GPU Optimization of Bayesian Localization Framework

Richmond J.D. Bhaskaran Gowri Ramachandran

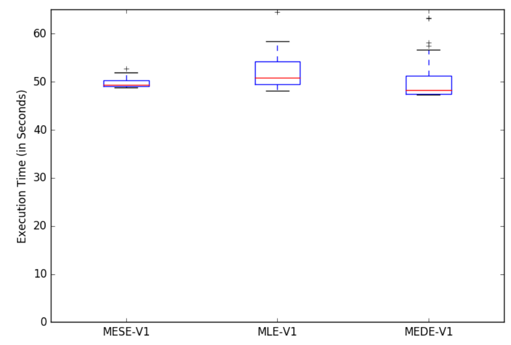
May 7, 2020

Abstract

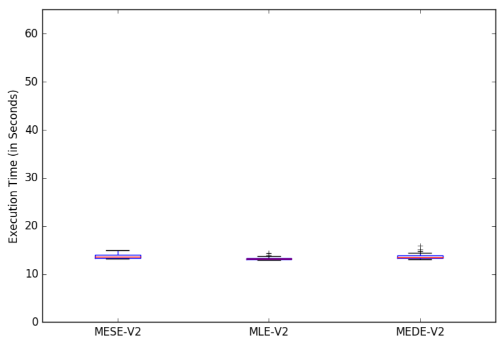
Bayesian localization framework locate the position of RF source based on RSSI measurements. The framework developed at the Autonomous Networks Research Group (ANRG) research lab was very slow and did not leverage GPU platforms. In particular, the location estimation function took between 12 seconds and 42 seconds on contemporary laptops. We have optimized the location estimation function using Nvidia GPUs and the CUDA library. This report presents the different optimization approaches and their performance, including run-time and energy consumption. The important finding from our evaluation is that the number of threads in the GPU may not necessarily lead to faster execution times - application developers has to carefully tune the thread configurations to achieve faster execution with optimal power consumption.

1 Introduction

The localization algorithm is expected to predict the location of the UAV in real-time to take effective action. However, our preliminary implementation of the Bayesian algorithm, developed in Python, requires tens of seconds to execute and estimate the position of the RF source. The long processing time is due to the implementation based on Python's built-in math libraries. Figure 1 summarizes the execution time of the Bayesian framework. We observe that such long estimation time is consistent with the literature [2]. Estimating the position of a moving RF source requires a much faster execution.



((a)) Execution Time of Bayesian Framework-Version 1.



((b)) Execution Time of Bayesian Framework-Version 2.

Figure 1: Execution Time of Bayesian Framework-Version 1 and Version 2.

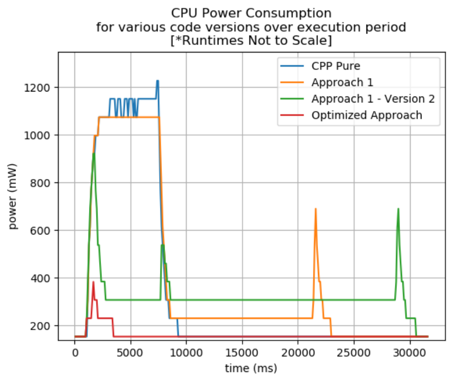
We optimized the Bayesian framework (Version-2) by executing its component operations using low-level libraries, providing support for parallel execution leveraging GPUs. The execution time of

Parameter	**Jetson TX2**	**GTX1060**	**GTX 1080 Ti**
Number of Parallel Multiprocessors	2	10	28 (Streaming Multiprocessors)
Number of Cores	256	1280	3584
Processor Frequency	1.3 GHz	1.48 GHz	1.5 GHz

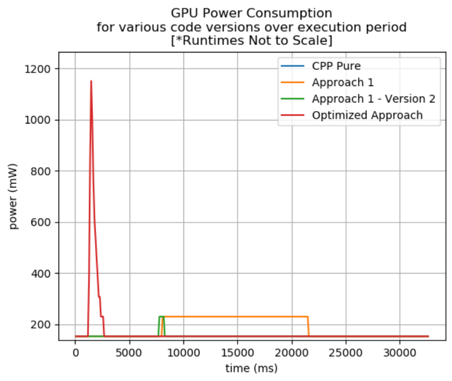
Table 1: High-level Overview of Nvidia GPUs Used in Our Evaluation.

the optimized version is 4 to 5 times faster compared to the previous version. As shown in Fig. 1 (right), the average execution time is 12 seconds.

In this report, we describe the different optimization approaches and their performance improvements. We report results from multiple Nvidia GPUs - Jetson TX2, a laptop running Nvidia-GTX-1060 GPU, and a server machine running Nvidia-1080 GPU. Table 1 provides a high-level overview of platforms used in our evaluation.

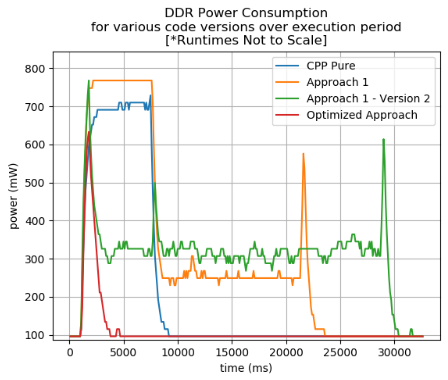


((a)) CPU Power Consumption over entire time period

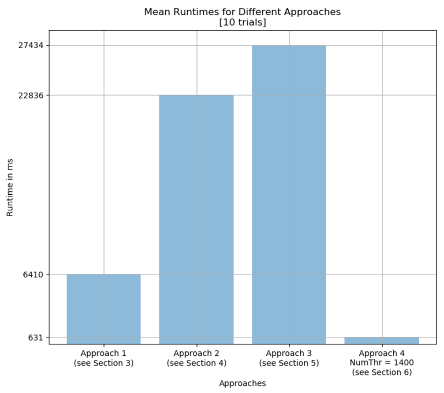


((b)) GPU Power Consumption over entire time period

Figure 2: Power Consumption and Runtime for various code versions in Jetson TX2

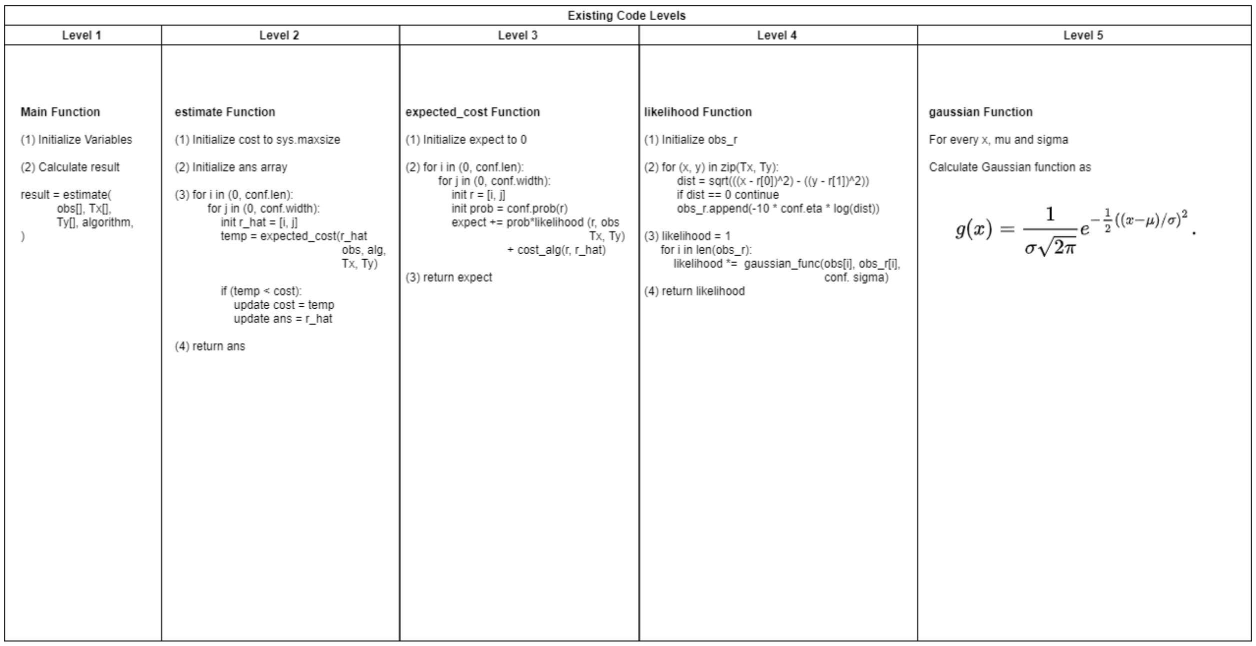


((a)) DDR Power Consumption over entire time period



((b)) Average Runtime over 10 trials

Figure 4: Power Consumption and Runtime for various code versions in Jetson TX2



Existing Code Levels

Level 1

```
Main Function

(1) Initialize Variables

(2) Calculate result

result = estimate(
        obs[], Tx[],
        Ty[], algorithm,
)
```

Level 2

```
estimate Function

(1) Initialize cost to sys.maxsize

(2) Initialize ans array

(3) for i in (0, conf.len):
        for j in (0, conf.width):
            init r_hat = [i, j]
            temp = expected_cost(r_hat
                                 obs, alg,
                                 Tx, Ty)

            if (temp < cost):
                update cost = temp
                update ans = r_hat

(4) return ans
```

Level 3

```
expected_cost Function

(1) Initialize expect to 0

(2) for i in (0, conf.len):
        for j in (0, conf.width):
            init r = [i, j]
            init prob = conf.prob(r)
            expect += prob*likelihood (r, obs
                                       Tx, Ty)
                    + cost_alg(r, r_hat)

(3) return expect
```

Level 4

```
likelihood Function

(1) Initialize obs_r

(2) for (x, y) in zip(Tx, Ty):
        dist = sqrt(((x - r[0])^2) - ((y - r[1])^2))
        if dist == 0 continue
        obs_r.append(-10 * conf.eta * log(dist))

(3) likelihood = 1
    for i in len(obs_r):
        likelihood *=  gaussian_func(obs[i], obs_r[i],
                                     conf. sigma)
(4) return likelihood
```

Level 5

gaussian Function

For every x, mu and sigma

Calculate Gaussian function as

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}((x-\mu)/\sigma)^2}.$$

Figure 5: Datapath for the existing code and callstack

2 Optimization Goal

The first version of the Bayesian framework requires between 45 and 60 seconds depends on the processing capacity of the hardware platform. And, this version does not use multi-threading and only uses the CPU. Figure 1 shows the execution time results of this version. This high execution time is due to the overhead introduced by the Python Interpreter layer and the sequential nature of the calculations that do not have any parallel constructs at all.

The second version of the framework performed better due to limited optimizations provided by Numba framework [1]. This version takes between 12 and 20 seconds, as shown in Figure 1. The Numba optimizer converts the function that it is applied on into a native C implementation. Therefore, it provides a certain level of increase in speed. This version is better than the previous version, because the overhead provided by passing through the Python Interpreter is avoided, and this version produces a C implementation upon the function it is applied on. However the nature of the calculation remains largely sequential.

Our goal is to optimize the Bayesian framework using GPU and enable parallel operations involving multiple threads to bring down the execution time.

3 Optimization Approach 1: Code Rewritten in C++

The low-level programming languages such as C and C++ are capable of producing optimal code, compared to high-level languages such as Python and Java. Therefore, we optimized the Bayesian framework by rewriting the entire framework in C++ without leveraging the GPU. This version takes about 4 seconds in 1080 and 1060 platforms (and nearly 6 seconds in Jetson TX2 platform) to run as a compiled binary using Nvcc, which is the default NVIDIA compiler (see Figure 2 and Figure 3), and between 28 and 30 seconds when run in debug mode in Visual Studio. Note Visual Studio was chosen because it is the default coding environment in Windows, where the Nvcc compiler is readily integrated. However, this version of the code uses only the CPU. Visual Studio is used as the IDE because of its support for the nvcc compiler.

In this version, the entire code is converted into C++. The execution speed was reduced in certain areas because of the standard template library (STL) items such as vectors and maps, which are optimized for ease of use. This version can be improved further if a much better code is written, but the execution speed will still be hampered due to sequential operations by the CPU.

We have chosen C++ because its execution speed is comparable to C while being flexible compared to high-level languages such as Python and Java. Besides, the source libraries for MQTT is also available in C++. MQTT is a publish-subscribe communication framework to allow different nodes in the network to exchange data between each other. Our Bayesian framework uses it to send and receive data between other nodes in the system.

4 Optimization Approach 2: Optimized C++ Code with GPU

We further optimized the Bayesian framework using GPU in this version. In particular, we identified Gaussian calculation segment to be the computationally heavy process in the pipeline, shown in Figure **??**. Therefore, we run the deepest level on the pipeline inside the GPU. Here, the initial stages of the computations are handled by the CPU. Compared to the previous version, the main design changes are as follows:

- CPU populates the entire matrix for which the calculation needs to be performed.
- From the populated matrix, the CPU feeds the GPU memory with 3000 entries at a time.
- The GPU acts on that data and produces the intermediary output. From that point, the CPU is responsible for producing the expected output.

This version takes approximately 4 seconds in 1080 and 1060 platforms (and nearly 22 seconds in Jetson TX2). In total, 9000 threads are performing 3000 Gaussian calculations at a time. C++ driver loads a large calculation matrix and then transfers it to GPU to perform the calculations. Once the GPU completes the calculation, it moves the result back to the CPU.

Initially, it was expected that this version would provide a better performance because a test bench code consisting of 2.25 Million Gaussian calculations was tested using the same threads and GPU configurations. The entire calculation was performed in 500 milliseconds. But, the execution time increases for the whole process since the CPU consumes a lot of time to populate the data and prepare the GPU for the computation.

The timing breakdown is given below:

- CPU Population time before sending to GPU is approximately 1.5 seconds.
- GPU calculation time is roughly 2 seconds.
- CPU Final Answer derivation time is approximately 1 second.

It is important to note that the above results are for 1060Ti and 1080 Nvidia GPUs only. For NVIDIA Jetson TX2, the performance is at 22 seconds because of the low computing power of the onboard CPU ((see Figure 2 and Figure 3).

5 Optimization Approach 3: Bursted Load and Run with GPU

This version is the same as the previous version, with the following difference. As soon as partial sets of input elements are produced by the CPU, it is loaded onto the GPU, the GPU is called to perform the calculations. Once the GPU has calculated the Gaussian values for that set of inputs, it is transferred back to the CPU. Now the CPU goes on to produce the next set and eventually derives the final answer and presents the output.

We thought that the performance would be improved as we do not wait for the CPU to populate the entire inputs, and then fire the GPU. Instead, we call the GPU as soon as 3000 entries are generated. But the performance remained the same ((see Figure 2 and Figure 3). But this is no different than the previous version since the CPU is waiting for the output of i^{th} batch of 3000 entries before producing $(i+1)^{th}$ batch entries for the GPU. However, there can be some promise in performance, if the CPU is allowed to produce the $(i+1)^{th}$ round of 3000 entries while the GPU works on the i^{th} set. This is possible if multi-threading is introduced in the CPU C++ environment. But considering that the maximum amount of multi-threading in CPU is around 4 or 8 threads, the runtime is not going to improve by more than 4 or 8 times.

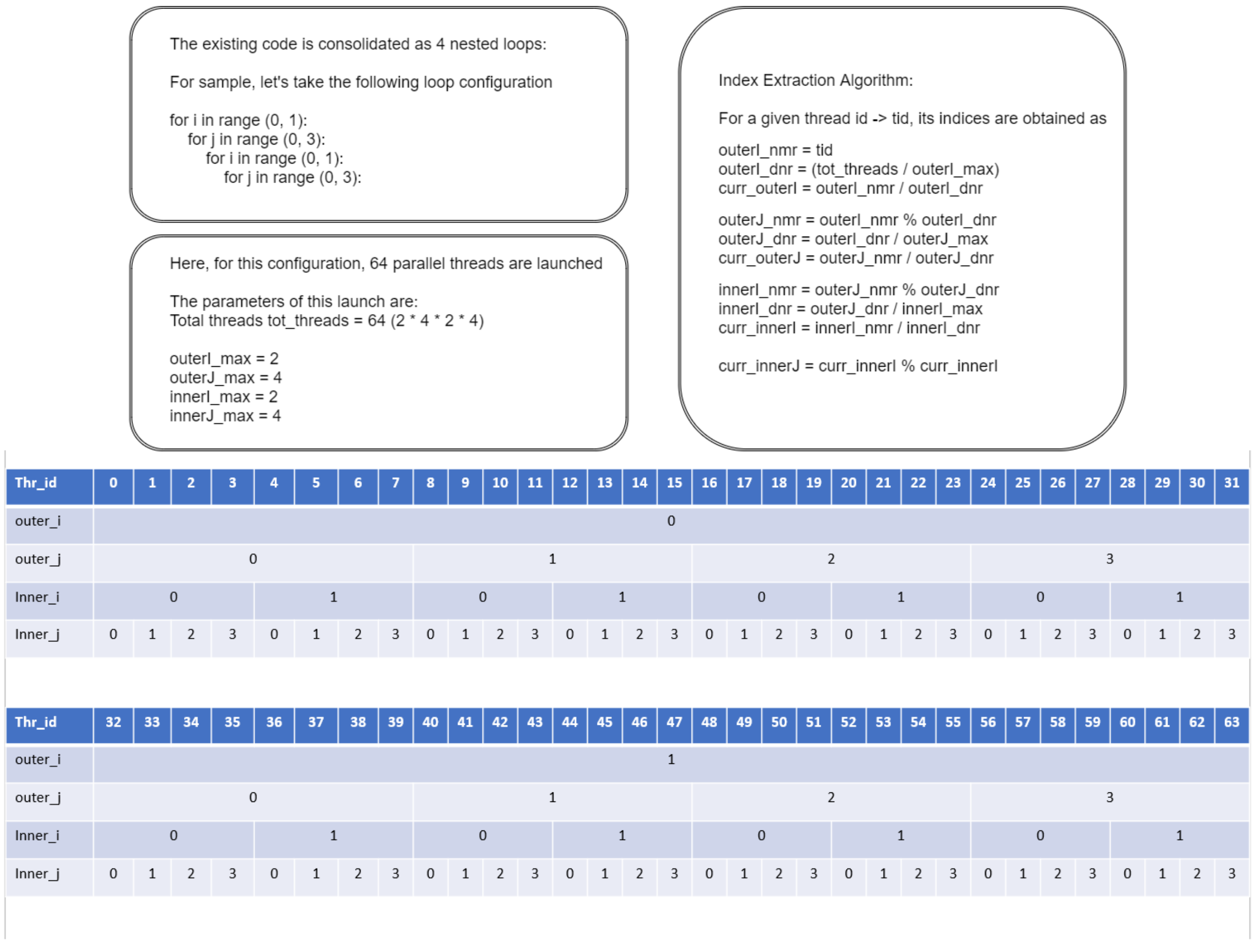
6 Optimization Approach 4: GPU-driven Localization

With the improvised algorithm, having near-zero CPU to GPU transfer time, the entire localization code is done by the GPU. Theoretically, 4096 parallel GPU threads can be launched. The GPU performs the entire calculation involving multiple threads and then transfers the result to CPU, which then deduces the final value.

It was identified that the CPU populating the values from the C++ layer was the primary reason for the slowdown in previous versions. Hence, it was decided that the entire workload calculation will be transferred to the GPU. In this version, we reduced the amount of data transferred to the GPU to near zero by loop unrolling in the earlier stages. Also, by using a sub-algorithm, namely the "Index Extraction Algorithm" (shown in Figure 6) to deduce array indices, the speed of the algorithm increases significantly while having zero transfer times.

One of the most critical observations from the previous versions is that, because the GPU needs to be invoked only in the Gaussian calculating stage, the input matrix needs to be populated by the CPU and then transferred to the GPU. Until Level 3, the amount of resources transferred is near zero. The nested loops start after level 3, and then subsequently, a large matrix is generated at level 4. In this version, the GPU is called at Level 3, without creating the large matrix. Instead, individual threads produce the necessary values and even perform the Gaussian calculation for that instance. Since the GPU is called at Level 3, the amount of resources that are copied from CPU memory to GPU memory is also reduced. The different levels in the execution and the data path are presented in Figure 5.

The data given to the GPU must be serialized. It is recommended that a multidimensional matrix be reduced to a single row matrix and then sent to the GPU for processing. Therefore the levels of nested loops from the original code need to be unrolled and flattened before GPU invocation. Although this flattening happened in an implicit sense in the previous versions, it did not matter much because the GPU was invoked in the terminal Gaussian calculation only. But in this version of the code, the GPU is expected to do much of the calculation work. Therefore every thread that is launched must need to know where it needs to fetch from the flattened input matrix and also needs to know where it needs to deposit in the output matrix. For this purpose, every thread that is launched needs to associate its thread id with its corresponding loop headers.



The existing code is consolidated as 4 nested loops:

For sample, let's take the following loop configuration

```
for i in range (0, 1):
    for j in range (0, 3):
        for i in range (0, 1):
            for j in range (0, 3):
```

Here, for this configuration, 64 parallel threads are launched

The parameters of this launch are:
Total threads tot_threads = 64 (2 * 4 * 2 * 4)

outerI_max = 2
outerJ_max = 4
innerI_max = 2
innerJ_max = 4

Index Extraction Algorithm:

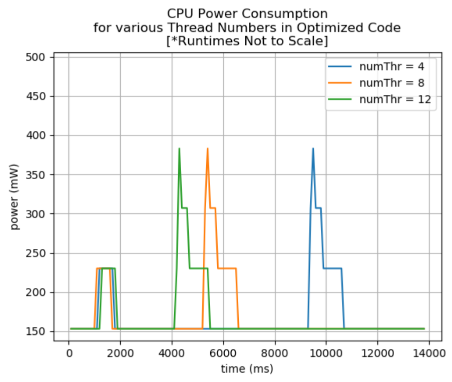
For a given thread id -> tid, its indices are obtained as

```
outerI_nmr = tid
outerI_dnr = (tot_threads / outerI_max)
curr_outerI = outerI_nmr / outerI_dnr

outerJ_nmr = outerI_nmr % outerI_dnr
outerJ_dnr = outerI_dnr / outerJ_max
curr_outerJ = outerJ_nmr / outerJ_dnr

innerI_nmr = outerJ_nmr % outerJ_dnr
innerI_dnr = outerJ_dnr / innerI_max
curr_innerI = innerI_nmr / innerI_dnr

curr_innerJ = curr_innerI % curr_innerI
```

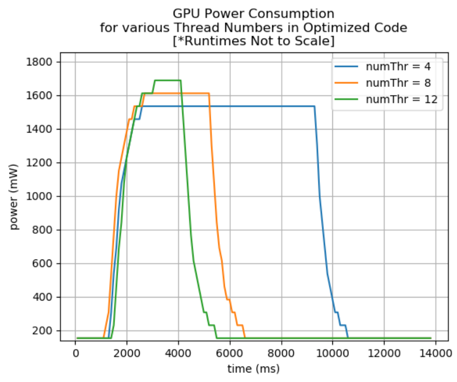
Thr_id	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
outer_i	0																															
outer_j	0								1								2								3							
Inner_i	0				1				0				1				0				1				0				1			
Inner_j	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

Thr_id	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
outer_i	1																															
outer_j	0								1								2								3							
Inner_i	0				1				0				1				0				1				0				1			
Inner_j	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

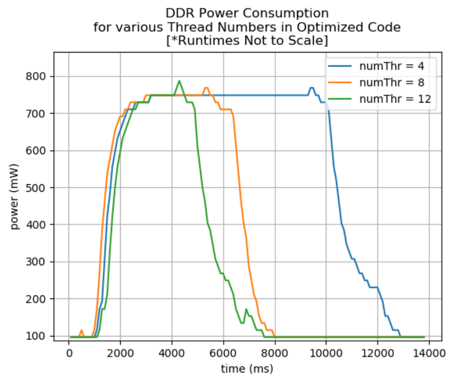
Figure 6: Index Extraction Algorithm



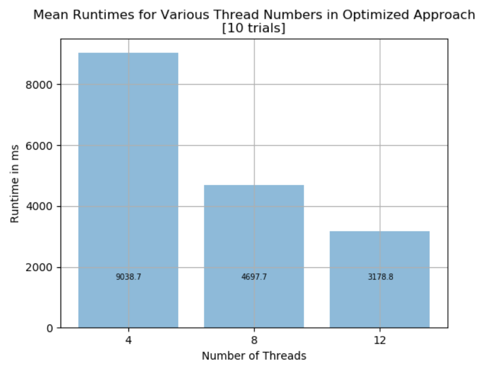
((a)) CPU Power Consumption over entire time period



((b)) GPU Power Consumption over entire time period



((c)) DDR Power Consumption over entire time period

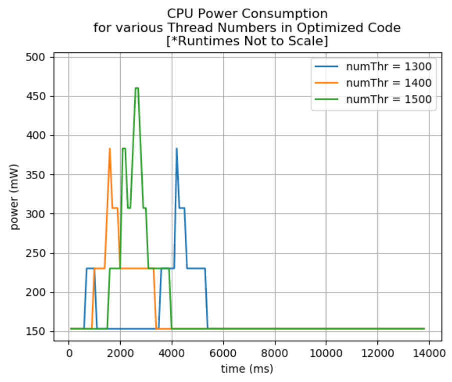


((d)) Average Runtime over 10 trials

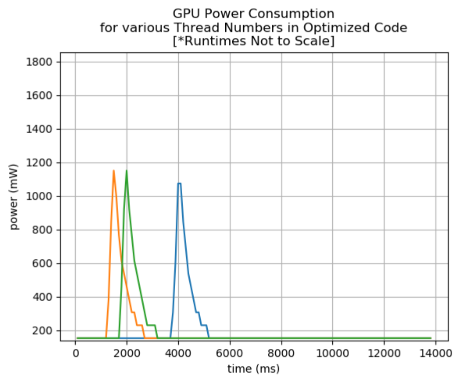
Figure 7: Power Consumption and Runtime for Experiments Conducted with 4, 8, and 12 Threads in Jetson TX2

If the loop headers are to be passed as an input for every element in the input matrix, 4 loop headers need to be appended. Such an approach instantly increases the input matrix size by 4, ultimately increasing the CPU to GPU input matrix transfer time, which defeats the purpose. Therefore, there needs to be an algorithm that can determine the loop headers based on the thread id. Figure 6, presents the "Index Extraction Algorithm," which explains how our algorithm reduces the execution time. It does leverage the number of parallel threads that can be launched on the underlying GPU platforms, but the improvements in runtime far outweigh the degree of parallelism.

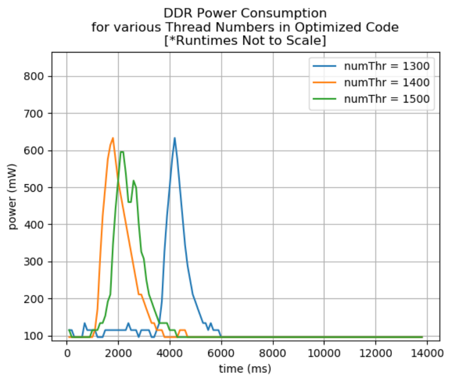
This version takes between 250 and 300 milliseconds on 1060 and 1080 Nvidia GPUs, while it takes approximately 650 milliseconds on the Nvidia Jetson TX2 platform (((see Figure 2 and Figure 3). By transferring the entire workload to the GPU, the amount of resources used by a thread increases significantly. This effect is visible in lower-end or edge GPUs such as Jetson TX2, wherein the maximum number of parallel threads that can be launched is approximately 3000. Therefore, there is a risk that the device runs out of memory very often if too many threads are started in parallel. Hence, the number of parallel threads needs to be reduced. There is a trade-off between how much workload we want to transfer to the GPU vs. the number of parallel threads that can be run.



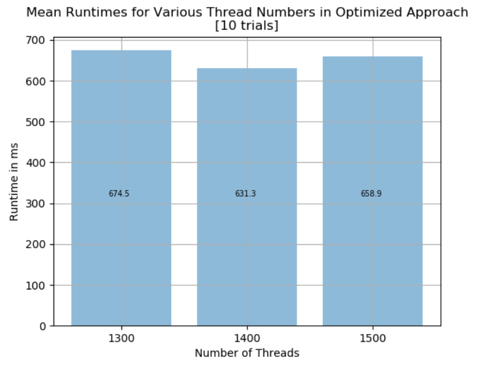
((a)) CPU Power Consumption over entire time period



((b)) GPU Power Consumption over entire time period



((c)) DDR Power Consumption over entire time period



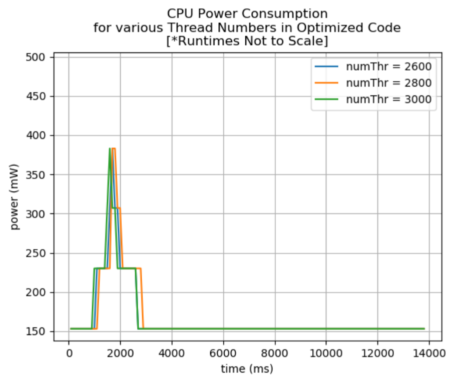
((d)) Average Runtime over 10 trials

Figure 8: Power Consumption and Runtime for Experiments Conducted with 1300, 1400, and 1500 Threads in Jetson TX2

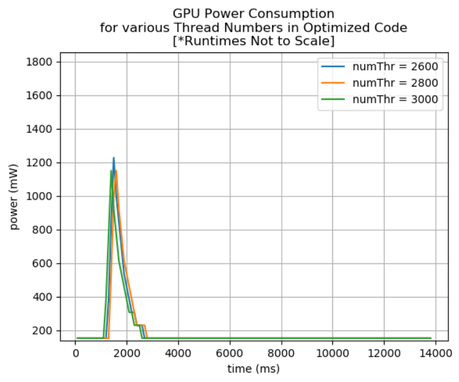
7 Results for Nvidia Jetson with Different Thread Configurations

To understand the relationship between the number of threads, power consumption, and execution time, we have performed an evaluation using 4, 8, 12, 1300, 1400, 1500, 2600, 2800, and 300 threads. Figure 7, Figure 8, Figure 9 show the CPU power consumption, GPU power consumption, DDR power consumption, and average run-time for the different configurations. As seen from Figure 10, the most optimized thread configuration lies in 1400 threads parallel threads. However, a few values from the upper and lower ends of the spectrum were chosen to understand the variance of critical parameters.

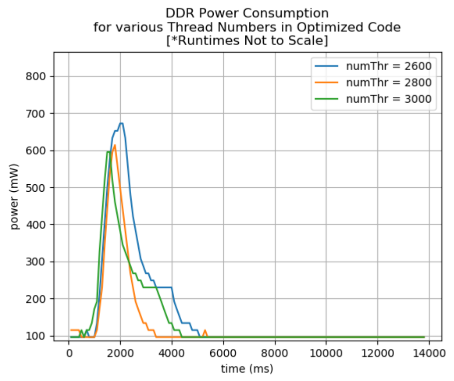
When threads are scheduled, they are done in blocks which contain a finite number of threads per block, The total number of cores in a GPU is separated into units of 128 cores called as a Streaming Multiprocessor (SM). At any time, two blocks run on an SM. Jetson TX2 supports 2 SMs, and each block can hold a maximum of 1024 parallel threads. Therefore, theoretically, 4096 parallel threads can be launched at once. When a block is scheduled, each block has access to shared memories and caches that will be shared up by all threads in that block. A scheduler expects the blocks, and a block execution goes out of scope only when all the threads in it have completed their execution. Because of this, some threads may need to wait until all threads in a block are fully executed.



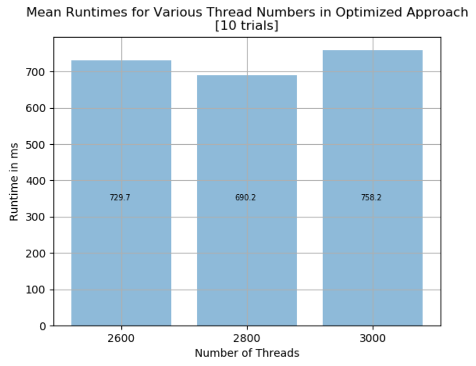
((a)) CPU Power Consumption over entire time period



((b)) GPU Power Consumption over entire time period



((c)) DDR Power Consumption over entire time period



((d)) Average Runtime over 10 trials

Figure 9: Power Consumption and Runtime for Experiments Conducted with 2600, 2800, and 3000 Threads in Jetson TX2

7.1 Evaluation using 4, 8, and 12 Threads

Figure 7 shows the CPU, GPU, and DDR power consumptions together with execution times for 4, 8, and 12 thread configurations in Jetson TX2 platform. When the number of threads is low, the number of parallel blocks that need to be created are high. In this case, when the number of threads is 4, the number of threads per block will be 1, which will lead to massive wastage of resources as the shared memories and caches will be wiped off every time another block is scheduled. Therefore much of the time is lost due to this context switching, resulting in high runtimes. The energy consumption is also high because the GPU operates at high power for a longer runtime.

7.2 Evaluation using 1300, 1400, and 1500 Threads

Figure 8 shows the CPU, GPU, and DDR power consumptions together with execution times for 1300, 1400, and 1500 thread configurations in Jetson TX2 platform. This is the most optimal configuration of thread numbers. At this point, the threads will effectively utilize the shared memories, and at the same time, the waiting time for all the threads in a block is also less. Therefore it is in this region we get the most efficient runtime. The power consumption is also optimal as it does not run to its peak power, as depicted in 4, 8, and 12 thread configurations.

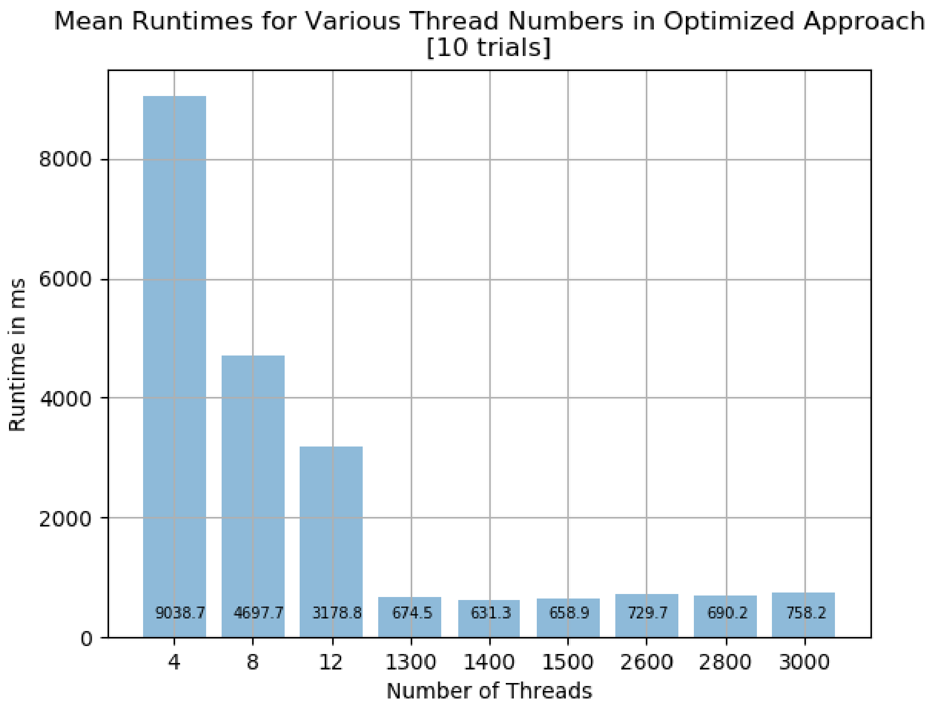


Figure 10: Comparison of runtimes for various thread configurations in Jetson TX2

7.3 Evaluation using 2600, 2800, and 3000 Threads

Figure 9 shows the CPU, GPU, and DDR power consumptions together with execution times for 2600, 2800, and 3000 thread configurations in Jetson TX2 platform. This configuration uses all the threads in the block. Here the shared resources are being utilized at maximum efficiency. Still, the waiting time for the threads to complete in a block and deposit the results are marginally high compared to 1300, 1400, and 1500 approach. The power consumption is almost the same as well.

7.4 Important Observation

Here it is noted that the thread number around 3000 is the maximum number of parallel threads that can be launched. Above that, the shared memories used by the threads in the block are exhausted, i.e., they run out of shared memory, and the GPU kernel code even fails to launch. Such behaviors are attributed to the temporary variables used in the Index Determination Algorithm. Interestingly, if one decides to do away with the algorithm and pass the array headers, from the CPU to GPU, it reduces the number of local variables used by a thread. In this case, we may be able to launch maximum number of parallel threads, but the overhead in performing a CPU to GPU memory copy becomes so large that it results in a terrible performance. Therefore, the trade-off is taken to go for a lesser thread number and use the Index Extraction Algorithm. The trade-off in choosing the thread numbers is also interesting as seen in Figure 10. It is seen that around 1300 - 1500 is an optimal number.

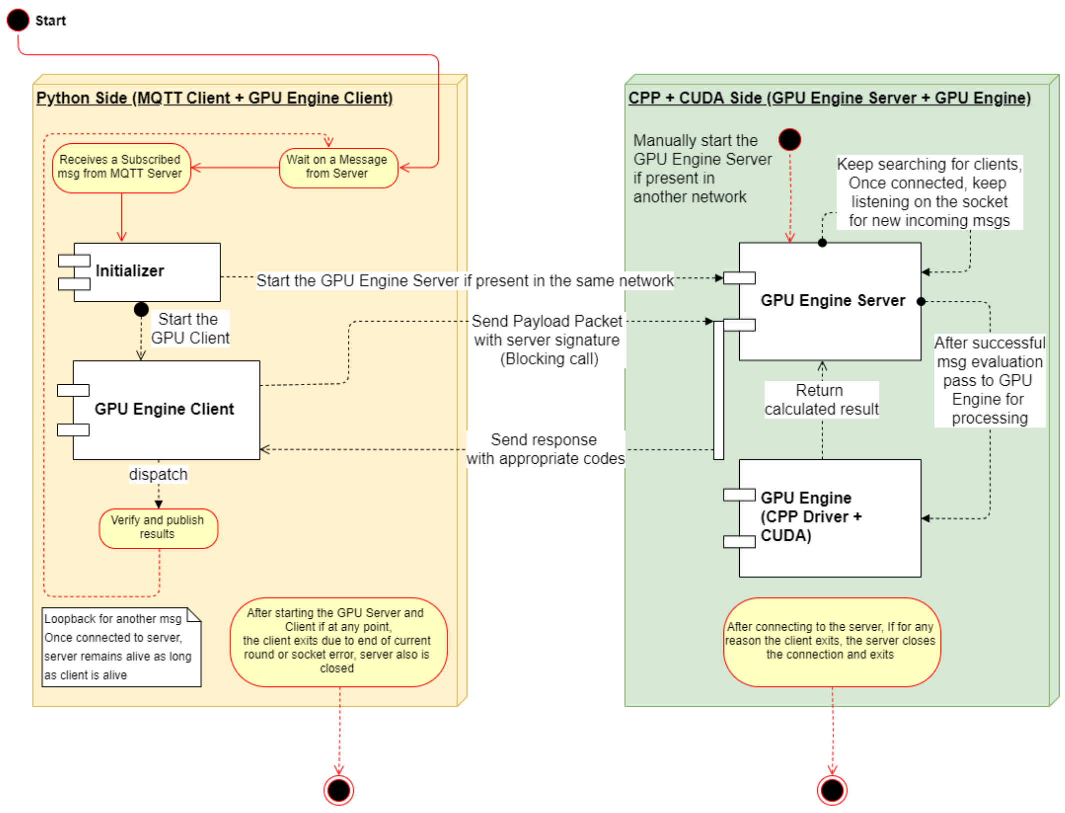

Start
Python Side (MQTT Client + GPU Engine Client)
Receives a Subscribed msg from MQTT Server
Wait on a Message from Server
Initializer
Start the GPU Client
GPU Engine Client
dispatch
Verify and publish results
Loopback for another msg Once connected to server, server remains alive as long as client is alive
After starting the GPU Server and Client if at any point, the client exits due to end of current round or socket error, server also is closed
Start the GPU Engine Server if present in the same network
Send Payload Packet with server signature (Blocking call)
Send response with appropriate codes
CPP + CUDA Side (GPU Engine Server + GPU Engine)
Manually start the GPU Engine Server if present in another network
Keep searching for clients, Once connected, keep listening on the socket for new incoming msgs
GPU Engine Server
Return calculated result
After successful msg evaluation pass to GPU Engine for processing
GPU Engine (CPP Driver + CUDA)
After connecting to the server, If for any reason the client exits, the server closes the connection and exits

Figure 11: Architecture diagram for experiment setup

8 Experimental Setup

The experiment is conducted by integrating with the main experiment setup that provides the input signal strength from 3 sources and the corresponding position is calculated. The experiment is divided into 3 individual components:

1. The signal gathering component
2. The GPU Engine Client Component
3. The GPU Engine Server Component

The signal gathering component lies outside this specific portion of the experimentation and therefore is not covered in the architecture diagram as shown in Figure 11.

8.1 Overall Working

These components work in tandem to deduce the position of the mobile platform. An overall working of this setup is

1. The signal strengths from the 3 sources is gathered by the signal gathering component
2. It is sent via the mqtt sender channel on a specific topic
3. The GPU Engine Client side will receive the message via the mqtt receive channel on the same topic

4. From this point on wards, it will formulate a message format (by adding a number of header data for parsing as well as authentication by the server)
5. The Client shares it with the GPU Engine Server for processing through a TCP port
6. The Server will calculate the result and embed its return message along with some header data
7. Finally the message is received at the GPU Engine Client where the message is further processed to be sent to the requester which is the signal gathering component
8. The return message is sent in another mqtt channel

8.2 The Signal Gathering Component

The signal gathering component is responsible for all the overhead code and circuitry that focuses on data acquisition from the 3 base stations, calculation of their signal strengths, setting up of the mqtt subscribe and receive channels for communication with the mobile device of interest and also the output display section. When this section of code is running, it is inherently assumed that the GPU Engine Client and server is always up and running. It will post a message to the mobile device and will block indefinitely till it receives a response after which it will perform output display functions and subsequently, the next round of data gathering for processing.

8.3 The GPU Engine Client Component

The GPU Engine Client component is an important middle-man such that it orchestrates the connection between the signal components and the actual GPU. Its primary responsibilities are:

1. waiting on an incoming mqtt request to process
2. Initializing the server (if the GPU server and client reside on the same machine. This is an optional step)
3. converting the data into a structure to be processed by the GPU server
4. adding of special keys to be verified by the GPU server
5. Opening of TCP ports to communicate with the GPU server
6. Parsing of return data from the GPU server to post the final response
7. Posting the final result on a seperate mqtt channel
8. Basic Error checking
9. Keeping the client alive by constantly blocking on an mqtt incoming request (This is especially important as the GPU server is liable to end its working if the client also ends the active TCP connection)
10. Various Timing profilers

It is to be noted that the sever and client can reside on the same or different machine and the client can choose to create a localhost connection to the server if required. If that is the case then the GPU server also needs to be started by the client. If the server resides in a different machine, the client can go ahead and trust that the server is already up and running and start forwarding the messages. In either case, the server needs to be set up before the client and start looking out for connections.

8.4 The GPU Engine Server Component

The GPU Engine Server Component is the wrapper C++ code and the actual CUDA code running atop the edge device and its GPU. This section contains the actual code explained in all the previous sections. A most optimized version of the code as in Approach 4 from Section 6 is chosen and is fixed at 1400 parallel threads (1400 parallel threads is chosen based on previous observations which provides with the maximum throughput to minimum power consumption characteristics). The input data arriving at the GPU Engine Server is always assumed to be correct and consisting of the basic information for the server to work on. It should also contain the server key for client authentication. One important aspect is that, once the server has established a connection with a client, it is to be noted that the server is alive as long as the client is alive. When the client program ends, the server also ends its execution. However this is not a concern as the client is constantly blocked on receiving incoming mqtt requests from the signal gathering components and therefore the server is kept alive.

9 Summary

In this report, we have presented how the Bayesian localization framework is optimized using Nvidia GPU platforms. Four different approaches have been investigated involving CPU and GPU platforms. As expected, GPU platforms are capable of speeding up the computation operations. However, the application developers are required to carefully optimize the operations, such as copying data between CPUs and GPUs. Furthermore, the number of threads and blocks must be carefully selected to achieve optimal performance on the GPUs. In other words, a powerful GPU may perform sub-optimally if the developer does not correctly optimize the memory transfer and computation operations. Through several optimization techniques, we have managed to increase the speed of the Bayesian framework by more than ten times without sacrificing core objectives.

References

[1] Lam, S.K., Pitrou, A., Seibert, S.: Numba: A llvm-based python jit compiler. In: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC. p. 7. ACM (2015)

[2] Nguyen, P., Truong, H., Ravindranathan, M., Nguyen, A., Han, R., Vu, T.: Matthan: Drone presence detection by identifying physical signatures in the drone's rf communication. In: Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services. pp. 211–224. MobiSys '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3081333.3081354, `http://doi.acm.org/10.1145/3081333.3081354`

A Experiment Procedure

The following sections contains the necessary procedures to run the experiment and also the location of important files

A.1 Location of files to run:

1. Location of the python client code (Present in the eclipse server PC): `/home/ripa/davidbha/ForIntegration/_V2_FinalIntegration/1080_Python_Client/mqtt-server.py`

2. Location of the CPP + CUDA server code (Present in the Jetson Board): `/home/nvidia/RichmondJohnson/ForIntegration/Integration_Phase1_ForExtendedTests/Jetson_CPP_Server/server`

A.2 Instructions to run:

1. First start the server by logging into the Jetson board
2. Navigate to the server binary preesent in the file path as in Section A.1
3. Start executing the server program using the format: `./server[port_number]`
4. Edit the python client code in the eclipse pc with the server ip and the port number entered in step 1 (In this case, enter the ip address of the Jetson board)
5. Start the python client code and wait for incoming mqtt requests

A.3 To edit thread Numbers or the GPU server code and recompile:

1. It is possible to change the parallel thread number by navigating to `/home/nvidia/RichmondJohnson/ForIntegration/Integration_Phase1_ForExtendedTests/Jetson_CPP_Server/server.cpp` and line number: 82 by editing the variable numThr.
2. The compile command for the entire CPP and CUDA files is available in the file: `/home/nvidia/RichmondJohnson/ForIntegration/Integration_Phase1_ForExtendedTests/Jetson_CPP_Server/Makefile`. To note that this is not a Makefile that can be executed with the make command. cat the contents of the file and enter it in the command line to execute and generate the required output binary

A.4 Tools and Dependencies:

The following are the tools and dependencies required to be set up before beginning with the experimentation

1. First check if the GPU is a CUDA enabled GPU. Get the underlying GPU details from the target machine. Using this information, the NVIDIA official website can be visited to check for CUDA compatibility. This experiment will run only on a CUDA enabled GPU. The link from the NVIDIA official website can be found *here*.
2. The necessary GPU drivers are needed to communicate with the underlying GPU. This can also be done by visiting the NVIDIA official website and comparing the GPU model and downloading the latest and most compatible drivers for the project. The link from the NVIDIA official website can be found *here*.
3. Download the latest CUDA Drivers from the NVIDIA official site as well. Kindly make sure if the latest CUDA versions are stable enough. This procedure will bring with itself the nvcc compiler which can compile both CPP and CUDA programs. The CUDA version used for the project is CUDA 9.1. The link from the NVIDIA official website can be found *here*.
4. When the CUDA drivers are downloaded, inside its folder structure is present a folder called samples folder. Run the deviceQuery file present in it, which will give a complete output of all the physical features of the GPU as in the number of cores and the maximum number of

parallel threads that can be launched. This is a very useful parameter. A sample screenshot is given below in Figure 12 for the NVIDIA Jetson Tx2 board. The last line of the image reflects as to where the deviceQuery program is present although it varies in different versions of CUDA

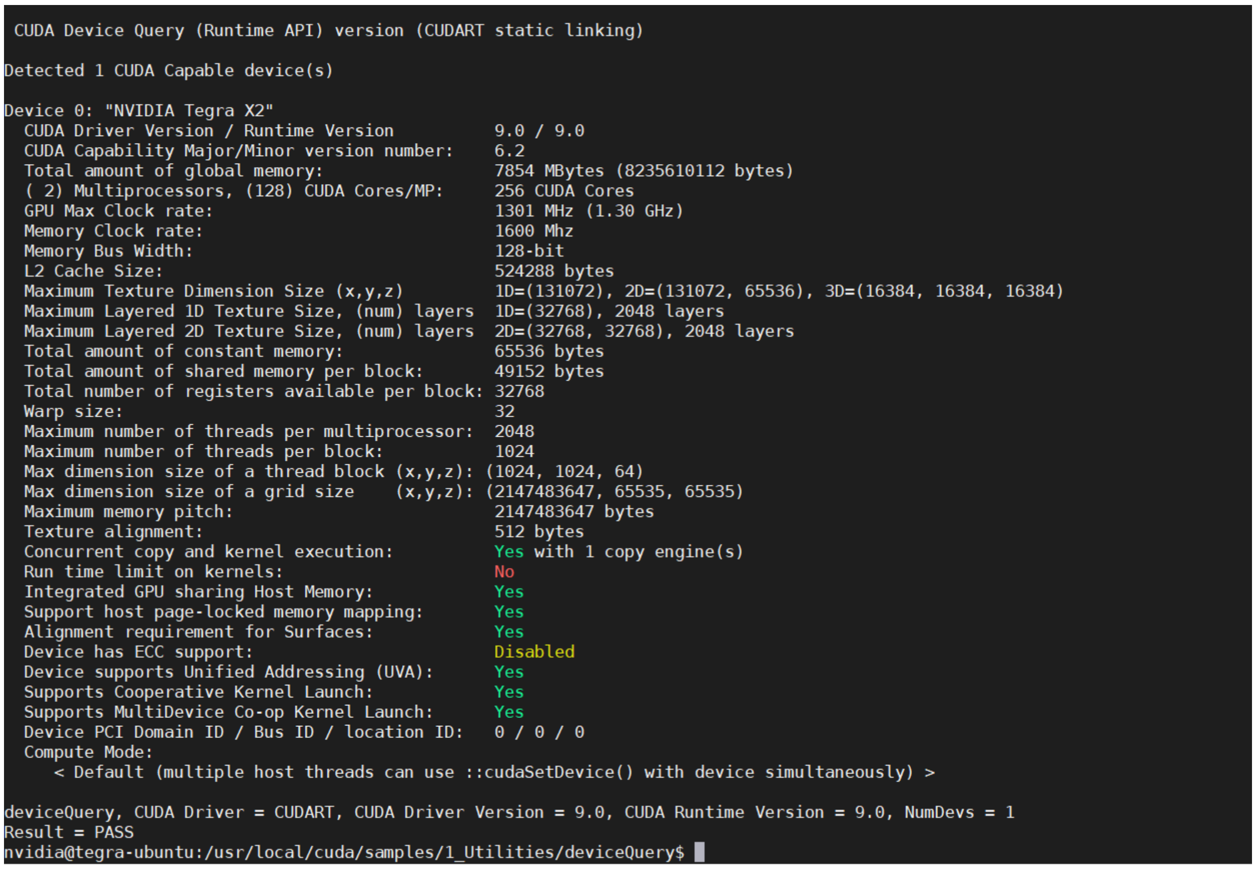
5. The C++ standard used is c++11 and is recommended to use that as it contains all the STL packages and the TCP port handling code

6. The major dependency from the Python part is the paho-mqtt client library. The link for the package can be found *here*.

7. Other minor dependencies in the Python part include the packages for json handling, tcp port handling and numpy for graphical output handling

8. The IDE used is VSCode. It is optional and can be chosen at will. But it is to be noted that if the code is run in debug mode from any IDE, the timing performance is worse than actually expected. This is because in the debug mode, the core clocks of the CPU and GPU run at lower speeds. Therefore it is always advised to run the code in the "Run" mode or launch the generated binary from a shell. The link for the VSCode download can be found *here*.

9. For profiling at the GPU end, various tools are available as per the model of the GPU and can be referenced from the NVIDIA Website as no one tool is available across all platforms. For example, in the Jetson Tx2 board there is a separate script called as tegrastats present in the home folder that does all the monitoring activity (Power and temperature), whereas in the 1080 chipset, there is a utility called nvidia-smi that does this activity.



```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA Tegra X2"
  CUDA Driver Version / Runtime Version          9.0 / 9.0
  CUDA Capability Major/Minor version number:    6.2
  Total amount of global memory:                 7854 MBytes (8235610112 bytes)
  ( 2) Multiprocessors, (128) CUDA Cores/MP:     256 CUDA Cores
  GPU Max Clock rate:                            1301 MHz (1.30 GHz)
  Memory Clock rate:                             1600 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            Yes
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.0, CUDA Runtime Version = 9.0, NumDevs = 1
Result = PASS
nvidia@tegra-ubuntu:/usr/local/cuda/samples/1_Utilities/deviceQuery$
```

Figure 12: deviceQuery program results for NVIDIA Jetson Tx2 board