Considerations and Conclusions from Building a Small Autonomous SLAM Vehicle

William Wagner, Praveen Bob, Graeme Fraser, Vanessa Casillas, John Sevier

Claremont Graduate University, GIS Practicum, Fall 2019

Table of Contents

Abstract

The following describes the technical considerations and methodologies employed while trying to assemble a small autonomous vehicle capable of Simultaneous Localization and Mapping (SLAM) from LIDAR, while simultaneously, but independently employing a Single Shot Detector Neural Network receiving input from only a single on-board 2D RGB camera for semantic object detection and collision avoidance. Further, our vehicle is intended to take a cube map set of six photos for import into a customized Esri pipeline, process the imagery and SLAM data, then export for near real-time viewing on the Oculus Go virtual reality system. This vehicle should serve as a proof of concept with regards to onboard AI, LIDAR, software pipeline, and hardware compatibility. Due to the technical curveballs, our group was unable to complete all the use case goals we had set for ourselves at the beginning of the semester, however this report serves as a roadmap for a solid foundation upon which to progress to the next level of research with confidence.

Introduction

According to the United Nations, “Demographic-economic projection of city population growth to 2050 suggests that exposure to earthquake and cyclone risk in developing country cities will more than double from today's levels.” [1] Specifically, “the signs of our vulnerability to urban risk are everywhere. An earthquake can bring hospitals, schools and homes tumbling down with unspeakably tragic consequences. A volcano can throw city airports into chaos. Flood waters can turn well-kept streets into detritus-strewn canals. The drug trade can turn an inner city into a war zone. An epidemic can spread rapidly through a crowded slum.” [2]

With this understanding, many organizations have been sponsoring research and development in this area. Two of the most competitive competitions are the DARPA Subterranean Challenge [3] and the Lockheed Martin Drone Racing Challenge [4]. Both of these competitions accept teams from individuals or institutions, and include teams from entities like JPL, Carnegie Mellon, MIT, CalTech, and many

others, as well as private enterprises setup solely for purposes of the competitions. 1st place prizes are over $1M for each competition.

While most of our research and development can be used universally, the two competitions mentioned above both have very different operating demands and optimizations. Lockheed Martin requires a small flying drone capable of very high-speed precision movement for a very short period of time (<30s). DARPA wants any viable solution for very long range (multiple kilometers of distance) and varied (underground and urban) SLAM.

This report details the research and development of an ultra-low-cost vehicle to provide proof of concept results with regards to first steps in competing in the above competitions. The MIT racecar platform has become the de facto development standard for racecar AI. While not robust enough for any off-road challenges, it is capable of supporting most types of indoor challenge, including speed tests, and it has sufficient size to carry multiple batteries, sensors, and processors, making it an ideal platform for development and research. The approximate cost of assembling this type of vehicle is around $2000, however, as this was a first-time project for many of the participants, our budget was substantially lower.

Related Work

See Appendix of Related Work

Problem Definition

The use case scenario for our project is an autonomous vehicle for use in emergency situations by C&C first responders. The vehicle should be capable of autonomous navigation to and from a location, using only on-board capabilities after deployment so that it can be used indoors or outside. Upon arrival at its destination, the vehicle should take six photos in a cube map formation for import into the Esri pipeline with an eventual goal to export to VR viewing on the Oculus Go. Our goal was to create a proof of concept (poc) and explore the capabilities of some existing development systems or platforms.

No one solution was in our price range, while also allowing on-board AI and a LIDAR component. Over the course of the semester, two companies (Nvidia and Yahboom!) have announced kits that compare to some of our solutions, however these were not available only 3 months ago and neither would be a turn-key solution even today.

Nvidia offers an AI development platform using their Jetson Nano, with a parts list from a variety of manufacturers, however this solution required a 3D printer, and greater electronics involvement than we were initially considering (ie. soldering at a component level), and it did not include LIDAR.

As mentioned earlier, MIT publishes a comprehensive parts list; however, these units exceeded our budget. Jetson Hacks / Nvidia also have an AI racecar which is based off the MIT solution, putting it out of our reach as well.

System Solution Development

Nvidia Jetson

We selected the Nvidia Jetson TX2 (See Appendix B for step-by-step setup) and Nano development boards as the principal development platform because of their small form factor combined with on-board support for multi-threaded machine learning libraries. These libraries are implemented in the form of TensorRT models accessed via an API using either C++ or Python, as well as PyTorch. Some functionality exists only in C++ libraries due to the nature of working with video images in memory. The API is installed as jetson.utils and they can be installed independently or as part of the jetson-inference package described below.

The TX2 processor is a Dual-Core NVIDIA Denver 1.5 64-Bit CPU and Quad-Core ARM® Cortex®-A57 MPCore processor which is the same family as the Raspberry Pi and the Arduino. What distinguishes the Jetson line is the inclusion of a GPU. The TX2 has a 256-core NVIDIA Pascal™ GPU capable of 1.33 TFLOPs (See Appendix A).

As illustrated in the diagram below, the TX2 serves as the central platform upon which all else is installed or connected. Integrating the disparate components required a detailed understanding of both the hardware and software components of establishing communication to and from the TX2 to all the other components of the bot via multiple busses and protocols.

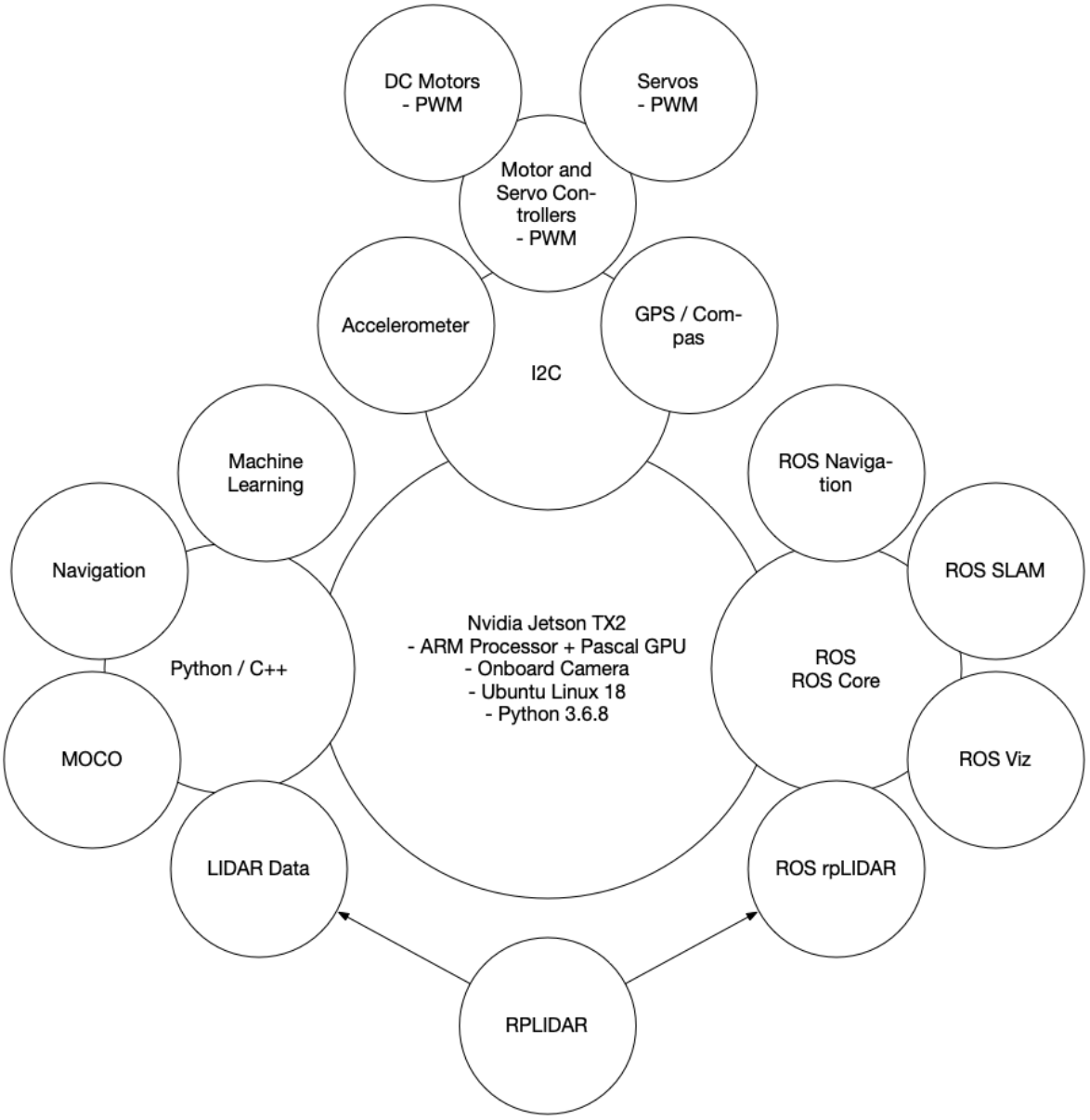

Figure 1 - GIS Bot System Overview

Jetson Inference

> The Python functionality of this project is implemented through Python extension modules that provide bindings to the native C++ code using the Python C API. While configuring the project, the repo searches for versions of Python that have development packages installed on the system and will then build the bindings for each version of Python that's present (e.g. Python 2.7, 3.6, and 3.7). It will also build NumPy bindings for versions of NumPy that are installed.
>
> - jetson.inference documentation

Jetson Inference Installation Summary

We followed the instructions in the Nvidia Jetson Inference Hello World tutorial [5] to setup and test the AI. This involved making sure **cmake** was installed properly and confirming the following steps completed without errors. We did encounter miscellaneous errors, and completing the above steps was not a trivial accomplishment (See Appendix C for detailed steps involved).

```
$ sudo apt-get update
$ sudo apt-get install git cmake libpython3-dev python3-numpy
$ git clone --recursive https://github.com/dusty-nv/jetson-inference
$ cd jetson-inference
$ mkdir build
$ cd build
$ cmake ../
$ make
$ sudo make install
$ sudo ldconfig
```

Flip the Camera Image on TX2:

One of the first challenges we faced was an obvious one. The image was upside down! It required some basic research in the Nvidia developer's forums to find that there is an error in the base code for the jetson.utils when working with the on-board camera for the TX2. It flips the image upside down on the TX2. Not only did this make the image unpleasant for viewing, but we also found that it confused

the AI. For example, it would identify a person walking with their arms outstretched, as a bird. The fix for this had to be implemented in the C++ code for the API, and the package recompiled.

We used gedit to set the flipMethod to 0 in the file utils/camera/gstCamera.cpp: line 416

```
#if NV_TENSORRT_MAJOR > 1 && NV_TENSORRT_MAJOR < 5      // if JetPack 3.1-3.3 (different flip-method)
                const int flipMethod = 0;                                       // Xavier (w/TRT5) camera is mounted inverted
        #else
                const int flipMethod = 0;
        #endif
```

Rebuild the package:

```
$ cd jetson-inference/build
$ make clean
$ cmake ../
$ make
$ sudo make install
```

Install VS Code – OSS version for ARM processors

None of the previous steps includes a full featured IDE so our solution was to build Microsoft's lightweight Code IDE from source for the aarch64 (See Appendix G for detailed installation steps). Code was chosen because it includes built-in support for virtual environments and Git, as well as full-featured linting and code highlighting for all the programming languages in our project. For anyone using Anaconda, it is now included by default. The OSS designates it as Open Source Software.

Set up LIDAR

We connected the RPLIDAR to the TX2 USB via the supplied UART to micro USB converter.

Then we installed the rplidar libraries

```
sudo pip3 install rplidar
```

We used the following Python test code from RPLIDAR.

```
from rplidar import RPLidar
lidar = RPLidar('/dev/ttyUSB0')

info = lidar.get_info()
print(info)

health = lidar.get_health()
print(health)

for i, scan in enumerate(lidar.iter_scans()):
    print('%d: Got %d measurments' % (i, len(scan)))
    if i > 10:
        break

lidar.stop()
lidar.stop_motor()
lidar.disconnect()
```

rplidar.RPLidarException

When we ran the above test code in Python we received the following error:

```
rplidar.RPLidarException: Failed to connect to the sensor due to: [Errno 13] could not open
port /dev/ttyUSB0: [Errno 13] Permission denied: '/dev/ttyUSB0'
```

The error was typical of those we faced while completing the various installations required. While the eventual solution appears simple when listed by the steps below, determining the appropriate modifiers and groups required some research and learning with regards to Linux. The solution was to add our autobot user to the tty and dialout Group then restart.

```
$ sudo gpasswd -a autobot tty
$ sudo usermod -a -G dialout autobot
```

Restart the bot.

Install and Test rplidar SDK

```
$ git clone https://github.com/Slamtec/rplidar_sdk.git
$ cd rplidar_sdk/sdk
$ make
$ cd output/Linux/Release
$ ./simple_grabber /dev/ttyUSB0
```

ROS

ROS is the Robot Operating System. It is a development framework for robotic development. ROS works off of a central core known as ROS Core. After starting the Core process, a variety of other nodes can be connected to the Core. This means that any given node can use any other node's output for input as long as it meets the Core's requirements for communicating.

We were able to access the point data from the rpLIDAR both via Python and via a ROS node and ROS Viz (See Appendix D for detailed installation notes). The data from the LIDAR arrives in the form of an angle (theta) and a distance (range).

It was our intent to use ROS to provide a solution for navigation and SLAM. Unfortunately, while implementing the solution, it was discovered that the ROS Navigation packages require odometry data and/or an Intel or AMD 64-bit processor we did not have (See Appendix E for ROS notes). Our current solution suffered from software / hardware incompatibility.

Our Navigation Solution – In Development

We are currently pursuing a custom solution, utilizing Python as the core programming language. We have written custom Python code to convert between a variety of coordinate systems to enable the bot to be geolocated when possible, but capable of operation indoors or without any GPS signal when necessary.

The primary location data is LAT/LON to provide for global functionality. We wrote a python module to convert from LAT/LON to a cartesian space of X/Y/Z coordinates. This model uses the radius of the Earth and a Spheroid shape, rather than a more accurate Ellipsoid model, because we were

implementing it for poc purposes only, and we wrote the code. However, the transforms doing the trigonometry can and should be easily swapped out for a more optimized library before moving to the next phase of competition or development.

```
R = 6371007.1810 # Radius of Earth in Meters per IUGG Spheroid - not WGS84 Ellipsoid
```

Further, our code is also capable of converting the radial and distance measurements from the LIDAR to and from the other two coordinate systems. In summary, we have a working package capable of handling all the required conversion to interchange any of three coordinate systems: LAT/LON , Cartesian X/Y/Z , Radial Theta/Distance.

The basic starting point of the navigation algorithm is to convert any starting GPS coordinates into X/Y/Z space then minimize the difference in the current location with respect to the destination, until the bot reaches its goal.

The navigation code seeks to emphasize the data coming from the Jetson TX2's onboard 2D RGB camera at 720p resolution. That information is fed into a Single Shot Detector neural network, based off the jetson.inference API. We have installed multiple TensorRT models on the Jetson capable of segmentation, detection, and classification. The models can be further tuned or substituted with custom models.

Figure 2 shows the flow from the onboard camera through the VRAM where it is accessed directly via CUDA as well as a NumPy array for color space transformations. This is the recommended methodology by Nvidia at this time for working with images in Array space. For us it also provides a simple solution for ingesting images intended for use in a machine learning environment.

From VRAM, the image can also be viewed directly using OpenGL. This allows for remote human operation of the bot using a standard joystick for control, as well as assisting in understanding what the camera is showing the AI.

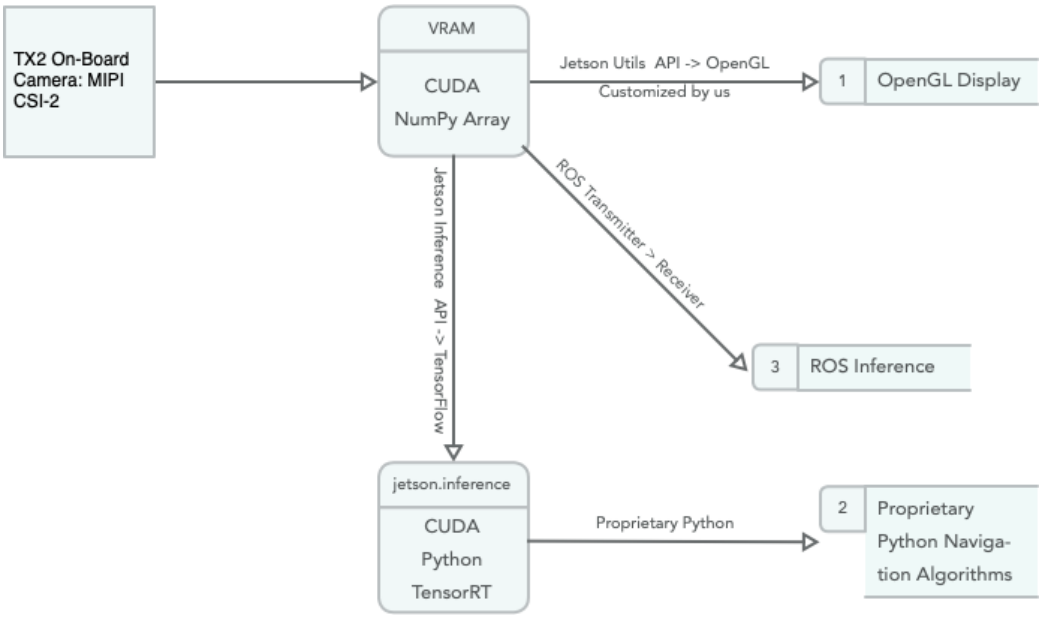

Figure 2 - Image Processing to Navigation Solutions

Resizing the OpenGL Window

When using the OpenGL window via jetson.utils, the window will display the image in the top left of the screen and fill the rest of the display with black. After significant research on how to resize the window using OpenGL, we learned this functionality has actually not been implemented yet via the jetson.utils API. In order to change the window size, we needed to edit the C++ code of the API and recompile. The following changes only hardcode the size of the window, rather than truly providing a general fix for the API.

```
/home/autobot/gis_bot/jetson-inference/utils/display/glDisplay.cpp
starting at line 155:
```

```
	// Uncomment to use the Display size instead of Custom (ie. Camera) Size
	// const int screenWidth = DisplayWidth(mDisplayX, screenIdx);
	// const int screenHeight = DisplayHeight(mDisplayX, screenIdx);

	// Comment out the below two lines if using the above code
	const int screenWidth = 1300;
	const int screenHeight = 750;
```

After editing the code, we re-compiled and re-installed jetson-inference

```
$ cd jetson-inference/build
$ make clean
$ make
$ sudo make install
```

Connecting the Jetson to the Motors, Servos, and Sensors.

Both the Nano and the TX2 list a 40-pin Raspberry Pi style connector on the development board. As we would discover, there are actually variations between the Pi version and the Nvidia versions, and even some differences in pins between the Nvidia models. This meant significant customization was required to use the hardware diagrams and software programs written by any other projects. It also meant that the motor control unit that was intended to control the bot was not usable in its current configuration via a simple 40 pin connector cable. It would require a re-mapping of pins to make this combination work with our TX2 or Nano developer boards.

We were able to make the bot move easily during week 3 when we set up the bot with a Raspberry Pi controller instead of the Nvidia boards. However, due to the difference in pin configurations, simply swapping out the Raspberry Pi board for the Nano or TX2 would not work. And the Pi does not have the processing power to handle the on-board AI. So, we decided we would use the Nvidia Jetson Racecar solution for motor and servo control.

The TX2 and NANO do not have the same support for on-board Pulse Wave Modulation (PWM) via the GPIO pins on the 40pin connector that the Pi has, and they have no software PWM via GPIO whereas the Pi has PWM control for every pin. There is a library available to mimic the software solution, and we would use that in conjunction with a separate PWM control module via I2C. The Nvidia development boards use the I2C standard for communication. This allows for more powerful types of serial communication than PWM, however it makes it a little trickier to use the smaller “hobby” servos and motors that our project was employing at this stage.

This communication between the logic on the processors and controlling the vehicle proved to be a significant pain point during this phase of the project. The lack of any definitive source for an appropriate component list combined with a steep learning curve with regards to the communications protocols supported by the individual components and boards proved to require significantly more learning and testing than originally anticipated. After successfully moving the bot in week three, the difficulties faced later in the process were all that more trying.

Miscellaneous Errors and Solutions

Throughout the project we have been challenged by a range of errors. These errors can often be deceptive, and often required significant research to solve. In fact, there were multiple occasions where the errors could only be remedied by changing source code or required a developer to point us in the right direction. We have listed some of the more universal errors, and some techniques we used to resolve them. Some of these may be well known to those more familiar with the Linux operating system, but they are included here for completeness.

System-wide environment variables

One challenge was the use of USB. USB is a hot swappable standard, and assigns IDs based on the order that a given device is plugged in or recognized at startup. This is very friendly from a user standpoint, allowing one to plug and unplug any number of devices on the same port. However, when using the USB IDs dynamically in our programs, we found it necessary to assign variables that could be used system-wide so that it did not matter if a given component was unplugged or when it was detected in the startup order (See Appendix F for detailed notes on our USB solution).

Next Steps

Complete the POC

We are at the final stages of a viable proof-of-concept for our use case. While not capable of fully autonomous operation, we will seek to complete a moving proof-of-concept over the coming weeks. This

bot should be capable of navigating a hallway at CGU, or the outside path north of the Academic Computing building at CGU, taking the required cube map photos, and returning to base. It will perform SLAM using LIDAR and rudimentary navigation based off the AI. Finally, the images should be imported to Esri tools and viewed on the Oculus Go.

Future improvements to this core set of functionalities will lie in optimizing the route-finding algorithms, further tuning of the AI models, development of ROS modules, and improvements to the motion control algorithms. However, due to the non-generalizable aspects of this type of development, these steps should happen with the specific hardware intended for use moving forward. This will make for both a more robust solution, as well as one that is far more generalizable for real-world use by first responders, C&C, emergency response, hostage rescue, and warfighter scenarios.

Iterate

Moving forward from this point we should be able to iterate on our solutions and deliver substantial gains with regard to the navigational coding and algorithms, as well as creating optimized datasets for training and tuning of the machine learning models. Further, the machine learning models need not be trained locally, allowing us to take advantage of cloud solutions (Google, AWS, Azure) on an as-needed or when-available basis.

Using the racecar sized vehicle as a baseline, we can introduce variations for locomotion like flying drones, tracked vehicles, and multi-pedal solutions. This should make competition at the national and international levels possible from a technical standpoint, however the cost of custom machined and one-off industrial caliber vehicles will be substantial. By employing a two-tiered approach, we can continue to improve software and algorithmic solutions while minimizing any on-going expenses. We can also compete in the DARPA Virtual Challenge for no significant additional charge.

Conclusion

The takeaways from this project are many. It has been a valuable research experience and informative not only of methodology, but of process as well. From dev/ops issues of bug tracking and issue resolution, to time constraints in the ordering process, to electrical engineering skills and Linux system administration – we overcame most of the challenges that can be associated with an early research and development project. I would suggest that in order to compete at the highest levels of the science, teams will require all members to be comfortable with Python, and the following skills should be found to varying degrees amongst the team members as a whole: machine learning, electrical engineering, mechanical engineering, C++, Linux system administration.

We have completed a necessary research and understanding phase of development. We have completed initial Python programs and algorithms to address multiple navigational challenges. We have identified compatible hardware and software and have created custom code to fill many of the gaps in functionality. Much of our work is relevant to more advanced levels of development, and we have done so for well under $1000 and in less than a semester.

In the future I look forward to exploring the intricacies of using machine learning for navigational and computer vision algorithms, as well as interpretive algorithms for understanding the point cloud data coming from LIDAR. While we explored some of the basic implementation of machine learning on the Jetson, I intend to further refine the models and data sets used for training. Having completed the development steps outlined in this paper, I am confident I am well prepared to advance to more capable and complex robotic vehicles and the software to control them.

Works Cited

[1] H. Brecht, U. Deichmann and H. G. Wang, "A Global Urban Risk Index. Policy Working Paper 6506," World Bank, 2013. [Online]. Available: https://openknowledge.worldbank.org/handle/10986/15865. [Accessed 2019].

[2] International Federation of Red Cross and Red Crescent Societies, "World Disasters Report," 2010.

[3] DARPA, "DARPA Subterranean Challenge," [Online]. Available: https://www.darpa.mil/program/darpa-subterranean-challenge. [Accessed 2019].

[4] Lockheed Martin, "Lockheed Martin AI Innovation Challenge," 2019. [Online]. Available: https://www.lockheedmartin.com/en-us/news/events/ai-innovation-challenge.html. [Accessed 2019].

[5] dusty-nv, "Jetson Inference Git Repo," 2019. [Online]. Available: https://github.com/dusty-nv/jetson-inference.git. [Accessed 2019].

[6] "Ubuntu Man Pages," [Online]. Available: https://help.ubuntu.com/community/CheckInstall. [Accessed 2019].

Supplementary Materials

Appendix A

Nvidia Jetson Comparison Table

	Jetson Nano	Jetson TX2 Series			Jetson Xavier NX	Jetson AGX Xavier Series	
		TX2 4GB	TX2	TX2i		AGX XAVIER 8GB	AGX XAVIER
AI Performance	472 GFLOPs	1.33 TFLOPs		1.26 TFLOPs	21 TOPs	20 TOPs	32 TOPs
GPU	128-core NVIDIA Maxwell™ GPU	256-core NVIDIA Pascal™ GPU			384-core NVIDIA Volta™ GPU with 48 Tensor Cores	384-core NVIDIA Volta™ GPU with 48 Tensor Cores	512-core NVIDIA Volta™ GPU with 64 Tensor Cores
CPU	Quad-Core ARM® Cortex®-A57 MPCore	Dual-Core NVIDIA Denver 1.5 64-Bit CPU and Quad-Core ARM® Cortex®-A57 MPCore processor			6-core NVIDIA Carmel ARM®v8.2 64-bit CPU 6MB L2 + 4MB L3	6-core NVIDIA Carmel Arm®v8.2 64-bit CPU 6MB L2 + 4MB L3	8-core NVIDIA Carmel Arm®v8.2 64-bit CPU 8MB L2 + 4MB L3
Memory	4 GB 64-bit LPDDR4 25.6GB/s	4 GB 128-bit LPDDR4 51.2GB/s	8 GB 128-bit LPDDR4 59.7GB/s	8 GB 128-bit LPDDR4 (ECC Support) 51.2GB/s	8 GB 128-bit LPDDR4x 51.2GB/s	8 GB 256-bit LPDDR4x 85.3GB/s	16 GB 256-bit LPDDR4x 136.5GB/s
Storage	16 GB eMMC 5.1 *	16 GB eMMC 5.1	32 GB eMMC 5.1	32 GB eMMC 5.1	16 GB eMMC 5.1	32 GB eMMC 5.1	
Power	5W / 10W	7.5W / 15W		10W / 20W	10W / 15W	10W / 20W	10W / 15W / 30W
PCIE	1 x4 (PCIe Gen2)	1 x1 + 1 x4 OR 1 x1 + 1 x1 + 1 x2 (PCIe Gen2)			1 x1 + 1 x4 (PCIe Gen3, Root Port &Endpoint)	1 x8 + 1 x4 + 1 x2 + 2 x1 (PCIe Gen3)	1 x8 + 1 x4 + 1 x2 + 2 x1 (PCIe Gen4, Root Port &Endpoint)
CSI Camera	Up to 4 cameras 12 lanes MIPI CSI-2 D-PHY 1.1 (up to 18 Gbps)	Up to 6 cameras (12 via virtual channels) 12 lanes MIPI CSI-2 D-PHY 1.2 (up to 30 Gbps) C-PHY 1.1 (up to 41Gbps)			Up to 6 cameras (36 via virtual channels) 12 lanes MIPI CSI-2 D-PHY 1.2 (up to 30 Gbps)	Up to 6 cameras (36 via virtual channels) 16 lanes MIPI CSI-2 | 8 lanes SLVS-EC D-PHY 1.2 (up to 40 Gbps) C-PHY 1.1 (up to 59 Gbps)	
Video Encode	250MP/sec 1x 4K @ 30 (HEVC) 2x 1080p @ 60 (HEVC)	500MP/sec 1x 4K @ 60 (HEVC) 3x 4K @ 30 (HEVC) 4x 1080p @ 60 (HEVC)			2x464MP/sec 2x 4K @ 30 (HEVC) 6x 1080p @ 60 (HEVC)	2x464MP/sec 2x 4K @ 30 (HEVC) 6x 1080p @ 60 (HEVC) 14x 1080p @ 30 (HEVC)	2x1000MP/sec 4x 4K @ 60 (HEVC) 16x 1080p @ 60 (HEVC) 32x 1080p @ 30 (HEVC)
Video Decode	500 MP/sec 1x 4K @ 60 (HEVC) 4x 1080p @ 60 (HEVC)	1000 MP/sec 2x 4K @ 60 (HEVC) 7x 1080p @ 60 (HEVC) 20x 1080p @ 30 (HEVC)			2x690MP/sec 2x 4K @ 60 (HEVC) 12x 1080p @ 60 (HEVC) 32x 1080p @ 30 (HEVC)	2x690MP/sec 2x 4K @ 60 (HEVC) 12x 1080p @ 60 (HEVC) 32x 1080p @ 30 (HEVC)	2x1500MP/sec 2x 8K @ 30 (HEVC) 6x 14k @ 60 (HEVC) 26x 1080p @ 60 (HEVC) 72x 1080p @ 30 (HEVC)
Display	2 multi-mode DP 1.2/eDP 1.4/HDMI 2.0 1 x2 DSI (1.5Gbps/lane)	2 multi-mode DP 1.2/eDP 1.4/HDMI 2.0 2 x4 DSI (1.5Gbps/lane)			2 multi-mode DP 1.4/eDP 1.4/HDMI 2.0 No DSI support	3 multi-mode DP 1.2/eDP 1.4/HDMI 2.0 No DSI support	
DL Accelerator	—	—			2x NVDLA Engines	2x NVDLA Engines	
Vision Accelerator	—	—			—	7-Way VLIW Vision Processor	
Networking	10/100/1000 BASE-T Ethernet	10/100/1000 BASE-T Ethernet	10/100/1000 BASE-T Ethernet, WLAN	10/100/1000 BASE-T Ethernet	10/100/1000 BASE-T Ethernet	10/100/1000 BASE-T Ethernet	
Mechanical	69.6 mm x 45 mm 260-pin SO-DIMM connector	87 mm x 50 mm 400-pin connector			69.6 mm x 45 mm 260-pin SO-DIMM connector	100 mm x87 mm 699-pin connector	

* The Jetson Nano module included as part of the Jetson Nano developer kit has a slot for using microSD card instead of eMMC as system storage device.
Please refer to NVIDIA software documentation online here for information about currently supported features.

Appendix B

Initial Setup of the TX2

1. Start Up the TX2 in Forced Reset Mode
2. Plug in to an Ubuntu Linux 16 or 18 Host Computer
3. Remember the host will probably not be ARM proc, but the TX2 is ARM aarch64 for purposes of building software packages
4. Installed Nvidia JetPack SDK on the host computer – small Lenovo w/Ubuntu 18
5. Needs a network connection
6. Check for all updates on the Host, then check again
7. Connect TX2 to host via supplied USB cable
8. Run the JetPack Installer on the Host to install on the TX2
9. Will take significant time to install JetPack on Host, download the Software and Flash the TX2 for the first time.
10. JetPack caches the downloads locally on the host for next time
11. The JetPack installation process will pause part way through
12. Requires the TX2 to complete its Initial Setup on the TX2
13. Complete setup on the TX2 only through the initial setup steps (including a static IP if you have one – but static is not required at this point)
14. Return to the Host Computer and complete the installation of JetPack software.
15. You can use the default IP address to complete the setup, but you will want to make sure you note the Username and Password of the TX2
16. Complete install will require a few hours and can run unattended; however, it will require you to click Finish and Exit to complete the install. If you click skip or go back at any point it will need to repeat the previous steps (potentially hours).
17. If the TX2 does not have an internet connection you will get “Connection Failed” errors, these can be ignored for the time being.

Appendix C

Jetson Inference Installation Details

By default, Ubuntu comes with the libpython-dev and python-numpy packages pre-installed (which are for Python 2.7). Although the Python 3.6 interpreter is pre-installed by Ubuntu, the Python 3.6 development packages (libpython3-dev) and python3-numpy are not. These development packages are required for the bindings to build using the Python C API.

So, if you want the project to create bindings for Python 3.6, install these packages before proceeding:

```
$ sudo apt-get install libpython3-dev python3-numpy
```

Installing these additional packages will enable the repo to build the extension bindings for Python 3.6, in addition to Python 2.7 (which is already pre-installed). Then after the build process, the jetson.inference and jetson.utils packages will be available to use within your Python environments.

Install the selected Models - Note the amount of space required – TX2 has 8GB available – installing all the models will take a little over 2GB

Compile the Models

```
$ cd jetson-inference/build    # omit if working directory is already build/ from above
$ make
$ sudo make install
$ sudo ldconfig
```

In the build tree, you can find the binaries residing in build/aarch64/bin/, headers in build/aarch64/include/, and libraries in build/aarch64/lib/. These also get installed under /usr/local/ during the sudo make install step.

The Python bindings for the jetson.inference and jetson.utils modules also get installed during the sudo make install step under /usr/lib/python*/dist-packages/. If you update the code, remember to run it again.

You will need to be in the correct directory to run the examples

```
$ cd jetson-inference/build/aarch64/bin
```

Run the Imagenet Object ID test

```
$ ./imagenet-console.py --network=googlenet images/orange_0.jpg output_0.jpg  # --network flag is optional
```

Appendix D

Install and Test ROS (Melodic) and ROS rplidar

cd to Home directory

```
$ cd ~
```

Configure your Ubuntu repositories to allow "restricted," "universe," and "multiverse." You can follow the Ubuntu guide for instructions on doing this.

https://help.ubuntu.com/community/Repositories/Ubuntu

Setup your computer to accept software from packages.ros.org.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Set up your keys

```
$ curl -sSL 'http://keyserver.ubuntu.com/pks/lookup?op=get&search=0xC1CF6E31E6BADE8868B172B4F42ED6FBAB17C654' | sudo apt-key add -
```

Update package list

```
$ sudo apt update
```

Install ROS Desktop (Not “Desktop-Full”)

```
$ sudo apt install ros-melodic-desktop
```

Initialize ROS Dep

```
$ sudo rosdep init
$ rosdep update
```

Environment Setup - It's convenient if the ROS environment variables are automatically added to your bash session every time a new shell is launched:

```
$ echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Dependencies for building packages

```
$ sudo apt install python-rosinstall python-rosinstall-generator python-wstool build-essential
```

Check your installation

```
$ printenv | grep ROS
```

Source the Environment

```
$ source /opt/ros/melodic/setup.bash
```

Install catkin_pkg

```
$ sudo pip3 install catkin_pkg
```

Create a catkin workspace

```
$ mkdir -p ~/bot_catkin_ws/src
$ cd bot_catkin_ws
```

You may need to install cmake

download the tar ball

extract

```
$ ./bootstrap && make && sudo make install
```

The first time in any new directory you must use the following, after that you can use just cmake

```
$ catkin_make -DPYTHON_EXECUTABLE=/usr/bin/python3 -DPYTHON_INCLUDE_DIR=/usr/include/python3.6m
-DPYTHON_LIBRARY=/usr/lib/libpython3.6m.so
```

Source the new catkin overlay

```
$ source devel/setup.bash
```

To make sure your workspace is properly overlayed by the setup script, make sure ROS_PACKAGE_PATH environment variable includes the directory you're in.

```
$ echo $ROS_PACKAGE_PATH
$ cd /src
```

clone the rplidar_ros

```
$ git clone https://github.com/Slamtec/rplidar_ros.git
$ cd ..
```

Confirm the overlay

```
$ source devel/setup.bash
```

Run catkin_make

```
$ catkin_make
$ roslaunch rplidar_ros view_rplidar.launch
```

Might need to change the USB port in the launch file

ProTip for figuring out which devices are what on USB:

Store a list of devices before plugging in Device

```
$ ls /dev/ > dev_list_1.txt
```

Then run this after you plug it in

```
$ ls /dev/ | diff --suppress-common-lines -y - dev_list_1.txt
```

RPLIDAR Won't stop – STILL NOT WORKING PROPERLY

Use a powered USB hub

Add 2 lines of command

```
$ source /home/{user}/catkin_ws/devel/setup.bash
$ source /opt/ros/melodic/setup.bash
```

RPLIDAR Demo using ROS Viz

```
$ cd ~/rplidar
$ source devel/setup.bash
$ roslaunch rplidar_ros view_rplidar.launch
```

ROS Notes

If you are ever having problems finding or using your ROS packages, make sure that you have your environment properly setup. A good way to check is to ensure that environment variables like ROS_ROOT and ROS_PACKAGE_PATH are set:

```
printenv | grep ROS
```

Need to run this in every new shell unless added to .bashrc

```
source /opt/ros/melodic/setup.bash
```

To make sure your workspace is properly overlayed by the setup script, make sure ROS_PACKAGE_PATH environment variable includes the directory you're in.

```
echo $ROS_PACKAGE_PATH
```

Make sure to allow access to the USB

```
sudo chmod 666 /dev/ttyUSB0
```

WARNING: The script jupyter-console is installed in '/home/autobot/.local/bin' which is not on PATH.

Consider adding this directory to PATH or, if you prefer to suppress this warning, use –no-warn-script-location.

Appendix E

ROS Deep Learning – Dusty nv – Nvidia

https://github.com/dusty-nv/ros_deep_learning

```
$ sudo apt-get install ros-melodic-image-transport
$ sudo apt-get install ros-melodic-image-publisher
$ sudo apt-get install ros-melodic-vision-msgs
```

Then, create a Catkin workspace (~/catkin_ws) using these steps:

http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment#Create_a_ROS_Workspace

```
$ cd ~/bot_catkin_ws/src
$ git clone https://github.com/dusty-nv/ros_deep_learning
$ cd ../
$ catkin_make
```

Note from Will: if you do this after the rplidar installation, it will see both cmake files and do them both.

Testing

Before proceeding, make sure that roscore is running first:

```
$ roscore
```

imageNet Node

First, to stream some image data for the inferencing node to process, open another terminal and start an image_publisher, which loads a specified image from disk. We tell it to load one of the test images that come with jetson-inference, but you can substitute your own images here as well:

```
$ rosrun image_publisher image_publisher __name:=image_publisher ~/jetson-
inference/data/images/orange_0.jpg
```

Next, open a new terminal, overlay your Catkin workspace, and start the imagenet node:

```
$ source ~/catkin_ws/devel/setup.bash
```

```
$ rosrun ros_deep_learning imagenet /imagenet/image_in:=/image_publisher/image_raw
_model_name:=googlenet
```

Here, we remap imagenet's image_in input topic to the output of the image_publisher, and tell it to load the GoogleNet model using the node's model_name parameter. See this table for other classification models that you can download and substitute for model_name.

In another terminal, you should be able to verify the vision_msgs/Classification2D message output of the node, which is published to the imagenet/classification topic:

```
$ rostopic echo /imagenet/classification
```

detectNet Node

Kill the other nodes you launched above, and start publishing a new image with people in it for the detectnet node to process:

```
$ rosrun image_publisher image_publisher __name:=image_publisher ~/jetson-
inference/data/images/peds-004.jpg
$ rosrun ros_deep_learning detectnet /detectnet/image_in:=/image_publisher/image_raw
_model_name:=pednet
```

See this table for the built-in detection models available. Here's an example of launching with the model that detects dogs:

```
$ rosrun image_publisher image_publisher __name:=image_publisher ~/jetson-
inference/data/images/dog_0.jpg
$ rosrun ros_deep_learning detectnet /detectnet/image_in:=/image_publisher/image_raw
_model_name:=coco-dog
```

To inspect the vision_msgs/Detection2DArray message output of the node, subscribe to the detectnet/detections topic:

```
$ rostopic echo /detectnet/detections
```

Appendix F

Install MatPlotLib and SciKit for aarch64

NOT WORKING w/Python 3.6.8

```
sudo apt-get update


pip3 install --upgrade setuptools
sudo pip3 install -U setuptools
sudo apt-get install libpcap-dev libpq-dev
sudo pip3 install cython

sudo apt-get update
sudo apt-get install -y build-essential libatlas-base-dev
sudo apt-get install gfortran



sudo pip3 install -U scikit-learn

sudo pip3 install git+https://github.com/scikit-learn/scikit-learn.git
python -m pip install --user numpy scipy matplotlib
```

PCA9685 Adafruit PCM

https://github.com/jetsonhacks/JHPWMDriver

https://www.jetsonhacks.com/2016/01/11/imu-and-pwm-driver-over-i2c-for-nvidia-jetson-tx1/

https://github.com/adafruit/Adafruit_CircuitPython_ServoKit

```
sudo pip3 install adafruit-circuitpython-servokit
```

Environment Variables and Settings

A suitable file for environment variable settings that affect the system as a whole (rather than just a particular user) is /etc/environment. An alternative is to create a file for the purpose in the /etc/profile.d directory.

/etc/environment

This file is specifically meant for system-wide environment variable settings. It is not a script file, but rather consists of assignment expressions, one per line.

```
FOO=bar
```

Note: Variable expansion does not work in /etc/environment.

/etc/profile.d/*.sh

Files with the .sh extension in the /etc/profile.d directory get executed whenever a bash login shell is entered (e.g. when logging in from the console or over ssh), as well as by the DisplayManager when the desktop session loads.

You can for instance create the file /etc/profile.d/myenvvars.sh and set variables like this:

```
export JAVA_HOME=/usr/lib/jvm/jdk1.7.0
export PATH=$PATH:$JAVA_HOME/bin
```

Setting USB Environment Variables

1. Find out what's on ttyUSB:

```
$ dmesg | grep ttyUSB
```

2. List all attributes of the device. Use your device ID instead of x.

```
$ udevadm info --name=/dev/ttyUSBx –attribute-walk
```

Pick out a unique identifier set, eg idVendor + idProduct. You may also need SerialNumber if you have more than one device with the same idVendor and idProduct. SerialNumbers ought to be unique for each device.

```
Info for our UART Adapter
ID_VENDOR=Silicon_Labs
ID_VENDOR_ID=10c4
ID_MODEL_ID=ea60
symlink for our bot :  uart_bridge
```

3. Create a file /etc/udev/rules.d/99-your_file.rules with something like this line in it:

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="1234", ATTRS{idProduct}=="5678",
SYMLINK+="your_device_name"
```

4. Load the new rule:

```
$ sudo udevadm trigger
```

5. Verify what happened:

```
$ ls -l /dev/your_device_name
```

will show what ttyUSB number the symlink went to. If it's /dev/ttyUSB1, then verify who owns that and to which group it belongs:

```
$ ls -l /dev/ttyUSB1
```

Then a final double check:

```
$ udevadm test -a -p $(udevadm info -q path -n /dev/your_device_name)
```

One Optimum Way to Install / Uninstall Packages

One significant challenge throughout the course of the project was the need to custom compile most of the software packages, and or code snippets. We were constantly presented with multiple authoritative solutions for a given installation. There are many “bad habits” that can easily become the norm when attempting to make all the pieces work together.

Given the unstable nature of a development environment, it is often necessary to uninstall after an incomplete installation. The variety of dependencies, and the need to compile much of the software specifically to this machine means that there will often be dependencies that need to be installed before a given package can be compiled. If a package fails to compile, it can leave files scattered around the system, making removal of a partial installation very difficult. The following method attempts to mitigate this scattering of files by tracking the installation, making the reversal of the process much easier.

We use checkinstall [6] instead of just install to create the list of files, and we use run ./configure instead of just ./configure to get a dialog box about any dependencies that are available for install.

Use CheckInstall with auto-apt

You can use auto-apt when you want to build a simple package from source with checkinstall.

You need to have auto-apt installed!

Instead of

```
$ ./configure
```

you use:

```
$ auto-apt run ./configure
```

If the dependencies are available, a dialog box opens and asks you to install them.

The rest remains the same

```
$ make
$ sudo checkinstall
```

Uninstalling

The installed package can then also easily be removed via Synaptic or via the terminal:

```
$ sudo dpkg -r packagename
```

Example:

```
$ sudo dpkg -r pidgin
```

Appendix G

Install Microsoft Code OSS for aarch64

Install CURL

```
$ sudo apt-get install curl
$ curl -s https://packagecloud.io/install/repositories/swift-arm/vscode/script.deb.sh | sudo bash
$ sudo apt-get install code-oss
```

Code OSS will be installed in /usr/share/applications/. Once installed you can pin it to the dock.

Install the Python Extension in VS Code

Install pylint

```
$ sudo pip3 install pylint
```

Or

```
$ pip3 install pylint --user
```

Or if using venv:

```
$ source venv/bin/activate
$ pip3 install pylint
```

Appendix of Related Work

https://emerj.com/ai-sector-overviews/search-and-rescue-robots-current-applications/

https://link.springer.com/content/pdf/10.1186%2Fs41018-018-0045-4.pdf

http://theconversation.com/autonomous-drones-can-help-search-and-rescue-after-disasters-109760

https://singularityhub.com/2019/04/12/ai-and-robotics-are-transforming-disaster-relief/

https://github.com/NVIDIA-AI-IOT/jetbot

https://blogs.nvidia.com/blog/2019/03/26/jetbot-diy-autonomous-robot/

https://www.nvidia.com/en-us/self-driving-cars/drive-constellation/

https://yadovr.com/

https://www.youtube.com/watch?v=wKMWjIKaU68

https://github.com/NVIDIA-AI-IOT/jetbot/wiki/Software-Setup

https://developer.nvidia.com/embedded/community/jetson-projects#keras_mobiledetectnet

https://developer.nvidia.com/embedded-computing

https://www.youtube.com/watch?v=ASIVJmlGNc8&feature=youtu.be

https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/?=jetsonDevkits

https://www.youtube.com/watch?v=FuRhYCFCaCc

https://www.youtube.com/watch?v=VhbFbxyOI1k

https://build5nines.com/raspberry-pi-4-vs-nvidia-jetson-nano-developer-kit/

https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html

https://stackoverflow.com/questions/7222382/get-lat-long-given-current-point-distance-and-bearing/7835325

https://www.earthdatascience.org/courses/earth-analytics-python/spatial-data-vector-shapefiles/intro-to-coordinate-reference-systems-python/

https://www.youtube.com/watch?v=xkut3yRL61U