Florida Tech IGVC

Software Design

Intelligent Ground Vehicle Competition (FIT-IGVC)

Members:

Brent Allard ballard2014@my.fit.edu

Adam Hill ahill2013@my.fit.edu

Chris Kocsis ckocsis2007@my.fit.edu

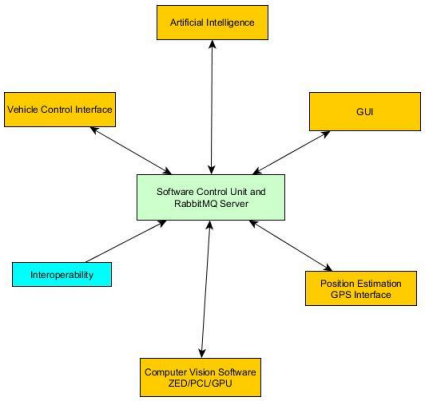
William Nyffenegger wnyffenegger2013@my.fit.edu

Faculty Sponsor:

Dr. Marius Silaghi msilaghi@fit.edu

Contents

Introduction



The Intelligent Ground Vehicle Competition (IGVC) is a challenge hosted by Oakland University. The competition promotes research and development on the problem of navigating unknown spaces autonomously. The goal of the competition is to complete an off-road course on grass. A team's vehicle must follow roads while intelligently avoiding obstacles using a variety of sensors. In addition to navigating through roads, the robot must find goals in free space and prioritize directions based on constraints.

Florida Tech (FIT) is partnering with Florida State University (FSU) to produce a vehicle capable of winning the competition. Concerning software, Florida Tech is tasked with implementing pathfinding and motion planning, image processing, obstacle detection, inter-process communication, a control unit, and a suite for satisfying SAE's JAUS standard. FSU is implementing a way to determine position with high accuracy and precision, as well as motor control and correction. FSU is also fabricating the robot.

For the purpose of this design document, we will only discuss the design for components of the software that Florida Tech is responsible for.

Overall Design Goals

To support the desired functionality, several overarching computer science concepts must be discussed:

- Concurrency
- GPU Programming
- Inter-process communication
- Software testing
- Modularity

Overall Software Architecture and Control

List of subsystems:

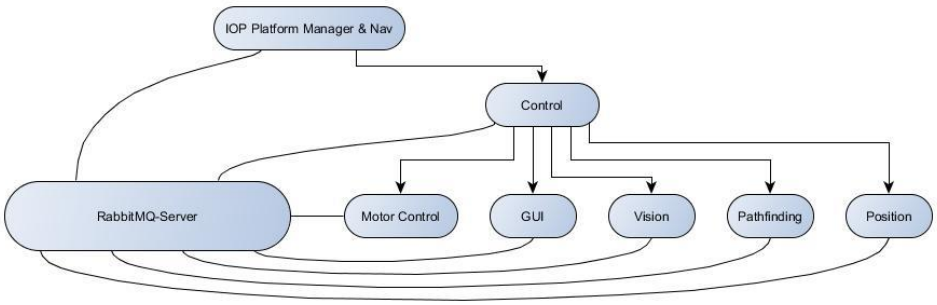
- Control: Modify the state of subsystems and log the robot's behavior
- Interoperability Platform Manager & Navigation: Interact with outside entities for testing, simulation, and benchmarking
- Pathfinding: Conduct motion planning and pathfinding based on position and obstacle input
- Position: Calculate the vehicle's velocity, acceleration, heading, and location
- Vision: Identify roads and obstacles viewable by the robot's sensors
- GUI: Display the robot's performance
- Motor Control: Actualize motion planning by executing commands on motors

The software system being created is complex and detailed. The detail required to discuss each class and module is too extensive for this document. However, when the code base has become more defined we will post a user manual, as well as Doxygen based documentation.

There exist three important aspects of the system: control flow, information flow, and subsystem behavior. Combined, these aspects feed into the control subsystem and communication framework.

One of the overarching goals is to maintain the independence of each subsystem in the software. Remote procedure calls may cause deadlocks, which are difficult to debug. Instead, the design focuses on the challenges. The ability of an outside entity to control the robot is paramount; however, having a software piece that responds to UDP requests and is actively trying to control the robot is complex. Therefore, separating interoperability from direct control of the robot is beneficial. With RabbitMQ, this separation only requires the creation of an additional publisher and subscriber. Additionally, separating control and interoperability allows the control flow to be maintained when an outside entity is not connected.

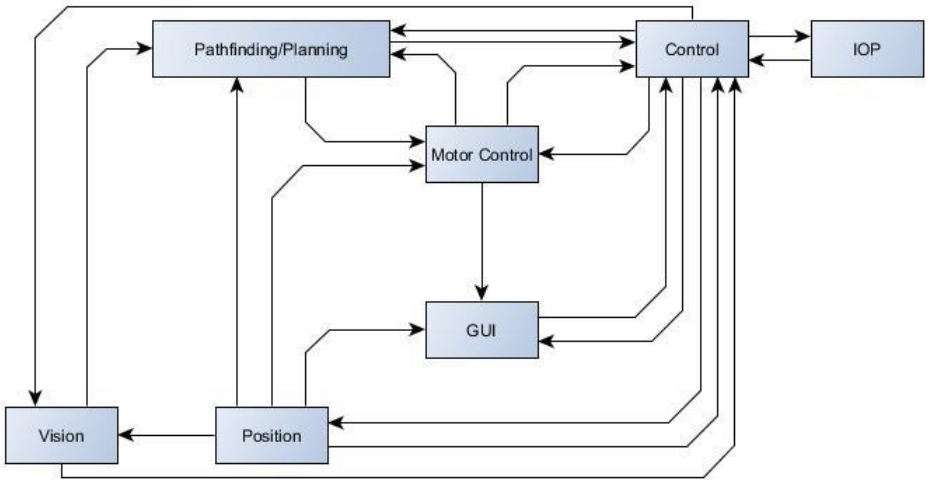
Control Flow



The control unit communicates with subordinate subsystems through RabbitMQ. Control mainly logs the behavior of the subsystems and interacts with subsystems on request from interoperability or to recover a subsystem.

Although control may not flow freely, information must. Several subsystems are primarily providers of information. The critical need for time-sensitive information forces Pathfinding and Motor Control to require information from Vision and Position, though the information itself is not critical. The GUI will also receive time-sensitive information, but only when that information is sent to another subsystem.

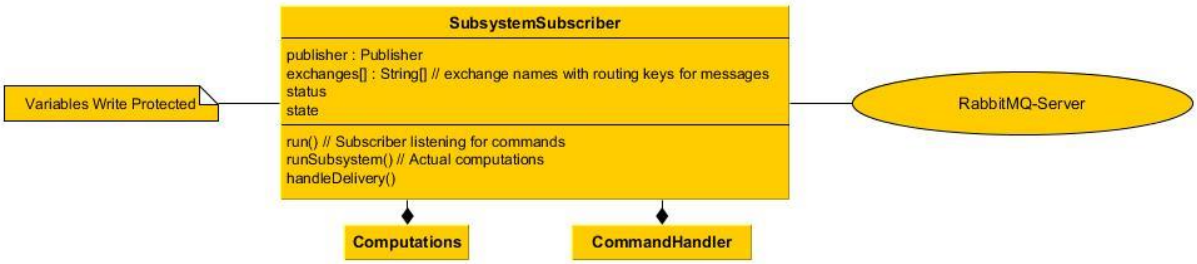
Information Flow



The main subsystems that need information are Pathfinding, the GUI, and Motor Control. Vision and Position are primarily providers; however, Position may need to provide information to both Motor Control, Vision, and Pathfinding on command. If Pathfinding runs fast enough, then the need for Position to report to Pathfinding and Motor Control diminishes slightly.

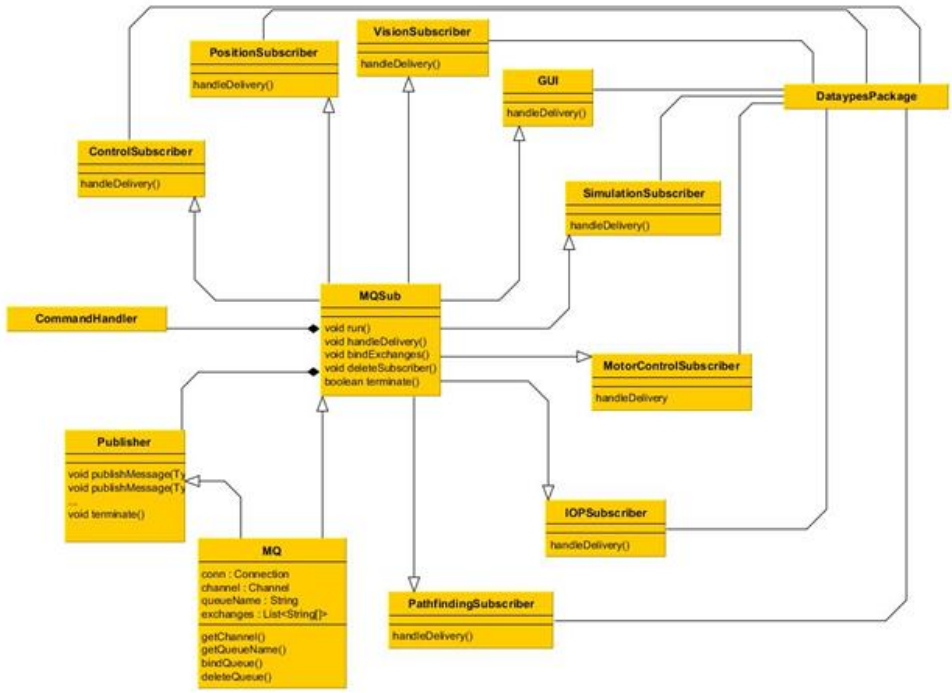
Each individual subsystem will be constructed around a subscriber which listens for information from RabbitMQ. Focusing on the subscriber will allow the system to maintain independent execution and listening. Each subscriber will contain a publisher that the subsystem may use to respond to requests. Additional publishers within each subsystem may be created without limiting the performance of the communication framework.

The particular structure of each subsystem is unique; however, the general structure is depicted below:



Each subsystem needs to parse all commands it receives. Parsing commands only needs to be implemented once per language in use. How those commands are handled after parsing depends on the subscriber.

Control Software and Communication Framework



RabbitMQ messaging server is the core of our communication framework, and will be under the direct control of the central control unit. RabbitMQ provides these features:

- Non-blocking message sending
- A standardized interface
- Language independence
- High throughput
- Support for heartbeats

The fact that RabbitMQ is non-blocking will allow the hardware driven components, such as computer vision, to be a simple producer of information; just passing the data along to the pathfinding subsystem instead of being concerned about messaging, timeouts and other more complicated IPC (Inter-process communication) mechanisms. This will also allow the pathfinding subsystem to process the information so that it can prioritize pieces of importance. This means that significant computation time taken for pathfinding does not have to be interrupted waiting on synchronized updates from, or to the other parts of the system.

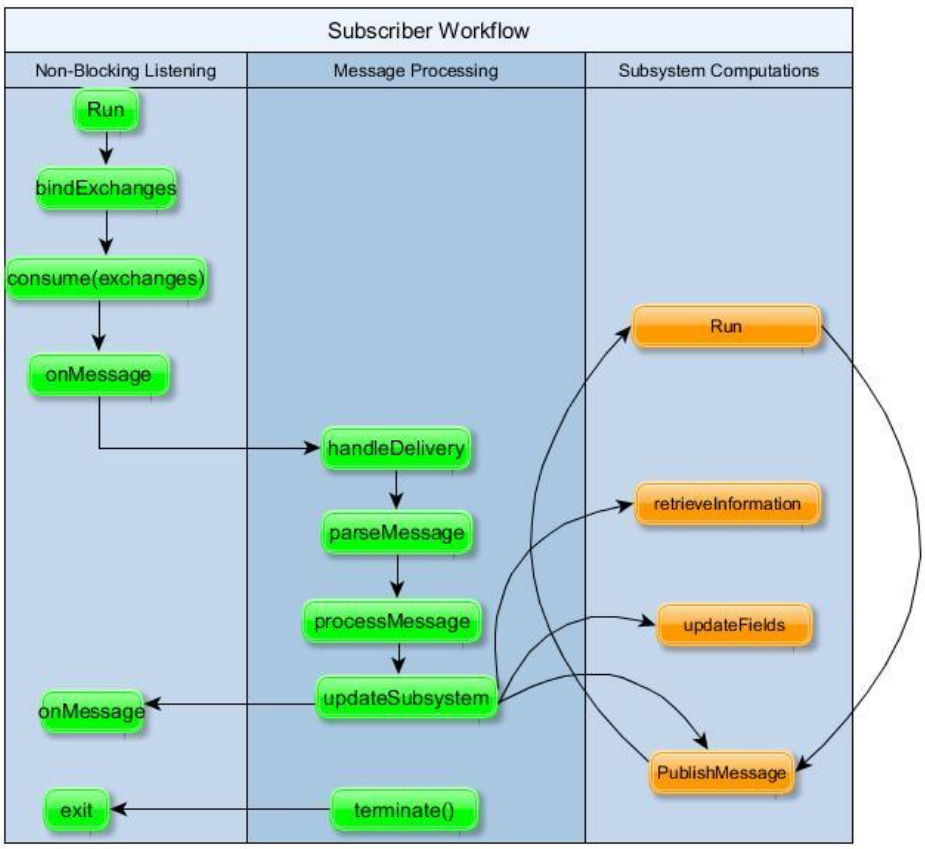
RabbitMQ Server has clients implemented in many languages, including Java and C++. The callback structure of RabbitMQ is preserved in all languages. In C++, additional tools, like

Boost's Asio library and ev.h (event handler with callbacks), must be used to support those callbacks.

Ultimately, every subscriber implements a common set of methods.

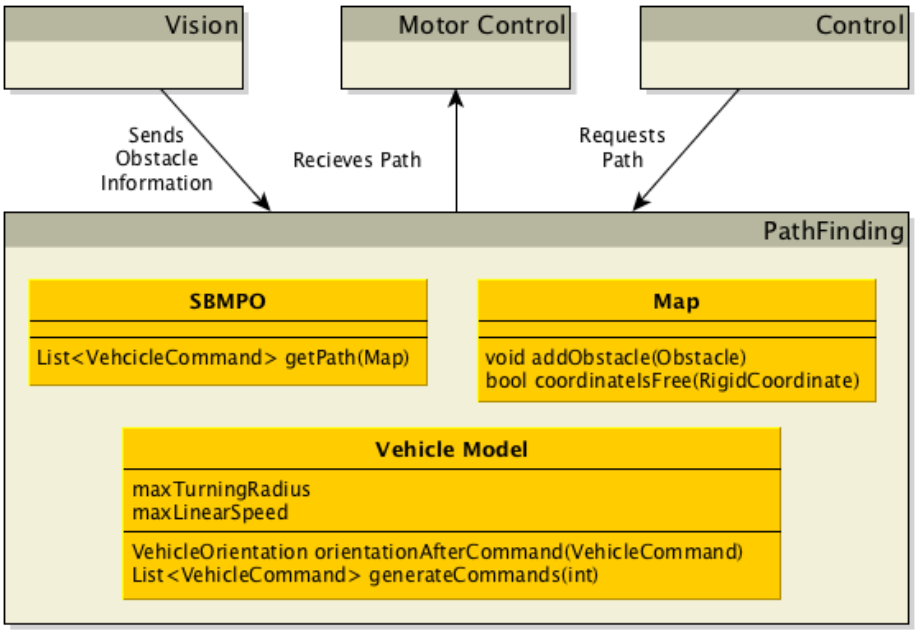
Steps in a subscriber:

1. At initialization, the subscriber queue is bound to exchanges with the specified routing keys.
2. Before consuming messages from exchanges, computations may be started in other threads.
3. Consume is called once per exchange and routing key. The subscriber will receive messages until termination.
4. On receiving a message, the subscriber calls handleDelivery, which parses the message into a string, which is then processed into a data type.
5. Commands are executed based on a message header, denoting the command, and the system is updated.
6. Once the command has been handled, the subscriber will again begin processing messages. Messages that arrive while the subscriber is busy are held in a queue.



At any point during the subscriber's execution, a message may be published. Often, the subscriber will publish in response to messages; however, messages may also be sent by the subsystem independently of the subscriber.

Pathfinding



The pathfinding component of the system will be comprised of a pathfinding algorithm, which will be taking inputs from the computer vision, hardware control, and position estimation components; and producing a set of actions for the robot to take. Each action consists of:

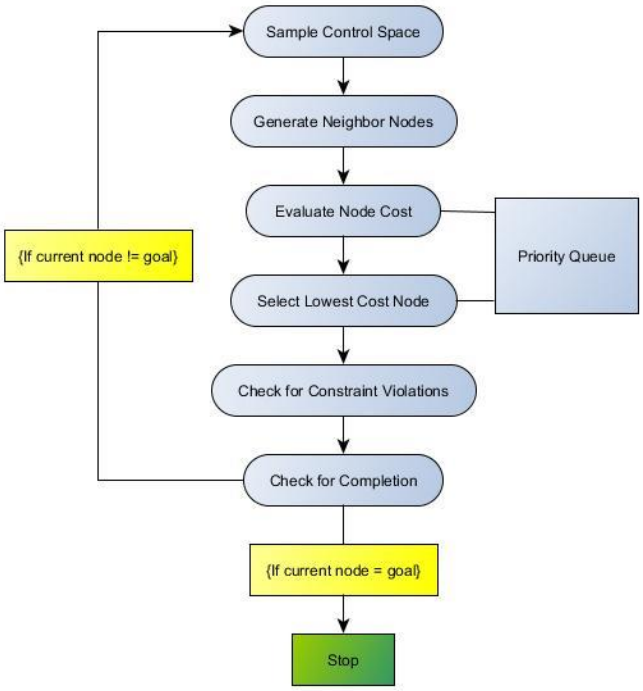
- A linear speed (forwards or backward)
- A duration for execution
- Turning information (turning radius or angular velocity)

The algorithm will create and update a map model based on messages sent from the computer vision module. At regular intervals, the pathfinding algorithm will re-compute the path to the goal, taking into account the most recent version of the map, and send the results to Motor Control. Each calculation produces a list of movement instructions, guiding the robot from its current position to the goal.

Map Representation

Because all space is assumed to be unoccupied until other information is given, the map will be represented by a function that maps coordinates to either a value (blocked), or nothing (unoccupied). This means that accessing information about a given coordinate, adding obstacles to the map, and removing obstacles from the map all occur in constant time.

SBMPC Algorithm

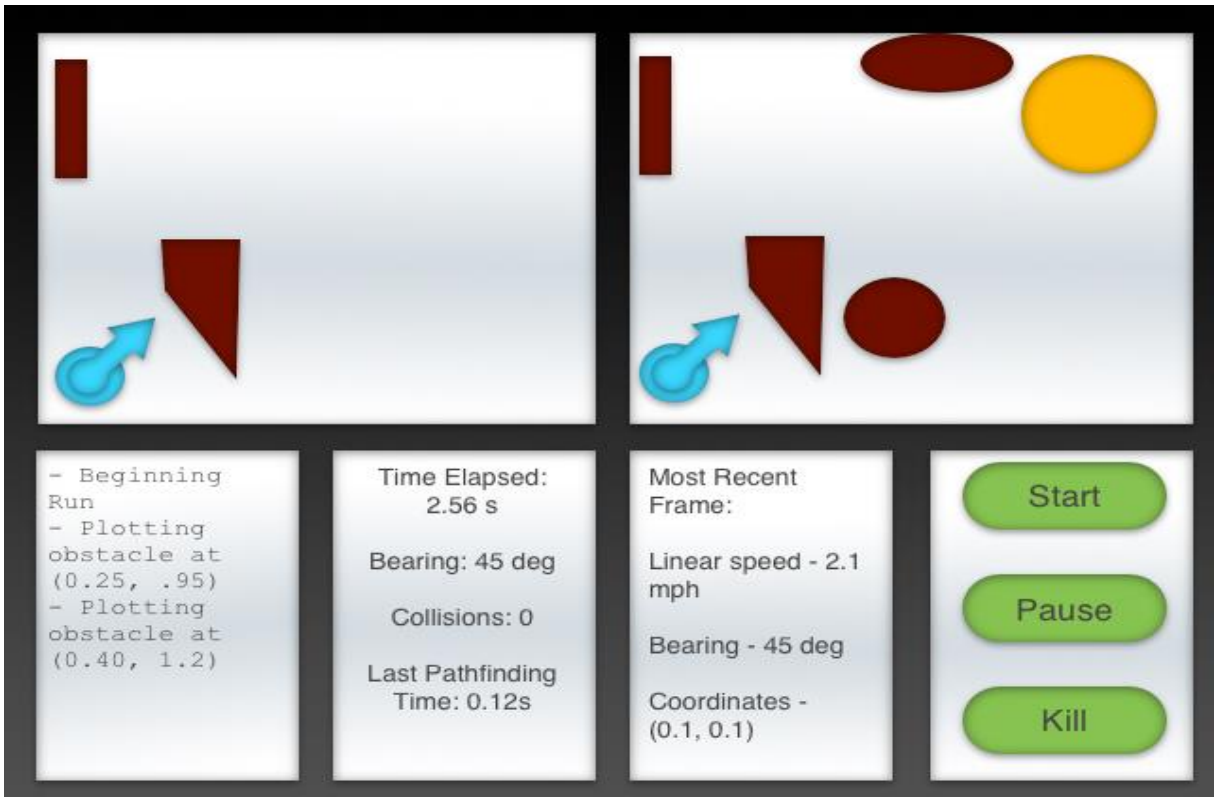


The main algorithm that our pathfinding will execute is as follows:

1. **Sample the Control Space:** Randomly choose samples from the given input that are feasible to navigate to within the robot's physical constraints.
2. **Generate the Neighbor Nodes:** Integrate our system model with the control samples that were chosen in the previous step. This will help us determine where to place the neighbor nodes.
3. **Evaluate the Node Costs:** Use the popular A* pathfinding algorithm's heuristic method to determine the cost of traveling to each neighbor node. In our case, the heuristic will be the shortest distance. Insert these nodes with their costs into a min-heap priority queue.
4. **Select the Lowest Cost Node:** Choose the node that has the lowest cost (top of the priority queue).
5. **Check for Constraint Violations:** Check to make sure that the robot really can get to the chosen node using the known physical restraints again. If no constraints are violated, then travel to the node.
6. **Check for Completion:** If the current node is the same as the goal then stop. If not, go back to step 1.

GUI Interface

The following image is a mockup of the GUI interface for use during simulations. On the top left is the incomplete map that the robot's pathfinding module has created, while the top right is the complete map the simulation has produced. Along the bottom, we have a detailed log of GUI commands as they occur in real time, a panel of robot metrics, the information from the most recent "Frame" (collection of data) received by the GUI, and buttons used for various user "interrupts" of the simulation.



All other GUI interfaces will be some variation of this general format.

Computer Vision

The primary need for vision is navigation. While traversing the course, the vehicle must locate and understand the different types of terrain, obstacles, objectives and course boundary lines it encounters. For this purpose, we have selected the ZED stereoscopic camera, supported by LIDAR. The ZED provides us with:

- Variable frame rate for capturing data
- GPU interoperability
- Point Cloud data for each frame
- 20-meter effective image range
- Depth and color perception

The ZED will produce a large quantity of image data for each frame. This requires a large amount of processing to locate objects. The TX1 board that the ZED will be operating with contains a powerful NVIDIA GPU that will be pivotal in processing the data returned in a timely manner. We have investigated implementations using the OpenCV and/or PCL libraries. Currently, PCL (Point Cloud Library) is the primary library involved in the implementation. PCL takes information in a point cloud format. Point cloud format consists of a set of points relative to the stereoscopic camera's current position, each containing information on the Cartesian coordinates and color of the point. PCL is capable of transforming and reducing point clouds using a variety of image processing algorithms. These will provide us with the ability to filter image data and identify clusters of points (obstacles).

Below is a sample of the data that the ZED produces:

X-Axis	Y-Axis	Z-Axis	Red	Green	Blue
-2706.072510	-1526.048218	2703.730957	67	67	64
-2699.805664	-1524.747559	2701.426758	67	67	64

From test data that we have captured, we have found that we get over 900,000 points represented in the previously described form per frame for 720p frame captures. PCL will be leveraged to process these large amounts of data.

Some of the algorithms already existing in PCL that are applicable include:

- PassThrough filter
- Noise reduction and removal
- Statistical Outlier Removal Noise reduction and removal
- Down-sampling Centroiding algorithm to assist in locating potential objects

The final piece of data generated by the ZED is a depth image map, which will be utilized in conjunction with the information generated using PCL, to then estimate ranges of objects located using the Point Cloud data.

Ideally, these operations will take place on the GPU as much as possible to take advantage of the massively parallel computations that are possible on the GPU. The processing will have to put some data in, and copy some information out to finalize the analysis, however, this will be kept as minimal as possible to not burden the TX1 system. Not only will the GPU-leveraged computations speed up analysis, it will free up potentially large amounts of system resources for use by the other pieces of the system.