International Journal of Advance Research in Science and Engineering
Volume No.07, Special Issue No. (02), January 2018
www.ijarse.com
IJARSE
ISSN: 2319-8354

Compuer Unified Device Architecture (CUDA)-Accelerated Visual SLAM for UAVs

Er.Gajendra Singh[1], Bharat Singh Hada [2] Insiya Bohra[3]

[1,2,3] *Department of Computer Science and Engineering,*

Vedant College of Engineering and Technology, Rajasthan Technical University (India)

ABSTRACT

CUDA is a Compute Unified Device Architecture and parallel computing platform and application programming interface (API) model created by Nvidia.It allows softwaredevelopers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

The CUDA platform is designed to work with programming languages such as C, C++, and FORTRAN. This accessibility makes it easier for specialists in parallel programming to use GPU resources, in contrast to prior APIs like Direct3D and OpenGL, which required advanced skills in graphics programming. Also, CUDA supports programming frameworks such as OpenACC and OpenCL. When it was first introduced by Nvidia, the name CUDA was an acronym for Compute Unified Device Architecture, but Nvidia subsequently dropped the use of the acronym.

Keywords: C, C++, FORTRAN,GPU, SLAM, UAV.

I. 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). 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. parallelized algorithm will be then by implemented on a small UAV.

II. OVERVIEW

2.1 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

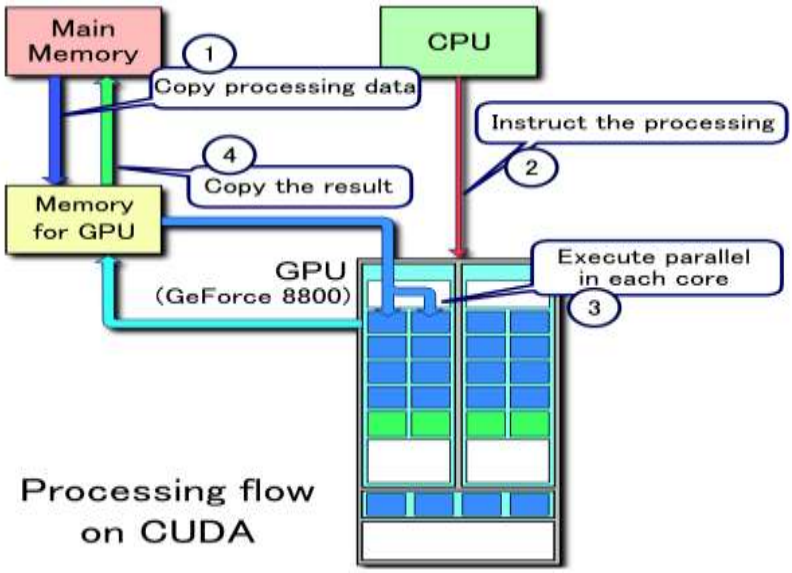

Fig1:- Processing flow on CUDA.

2.2 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

International Journal of Advance Research in Science and Engineering
Volume No.07, Special Issue No. (02), January 2018
www.ijarse.com
IJARSE
ISSN: 2319-8354

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 of SLAM pose estimation for a robot navigating an environment.

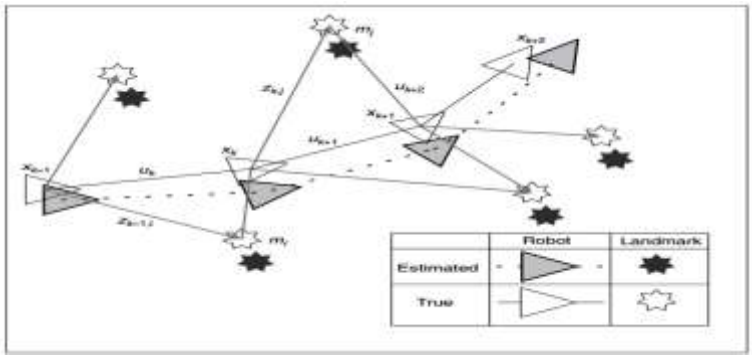

Fig 2:- The estimation problem of SLAM

2.3 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. 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.

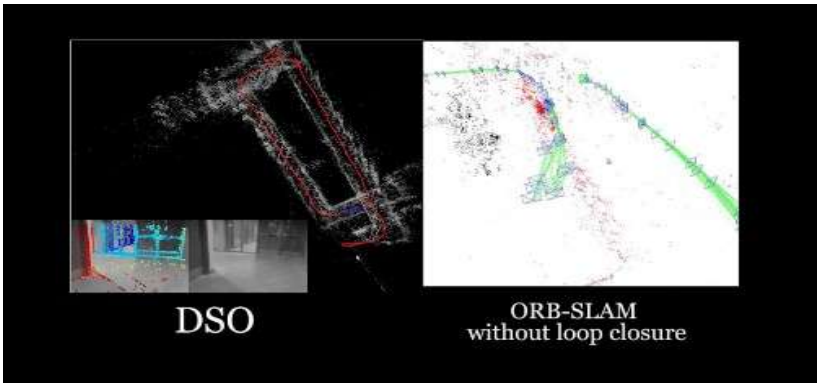

Fig3:- ORB-SLAM sparse reconstruction

International Journal of Advance Research in Science and Engineering
Volume No.07, Special Issue No. (02), January 2018
www.ijarse.com
IJARSE
ISSN: 2319-8354

2.4 GPU-Acceleration and Parallel

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 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.

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.

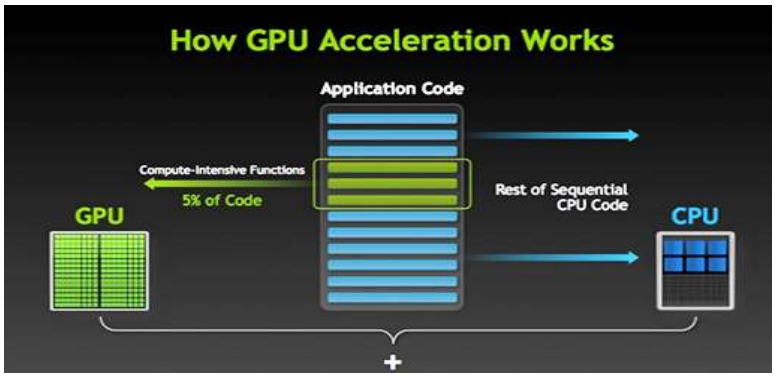

Fig4:- How GPU Acceleration works by NVidia

One of the major developers parallel computing platforms is NVidia. NVidia developed it's own parallel computing platform API called CUDA.

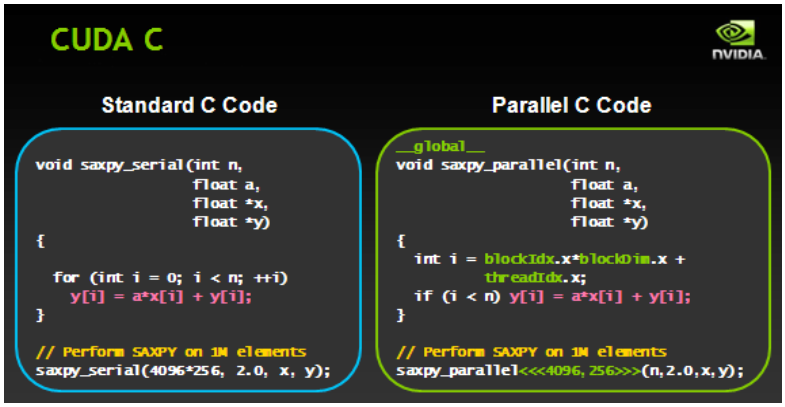

Fig5:- example of parallelization of serial code

```
texture<float, 2, cudaReadModeElementType> tex;
void foo()
{
  cudaArray* cu_array;
  // Allocate array
  cudaChannelFormatDesc description = cudaCreateChannelDesc<float>();
```

International Journal of Advance Research in Science and Engineering
Volume No.07, Special Issue No. (02), January 2018
www.ijarse.com
IJARSE
ISSN: 2319-8354

```
 cudaMallocArray(&cu_array, &description, width, height);
 // Copy image data to array
 cudaMemcpyToArray(cu_array, image, width*height*sizeof(float), cudaMemcpyHostToDevice);
 // Set texture parameters (default)
 tex.addressMode[0] = cudaAddressModeClamp;
 tex.addressMode[1] = cudaAddressModeClamp;
 tex.filterMode = cudaFilterModePoint;
 tex.normalized = false; // do not normalize coordinates

 // Bind the array to the texture
 cudaBindTextureToArray(tex, cu_array);
 // Run kernel
 dim3 blockDim(16, 16, 1);
 dim3 gridDim((width + blockDim.x - 1)/ blockDim.x, (height + blockDim.y - 1) / blockDim.y, 1);
 kernel<<< gridDim, blockDim, 0 >>>(d_data, height, width);
 // Unbind the array from the texture
 cudaUnbindTexture(tex);
} //end foo()
__global__ void kernel(float* odata, int height, int width)
{
  unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
  unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
  if (x < width && y < height) {
    float c = tex2D(tex, x, y);
    odata[y*width+x] = c;
  }
}
```

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.

International Journal of Advance Research in Science and Engineering
Volume No.07, Special Issue No. (02), January 2018
www.ijarse.com
IJARSE
ISSN: 2319-8354

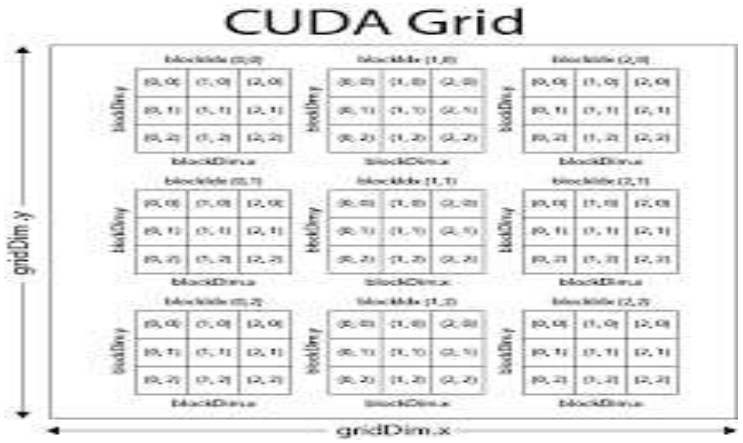

Fig6:- CUDA block and thread diagram.

III. 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

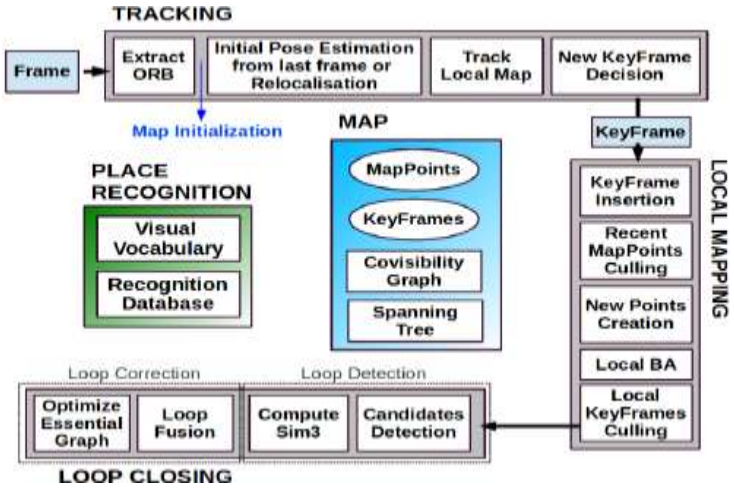

Fig7:- Threads & modules of ORB-SLAM algorithm.

3.1 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.

3.2 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

IV. 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:

4.1 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.

International Journal of Advance Research in Science and Engineering
Volume No.07, Special Issue No. (02), January 2018
www.ijarse.com
IJARSE
ISSN: 2319-8354

V. UAV PLATFORM

5.1 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:

5. 2 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.

requiring a battery recharge after every test.

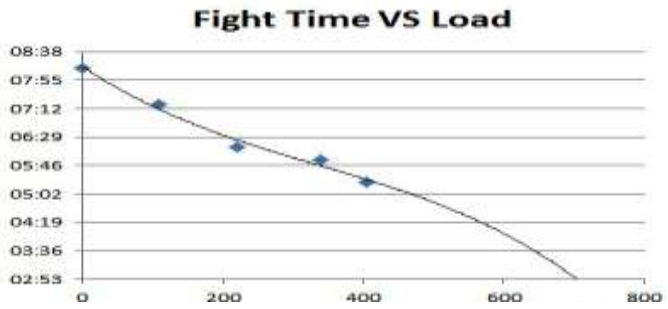

Fig8:- Predicted flight time vs current per motor

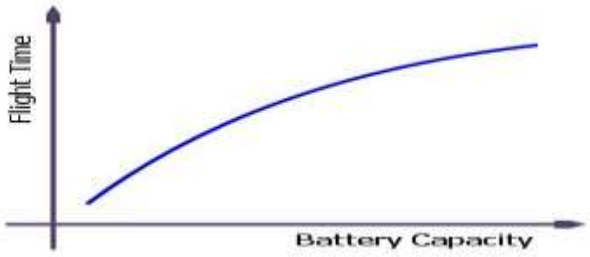

Fig9:- Predicted flight time vs all-up weight of UAV.

International Journal of Advance Research in Science and Engineering
Volume No.07, Special Issue No. (02), January 2018
www.ijarse.com
IJARSE
ISSN: 2319-8354

VI. EXPERIMENTS AND RESULTS

6.1 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.

Fig10:- 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.

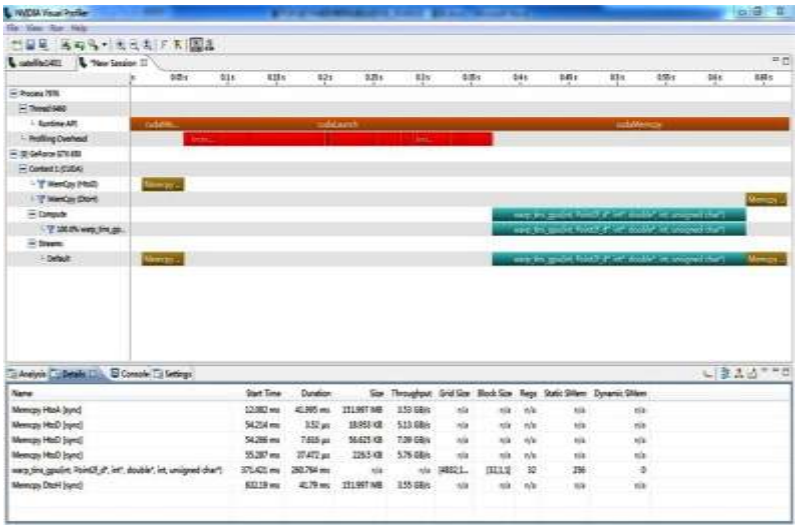

Fig11:- NVidia Visual Profiler thread analysis of CUDA-accelerated ORB-SLAM

6.2 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.

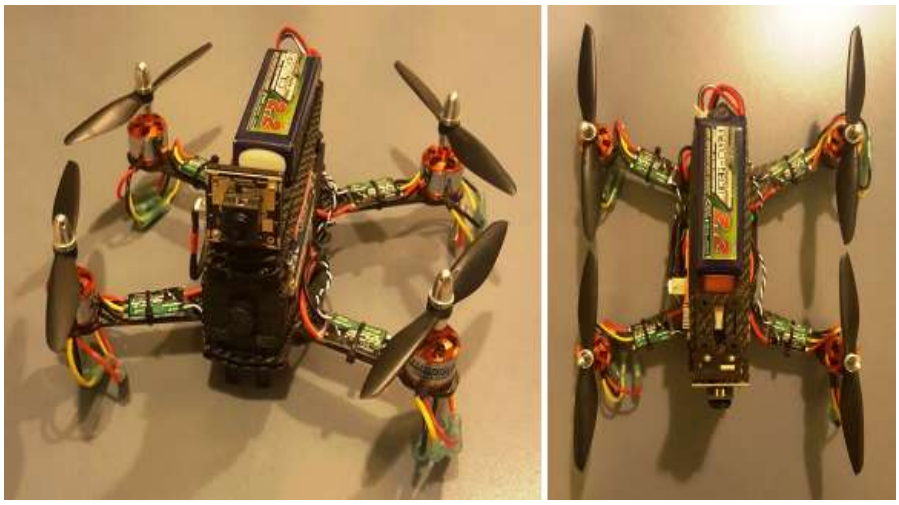

Fig12:- Constructed final UAV

VII. CONCLUSION

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.

REFRENCES

[1] L. De Giovanni and F. Pezzella, “An improved genetic algorithm for the distributed and flexible jobshop scheduling problem”, European journal of operational research, vol. 200, no. 2, (2010), pp. 395- 408.

[2] J. H. Holland, “Genetic algorithms and the optimal allocation of trials”, SIAM Journal on Computing, vol. 2, no. 2, (1973), pp. 88-105.

[3] K. Ghoseiri and S. F. Ghannadpour, “Multi-objective vehicle routing problem with time windows using goal programming and genetic algorithm”, Applied Soft Computing, vol. 10, no. 4, (2010), pp. 1096-1107.

[4] K. P. Murphy, “Machine learning: a probabilistic perspective”, MIT press, (2012).

[5] H. Xing and R. Qu, “A compact genetic algorithm for the network coding based resource minimization problem”, Applied Intelligence, vol. 36, no. 4, (2012), pp. 809-823.

[6] S. Yang, H. Cheng and F. Wang, "Genetic algorithms with immigrants and memory schemes for dynamic shortest path routing problems in mobile ad hoc networks", Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, vol. 40, no. 1, (2010), pp. 52-63.

[7] A. Prakash, F. T. Chan and S. Deshmukh, "Fms scheduling with knowledge based genetic algorithm approach", Expert Systems with Applications, vol. 38, no. 4, (2011), pp. 3161-3171.

[8] J. Adams, D. L. Woodard, G. Dozier, P. Miller, K. Bryant and G. Glenn, "Genetic-based type ii feature extraction for periocular biometric recognition: Less is more", Pattern Recognition (ICPR), 2010 20th International Conference on. IEEE, (2010), pp. 205-208.

[9] A. Quteishat, C. P. Lim and K. S. Tan, "A modified fuzzy min–max neural network with a geneticalgorithm-based rule extractor for pattern classification", Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions, vol. 40, no. 3, (2010), pp. 641-650.

[10] Y. Zhang and L. Wu, "Artificial bee colony for two dimensional protein folding," Advances in Electrical Engineering Systems, vol. 1, no. 1, (2012), pp. 19-23.