PROFILING NVIDIA JETSON EMBEDDED GPU DEVICES FOR AUTONOMOUS MACHINES

Yazhou Li[1] and Yahong Rosa Zheng[2]

[1]School of Computer Science and Engineering, Beihang University, Beijing, China
[2]Department of Electrical and Computer Engineering, Lehigh University, Bethlehem, PA, 18015, USA

ABSTRACT

This paper presents two methods, tegrastats GUI version jtop and Nsight Systems, to profile NVIDIA Jetson embedded GPU devices on a model race car which is a great platform for prototyping and field testing autonomous driving algorithms. The two profilers analyze the power consumption, CPU/GPU utilization, and the run time of CUDA C threads of Jetson TX2 in five different working modes. The performance differences among the five modes are demonstrated using three example programs: vector add in C and CUDA C, a simple ROS (Robot Operating System) package of the wall follow algorithm in Python, and a complex ROS package of the particle filter algorithm for SLAM (Simultaneous Localization and Mapping). The results show that the tools are effective means for selecting operating mode of the embedded GPU devices.

KEYWORDS

Nvidia Jetson, embedded GPU, CUDA, Automous Driving. Robotic Operating Systems (ROS).

1. INTRODUCTION

NVIDIA Jetson is a complete embedded system-on-module (SoM) device that integrates the CPU, GPU, PMIC, DRAM, and flash storage on a small-form-factor platform. The current Jetson series include Jetson Nano, Jetson TX2, and Jetson Xavier (NX and AGX), which are commonly used for edge computing, autonomous machines, machine learning, and artificial intelligence. An example application is the toy race car F1/10 which is of 1/10 of a real car size (https://f1tenth.org) and uses a Jetson TX2 as its computing hardware, as shown in Fig. 1, where the Connect Tech Obitty Carrier board is on top of the Jetson TX2 module, the power distribution board is on top of the VESC motor controller, and the Lidar is in the front of the car. All electronics are mounted on a plate above the brushless motor and battery compartment, making it a compact and powerful platform for testing autonomous driving algorithms. The Jetson device can connect to host computer via WiFi antennas and a dedicated WiFi router with static IP address assignment is recommended.

David C. Wyld et al. (Eds): CSEA, DMDBS, NSEC, NETWORKS, Fuzzy, NATL, SIGEM - 2020
pp. 133-144, 2020. CS & IT - CSCP 2020 DOI: 10.5121/csit.2020.101811

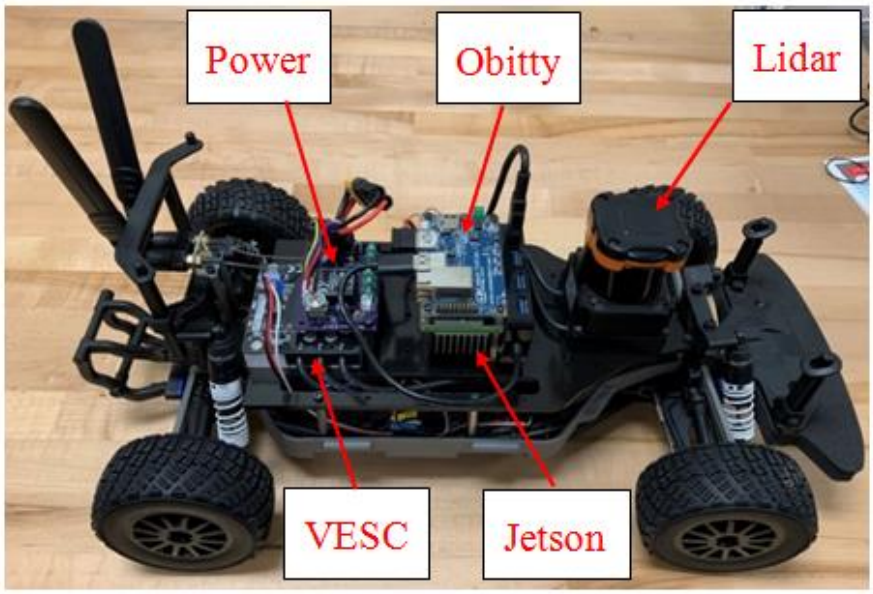


Figure 1. The F1/10 race car with Nvidia Jetson TX2, Connect Tech Obitty Carrier board, Hokuyo UMT-30 Lidar, and VESC 6 MK-III motor controller.

With the embedded GPUs, the NVIDIA Jetson systems provide the performance and power efficiency to run autonomous machines software faster and with less power than CPU-only embedded systems. However, how do we profile the performance of the Jetson devices? How do we use the profiling results to help improve programs written for Jetson devices? Answering these questions requires some serious effort because Jetson embedded devices require special versions of profilers than the commonly used nvidia-smi utility and Nsight Systems for desktop or workstations.

In this paper, we explore two methods for profiling the performance of Jetson TX2 8GB: one is the tegrastats utility and its graphical APIs which can be used directly on the Jetson device; one is Nsight Systems for Tegra target systems which is used on a host computer to remote access the Jetson device.

Several coding examples are used to bench mark the performance of Jetson TX2 under different working modes: a vector-add program in C or CUDA C, a wall-follow python program for f1/10 race cars [1], and a particle filter algorithm for Simultaneous Localization and Mapping (SLAM) [2]. The profilers measure the power consumption, the run time of the CUDA C threads, and the CPU/GPU utilization under five operating modes of the Jetson TX2. The program files used in this paper can be found at https://github.com/li630925405/jetson-profile.

1.1. Jetson TX2 Working Modes

The Jetson TX2 consists of a 256-core Pascal GPU along with a CPU cluster. The CPU cluster consists of a dual-core NVIDIA 64-bit Denver-2 CPU and a quad-core ARM Cortex-A57. By configuring the 6 CPU cores and the GPU, a Jetson TX2 typically runs in five different modes, as shown in Table 1, where the GPU is enabled in all five modes, but at different clock speeds. The two types of CPUs are enabled or disabled in different configurations. Different modes show different speed-power performance trade-offs. It is clear that the Max-N mode is the fastest and consumes the most power as all CPUs and GPU are enabled and they run at their maximum speeds.

Table 1. Working modes of typical Jetson devices configurable by the NVPmodel utility [3].

Mode	Mode Name	Denvor 2 CPU Core		ARM A57 Core		GPU Frequency
		# of Cores	Frequency	# of Cores	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	1	2.0 GHz	1	2.0 GHz	1.12 GHz

The Jetson operation modes are enabled by the Dynamic Voltage and Frequency Scaling (DVFS) technology and can be configured at run time by a command-line tool NVPModel [3]. For example, we use 'sudo nvpmodel -m 2' to change the working mode of a Jetson to Mode 2 Max-P Core-All mode. The configuration of the five modes are saved in the file /etc/nvpmodel. conf which can also be customized to produce user flexible modes. The current mode of the Jetson is queried by 'sudo nvpmodel -q –verbose'.

1.2. NVIDIA Profiling Tools

NVIDIA profiling tools help the developers to optimize their programs and applications. The newly announced NVidia Nsight Systems [4], [5] supersedes the command-line nvprof and visual profiler NVVP tools and combine them into one unified tool. To apply it to profile Jetson devices, NSight Systems for Tegra target system package has to be used on the host computer which remote accesses the jetson device profilee. The package is part of the JetPack 4.3 installation [6] and is installed on the host computer. Alternatively, a command-line utility 'tegrastats' and its graphical APIs [7], [8] are available to profile Jetson devices directly. In addition, user-defined functions or python wrappers may utilize the system APIs such as timer to profile the performance.

```
robot-1@robot-1:~$ tegrastats
RAM 2268/7861MB (lfb 49x4MB) SWAP 0/3930MB (cached 0MB) CPU [6%@345,off,off,2%@345,6%@345,5%@345] EMC_FREQ 0% GR3D_FREQ 0%
PLL@39C MCPU@39C PMIC@100C Tboard@36C GPU@38C BCPU@39C thermal@38.6C Tdiode@36.25C VDD_SYS_GPU 143/143 VDD_SYS_SOC 286/286
VDD_4V0_WIFI 305/305 VDD_IN 1909/1909 VDD_SYS_CPU 95/95 VDD_SYS_DDR 229/229
```

Figure 2. profile jetson GPU status using tegrastats

The tegrastats utility reports memory usage and processor usage for Jetson-based devices [7] similar to the Nvidia-smi utility which is not supported on Jetson. An example of tegrastats is shown in Figure 2, which means CPU2 and CPU3 are off, CPU1, CPU5 and CPU6 are using 6% of their loads, CPU4 is using 2% of its load and their current running frequency is 345 MHz. The details of the tegrastats output is explained in [7], but it is rather user unfriendly. The tegrastats output can be visualized by a tool called jetson_stats [8], a package that combines both tegrastas and NVPmodel into a GUI to profile and control Jetson. It contains five different tools, among which the jtop tool is the most useful for profiling.

Nsight Systems for Tegra targets is a part of the JetPack SDK [9] for Jetson devices and is installed on a host computer to remotely profile the Jetson target device. Nsight Systems is a low-overhead sampling, tracing and debugging tool for C or CUDA C programs and may not be effective to profile a python program.

Therefore, we rely on the system APIs such as high resolution clock in the C++ library and time.time() in the python library and write custom functions and evaluate the performance of python code on Jetson devices.

2. HARDWARE PLATFORM AND SAMPLE PROGRAMS

The Jetson TX2 device in the F1/10 race car platform [1] is used to profile the example programs. The Jetson TX2 is flushed with the Ubuntu 18.04.4, ROS Melodic, and Jetpack 4.3 L4T 33.3.1 packages. The F1/10 race car ROS simulator is also installed. We profile three examples on the f1/10 race car platform: Example 1 is the vector-add program in two versions [10]: vector_add.cpp and vector_add.cu. The cpp version uses the CPU cores only, while the cu version explicitly utilizes the GPU parallel programming APIs and memory management via CUDA C/C++.

Example 2 is a wall follow python program [11] running on the f1/10 simulator platform [1]. As shown in Fig. 3, the simulator takes the map and drive information to generate the Lidar sensing data. It also interfaces the joy pad or keyboard inputs for control and runs the autonomous algorithms via the New Planners module and the /nav and /brake topics. The RViz package is used to visualize the race track, the car motion, and the Lidar sensing data.

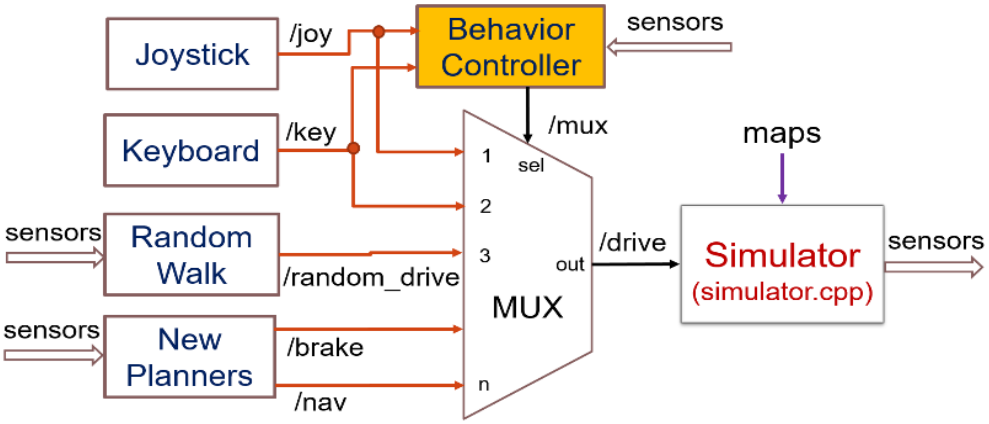


Figure 3. Block diagram of the F1/10 race car ROS simulator.

The wall follow python program subscribes to the LidarScan data and computes the distance of the race car to the left or right wall, then determines the PID control parameters according to a pre-set distance, and publishes the Ackermann steering parameters to the /drive topic to control the race car. When the car receives data from the lidar, the function lidar_callback will be called to calculate the control signal and send it to control the car. So the speed of wall follow program can be represented by time consumed by lidar_callback.

Example 3 is a particle filter algorithm for SLAM and is available at github [2]. The python code implements a fast particle filter localization algorithm and uses the RangeLibc library for accelerated ray casting. By specifying the range method "rmgpu", the program explicitly use the GPU to achieve fast ray casting, while other range methods, such as "bl", "rm", and "glt", etc., use the CPUs only. The bench mark performed in C++ is reprinted in Table. 2. The range method options can be specified in localize.launch of the particle filter package.

Table 2. Different ray casting algorithms in the particle filter example [2].

Method	Init time (sec)	Random queries Thousand/ sec	Grid queries Thousand/ sec	Memory (MB)
BL	0.02	297	360	1.37
RM	0.59	1,170	1,600	5.49
RMGPU	0.68	18,000	28,000	5.49
CDDT	0.25	2,090	4,250	6.34
PCDDT	7.96	2,260	4,470	4.07
LUT	64.5	2,160	4,850	296.6

3. TEGRASTATS PROFILING PROCEDURES AND RESULTS

The performance of Example 1 is profiled by three methods: the average time to run the program is extracted by the clock() function in cpp and cu versions of the vector_add programs; the jetson-stats is used to extract the tegrastats results on CPU/GPU utilization and power consumption and graph them; and Nsight Systems is used to trace the APIs of GPU synchronization and memory management in details.

The performance of Example 2 is evaluated by two methods: the first is to utilize a python wrapper with python library function time.time() to calculate time spent for each function in the python program; the second is to use jetson-stats to graph the CPU usage and power consumption. The wall-follow program runs for minutes and the python wrapper outputs more than 60,000 calls of each function and the time of each function run is averaged over the total numbers of the calls. The python wrapper is disabled when using jetson-stats to profile Example 2.

Example 3 has a built-in performance calculator and it outputs the average number of iterations of ray casting in the particle filter algorithms. The jetson-stats is used to profile its power consumption and CPU/GPU utilization.

3.1. Example 1: Vector Add

The two versions of vector-add programs are run on Jetson TX2 in modes 0 -- 4, respectively. Although all modes have the GPU enabled, only the vector_add.cu makes use of the GPU, while vector_add.cpp utilizes the CPU cores only. The results of the execution time is shown in Table 2, where the time is measured by high resolution clock from the C++ library "chrono". Without utilizing the GPU, Modes 0, 2 and 4 run in similar speeds which are faster than Modes 1 and 3. This is because Modes 0, 2 and 4 use the Denver-2 and Modes 1 and 3 use ARM cores only. The example code is a single-threaded kernel and Denver 2 CPU has better single-threaded performance than the ARM core.

Table 3. Power consumption of Example 1 in different Jetson modes.

Mode	GPU	CPU power	GPU power	IN power
0	No	1983	95	7151
0	Yes	567	2315	8229
1	No	524	95	2527
1	Yes	190	1045	5155
2	No	1426	95	4660
2	Yes	379	1706	6568
3	No	1237	95	4187
3	Yes	331	1707	6663
4	No	1900	95	5108
4	Yes	521	1706	6682

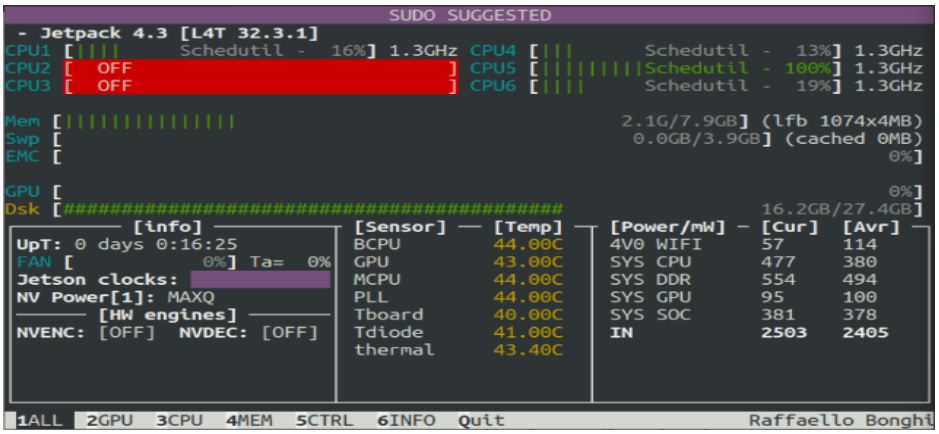
Utilizing the GPU, all five modes run at the similar speed as most of the processing is offloaded to the GPU running at similar speeds in all modes. Comparing the performance between the CPU+GPU and the CPU only modes, performances vary significantly in different GPU execution configurations, as shown in Table 3. Note that the streaming multiprocessors in Jetson TX2 is 2 and the maximum number of thread per block is 1024. Also note that the size of the vectors to be added is 224. With a small number of thread per block and a small number of blocks per grid, the performance of the GPU version is worse than the CPU only version, as the potential of the GPU is under utilized and the overhead of memory management is relatively large. Let the number of threads per block and the number of block per grid be N_t and N_b, respectively. If we increase N_t or N_b such that $N_tN_b \geq 211$, then the GPU performs faster than the CPU only version, as shown in Table 4. Therefore, the try and error method is used to find out the best combination of the numbers of threads and blocks.

Table 4. Time needed to run Example 1 with different CUDA execution configurations

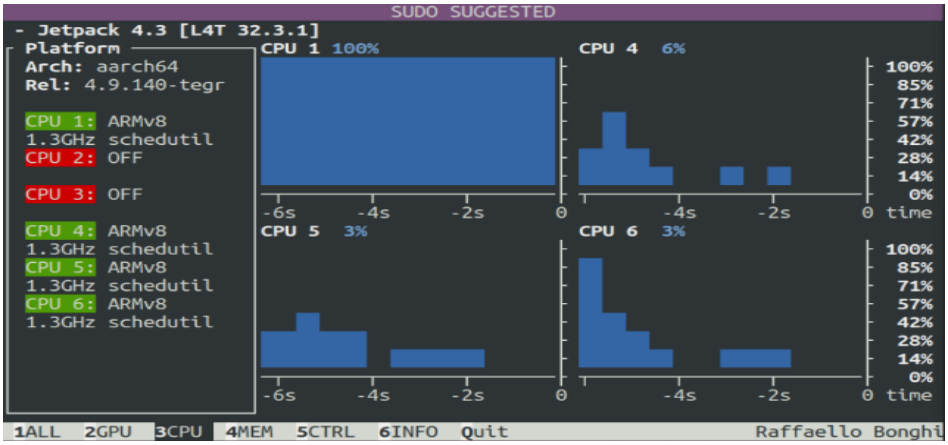
# threads / block	# blocks /grid	run time (s)
32	2	240
128	2	71
1024	2	15
32	64	15

The two versions of vector_add programs are also profiled by tegrastats via jtop and the results for Mode 1 (Max-Q) are shown in Fig. 4 and 5. Mode 1 uses all four ARM cores at 1.2 GHz clock rate while the GPU can run at 0.85 GHz speed. The CPU version of the program utilizes all four ARM cores and its GPU utilization is 0%, as shown in Fig. 4a. CPU 2 and 3 are off as they are the Denver 2 cores which are disabled in Mode 1. The average power consumption of the program in Mode 1 is 2405 miliwatts. It is interesting to note that the GPU still consumes 100 mW power on average even though the program does not utilize the GPU. This is because the GPU is used by Jetson to support graphics in the operating system.

The CPU utilization is further detailed by going to menu 3CPU at the bottom of the jtop window, as shown in Fig. 4b, where the time snap shots of the CPU utilization is captured. All four ARM cores run at the 1.3 GHz clock rate and CPU 1 is utilized fully at 100% at the time of the capture, while other three ARM cores are used occasionally. Observing the CPU graphs over time reveals that the four ARM cores are utilized uniformly by the "schedutil".



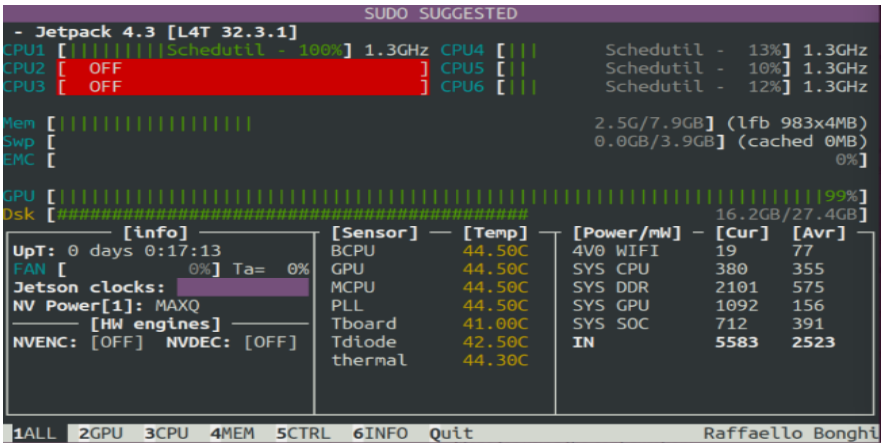
(a) Overall performance of vector_add.cpp w/o using the GPU



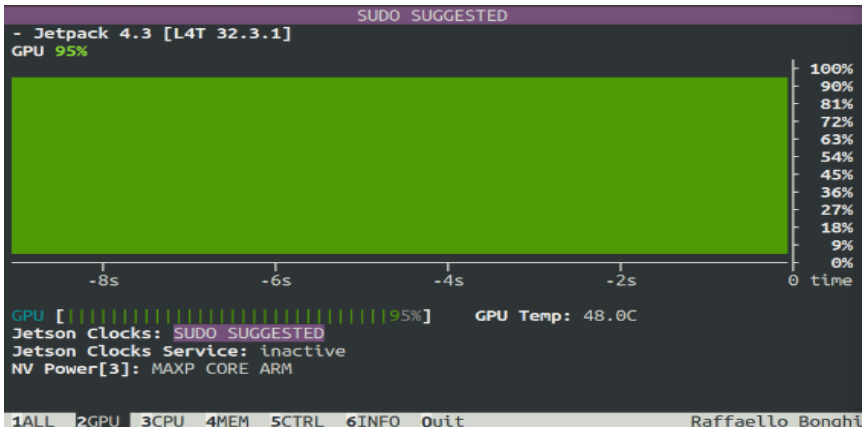
(b) CPU utilization of vector_add.cpp w/o using the GPU

Figure 4. Results profiled by jtopfor vector_add.cpp program which uses the CPU only. Jetson mode 1 was used to run the program.

In comparison, vector_add.cu utilizes the GPU almost fully at 99%, as shown in Fig. 5a. The CPU1 ARM core is also fully utilized, while other ARM cores are used by 10% -- 13%. The power consumption of the CPU+GPU mode is 2523 milliwatts which is slightly higher than the CPU only version. Note, jtop menu 5CTRL also provides a user-friendly interface for the NVPmodel utility so the user can change the Jetson operation mode and CPU/GPU clock rates easily.



(a) Overall performace of vectoradd.cu utilizing the GPU



(b) GPU utilization of vector_add.cu using the GPU

Figure 5. Results profiled by jtop for vector_add.cpp program which uses the CPU only. Jetson mode 1 was used to run the program.

The profiling results of vector_add.cu using Nsight Systems is shown in Fig. 6. The host computer runs the Nsight Systems while the Jetson TX2 is running the vector_add.cu. The host computer remote accesses the Jetson via SSH to profile the performance of each CPU core and GPU thread. It is clear that the utilization of CPU is decreased when the GPU kernel function is being executed.

3.2. Example 2: Wall Follow Python Code

The wall-follow python code is run on the f110 simulator with time.time() to calculate the time consumed in each function. The car is placed at the right bottom corner of the track when the wall-follow program starts. In all Jetson modes except Mode 4, the car completes the whole track without collision. The time spent on function lidar_callback is shown in Table 5. It is clear that Mode 4 runs the slowest because it only uses 2 CPUs, thus the car reacts slowly and collides to the wall easily; No differences are observed in car behaviors in other modes, although the times spent by modes 0 - 3 are slightly different and Mode 1 is notably slower than Mode 0, 2 and 3.

Figure 6. Nishgt System profiling timeline results for Example 1

Table 5. Time spent by the lidar_callback function of Example 2 in five working modes of Jetson TX2

Function	Mode 0	Mode 1	Mode 2	Mode 3	Mode 4
lidar_callback	626us	967us	511us	541us	1010us

The jtop profiler results are shown in Fig. 7 with Mode 4 as an example. Since only one Denver2 core and one ARM core are enabled in Mode 4, both of the CPU cores are utilized at 100% and the GPU is utilized at 10% mainly to support Rviz graphical display.

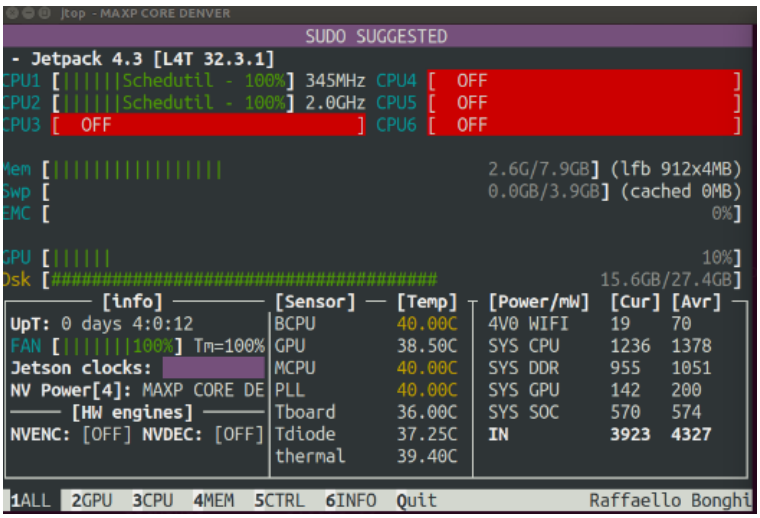
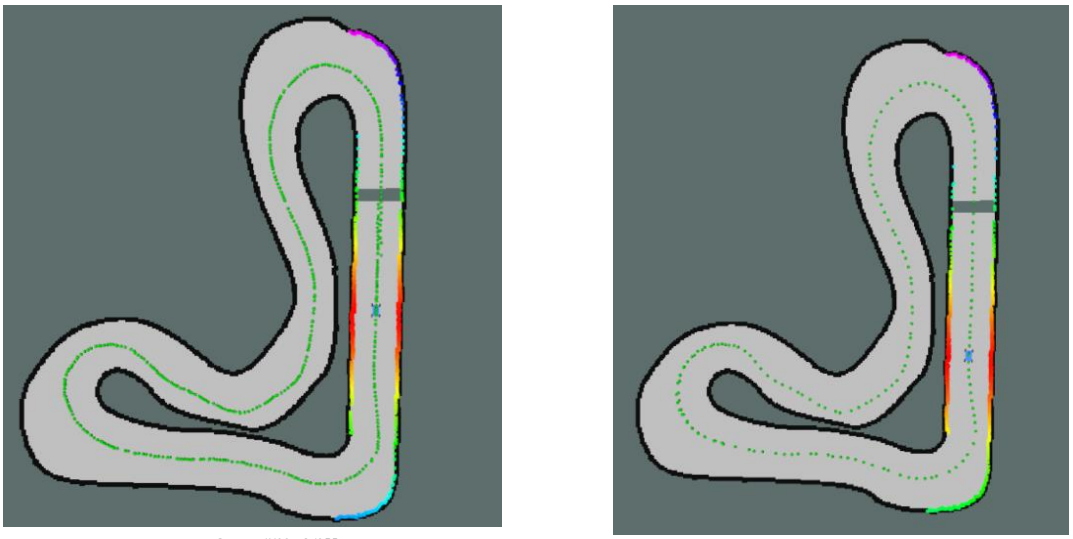


Figure 7. Performance of Example 2: Wall follow python program run in Mode 4

3.3. Example 3: Particle Filters

The particle filter algorithm is run in the f110 simulator on the Jetson TX2 device first, and then run in ROS directly with the hardware. The operating Mode 0 is used for both cases. In the f110 simulator, the car runs the wall-following algorithm through one lap of the track while the particle filter runs simultaneously with a way-point logger, so that the localized waypoints are logged in to a file as the car running through the track. When the range method is set to "rm" (ray marching), the points in the logged file are sparse, as shown in Fig. 8b, because the algorithm runs slow with CPU cores only. When the range method is set to "rmgpu" (ray marching using GPU), the particle filter performance is faster than the "rm" option, and the number of way points logged in the track is much higher, as shown in Fig. 8a.



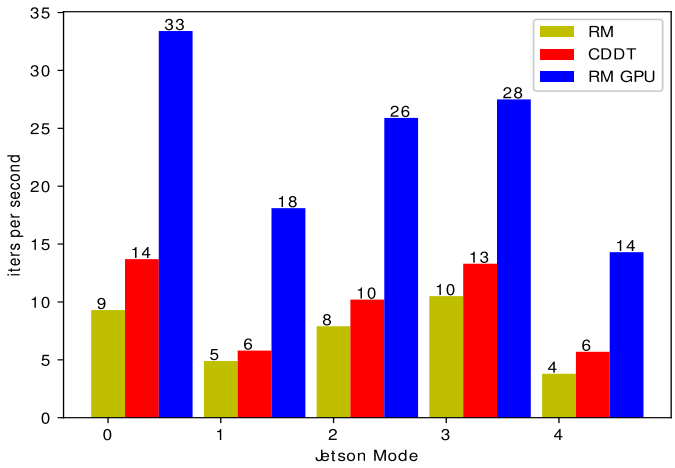
a. rm gpu b. rm

Figure 8. Way points generated by ray casting methods: ray marching with or without GPU. The number of way points is proportional to the processing speed of the algorithm.

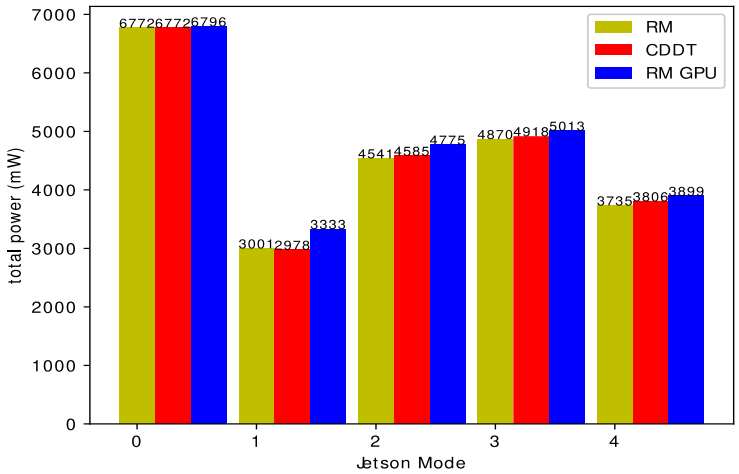
The speed and power efficiency of Example 3 in different modes are shown in Fig. 9. Utilizing the GPU, the rmgpu method gains about 2x to 4x speedup over the CDDT and rm methods in all Jetson modes, as shown in Fig. 9a. Meanwhile, the power consumption of the rmgpu method is slightly more than the rm and CDDT methods running in the same operation mode, as shown in Fig. 9b. This demonstrates that it is beneficial to utilize the GPU when ever is possible.

4. CONCLUSION

This paper has investigated two profiling tools for Jetson devices using three examples that involve CUDA C, python or ROS packages. The two profiling tools are tegrastats and Nsight Systems. While tegrastats and its GUI version jtop can run directly on the Jetson devices, the Jetson version of Nsight Systems has to run on a host computer to remotely profile a Jetson devices. Jtop provides summary of the Jetson CPU and GPU utilization, power consumption and dynamic flow in coarse time scale, while Nsight Systems can provide detailed CPU/GPU and memory activities of individual threads in fine time scale. Therefore, Nsight Systems is a better tool for performance optimization. Performance of five working modes of Jetson TX2 has been compared to illustrate the capabilities of the two profiling tools.



(a) iters per second



(b) power consumption

Figure 9. Example 3 Performance in different modes: RM and CDDT run w/o GPU, rmgpu runs with GPU fully utilized.

REFERENCES

[1] University of Pennsylvania. F1/10 race car. [Online]. Available: https://f1tenth.org

[2] MIT-racecar. Particle filter localization. Retrieved Feb. 2020. [Online]. Available:https://github.com/mit-racecar/particle_filter/blob/master/README.md

[3] J. Benson. NVPModel - NVIDIA Jetson TX2 DevelopmentKit. [Online]. Available: https://www.jetsonhacks.com/2017/03/25/nvpmodel-nvidia-jetson-tx2-development-kit/

[4] NVIDIA. (2020, Feb.) Nvidia releases nsight systems 2020.2. [Online]. Available: https://news.developer.nvidia.com/nvidia-announces-nsight-systems-2020-2/

[5] S. McMillan. (2019, Aug.) Transitioning to nsight systems from nvidia visual profiler/nvprof. [Online]. Available: https://devblogs.nvidia.com/transitioning-nsight-systems-nvidia-visual-profiler-nvprof/

[6] NVidia. Profiler user's guide. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[7] NVIDIA. Tegrastats utility. [Online]. Available: https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-231/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/AppendixTegraStats.html

[8] R. Bonghi. Jetson stats. [Online]. Available: https://github.com/rbonghi/jetsonstats

[9] NVidia. Jetpack SDK: Jetpack 4.3. [Online]. Available: https://developer.nvidia.com/embedded/jetpack

[10] M.Harris. (2017, Jul.) Unified memory for CUDA beginners. [Online]. Available: https://devblogs.nvidia.com/unified-memory-cuda-beginners/

[11] f1tenth team. f1tenth racecar wall follow. [Online]. Available: https://github.com/f1tenth/f110_ros/tree/master/wall_follow