EMPIRICAL CHARACTERIZATION OF MPS

1. OBJECTIVE

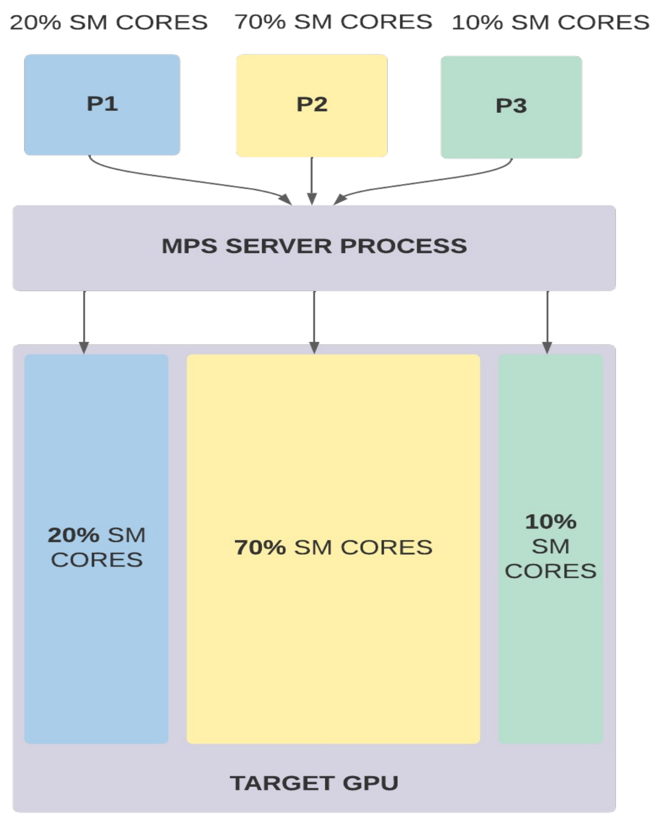
To understand application performance with fractional allocation of GPU Resources (via MPS) and colocation.

2. BACKGROUND

a. MPS Overview:

   i. NVIDIA's MPS (Multi-Process Service) is a feature provided by NVIDIA GPUs that allows multiple CUDA applications to share GPU resources efficiently.

   ii. It enables concurrent execution of multiple CUDA applications without the need to serialize their operations, resulting in improved GPU utilization and performance.
   Example: Consider three CUDA processes – P1, P2 and P3 requesting 20%, 70% and 10% of SM Cores.
   MPS manages the allocation and scheduling of GPU resources, ensuring that each process gets its requested percentage of SM cores (by enabling their concurrent execution).



b. MPS Components:
   MPS is a binary-compatible client-server runtime implementation of the CUDA API which consists of several components:
   i. <u>Control Daemon Process:</u> The control daemon is responsible for starting and stopping the server, as well as coordinating connections between clients and servers.
   ii. <u>Client Runtime:</u> The MPS client runtime is built into the CUDA Driver library and may be used transparently by any CUDA application.
   iii. <u>Server Process:</u> The server is the clients' shared connection to the GPU and provides concurrency between clients.

3. SETUP DESCRIPTION

a. System Setup / Requirements:
   i. NVIDIA Drivers and CUDA Toolkit
   ii. Machine with a MPS enabled NVIDIA GPU (hard requirement)

b. Experimental Setup / Description:
   i. We run a CUDA program “vect_add.cu” which is based on recursive vector addition.
      The description of the same are as follows:
      - Each CUDA thread runs the kernel 5000 times
      - Array size of each of the 3 arrays= 0.4G
      - Num_threads = $2 * 10^8$
      - Threads/block = 512
      - Num_blocks = (arr_size + thr/bl - 1) / (thr/bl)

   ii. We wish to allocate fractional GPU resources to our “vect_add.cu” via MPS and observe its performance as we vary % SM Cores under different collocation factors.

4. STEPS TO REPLICATE

a. Installation Dependencies:
   i. <u>NVIDIA Drivers and CUDA Toolkit Installation:</u>
      Install an appropriate version of the CUDA Toolkit (which comes in with an appropriate compatible version of NVIDIA Drivers) from NVIDIA’s official website [1]. You may select the required OS, architecture, Distribution, Version and Installer Type for which you will get the download commands.
   ii. Ensure that your machine has a MPS enabled NVIDIA GPU.

b. Running Commands:

Now that we have our setup ready with the required installation dependencies, we can dive right into running the commands to achieve our objective.

i. Building an executable for “vect_add.cu”
We must first build an executable for our “vect_add.cu” program. Go to the directory containing “vect_add.cu” and run the following command:
⇒ nvcc -o vect_add vect_add.cu

ii. Starting MPS
Before even beginning to use MPS, we must know how to start the MPS Control Daemon. Run the following commands using root privileges (sudo su -):
⇒ export CUDA_VISIBLE_DEVICES="0"
⇒ nvidia-smi -i 0 -c EXCLUSIVE_PROCESS
⇒ nvidia-cuda-mps-control -d

The first command sets the environment variable "CUDA_VISIBLE_DEVICES" to specify which CUDA-capable devices (in our case GPU with device ID “0”) are visible to CUDA applications.
The second command sets the compute mode of GPU ID “0” to "EXCLUSIVE_PROCESS" using the nvidia-smi tool.
The last command finally starts our MPS control daemon process.

iii. Restricting Cores and Memory via MPS
Now that our MPS server is up and running we will try to restrict GPU resources via it. To do so, firstly create a bash script (in the same directory as your “vect_add.cu”) called “set_gpu_limits.sh” and include the following:

```bash
#!/bin/bash
export CUDA_MPS_PINNED_DEVICE_MEM_LIMIT=$1
export CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=$2
```

Next open the terminal and run the following command:
⇒ source ~/directory_name/set_gpu_limits.sh "0=2G" 10

By running the above command, for GPU-ID: 0, we set the CUDA_MPS_PINNED_DEVICE_MEM_LIMIT (which represents GPU memory) to 2G and set the CUDA_MPS_ACTIVE_THREAD_PERCENTAGE (which represents the % SM Cores) to 100%

iv. Allocating restricted cores and memory to our “vect_add”

So far we have:-

⇒ Built an executable for “vect-add.cu” program

⇒ Started our MPS control daemon

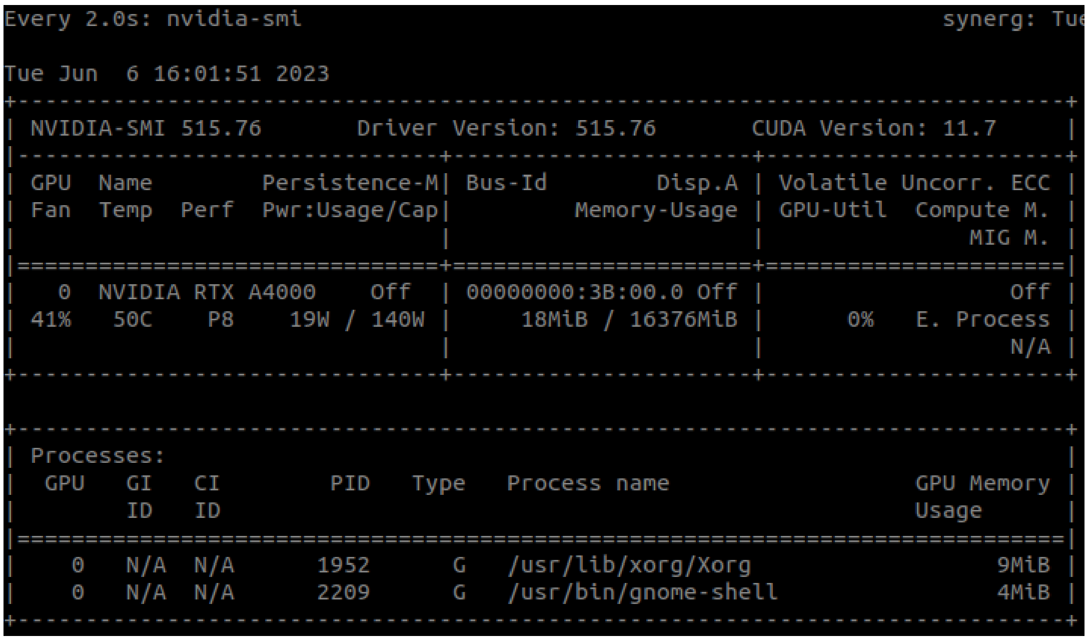
⇒ Restricted GPU cores and memory via MPS

Now, we just have to allocate the restricted set cores and memory to our compiled program. However, before we do this we must run the “nvidia-smi” command to ensure that the “nvidia- cuda-mps-server” process is running after we compile “vect_add”

Preferably in a split terminal run:

⇒ watch nvidia-smi

This command provides the information regarding all processes accessing the GPU besides providing GPU utilization and performance metrics



```
Every 2.0s: nvidia-smi                                                    synerg: Tu

Tue Jun  6 16:01:51 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 515.76       Driver Version: 515.76       CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA RTX A4000    Off  | 00000000:3B:00.0 Off |                  Off |
| 41%   50C    P8    19W / 140W |     18MiB / 16376MiB |      0%   E. Process |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1952      G   /usr/lib/xorg/Xorg                  9MiB |
|    0   N/A  N/A      2209      G   /usr/bin/gnome-shell                4MiB |
+-----------------------------------------------------------------------------+
```

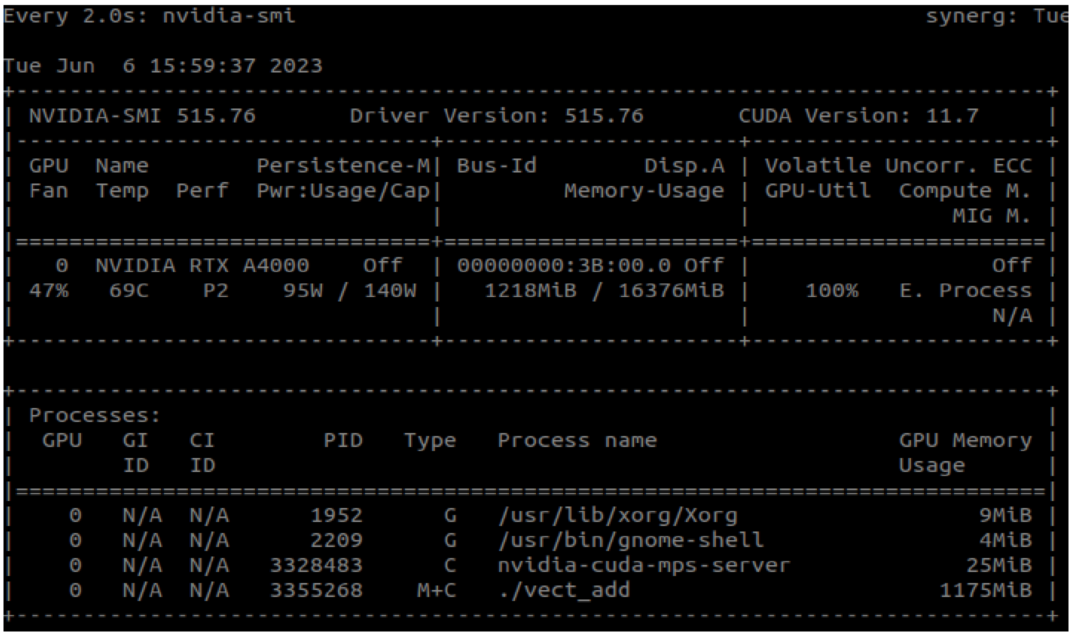
Once this is setup, we can finally compile our “vect_add.” After running the command in the below terminal:

⇒ source ~/directory_name/set_gpu_limits.sh "0=2G" 10

In the same terminal, simply execute the compiled “vect_add” by:

⇒ ./vect_add

Watch how the “nvidia-cuda-mps-server” process springs up along with the “./vect_add”


```
Every 2.0s: nvidia-smi                                                          synerg: Tue

Tue Jun  6 15:59:37 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 515.76       Driver Version: 515.76       CUDA Version: 11.7     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA RTX A4000    Off  | 00000000:3B:00.0 Off |                  Off |
| 47%   69C    P2    95W / 140W |   1218MiB / 16376MiB |    100%   E. Process |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1952      G   /usr/lib/xorg/Xorg                  9MiB |
|    0   N/A  N/A      2209      G   /usr/bin/gnome-shell                4MiB |
|    0   N/A  N/A   3328483      C   nvidia-cuda-mps-server             25MiB |
|    0   N/A  N/A   3355268    M+C   ./vect_add                       1175MiB |
+-----------------------------------------------------------------------------+
```

v. Repeating iii. and iv.

- For colocation factor = 1 (x cores for P1 and vary x)

  You can try repeating steps iii. and iv. by varying the %SM cores and keeping the GPU memory same.

  A few example commands include:

  ⇒ source ~/directory_name/set_gpu_limits.sh "0=2G" 5

  ⇒ source ~/directory_name/set_gpu_limits.sh "0=2G" 10

  ⇒ source ~/directory_name/set_gpu_limits.sh "0=2G" 20

  And so on all the way until:

  ⇒ source ~/directory_name/set_gpu_limits.sh "0=2G" 100

  Another interesting observation can be what happens if we provide the following command(s):

  ⇒ source ~/directory_name/set_gpu_limits.sh "0=xG" y

  Where x<(0.4*1024*3) and y<=100

  (Hint: Understand how we arrived at 0.4*1024*3

  Array size mentioned was 0.4G. Insufficient memory?)

  FINAL OUTPUT: Plot a graph for execution time v/s % SM Cores

- For colocation factor = 2 (x cores for P1 and 100-x cores for P2, vary x)
  Open 3 terminals – one terminal for P1, one for P2 and one for “watch nvidia-smi”
  In terminal - 1, restrict GPU resources for P1 as:
  ⇒ source ~/directory_name/set_gpu_limits.sh "0=2G" x

  In terminal -2, restrict GPU resources for P2 as:
  ⇒ source ~/directory_name/set_gpu_limits.sh "0=2G" 100-x

  Do not run these processes one by one, **but rather concurrently**
  Observe terminal-3 which has “watch nvidia-smi” command running
  Once again vary the %SM cores for P1 & P2 and keep the GPU memory same.

  A few example commands include:
  ⇒ P1: source ~/directory_name/set_gpu_limits.sh "0=2G" 5
  P2: source ~/directory_name/set_gpu_limits.sh "0=2G" 95
  ⇒ P1: source ~/directory_name/set_gpu_limits.sh "0=2G" 10
  P2: source ~/directory_name/set_gpu_limits.sh "0=2G" 90

  And so on all the way until:
  ⇒ P1: source ~/directory_name/set_gpu_limits.sh "0=2G" 95
  P2: source ~/directory_name/set_gpu_limits.sh "0=2G" 5

  FINAL OUTPUT:
  ⇒ Plot a graph for execution time v/s % SM Cores (wrt P1)
  ⇒ Compare P1’s performance for CF=1 and CF=2. Plot the results.

- Bonus: For colocation factor = 3 (x cores for P1 and [100-x]/2 cores for P2 and P3, vary x)
  Repeat the steps as described above.

  FINAL OUTPUT:
  ⇒ Plot a graph for execution time v/s % SM Cores (wrt P1)
  ⇒ Compare P1’s performance for CF=1, CF=2 and CF=3. Plot the results.

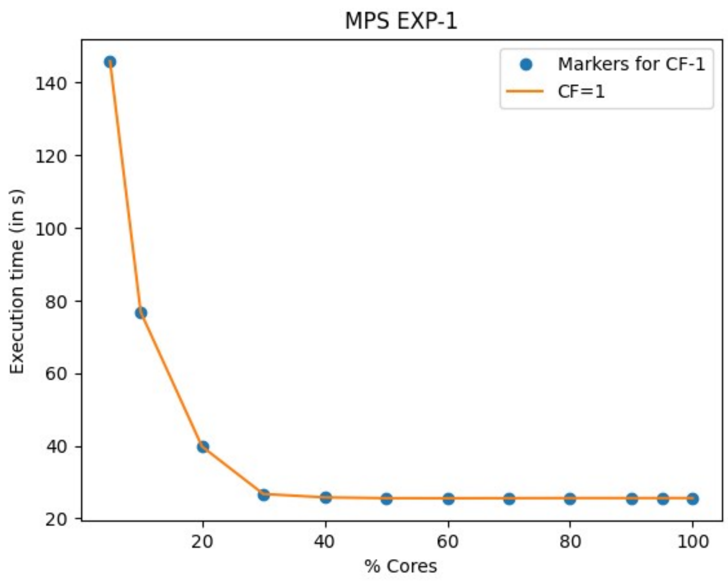
vi. Stopping MPS
Once we have concluded our experiment we may stop our MPS Control Daemon by running the following command with root privileges (sudo su -):
⇒ `echo quit | nvidia-cuda-mps-control`

vii. Exploring MPS and beyond:
For a more detailed guide to MPS you may refer [2] NVIDIA's official MPS Documentation.

5. EXPERIMENTAL RESULTS

a. For colocation factor = 1

% CORES	**GPU Mem (in MiB)**	**T (in s)**	**Max SM Cores**
5	1171	145.93	4
10	1175	76.818	8
20	1187	39.772	17
30	1205	26.704	29
40	1217	25.773	37
50	1233	25.548	50
60	1243	25.525	58
70	1255	25.551	67
80	1271	25.584	79
90	1283	25.583	87
95	1289	25.585	92
100	1299	25.589	100

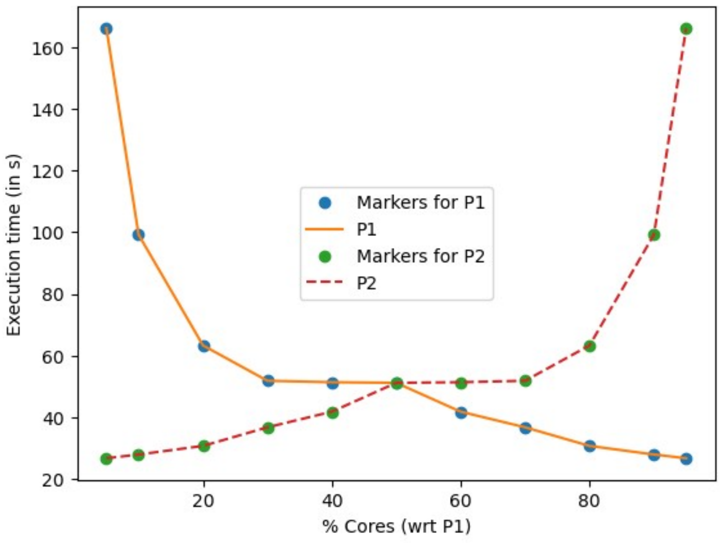


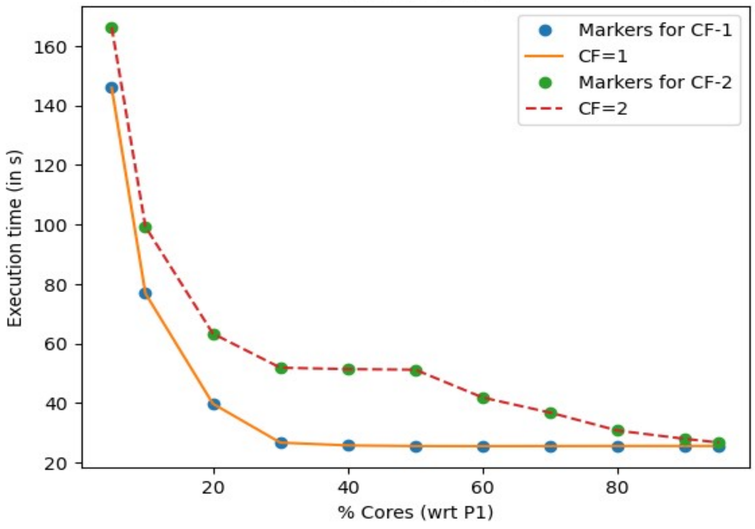
As we can observe, execution time of P1 decreases as we allot it a greater %SM cores, and is almost constant beyond the 50% mark.

b. For colocation factor = 2

% CORES_1	% CORES_2	GPU MEM_1 (MiB)	GPU MEM_2 (MiB)	T_1 (in s)	T_2 (in s)
5	95	1171	1289	166.226	26.750
10	90	1175	1283	99.239	27.956
20	80	1187	1271	63.335	30.746
30	70	1205	1255	51.865	36.782
40	60	1217	1243	51.396	41.839

50	50	1233	1299	51.217	51.167

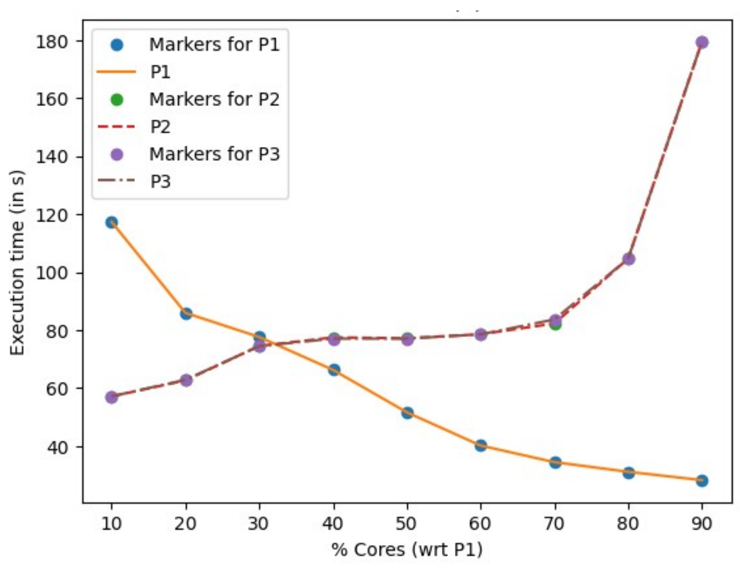


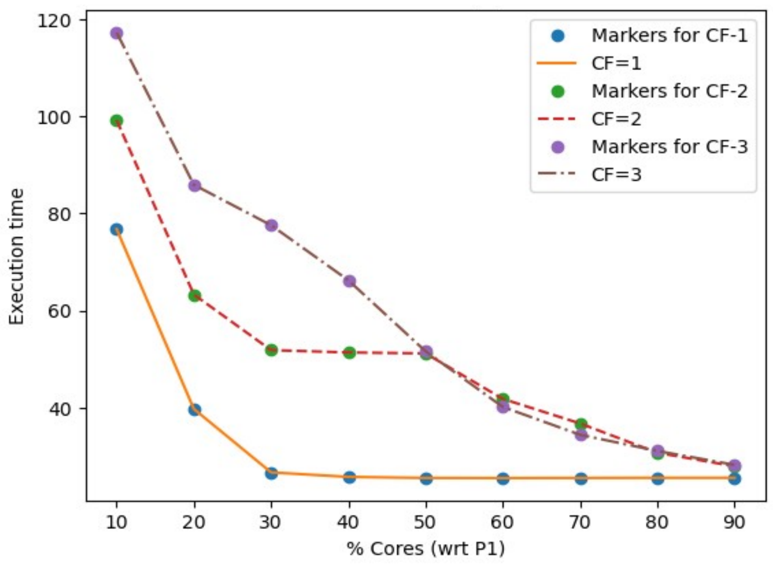


As we can observe, P1 takes a longer time for cf=2 than for cf=1
Which is quite explanatory as for cf =2 P1 and P2 run concurrently and compete for SM cores than for cf = 1 (when only P1 runs alone)

c. For colocation factor = 3

% CORES_1	**% CORES_2**	**% CORES_3**	**GPU MEM_1 (MiB)**	**GPU MEM_2 (MiB)**	**GPU MEM_3 (MiB)**	**T_1 (in s)**	**T_2 (in s)**	**T_3 (in s)**
10	45	45	1175	1221	1221	117.291	57.115	57.153
20	40	40	1187	1217	1217	85.928	62.649	62.815
30	35	35	1205	1209	1209	77.603	74.501	74.452
40	30	30	1217	1205	1205	66.269	77.455	77.053
50	25	25	1233	1199	1199	51.638	77.170	77.037
60	20	20	1243	1187	1187	40.180	78.556	78.508
70	15	15	1255	1183	1183	35.450	82.255	83.655
80	10	10	1271	1175	1175	31.119	104.664	104.641
90	5	5	1283	1171	1171	28.249	179.405	179.485





As we can observe, P1 takes the longest time for cf=3 followed by cf=2 than for cf=1.
Which is quite explanatory as for cf = 3 P1, P2 and P3 run concurrently whereas for cf = 2 P1 and P2 run concurrently and compete for SM cores than for cf = 1 (when only P1 runs alone)

6. REFERENCES

[1] NVIDIA's CUDA Toolkit Installation Guide:
https://developer.nvidia.com/cuda-downloads
[2] NVIDIA's MPS Documentation:
https://docs.nvidia.com/deploy/mps/index.html