM was selected for this research.

OpenCV version used in this research was 3.2, the latest at the time. OpenCV3.2 needed to be built from source with CUDA support for the NVidia Pascal GPU. This was done with the cmake argument “-DCUDA_ARCH_BIN=6.2”. A software edit in the g2o third-party library allows for successful building with OpenCV3.2 and CUDA [38].

ORB-SLAM consists of three main parallel threads for tracking, mapping, and loop closing. An overview of these threads can be seen below in Figure 12. Due to the nature of the parallel threads, the limitation factor for Frame processing is the time required by the Tracking thread - a new Frame cannot be input into the system until the Tracking thread has finished with the last Frame. During initial tests, it also became apparent that the bottleneck of the algorithm lies in the Tracking thread. The Tracking thread thus became the focus of the CUDA-acceleration effort.

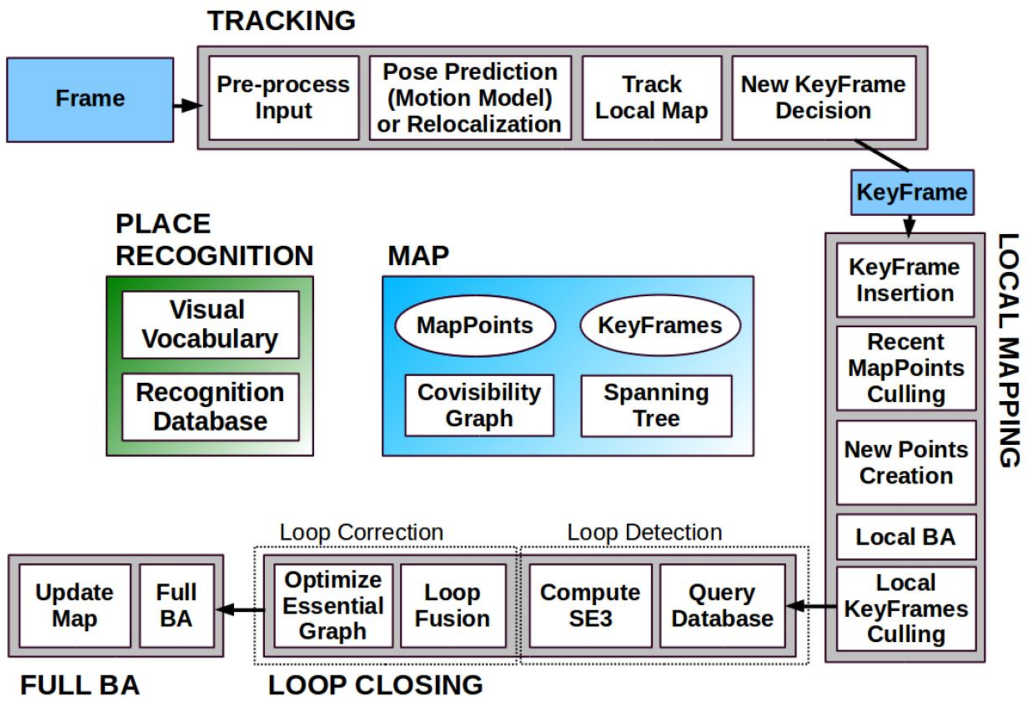


Figure 12: Threads and modules of ORB-SLAM algorithm.

3.1.1 ORB-SLAM Tracking Thread

The algorithm initializes the system by loading ORB vocabulary, then creating a database for the KeyFrames, and other necessary objects. Multiple threads are initialized and the Tracking thread is adopted as the main thread while the other threads are launched. During system initialization, the camera parameters are loaded from a camera settings file and the ORB runtime parameters are loaded as well. These parameters include the number of features per image, the scale factor for image pyramids, the number of levels in the image pyramid, as well as minimum and maximum FAST thresholds.

After system initialization, an input image is passed into the Tracking thread of the system as a Frame. Only after this, image is converted to grayscale and ORB features are extracted in the “Preprocessing” step. This includes FAST corner detection, computing the orientation of the FAST corners, and generating the binary descriptors for the features. Then the Pose Prediction (Motion Model) or Relocalization step attempts to match and track the ORB features. After the Track Local Map step, a check will be performed to determine if a new

KeyFrame should be created. Any new KeyFrames generated would be processed by the Local Mapping thread. The Tracking thread algorithm steps are summarized below:

Tracking Thread Algorithm:

```
1) System initialization
    a) Load ORB Vocabulary
    b) Create KeyFrame Database and other necessary objects
    c) Initialize Threads
    d) Load Camera Parameters from settings file
    e) Load ORB Parameters
2) Create New Frame with input image
3) Pre-process Input
    a) Convert image to grayscale
    b) Extract ORB features**
        i) Pre-compute the scale pyramid
        ii) Compute keypoints
        iii) Gaussian Blur
        iv) Compute descriptors for keypoints
        v) Scale keypoint coordinates
4) Pose Prediction (Motion Model) or Relocalization
    a) Initialization if not initialize
        i) Set reference frame
        ii) Try to initialize by finding correspondences
        iii) Set Frame poses
        iv) Create initial Map
    b) Track Frame by matching keypoints
5) Track Local Map
6) Check if a KeyFrame should be inserted
```

**Determined to be the bottleneck of the Tracking thread.

3.1.2 Parallelization of ORB-SLAM

The high-level steps of the algorithm remain similar after parallelizing the extraction of the ORB features with CUDA. First, the scale pyramid is precomputed, but using OpenCV's CUDA GpuMat methods in this implementation. Then, for each pyramid level, an asynchronous CUDA kernel is launched for FAST on the tiles of the image to compute the keypoints. After that, OpenCV's CUDA implementation of Gaussian Blur is used before computing descriptors for the keypoints by using the same asynchronous kernel tactic as before. In general, all OpenCV functions were replaced with OpenCV CUDA functions where possible. The outline of the this parallelized implementation is summarized below:

Extract ORB features (After Parallelization):

1. Pre-compute the scale pyramid
    a. Using OpenCV CUDA GpuMat methods
2. Compute keypoints
    a. For each pyramid level a CUDA asynchronous kernel is launched for FAST on tiles
3. OpenCV CUDA Gaussian Blur
4. Compute descriptors for keypoints
    a. For each pyramid level a CUDA asynchronous kernel is launched for computing descriptors for keypoints

3.2 UAV Platform

3.2.1 Requirements and Goals

The following list of requirements pertain to the UAV platform for this research:

Requirements:

1) Size: The UAV shall be able to fit through a standard window of 24” wide. [25]
2) Weight: As per FAA regulations, the all-up weight of the UAV shall be no more than 55 lbs. [26]
3) Lifting Capacity: The UAV shall be capable of hovering at no more than 70% throttle.
4) Flight Time: The UAV shall be capable of flying for at least 1 minute.

Requirement 1 is designed for the application scenario where a first responder using the UAV to map a floor of a building before the firefighters arrive. Requirement 2 is mandated by the FAA, and Requirement 3 defines a limit such that the UAV will have enough power to respond timely to input commands, both teleoperation commands, and flight controller commands. Lastly, Requirement 4 is a minimum flight time to showcase a functioning system. Each requirement also has an associated goal. The UAV platform will be deemed successful by meeting all requirements, but the following goals represent a more practical end system:

Associated List of Goals:

1) Size: The UAV will use a 250 mm racing quadcopter frame.
2) Weight: The all-up weight of the UAV will be no more than 1.5kg.
3) Lifting Capacity: The UAV will hover at ~50% throttle.
4) Flight: The UAV will be capable of flying for at least 3 minutes.

Goal 1 is challenging as a 250 mm racing quadcopter frame which is designed to have just enough space for teleoperation of an FPV UAV has limited spatial capacity. The hardware required for onboard Visual SLAM certainly takes up more space than this design, thus 3D printing was exploited to create custom fixtures to maximize space usage and minimize weight. If the hardware could not be mounted on such a small frame, a custom frame would be built, or larger frames would be investigated. Goal 2 was set with this alternative in mind and after an initial investigation of the predicted all-up weights of various sets of components. Goal 3 is a more desirable quality to have in a UAV than its associated Requirement 3. Hovering at 70% throttle will result in rather sluggish responses to input commands, where 50% guarantees optimal agility. Lower throttle percentages at hover than 50% can prove to be too responsive and unstable to input commands. Goal 4 is based on having enough time to gather data during test runs without requiring a battery recharge after every test.

Combinations of motors and propellers were researched with the requirements and goals stated above. The 250 mm racing quadcopter frame goal limited propeller selection to 6" and smaller. The 1.5kg all-up weight of the UAV and the hover at 50% throttle goals suggests that motor and propeller combinations should yield about 750g at full power per motor. Other selections to consider are battery size, type, and cell count, and electronic speed controllers (ESCs).

3.2.2 Hardware Selection

The NVidia Jetson TX2 board was selected as it contains an impressive 1.3TFLOPS NVidia Pascal GPU, 2GHz quad-core CPU, as well as a 2GHz dual-core NVidia Denver2 CPU with a power draw of only 7.5W. It is operated by a 64-bit Ubuntu Linux, and also has built-in Wifi. The Jetson board can come in a developer kit, or it can be used with a carrier board to interface the module with peripherals, An Orbitty Carrier board was chosen to interface with the Jetson TX2 module due to its 9V-19V input voltage regulation, 1x GbE, 1x USB 3.0, 1x USB 2.0, 1x HDMI, and 1x MicroSD, as well as others. This Orbitty Carrier board would allow the Jetson TX2 board to be power directly by a 3S LiPo battery.

3.2.2.1 Motor and Propeller Selection

With a 3S LiPo battery, a Turnigy D2826-6 2200kv Outrunner motor was selected. At 342W max power, 34A max current, and at 12.4V supplied by a 3S LiPo battery, the motor will be limited by max current at 27.5A. Test data with these motors and 6x4 propellers yields ~900g thrust at 27A. A plotted thrust vs current graph for these tests can be seen below in Figure 13.

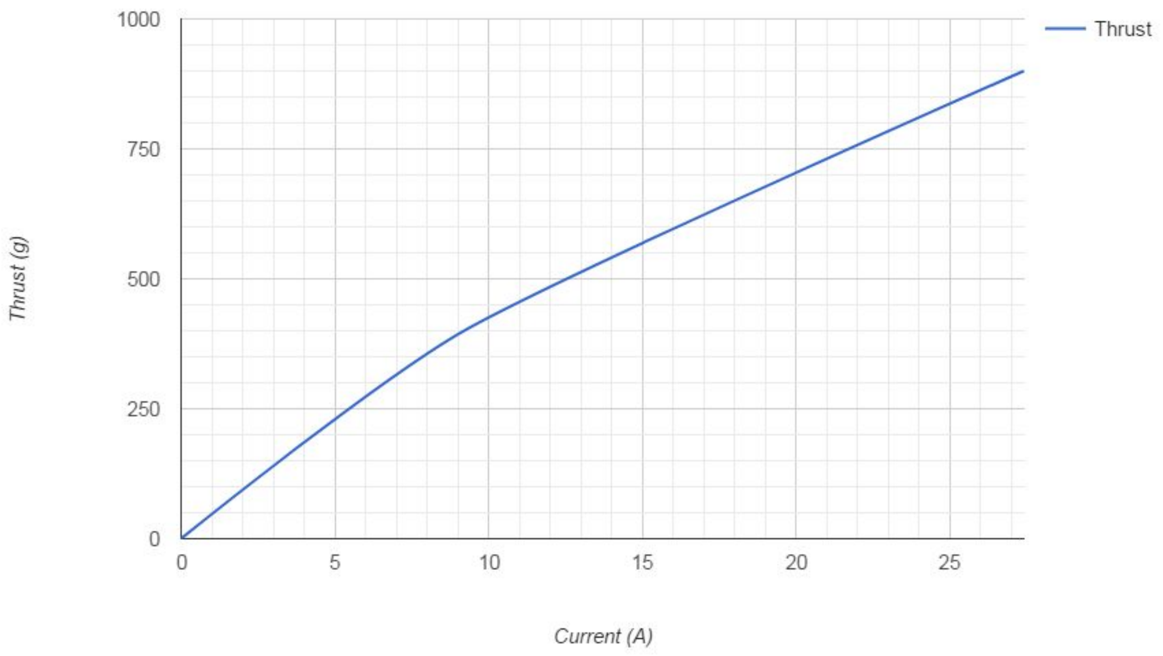


Figure 13: Estimated thrust vs current at 12.4V with 6x4 propellers. Using interpolated values based on test results with scale and power supply.

With these thrust and current values at 12.4V, and estimated flight time vs current per motor can be created, given a battery capacity. The selected battery for these estimations was a Turnigy Nano-tech 2200mAh 3S LiPo battery. The equation derived to predict flight time given current per motor and a battery capacity was:

$$FlightTime \ = \ 0.7 * 60 * ((BatteryCapacity/1000)/(4 * CurrentPerMotor + (7.5/12.4)))$$

With *FlightTime* being in minutes, *BatteryCapacity* in mAh, and *CurrentPerMotor* in Amps. The equation gives us the maximum flight time in hours imposed by the current drawn from the battery by the four quadcopter motors and the Jetson TX2 board. Note that the Jetson TX2 draws 7.5W and the battery is assumed to supply 12.4V. Flight time in hours is then converted to minutes and 70% is a safety factor for voltage cutoff. The estimated flight time vs current per motor is graph can be seen below in Figure 14.

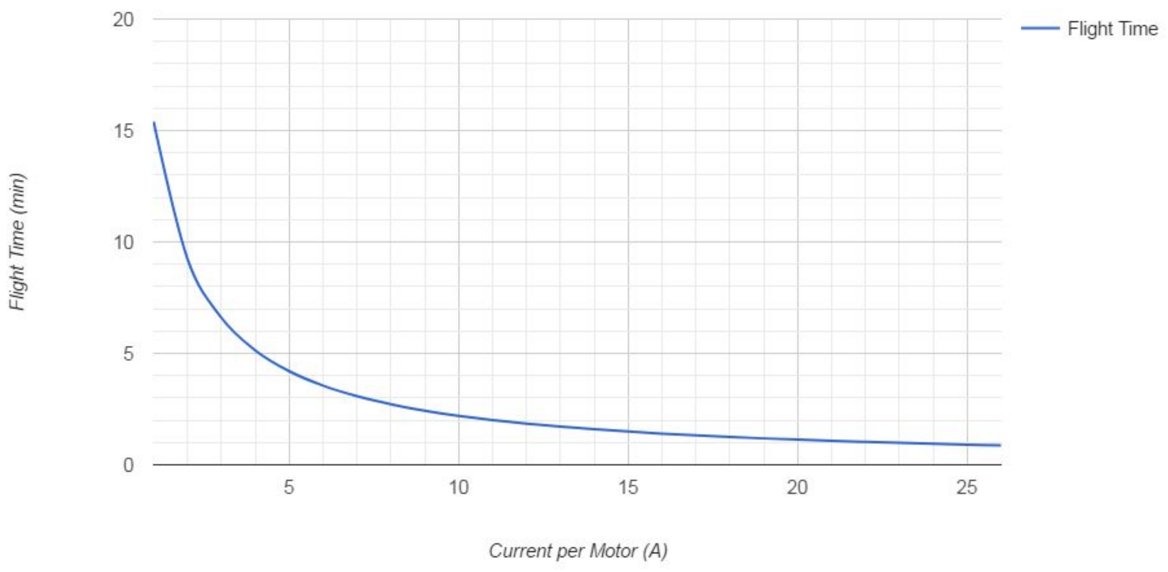


Figure 14: Predicted flight time vs current per motor at 12.4V with 6x4 propellers.

Lastly, Figure 15 below shows the estimated flight time vs all-up weight of the UAV. This can now be generated given the flight time and total thrust values.

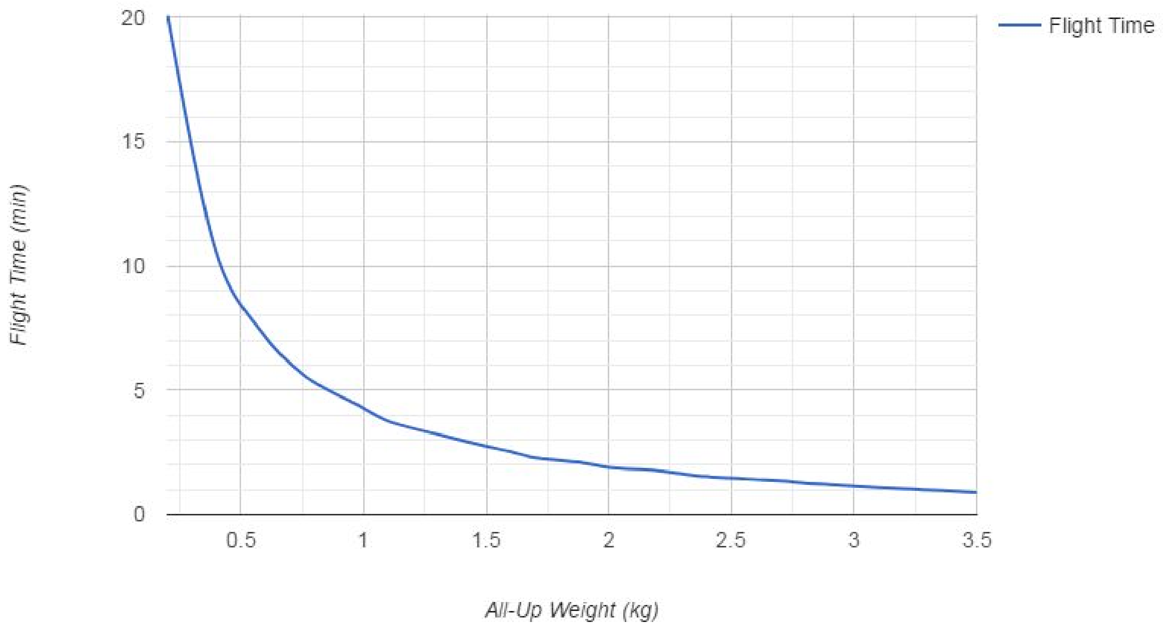


Figure 15: Predicted flight time vs all-up weight of UAV.

3.2.2.2 Teleoperation of UAV

A Naze32 Rev 6 10DoF Flight Controller was selected as the flight controller for the UAV. Naze32 flight controllers are commonly used in teleoperated racing quadcopters. The Jetson TX2 is the main board on the UAV and teleoperation commands will be received via Wifi. These input commands will then be transmitted via USB to a Switch Science mbed LPC824. The mbed converts the teleoperation commands to PWM signals that connect to the Naze32 flight controller. The flight controller then sends PWM signals to the ESCs which control the motors.

The mbed also reads from a downward-facing MB1040 LV-MaxSonar-EZ4 Range Finder. With one-inch accuracy up to 20ft, this distance reading is used within the mbed logic as a simple altitude-hold controller. The Jetson TX2 is capable of being powered directly by the 3S LiPo through the carrier board, but a power distribution board (PDB) is used to power the different components. 5V is supplied to the flight controller, 5V to the ultrasonic sensor, and the ESCs and the Jetson board are powered by the terminal pads.

3.2.3 UAV Software System

A password-encrypted hidden Wireless Access Point was set up on the Jetson TX2 board using Hostapd and Dnsmasq. SSH key authentication was also set up and is required for SSH access to the board. For teleoperation, an Xbox controller was used, and a Linux joystick python program was modified to read from the XBox controller [39]. An SSL socket program between a laptop, with the Xbox controller, and the Jetson board was implemented in Python for teleoperation. The SSL socket program utilized two-way authentication with self-signed certificates for security.

4. Experiments and Results

4.1 Visual SLAM Software

When comparing framerate averages on the TUM RGB-D image sequences over twenty runs, the CUDA-accelerated ORB-SLAM implementation with OpenCV 3.2 showed between 26% and 55% improvement in speed versus ORB-SLAM, without CUDA, with OpenCV 2.4 and OpenCV3.2. The average was a ~33% speed improvement. These results are shown below in Figure 16.

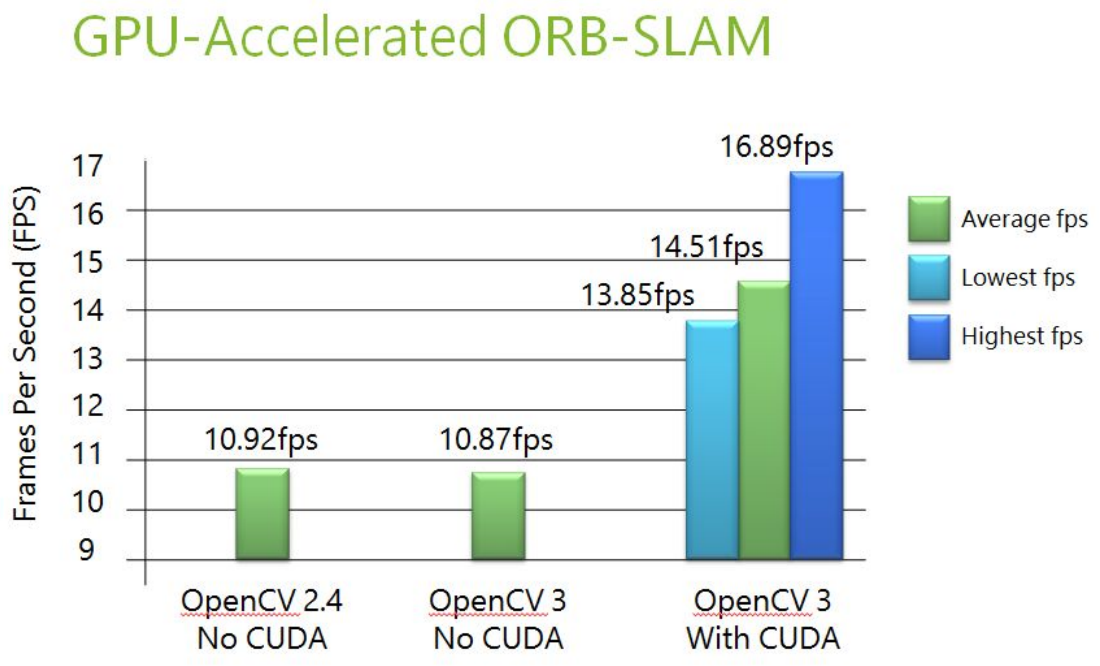


Figure 16: FPS improvement with CUDA parallelization.

The NVidia Visual Profiler was used to analyze the parallelized ORB-SLAM implementation. The final thread analysis is shown below in Figure 17. The figure shows that the GPU is being utilized during the ORBextractor portion of code, an apparent bottleneck in the Tracking thread. It also shows small amounts of parallelized CPU alongside GPU computation resulting from launched asynchronous CUDA kernels.

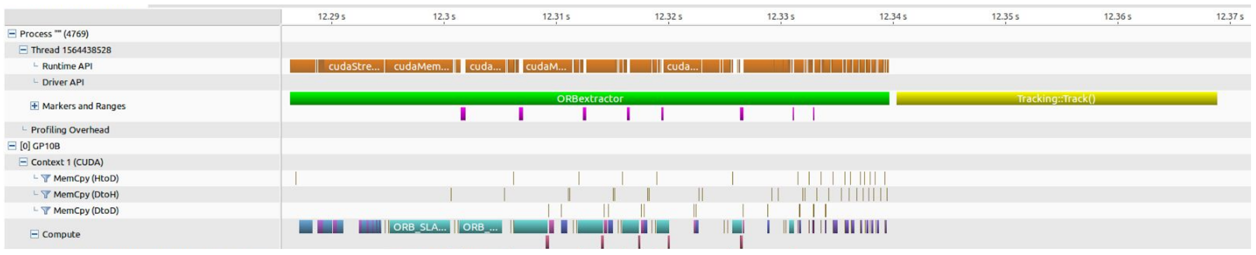


Figure 17: NVidia Visual Profiler thread analysis of CUDA-accelerated ORB-SLAM.

Table 1 below shows the translational RMSE between the estimated trajectory by ORB-SLAM and the ground-truth trajectory. The RMSE is averaged across 5 runs for popular image sequences in the TUM RGB-D dataset. Column 1 shows the name of the image sequence in the dataset, and column 2 shows the best published results for ORB-SLAM with the image sequence. Column 3 shows the average translational RMSE achieved with default ORB-SLAM parameters and no estimated scale factor for the unmodified ORB-SLAM (without CUDA). Next column (4) shows the averaged translational RMSE achieved with default ORB-SLAM parameters and no estimated scale factor for ORB-SLAM with CUDA accelerations. These two columns verify that the CUDA-accelerated ORB-SLAM is comparable in accuracy to the unmodified ORB-SLAM without CUDA accelerations. These two columns also show the variability in the estimated trajectory across runs. This is due to monocular SLAM not being able to determine map scale. Also, subtle differences in the duration of keyframe generations can have large effects on the output trajectory. Column 5 shows the averaged translational RMSE achieved with default ORB-SLAM parameters and an estimated scale for ORB-SLAM with CUDA accelerations for the fr2/desk image sequence of the TUM RGB-D dataset.

	Best Published [8]	No CUDA, No Scale Estimate	CUDA, No Scale Estimate	CUDA, With Scale Estimate
fr1/desk	0.016	0.083	0.058	N/A
fr1/desk2	0.022	0.266	0.306	N/A
fr1/room	0.047	0.204	0.233	N/A
fr2/desk	0.009	0.885	0.836	0.009
fr2/xyz	0.004	0.136	0.124	N/A
fr3/office	0.010	1.212	1.239	N/A

Table 1: TUM RGB-D Dataset. Comparison of Translation RMSE (m).

Figure 18a shows the unscaled estimated trajectory vs ground truth trajectory for the fr2/desk image sequence run with CUDA-accelerated ORB-SLAM. The averaged translational RMSE was 0.797m across 5 runs. Figure 18b shows the scaled estimated trajectory versus ground truth trajectory of the same run. When scaled by 1.887, the translational RMSE becomes 0.009m. This matches the best published results and verify the accuracy of the CUDA augmented ORB-SLAM. The scale was chosen to minimize the translational RMSE error across the 5 runs.

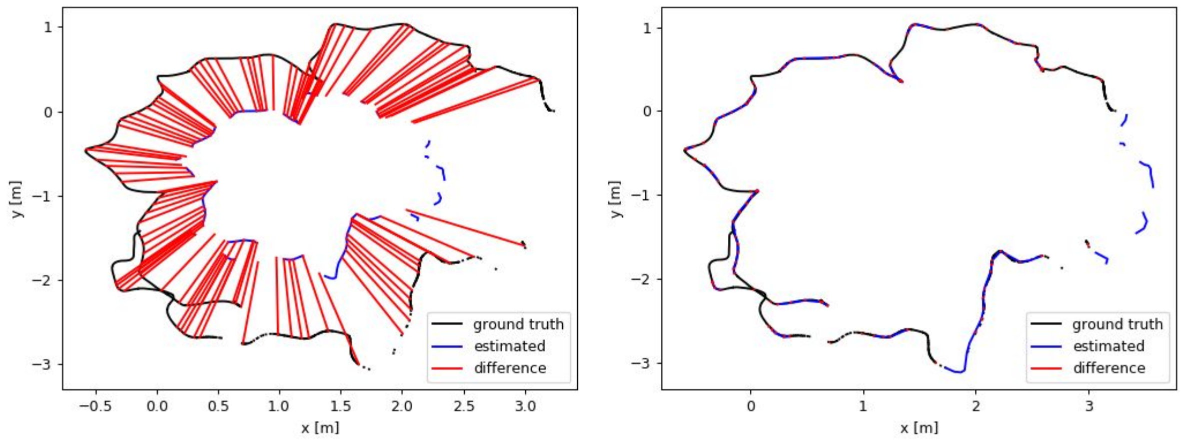


Figure 18a-b: Trajectory results of CUDA-accelerated ORB-SLAM. Left: Unscaled estimated trajectory. Right: Scaled estimated trajectory.

4.2 UAV Platform

The final construction of the UAV can be seen below in Figure 19. All requirements were met as listed in Table 2 below. The final UAV design used was a 250 mm racing quadcopter frame and was 12.5” long and 14” wide including propellers fully extended for both measurements. It weighed 1 kg and was capable of hovering at ~33% throttle. As the drone behaved too aggressive to controller input at 50% throttle, the requirement of hovering at 50% was dropped for a lower value. Adding a larger battery such as a 4000mAh LiPo would add more weight and work towards getting closer to the 50% throttle goal and would also allow for longer flight times. The UAV was capable of flying for more than 3 minutes with minimal load, but it was not tested beyond this weight due to lack of replacement parts.

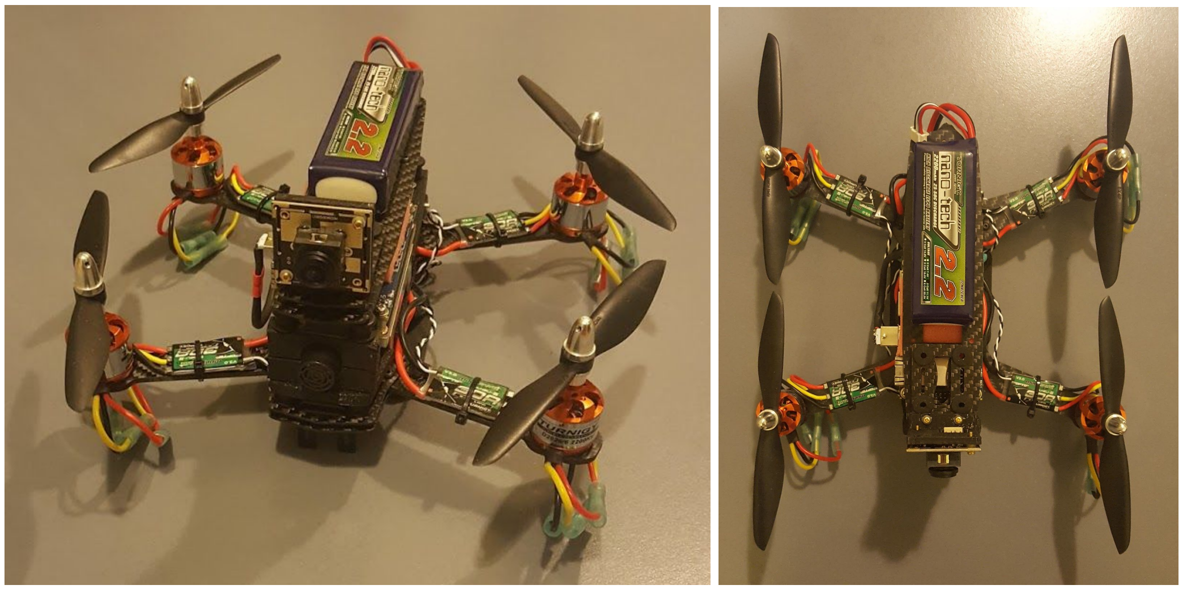


Figure 19: Constructed final UAV.

	Size:	Weight:	Payload and Lifting Capacity:	Flight Time:
Requirement:	Shall be less than 24 inches wide	Shall be less than 25 kg (55 lbs)	Shall hover at not more than 70% throttle	Shall be greater than 1 minute
Goal:	Use a 250 mm (9.84 inch diagonal) racing quadcopter frame	1.5 kg	Hover at 50% throttle	3 minutes
Actual:	250mm racing quadcopter frame 12.5 inches long and 14 inches wide with propellers	1 kg	Hover at ~33% throttle	Greater than 3 minutes (5 minutes predicted flight time)

Table 2: Results of UAV design.

3D printing was utilized for much of the custom fixtures and mounts for the UAV. A custom camera module holder, Jetson board fixture, vibrationally isolated mount for the flight controller using vibration dampeners, and feet for the UAV were are created with 3D printing. The 3D printed parts assisted in being able to fit all the hardware into the small frame while providing adequate strength with a negligible addition to weight.

5. Conclusion and Future Work

Overall, using GPU-acceleration with CUDA on the ORB-SLAM algorithm showed a speed increase of ~33% on average, Along with the improved performance, this significant boost is important to overcome motion blur, a main cause of poor tracking quality that blurs features and is prevalent in systems using rolling shutter cameras. Further optimization of ORB-SLAM algorithm will reduce motion blur even further. As shown in the results, this speed increase comes with no accuracy loss.

Future parallelization of ORB-SLAM would require restructuring of the original code. While ORB-SLAM was designed to be scaleable, it was not designed with future parallelization in mind. Restructuring and parallelizing the code would allow for even greater speed-up. However, there are limitations in reality that must be faced. Based on Amdahl's law, there are limits on performance gains with parallelization as the speedup will always be limited by serial portions of code [40]. ORB-SLAM cannot completely be parallelized, as is the case with most software, though large sections of it could still be parallelized. The Jetson TX2 is limited to single-thread analysis with NVidia profiling software, where further thread analysis could reveal the next algorithm segments to parallelize.

Works Cited

1) Scaramuzza, Davide, and Michael Blösch. "Visual SLAM—An Overview."
2) Engel, Jakob, Thomas Schöps, and Daniel Cremers. "LSD-SLAM: Large-scale direct monocular SLAM." European Conference on Computer Vision. Springer International Publishing, 2014.
3) Engel, Jakob, Jurgen Sturm, and Daniel Cremers. "Semi-dense visual odometry for a monocular camera." Proceedings of the IEEE international conference on computer vision. 2013.
4) Engel, Jakob, Jörg Stückler, and Daniel Cremers. "Large-scale direct SLAM with stereo cameras." Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on. IEEE, 2015.
5) Engel, Jakob, Vladlen Koltun, and Daniel Cremers. "Direct sparse odometry." arXiv preprint arXiv:1607.02565 (2016).
6) Engel, Jakob, Vladyslav Usenko, and Daniel Cremers. "A photometrically calibrated benchmark for monocular visual odometry." arXiv preprint arXiv:1607.02555 (2016).
7) Klein, Georg, and David Murray. "Parallel tracking and mapping for small AR workspaces." Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on. IEEE, 2007.
8) Mur-Artal, Raul, J. M. M. Montiel, and Juan D. Tardós. "Orb-slam: a versatile and accurate monocular slam system." IEEE Transactions on Robotics 31.5 (2015): 1147-1163.
9) Newcombe, Richard. Dense visual SLAM. Diss. Imperial College London, 2012.
10) von Stumberg, Lukas, et al. "Autonomous Exploration with a Low-Cost Quadrocopter using Semi-Dense Monocular SLAM." arXiv preprint arXiv:1609.07835 (2016).
11) Engel, Jakob, Jürgen Sturm, and Daniel Cremers. "Camera-based navigation of a low-cost quadrocopter." 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 2012.

12) G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. g2o: A general framework for graph optimization. In IEEE International Conference on Robotics and Automation, 2011.
13) Brief Review on Visual SLAM: A Historical Perspective. N.p., n.d. Web. 01 June 2017.
14) Klein, Georg. "Parallel Tracking and Mapping for Small AR Workspaces - Source Code" *University of Oxford.* University of Oxford, Feb 2014. Web. 5/1/2017.
15) Yunchih. "ORB-SLAM2 GPU Optimization" *Github.* n.p., 2016. Web. 4/5/2017.
16) Riisgaard, Søren. Blas , Morten . *A Tutorial Approach to Simultaneous Localization and Mapping* . Boston: MIT, 2005. Print.
17) *ORB-SLAM2: An Open-Source SLAME for Monocular, Stereo and RGB-D Cameras.*Raul Mur-Artal and Juan D. Tardos. 21 Oct. 2016. Web. 1 Feb. 2017
18) *LSD-SLAM: Large-Scale Direct Monocular SLAM.* J. Engel, T. Schöps, D. Cremers. 6 Jul. 2014. Web. 5 Feb. 2017.
19) *DSO: Direct Sparse Odometry.* J. Engel, T. Schöps, D. Cremers. 13 Jul. 2014. Web. 5 Feb. 2017.
20) Zheng, F. *Brief Review on Visual SLAM: A Historical Presepective.* N.p., 20 May 2016. Web 5 Mar. 2017
21) Mur-Artal, Rual. *Real-Time SLAM for Monocular, Stereo and RGB-D Cameras, with Loop Detection and Relocalization Capabilities*, Github. 23 Jan, 2016.
22) Mujtaba, Hassan. *NVidia and Microsoft Announce The Tesla P100 based HGX-1 Hyperscale GPU Accelerator To Drive AI Cloud Computing – Jetson TX2 Launched Too*, 8 Mar. 2017. Web 4 Apr. 2017.
23) NVidia, *What is GPU Computing*, 2017. Web. 3 Mar. 2017
24) Larabel, Michael. *Raspberry Pi 3 Benchmarks vs. Eight Other ARM Linux Boards.* 5 Mar. 2016. Web. 15 Mar. 2017.
25) Modernize. *Standard Window Sizes.* 2017. Web. 15 Apr. 2017.
26) Federal Aviation Administration, *Unmanned Aircraft Systems – Getting Started*. FAA. 2017. Web 28 May 2017.

27) Rublee, E. Rabaud, V. Konolige, K. Bradski, G. *ORB: an efficient alternative to SIFT or SURF.* Willow Garage. 2011.

28) OpenSlam. *What is SLAM?*. 2017. Web. 1 Jun. 2017.

29) Rissgaard, S. Rufus-Blas, M. *SLAM for Dummies: A tutorial Approach to Simultaneous Localization and Mapping.* MIT. 2005.

30) *UAV Applications.* Ascending Technologies. 2017. Web. 1 Jun. 2017

31) Lategahn, H. Geiger, A. Kitt, B. *Visual SLAM for autonomous ground vehicles.* IEEE International Conference on Robotics and Automation, Shanghai, 2011. p.g. 1732-1737.

32) *Loop Closure Detection.* Robotics and Computer Vision Laboratory, ENSTA Paris Tech Universite. 2017. 26 May 2017.

33) NVidia, *Accelerated Computing*, 2017. Web. 3 Mar. 2017

34) NVidia, *What's the Difference Between a CPU and a GPU?*, 2017. Web. 3 Mar. 2017

35) NVidia, *What is CUDA?*, 2017. Web. 3 Mar. 2017

36) Oosten, Jeremiah. *CUDA Thread Execution Model.* 3D Game Engine Programming. 15 Nov. 2011. Web 5 Mar. 2017

37) NVidia, *NVidia Jetson TX2 Delivers Twice the Intelligence to the Edge*, 2017. Web. 4 Mar. 2017

38) Mur-Artal, Rual. *Failed to Build ORB-SLAM on Ubuntu 16.04, Issue 202, ORB-SLAM2*, Github. 23 Nov, 2016.

39) Rdb, *Access joysticks/game controllers from Python in Linux via the joystick driver-js_linux.py.* GithubGist. 2014.

40) Gillespie, Matthew. *Amdahl's Law, Gustafson's Trend and the Performance Limtis of Parallel Application*s. Intel. 1 Jan. 2015. Web. 26 Dec 2016.