Fall 2018 Indenpendent Study Report

Feiyang Jin

December 2018

Contents

1 Introduction

This report is written by Feiyang Jin to record his independent study during Fall 2018, under the supervise of Xuan Zhang. There are a lot of things being done during this independent study, and I organize them into three sections: GPU and CUDA study, Embedded system and Nvidia Jetson TX2, MAVbench paper.

1.1 GPU and CUDA

Parallel computing has been growing extremely fast during past years. To utilize and maximize the potential of GPU (graphics processing unit), we need the help from CUDA, a parallel computing platform and application programming interface (API) model created by Nvidia[1].

1.2 Embedded system and Nvidia Jetson TX2 board

An embedded system is a programmed controlling and operating system with a dedicated function within a larger mechanical or electrical system[2]. The goal for this study is to choose a suitable embedded system for Picar project. This report will make comparison between two popular embedded system, Samsung Exynos and Nvidia Jetson.

1.3 MAVbench paper

MAVBench is a framework targeting design and development of Micro Aerial Vehicles for hardware/software designers and roboticists. It consists of a closed-loop simulator and an end-to-end application benchmark suite[3]. My study focus on understanding the paper material, and try to rebuild their work on Jetson TX2 board.

2 Goal of the study

The goal of my independent study is to develop a platform or tool for instrumenting GPU (precision measurement of certain code snippets running on an actual hardware processor). This would be part of a huge project: meso-scale power orchestration. For meso-scale robotics, it would be a great improvement in power if we can run some computing intensive work on GPU. Thus, we need a tool to help us profile activity on GPU precisely. To achieve this goal, I go through several different topics this semester, and inspired by them greatly.

3 GPU and CUDA

3.1 GPU

Typically,CPU is required to run the majority of engineering and office software; however, there is a multitude of tasks that can overwhelm CPU. If we do not want computing-intensive work to crash our CPU, we need a powerful GPU to to that work. The reason is that GPU uses thousands of smaller and more efficient cores for a massively parallel architecture, which is aimed at handling multiple functions at the same time. There is an analogy between CPU,GPU and brain and brown. A CPU (the brain) can work on a variety of different calculations, while a GPU (the brawn) is best at focusing all the computing abilities on a specific task.

3.2 CUDA

Designed by Nvidia, CUDA increases computing performance by harnessing the power of GPU[1]. As mentioned in last section, CPU still performs as the brain for work (like assigning tasks), so sequential part still runs on CPU, while the compute intensive portion runs on GPU cores in parallel. In CUDA, the CPU is called "host", and "GPU" is called "device"

3.2.1 CUDA program

A typical CUDA program would look following[4]:

1. Declare and allocate CPU(host) and GPU(device) memory.
2. Initialize host data.
3. Transfer data from the host to the device.
4. Execute one or more kernels.
5. Transfer results from the device to the host.

A great advantage of CUDA is its similarity with C or C++ program. The coding syntax is almost the same, with some CUDA extensions the file would become a CUDA program that is compiled by CUDA compiler called nvcc.Here are some basic CUDA language extensions and functions

- __global__ specifier: tells CUDA that GPU runs this function
- cudaMallocManaged, cudaFree. Similar to C function malloc and free, but these two CUDA functions manage memory on GPU (malloc and free memory on GPU).
- cudaDeviceSynchronized(): let CPU wait for GPU work done
- cudaMemcpy: transfer data between host and device

3.2.2 CUDA profiling

CUDA also comes with its own profiling tools. "nvprof" is the command to profiling a certain CUDA program in terminal.
Beside the terminal command nvprof, CUDA has a visual profiler as well. The visual profiler helps to translate nvcc result to easy understanding graph with nice interface.
With the help of nvprof and nvcc, we are able to measure how well the program runs on GPU, even some details about CPU are also available from the two.
There are even details about hardware from profiling. The profiler is able to record the temperature and power of GPU during the running time. However, the data is not update frequently, and how precise the measurement is not the scope of this report.

```
C:\Users\feiya\Desktop\cuProgram>nvprof saxpy.exe
==9916== NVPROF is profiling process 9916, command: saxpy.exe
==9916== Warning: Unified Memory Profiling is not supported on the current configuration because a pair of devices witho
ut peer-to-peer support is detected on this multi-GPU setup. When peer mappings are not available, system falls back to
using zero-copy memory. It can cause kernels, which access unified memory, to run slower. More details can be found at:
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-managed-memory
Max error: 0.000000
==9916== Profiling application: saxpy.exe
==9916== Warning: Found 7 invalid records in the result.
==9916== Warning: This can happen if device ran out of memory or if a device kernel was stopped due to an assertion.
==9916== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   60.15%  2.0676ms         2  1.0338ms  962.64us  1.1049ms  [CUDA memcpy HtoD]
                   29.31%  1.0075ms         1  1.0075ms  1.0075ms  1.0075ms  [CUDA memcpy DtoH]
                   10.55%  362.57us         1  362.57us  362.57us  362.57us  saxpy(int, float, float*, float*)
      API calls:   72.72%  224.99ms         2  112.50ms  844.52us  224.15ms  cudaMalloc
                   23.86%  73.817ms         1  73.817ms  73.817ms  73.817ms  cuDevicePrimaryCtxRelease
                    1.59%  4.9196ms         3  1.6399ms  1.2985ms  2.1259ms  cudaMemcpy
                    1.46%  4.5148ms        89  50.727us     463ns  2.9973ms  cuDeviceGetAttribute
                    0.21%  642.32us         2  321.16us  286.15us  356.17us  cudaFree
                    0.08%  244.87us         1  244.87us  244.87us  244.87us  cuModuleUnload
                    0.03%  106.20us         1  106.20us  106.20us  106.20us  cuDeviceGetName
                    0.03%  86.261us         1  86.261us  86.261us  86.261us  cudaLaunchKernel
                    0.01%  32.927us         1  32.927us  32.927us  32.927us  cuDeviceTotalMem
                    0.01%  15.768us         1  15.768us  15.768us  15.768us  cuDeviceGetPCIBusId
                    0.00%  6.9560us         3  2.3180us     463ns  5.5650us  cuDeviceGetCount
                    0.00%  2.7820us         2  1.3910us     463ns  2.3190us  cuDeviceGet

C:\Users\feiya\Desktop\cuProgram>
```

Figure 1: sample nvprof result

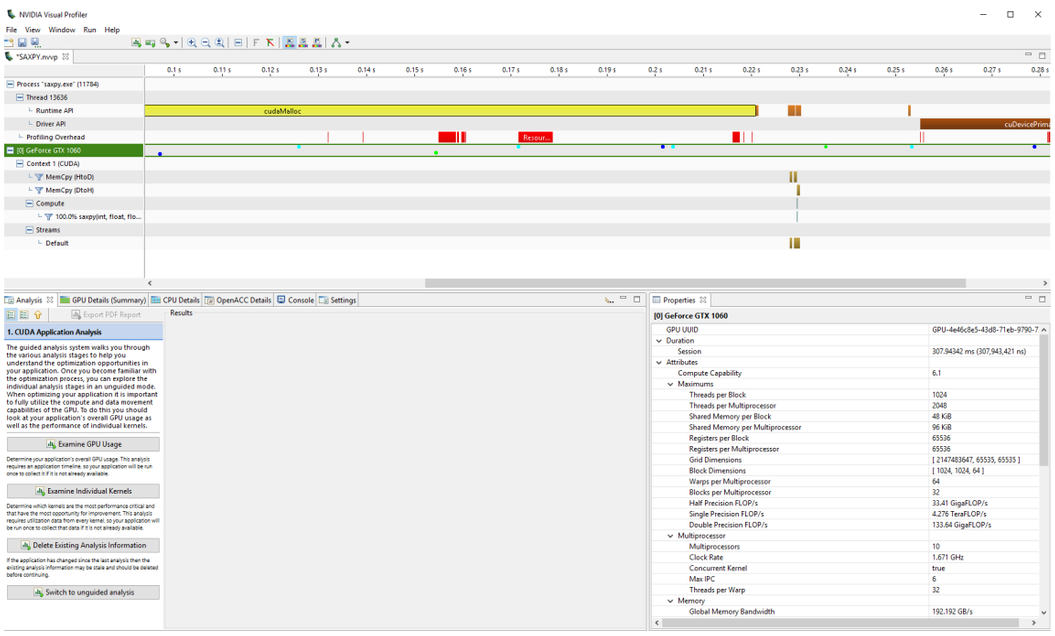

Figure 2: sample visual profiler result

3.2.3 Tips for installing CUDA on windows

Despite the fact that CUDA is a well documented platform, the installing process on my windows laptop almost kills me. The following tip,as I believe, would be great help for you if you are stuck with CUDA installation:

1. Set up: In order to best use CUDA, Microsoft visual studio is a necessary platform. However, the integration of CUDA and VS would fail because CUDA does not update its default setting. The reason is that as VS keeps updating, its "serial number" also changed while in CUDA setting file, there is one statement that if "serial number" ¿ some model, the integration will not execute.

2. To solve the problem, we need to copy several files such as cl.exe, from CUDA directory to VS directory. We also need to change the CUDA setup file to make it recognize the "Serial number".

3. Due to the lack of integration with VS, NVCC, the CUDA compiler, will fail to compile CUDA program from .cu file as well. To solve this, we need ccbin flag to indicate the VS compiler for nvcc.

4 Embedded system and Nvidia Jetson TX2 board

The second part of my study is to choose an embedded system for the Picar project[5]. Currently, Picar is a miniature four-wheeled car powered by a Raspberry Pi 3 board. However, it does not contain any computing intensive work when running. If we wish to improve its computing ability, a more powerful and portable board would be required.
The board should have a reasonable size for our car to carry, and an embedded system typically would fit our requirement. After doing some search and comparison, finally I focus on two embedded system: Samsung Exynos 8895 and Nvidia Jetson TX2. However, Samsung does not seem to have official board for developers. The board using Exynos processor is only available from third party, while Nvidia Jetson board is supported by Nvidia officially. The following table is a comparison of Exynos 8895, which is launched last year and used on Samsung Galaxy S8, and Nvidia Jetson TX2, which was also launched last year.

	Exynos 8895	Jetson TX2
CPU	2.3GHz Quad-Core (Custom CPU) + 1.7GHz Quad-Core (Cortex®-A53)	ARM Cortex-A57 (quad-core) @ 2GHz + NVIDIA Denver2 (dual-core) @ 2GHz
GPU	Mali-G71 MP20 @ 900 (Boost) MHz	256-core Pascal @ 1300MHz
Memory	8GB 64 bits LPDDR4X 3733 MBPS	8GB 128-bit LPDDR4 @ 1866Mhz | 59.7 GB/s
Camera	Rear 28MP, Front 28MP, Dual Camera 16MP+16MP	12 lanes MIPI CSI-2 | 2.5 Gb/sec per lane | 1400 megapixels/sec ISP
Video	4K UHD 120fps encoding and decoding with HEVC (H.265), H.264, VP9 Codec	Supported video codecs: H.264, H.265, VP8, VP9
Power	No official data	7.5W Max-Q profile 15W Max-P profile
Available develop board	Howchip ExSOM-8895 http://www.howchip.com/products/exsom8895/ExSOM_8895.php	itself

Figure 3: compare between Exynos 8895 and Jetson TX2

At last, we chose to use Jetson TX2 as our embedded system for Picar.

4.1 Nvidia Jetson TX2 board

Jetson TX2, by default, runs on Ubuntu. With some embedded system variables and function, it is extremely powerful.For instance, you can measure the power of this board as following:

- /sys/bus/i2c/drivers/ina3221x, This directory saves voltage, current and power for different rails(e.g. VDD_GPU,VDD_SOC,VDD_WIFI)
- You can "cat" any variable and it will show corresponding data (voltage, current, etc)

The CPU cluster consists of a dual-core Denver 2 processor and a quad-core ARM Cortex-A57, connected by a high-performance coherent interconnect fabric; there are totally 6 CPU cores, and you can set different mode to decide how many CPU cores you want to use. The command to use is nvpmodel, The following table breaks down the modes, which CPU cores are used, and the maximum frequency of the CPU and GPU being used[6].

Mode	Mode Name	Denver 2	Frequency	ARM A57	Frequency	GPU Frequency
0	Max-N	2	2.0 GHz	4	2.0 GHz	1.30 Ghz
1	Max-Q	0		4	1.2 Ghz	0.85 Ghz
2	Max-P Core-All	2	1.4 GHz	4	1.4 GHz	1.12 Ghz
3	Max-P ARM	0		4	2.0 GHz	1.12 Ghz
4	Max-P Denver	2	2.0 GHz	0		1.12 Ghz

Figure 4: Jetsont TX2 model and cores

The following two screenshots are what I did on my nvidia jetson board.

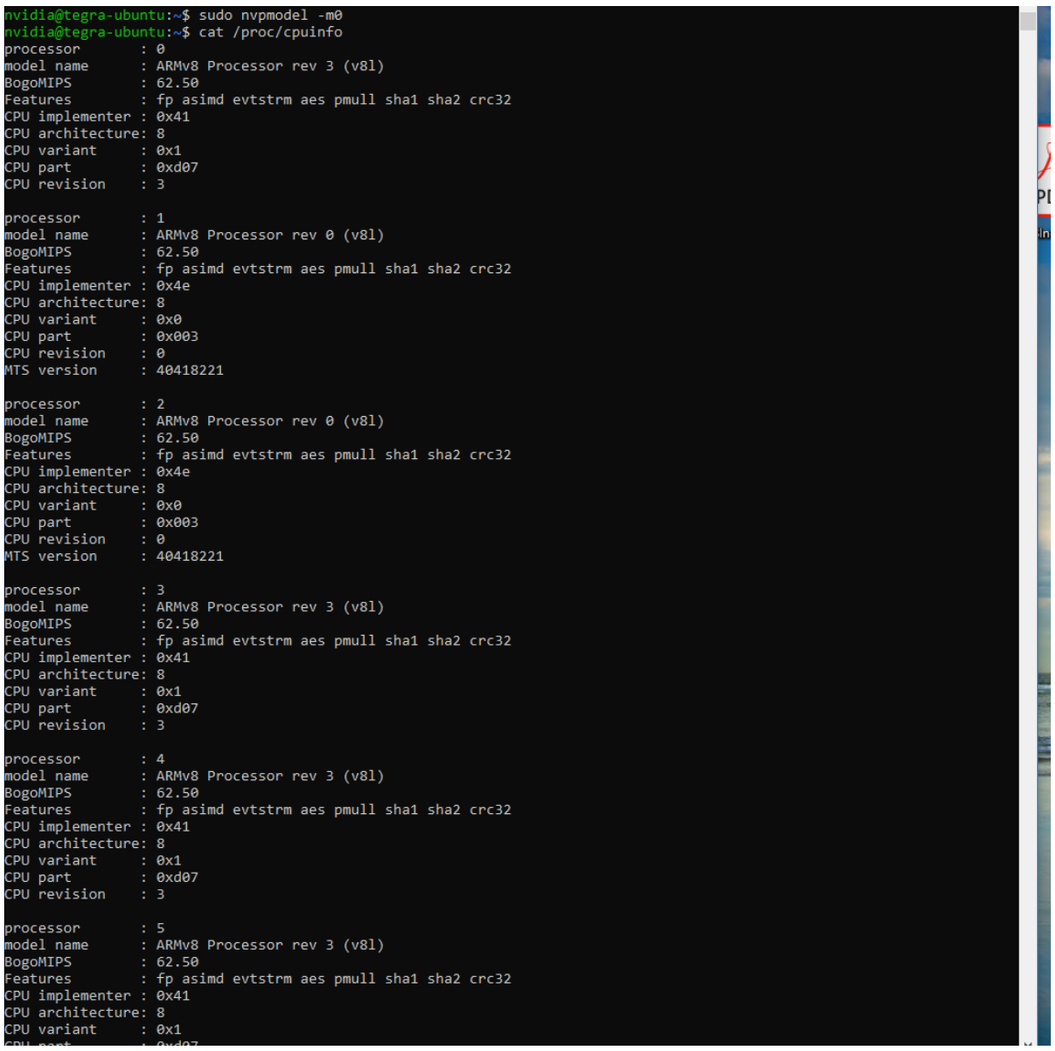

Figure 5: Jetsont TX2 model0

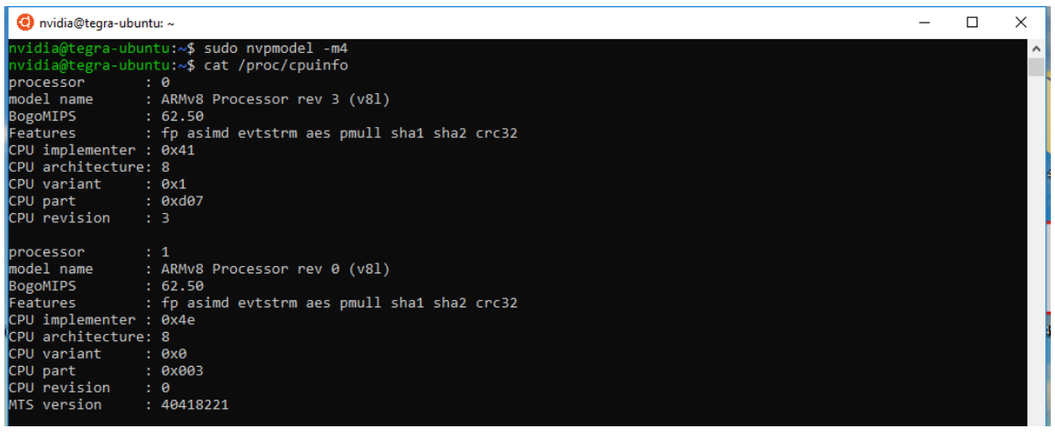

Figure 6: Jetsont TX2 model4

As we can see, by switching between different modes, we can decide how many cores to use. Notice that CPU0 is always available.
ROS (robot operating system), which will be introduced in next section, seems to spread out its programs through all available cores. The following two screenshots are what I get from htop, a running process monitor, when launching one of the MAVbench apps (discussed in next section) using roslaunch.

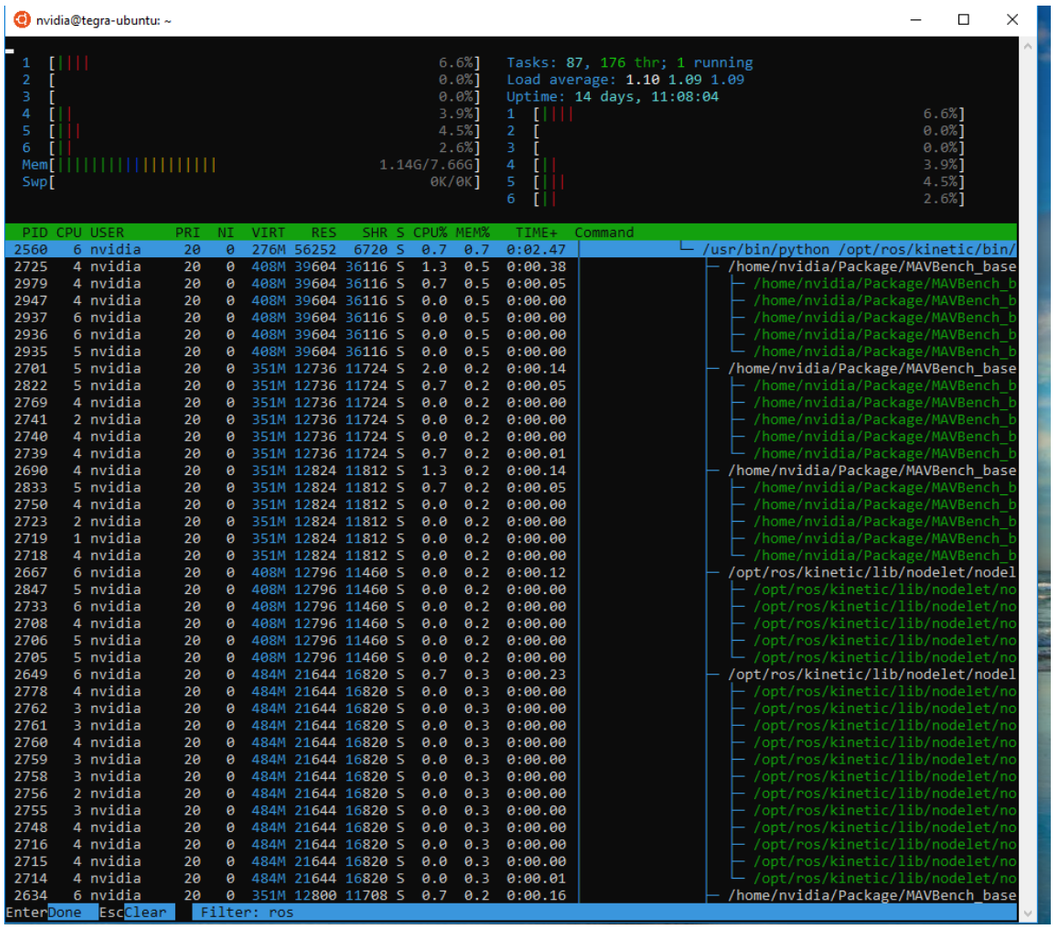

Figure 7: ros running in mode 0 (all cores available)

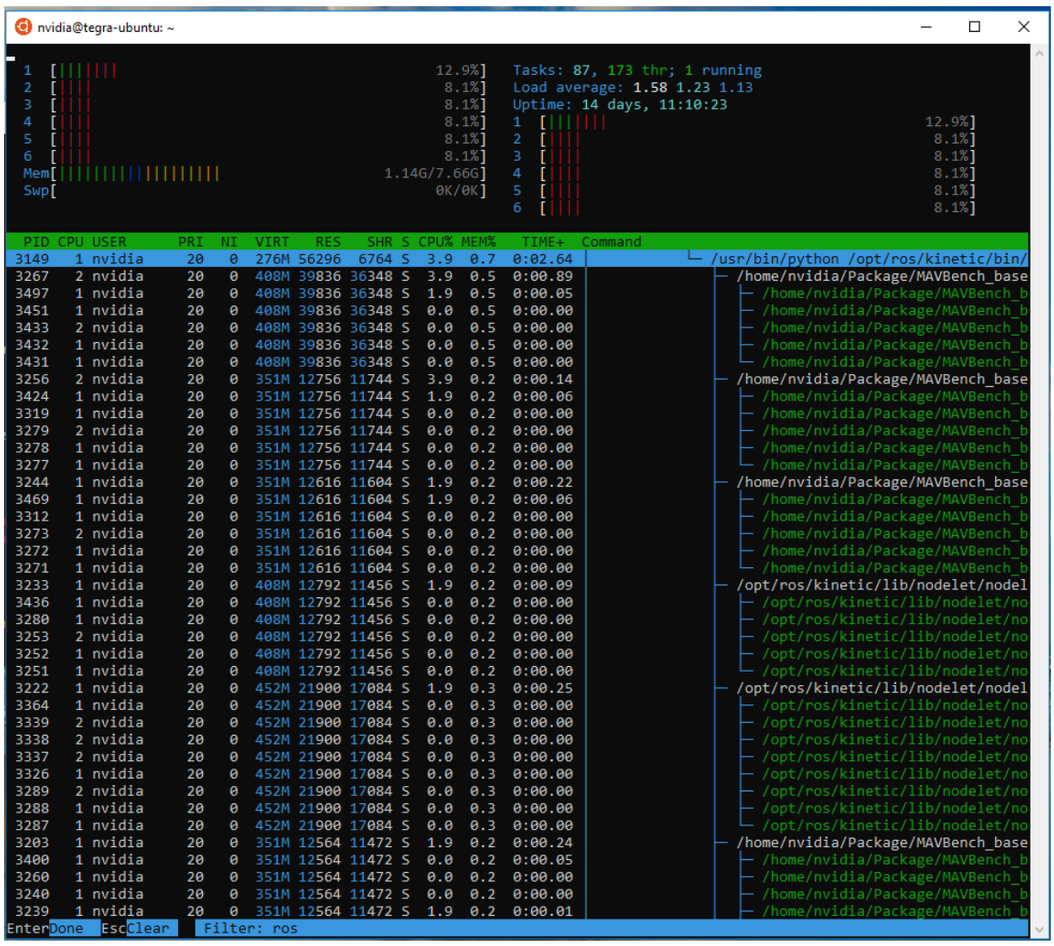

Figure 8: ros running in mode 4 (only Denver cores available)

5 MAVbench paper

The MAVbench includes a closed-loop simulator, which integrates simulated environment, simulated flight controller and one companion computer(one Jetson TX2 board) responsible for computing intensive work. The following figure shows this structure.

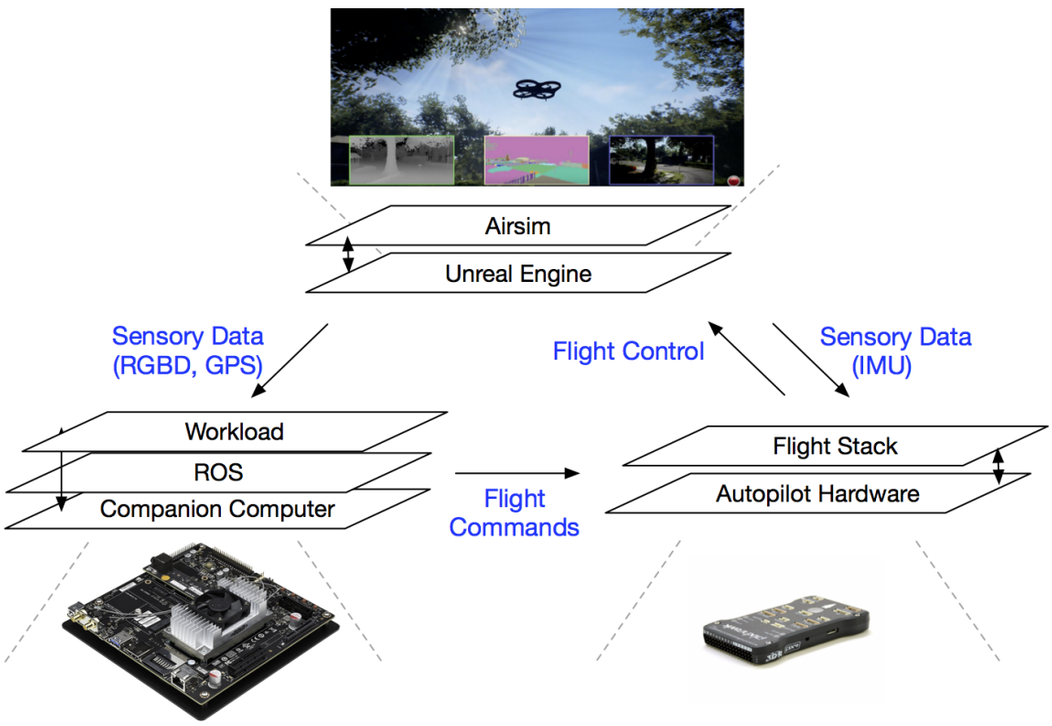

Figure 9: closed loop simulator for MAVbench

The top of closed-loop is simulated environment and sensors(for drone) in Airsim, which is built on Unreal Engine. This part is running on the host computer with a powerful GPU to simulate all the things.
The left corner is the companion computer. It is a powerful compute unit (compared to the FC) that is responsible for the processing of the high level, computationally intensive tasks such as vision processing[3]. Robot operating system (ROS) also runs on companion computer. It is used for many purposes such as low-level control or processes communication.
The right corner is flight controller, which is also simulated on host computer in Airsim. The flight controller(FC) is an autopilot system responsible for the MAV's stabilization and conversion of high-level to low-level actuation commands.FCs take high-level flight commands such as"take-off" and lower them to a series of instructions understandable by actuators such as rotors[3].
The workload of MAVbench is best explained by walking through an example. The object(e.g. a person) and its environment (e.g. urban) are modeled in the Unreal Engine. The MAV's sensors (e.g. accelerometer and RGB-D Camera), modeled in Airsim, feed their data to the flight controller (e.g. physics data to PX4) and the companion computer (e.g. visual and depth to TX2) using MAVLink protocol. The kernel (e.g. object detection), running within the ROS runtime environment on the companion computer, is continuously invoked until the object is detected. Once so, flight commands (e.g. move forward) are sent back to flight controller, where they get converted to a low level rotor instruction stream flying the MAV closer to the person.[3]
The thing to be noticed is that most of the part of this closed loop is simulated except the companion computer. This means the MAVbench is actually a good platform to do further research because it helps us to avoid physical problems in read world.
In next two subsections, I will do case study on two of the sample applications to further how MAVbench works and its design thoughts.

5.1 Case study: Package delivery

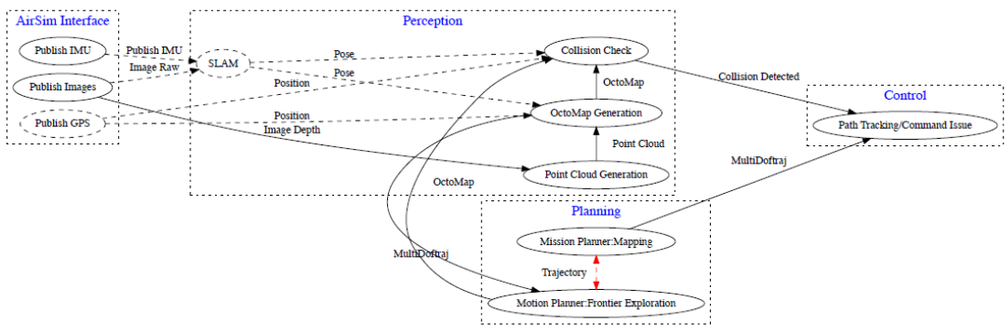

Figure 10: package delivery flow

Every application of MAVbench has three stages: Perception, Planning and Control. In short, perception stages will read environment data, generate necessary data for planning such as OctoMap. For planning, it will choose the best route, do some smoothing for stability; finally, control stage controls the real flight and makes it stable.
During perception and planning stages, many ROS packages are important to do a variety of tasks. For instance, to achieve SLAM (Simultaneous localization and mapping), there is a ROS program (running on companion computer) responsible for it. In MAVbench, ROS plays an essential role; it contains many ROS packages (SLAM, point cloud generation, OctoMap, mav-trajectory) from other source as well as original ROS applications (scanning, mapping,etc).

5.1.1 Overview for code

Package delivery is a relatively complicated application because it involves many small tasks such as scanning and generating trajectory based on map. This section goes into its main .cpp file (the file responsible for generating the final ROS application) to understand how it behaves generally.
The structure of the main cpp file consists of preparation part and a finite state machine. After necessary setup, the program goes into the finite state machine, which controls the drone at different stages. Following part is a summary and snippet of the finite state machine:

- For(State state = setup; ros::ok())
- If (state == setup): Get_goal, get_stat, next_state = waiting.....
- If (state == waiting): Request_trajectory,.....
- If (state = flying): do something
- If (state = trajectory_completed): do something
- state = next_state

As I mentioned above, there are many ROS interactions integrated with the code. For instance, when we try to request trajectory when state is waiting, what we do is actually let ROS call a ROS program called Request_trajectory, and ROS will help to communicate with this program; if the calling is successful, ROS will return the generated trajectory to the cpp file so we can use it.

5.2 Case study: scanning

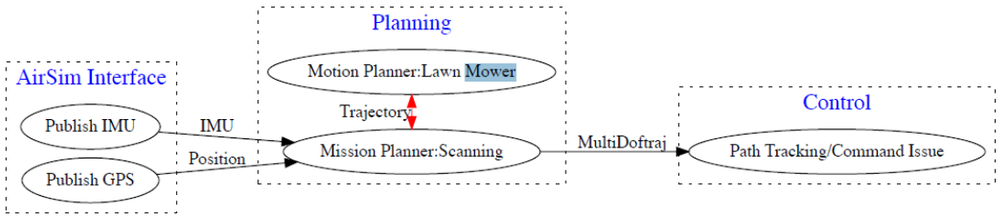

Figure 11: scanning flow

The scanning application is quite different from package delivery application. It does not need to plan too much as the scanning environment has less obstacles. It is like a simple "lawnmower" path planning application. This difference is well demonstrated by lines of code in the main cpp file: 293 lines vs 587 lines.

5.3 Rebuild MAVbench from its Github

It is not easy to create a tool, such as MAVbench; it is even harder to organize the code and merge them correctly. The MAVbench Github repository, at least up to now, still contains some error in its setup file. During my study I tried to fix the problems, but failed to do so. I believe there will be a day when we would rebuild all their work from Github repository.

5.4 Summary

From the goal section, we can see MAVbench is a great platform for measuring the power usage of drone. To modify it so we can create a similar platform for Picar (it already provides us with plenty background material), there are some things we need to change, and area less studies we need to develop. The following thoughts are what I have during studying the MAVbench paper, and hope them can inspire myself in the future.

1. As MAVbench team does not apply anything on GPU, how could we run their computing intensive work on GPU?

2. Because many of their work is ROS package, if we run them on GPU by changing them into CUDA program, how can we communicate and profile them using ROS (is there a way to combine CUDA and ROS)?

3. MAVbench simulates a lot of things, while many of them are already built in Airsim, the platform developed by Microsoft. Compared with it, our Picar platform is totally built by ourselves, so to simulate it in Unreal engine, how could we do this?

A Jetson TX2 Datasheet

Description	Jetson TX2 Series System-on-Module*	
	TX2	**TX2i**
Pascal GPU ◊		
256-core GPU | End-to-end lossless compression | Tile Caching | OpenGL® 4.6 | OpenGL® ES 3.2 | Vulkan® 1.0 | CUDA® 9.0		
Maximum Operating Frequency	1.12GHz	
CPU Complex ‡		
ARMv8 (64-bit) heterogeneous multi-processing (HMP) CPU architecture; two CPU clusters (6 processor cores) connected by a high-performance coherent interconnect fabric. NVIDIA Denver 2 (Dual-Core) Processor: L1 Cache: 128KB L1 instruction cache (I-cache) per core; 64KB L1 data cache (D-cache) per core | L2 Unified Cache: 2MB ARM® Cortex® -A57 MPCore (Quad-Core) Processor: L1 Cache: 48KB L1 instruction cache (I-cache) per core; 32KB L1 data cache (D-cache) per core | L2 Unified Cache: 2MB		
Maximum Operating Frequency per Core NVIDIA Denver 2 ARM Cortex-A57	 2.0GHz 2.0GHz	 1.95GHz 1.92GHz
HD Video & JPEG		
Video Decode (Number of Streams Supported): H.265 (†): Main 10, Main 8 H.265 (†): Main 444 H.264 (†): Baseline, Main, High H.264 (†): MVC Stereo (per view) VP9 (†): Profile 0 (8-bit) and 2 (10 and 12-bit) VP8: All MPEG1/2: Main MPEG4: SP/AP VC1: SP/MP/AP	 (2x) 2160p60 | (4x) 2160p30 | (7x) 1080p60 | (14x) 1080p30 2160p60 | (2x) 2160p30 | (3x) 1080p60 | (7x) 1080p30 (2x) 2160p60 | (4x) 2160p30 | (7x) 1080p60 | (14x) 1080p30 2160p60 | 2160p30 | 1080p60 | 1080p30 (2x) 2160p60 | (4x) 2160p30 | (7x) 1080p60 | (14x) 1080p30 2160p60 | (2x) 2160p30 | (4x) 1080p60 | (8x) 1080p30 2160p60 | (2x) 2160p30 | (4x) 1080p60 | (8x) 1080p30 (4x) 1080p60 | (8x) 1080p30 (2x) 1080p60 | (4x) 1080p30	
Video Encode (Number of Streams Supported): H.265 H.264: Baseline, Main, High WEBM VP9 WEBM VP8	 2160p60 | (3x) 2160p30 | (4x)1080p60 | (8x) 1080P30 2160p60 | (3x) 2160p30 | (7x) 1080p60 | (14x) 1080p30 2160p30 | (3x) 1080p60 | (7x) 1080p30 2160p30 | (3x) 1080p60 | (6x) 1080p30	
JPEG (Decode & Encode)	600 MP/sec	
Audio Subsystem		
Industry-standard High Definition Audio (HDA) controller provides a multi-channel audio path to the HDMI interface | 4 x I2S | DMIC | DSPK | 2 x I and Q baseband data channels | PDM in/out		
Display Controller Subsystem		
Support for DSI, HDMI, DP and eDP | Two multi-mode eDP/DP/HDMI outputs.		
Captive Panel		
MIPI-DSI (1.5Gbps/lane) — Max Resolution	Support for Single x4 or Dual x4 links | 2560x1600 at 60Hz	
eDP 1.4 (HBR2 5.4Gbps) — Max Resolution	3840x2160 at 60Hz	
External Display		
HDMI 2.0a/b (6Gbps) — Max Resolution	3840x2160 at 60Hz	
DP 1.2a (HBR2 5.4 Gbps) — Max Resolution	3840x2160 at 60Hz	
Imaging System		
Dedicated RAW to YUV processing engine process up to 1.4Gpix/s | MIPI CSI 2.0 up to 2.5Gbps (per lane) | Support for x4 and x2 configurations (up to 3 x4-lane or 6 x2-lane cameras)		
Clocks		
System clock: 38.4 MHz | Sleep clock: 32.768 KHz | Dynamic clock scaling and clock source selection		
Boot Sources		
Internal eMMC and USB (recovery mode)		

COPYRIGHT © 2014 – 2018 NVIDIA CORPORATION. ALL RIGHTS RESERVED.

Description	Jetson TX2 Series System-on-Module*	
	TX2	**TX2i**
Security		
Secure memory with video protection region for protection of intermediate results | Configurable secure DRAM regions for code and data protection | Hardware acceleration for AES 128/192/256 encryption and decryption to be used for secure boot and multimedia Digital Rights Management (DRM) | Hardware acceleration for AES CMAC, SHA-1, SHA-256, SHA-384, and SHA-512 algorithms | 2048-bit RSA HW for PKC boot| HW Random number generator (RNG) SP800-90 | TrustZone technology support for DRAM, peripherals | SE/TSEC with side channel counter-measures for AES | RSA-3096 and ECC-512/521 supported via PKA		
Memory ††		
128-bit DRAM interface | Secure External Memory Access Using TrustZone Technology | System MMU | ECC (enabled by software for TX2i only)		
Memory Type	4ch x 32-bit LPDDR4	
Maximum Memory Bus Frequency (up to)	1866MHz	1600MHz
Memory Capacity		
Storage		
eMMC 5.1 Flash Storage		
Bus Width	8-bit	
Maximum Bus Frequency	200MHz (HS400)	
Storage Capacity	32GB	
Connectivity (TX2 only)		
WLAN		
Radio type	IEEE 802.11a/b/g/n/ac dual-band 2x2 MIMO	–
Maximum transfer rate	866.7Mbps	–
Version level	4.1	–
Maximum transfer rate	3MB/s	–
Networking		
10/100/1000 BASE-T Ethernet | IEEE 802.3u Media Access Controller (MAC) | Embedded memory		
Peripheral Interfaces Δ		
XHCI host controller with integrated PHY: (up to) 3 x USB 3.0, 3 x USB 2.0 | USB 3.0 device controller with integrated PHY | 5-lane PCIe: two x1 and one x4 controllers | SATA (1 port) | SD/MMC controller (supporting eMMC 5.1, SD 4.0, SDHOST 4.0 and SDIO 3.0): **1 x SD/SDIO at connector (TX2)**, **2 x SD/SDIO at connector (TX2i)** | 5 x UART | 3 x SPI | 8 x I^2C | 2 x CAN | 4 x I2S: support I^2S, RJM, LJM, PCM, TDM (multi-slot mode) | GPIOs		
Operating Requirements ♦		
Temperature Range	-25C – 80C	-40C to 85C
Module Power	7.5W (Max-Q) / 15W (Max-P)	10W (Max-Q) / 20W (Max-P)
Power Input	5.5V – 19.6V	9.0V – 19.6V
Applications		
Intelligent Video Analytics, Drones, Robotics, Industrial automation, Gaming, and more.		

* Refer to the software release feature list for current software support.

◊ GPU Maximum Operating Frequency: 1.3GHz supported in boost mode for Jetson TX2 and 1.23GHz for Jetson TX2i
Product is based on a published Khronos Specification and is expected to pass the Khronos Conformance Process. Current conformance status can be found at www.khronos.org/conformance.

‡ CPU Maximum Operating Frequency: 1-4 core = up to 2.0GHz; greater than 4-core = up to 1.4GHz

(†) For max supported number of instances: bitrate not to exceed 15 Mbps per HD stream (i.e., 1080p30), overall effective bitrate is less than or equal to 240 Mbps

†† Dependent on-board layout. Refer to *Jetson TX2/TX2i OEM Product Design Guide* for layout guidelines.

Δ Refer to *Jetson TX2/TX2i OEM Product Design Guide* and *Parker Series SoC Technical Reference Manual* to determine which peripheral interface options can be simultaneously exposed.

♦ Refer to the *Jetson TX2/TX2i OEM Product Design Guide* and the appropriate version of the Thermal Design Guide (either Jetson TX2 or TX2i) for evaluating product power and thermal solution requirements. See the software documentation for information on changing the default power mode (default: Max-P).

 COPYRIGHT © 2014 – 2018 NVIDIA CORPORATION. ALL RIGHTS RESERVED.

References

[1] CUDA-contributors, "Cuda–introduction." https://www.geforce.com/hardware/technology/cuda. Accessed on Dec 2018.

[2] Wikipedia-contributors, "Embedded system — Wikipedia, the free encyclopedia." https://en.wikipedia.org/wiki/Embedded_system, Dec 2018. Accessed on Dec 2018.

[3] Behzad Boroujerdian et al., "Mavbench:micro aerial vehicle benchmarking." https://d.pr/f/fqspYT. Accessed on Dec 2018.

[4] Mark Harris, "An easy introduction to cuda c and c++." https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/, Oct 2012. Accessed on Dec 2018.

[5] Picar contributor, "Picar project." https://github.com/xz-group/PiCar. Accessed on Dec 2018.

[6] kangalow, "Nvpmodel – nvidia jetson tx2 development kit." https://www.jetsonhacks.com/2017/03/25/nvpmodel-nvidia-jetson-tx2-development-kit/. Accessed on Dec 2018.