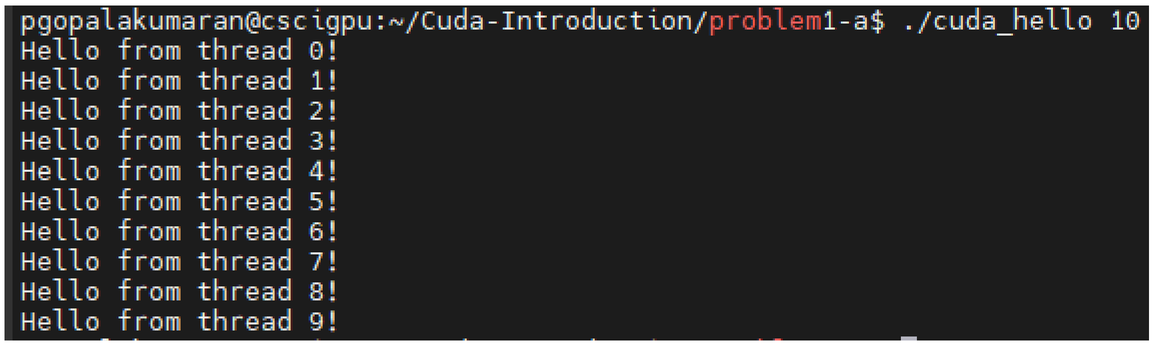
CSCI 551 - Numerical and Parallel Programming Exercise #4 SPMD (MPI and CUDA) Parallel Scaling, Interpolation and Methods of Integration for Simulation

By: Pranav Gopalakumaran

1) Introduction to SPMD parallel programing with CUDA

a) Build, Run and Test CUDA hello programs

The two versions of CUDA hello programs (hello_cuda.cu, hello_cuda1.cu) were built, and executed. The following are the output of the execution:



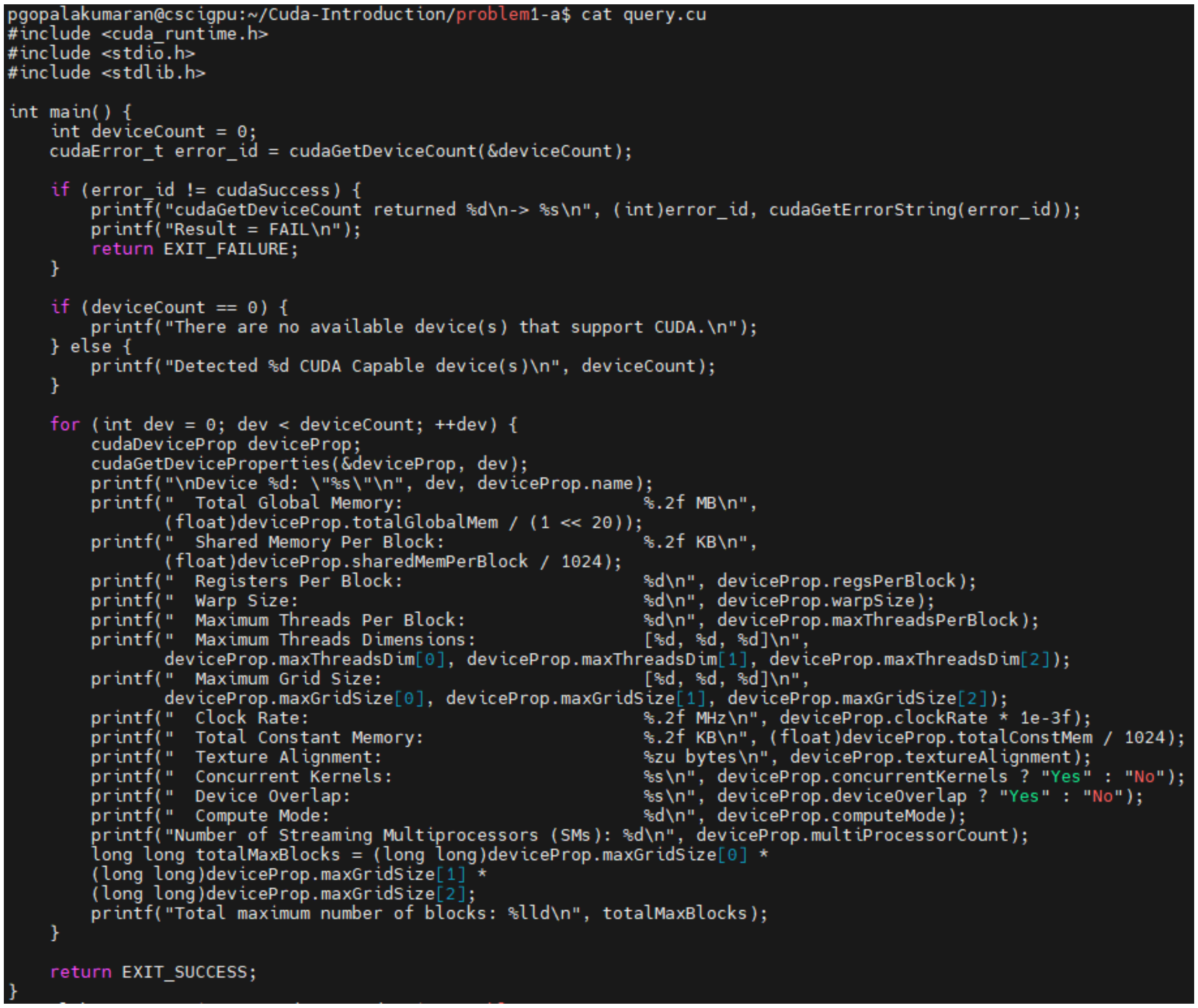
```
pgopalakumaran@cscigpu:~/Cuda-Introduction/problem1-a$ ./cuda_hello 10
Hello from thread 0!
Hello from thread 1!
Hello from thread 2!
Hello from thread 3!
Hello from thread 4!
Hello from thread 5!
Hello from thread 6!
Hello from thread 7!
Hello from thread 8!
Hello from thread 9!
```

<u>How many threads can print hello using hello_cuda.cu? :</u>

Using the **`hello_cuda.cu`** program, the maximum number of threads that can print "hello" is 2048, because it uses a single block and the maximum number of threads per block is 2048.

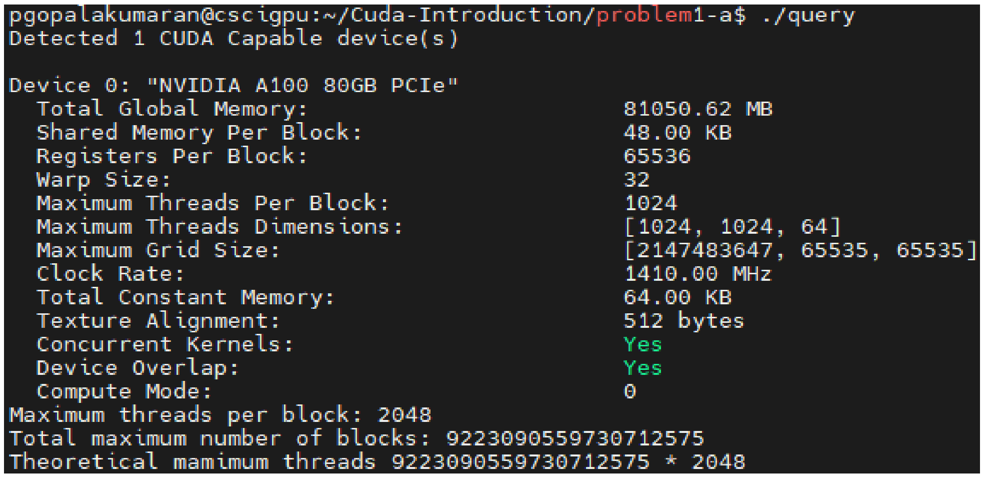
<u>With blocks and threads, how many threads can print hello using hello_cuda1.cu? Can we scale bigger using blocks and threads?</u>

The number of blocks and threads per block can help scale to print a large number of “hello” messages. In order to determine the number of blocks and threads in a block in the A100 GPU, I wrote a program to query these details from the GPU driver called “query.cu”. The screenshot of the code is provided below and in the submission files.



```
pgopalakumaran@cscigpu:~/Cuda-Introduction/problem1-a$ cat query.cu
#include <cuda_runtime.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    int deviceCount = 0;
    cudaError_t error_id = cudaGetDeviceCount(&deviceCount);

    if (error_id != cudaSuccess) {
        printf("cudaGetDeviceCount returned %d\n-> %s\n", (int)error_id, cudaGetErrorString(error_id));
        printf("Result = FAIL\n");
        return EXIT_FAILURE;
    }

    if (deviceCount == 0) {
        printf("There are no available device(s) that support CUDA.\n");
    } else {
        printf("Detected %d CUDA Capable device(s)\n", deviceCount);
    }

    for (int dev = 0; dev < deviceCount; ++dev) {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, dev);
        printf("\nDevice %d: \"%s\"\n", dev, deviceProp.name);
        printf("  Total Global Memory:                         %.2f MB\n",
               (float)deviceProp.totalGlobalMem / (1 << 20));
        printf("  Shared Memory Per Block:                     %.2f KB\n",
               (float)deviceProp.sharedMemPerBlock / 1024);
        printf("  Registers Per Block:                         %d\n", deviceProp.regsPerBlock);
        printf("  Warp Size:                                   %d\n", deviceProp.warpSize);
        printf("  Maximum Threads Per Block:                   %d\n", deviceProp.maxThreadsPerBlock);
        printf("  Maximum Threads Dimensions:                  [%d, %d, %d]\n",
               deviceProp.maxThreadsDim[0], deviceProp.maxThreadsDim[1], deviceProp.maxThreadsDim[2]);
        printf("  Maximum Grid Size:                           [%d, %d, %d]\n",
               deviceProp.maxGridSize[0], deviceProp.maxGridSize[1], deviceProp.maxGridSize[2]);
        printf("  Clock Rate:                                  %.2f MHz\n", deviceProp.clockRate * 1e-3f);
        printf("  Total Constant Memory:                       %.2f KB\n", (float)deviceProp.totalConstMem / 1024);
        printf("  Texture Alignment:                           %zu bytes\n", deviceProp.textureAlignment);
        printf("  Concurrent Kernels:                          %s\n", deviceProp.concurrentKernels ? "Yes" : "No");
        printf("  Device Overlap:                              %s\n", deviceProp.deviceOverlap ? "Yes" : "No");
        printf("  Compute Mode:                                %d\n", deviceProp.computeMode);
        printf("Number of Streaming Multiprocessors (SMs): %d\n", deviceProp.multiProcessorCount);
        long long totalMaxBlocks = (long long)deviceProp.maxGridSize[0] *
        (long long)deviceProp.maxGridSize[1] *
        (long long)deviceProp.maxGridSize[2];
        printf("Total maximum number of blocks: %lld\n", totalMaxBlocks);
    }

    return EXIT_SUCCESS;
}
```

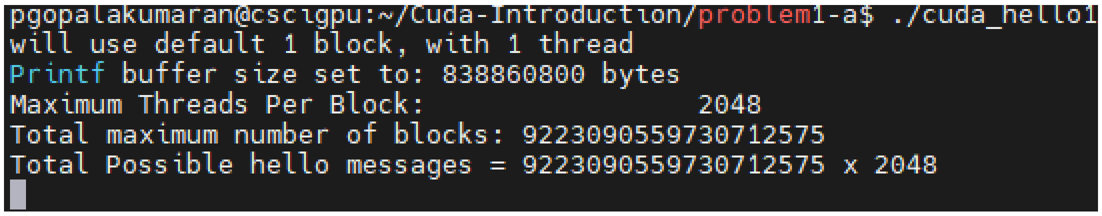
By executing this program, I found that the threads per block are 2048 and there is an enormous number of blocks, 9,223,090,559,730,712,575, as observed below.



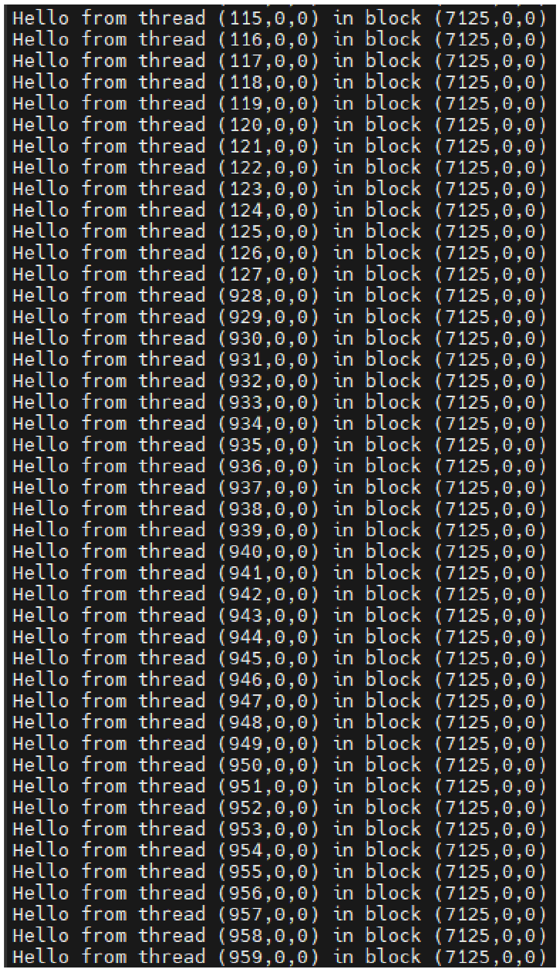
```
pgopalakumaran@cscigpu:~/Cuda-Introduction/problem1-a$ ./query
Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA A100 80GB PCIe"
  Total Global Memory:                         81050.62 MB
  Shared Memory Per Block:                     48.00 KB
  Registers Per Block:                         65536
  Warp Size:                                   32
  Maximum Threads Per Block:                   1024
  Maximum Threads Dimensions:                  [1024, 1024, 64]
  Maximum Grid Size:                           [2147483647, 65535, 65535]
  Clock Rate:                                  1410.00 MHz
  Total Constant Memory:                       64.00 KB
  Texture Alignment:                           512 bytes
  Concurrent Kernels:                          Yes
  Device Overlap:                              Yes
  Compute Mode:                                0
Maximum threads per block: 2048
Total maximum number of blocks: 9223090559730712575
Theoretical mamimum threads 9223090559730712575 * 2048
```

This would make the total number of threads in the GPU is 9,223,090,559,730,712,575 x 2048 which is an enormous number resulting in a theoretical maximum of over 9 sextillion threads that is available to the **hello_cuda1.cu** program, resulting in a theoretical maximum of over 9 sextillion threads that can print "hello".

cuda_hello1 Output:



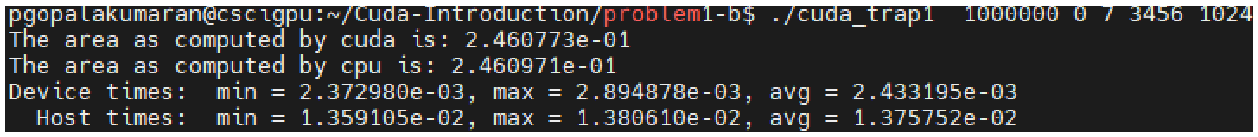
```
pgopalakumaran@cscigpu:~/Cuda-Introduction/problem1-a$ ./cuda_hello1
will use default 1 block, with 1 thread
Printf buffer size set to: 838860800 bytes
Maximum Threads Per Block:              2048
Total maximum number of blocks: 9223090559730712575
Total Possible hello messages = 9223090559730712575 x 2048
```



```
Hello from thread (115,0,0) in block (7125,0,0)
Hello from thread (116,0,0) in block (7125,0,0)
Hello from thread (117,0,0) in block (7125,0,0)
Hello from thread (118,0,0) in block (7125,0,0)
Hello from thread (119,0,0) in block (7125,0,0)
Hello from thread (120,0,0) in block (7125,0,0)
Hello from thread (121,0,0) in block (7125,0,0)
Hello from thread (122,0,0) in block (7125,0,0)
Hello from thread (123,0,0) in block (7125,0,0)
Hello from thread (124,0,0) in block (7125,0,0)
Hello from thread (125,0,0) in block (7125,0,0)
Hello from thread (126,0,0) in block (7125,0,0)
Hello from thread (127,0,0) in block (7125,0,0)
Hello from thread (928,0,0) in block (7125,0,0)
Hello from thread (929,0,0) in block (7125,0,0)
Hello from thread (930,0,0) in block (7125,0,0)
Hello from thread (931,0,0) in block (7125,0,0)
Hello from thread (932,0,0) in block (7125,0,0)
Hello from thread (933,0,0) in block (7125,0,0)
Hello from thread (934,0,0) in block (7125,0,0)
Hello from thread (935,0,0) in block (7125,0,0)
Hello from thread (936,0,0) in block (7125,0,0)
Hello from thread (937,0,0) in block (7125,0,0)
Hello from thread (938,0,0) in block (7125,0,0)
Hello from thread (939,0,0) in block (7125,0,0)
Hello from thread (940,0,0) in block (7125,0,0)
Hello from thread (941,0,0) in block (7125,0,0)
Hello from thread (942,0,0) in block (7125,0,0)
Hello from thread (943,0,0) in block (7125,0,0)
Hello from thread (944,0,0) in block (7125,0,0)
Hello from thread (945,0,0) in block (7125,0,0)
Hello from thread (946,0,0) in block (7125,0,0)
Hello from thread (947,0,0) in block (7125,0,0)
Hello from thread (948,0,0) in block (7125,0,0)
Hello from thread (949,0,0) in block (7125,0,0)
Hello from thread (950,0,0) in block (7125,0,0)
Hello from thread (951,0,0) in block (7125,0,0)
Hello from thread (952,0,0) in block (7125,0,0)
Hello from thread (953,0,0) in block (7125,0,0)
Hello from thread (954,0,0) in block (7125,0,0)
Hello from thread (955,0,0) in block (7125,0,0)
Hello from thread (956,0,0) in block (7125,0,0)
Hello from thread (957,0,0) in block (7125,0,0)
Hello from thread (958,0,0) in block (7125,0,0)
Hello from thread (959,0,0) in block (7125,0,0)
```

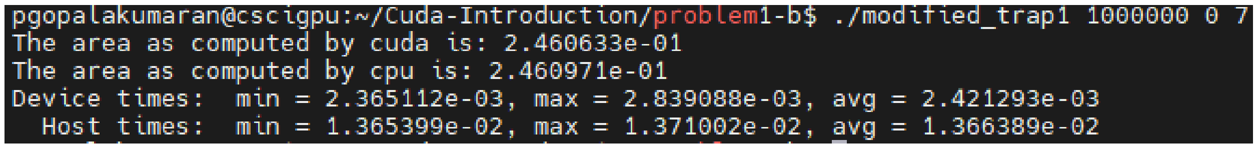
B) Build, Run and Test cuda_trap1.cd

The following is the output of the cuda_trap1 run:



```
pgopalakumaran@cscigpu:~/Cuda-Introduction/problem1-b$ ./cuda_trap1  1000000 0 7 3456 1024
The area as computed by cuda is: 2.460773e-01
The area as computed by cpu is: 2.460971e-01
Device times:  min = 2.372980e-03, max = 2.894878e-03, avg = 2.433195e-03
  Host times:  min = 1.359105e-02, max = 1.380610e-02, avg = 1.375752e-02
```

I have copied cuda_trap1 to modified_trap1 to compare the area computed by both sequential trap.c and the modified trap code. The computed values from both approaches are accurate to 4 decimals. The output of the modified code is provided below. The area computed by cpu below is for sequential trap code.



```
pgopalakumaran@cscigpu:~/Cuda-Introduction/problem1-b$ ./modified_trap1 1000000 0 7
The area as computed by cuda is: 2.460633e-01
The area as computed by cpu is: 2.460971e-01
Device times:  min = 2.365112e-03, max = 2.839088e-03, avg = 2.421293e-03
  Host times:  min = 1.365399e-02, max = 1.371002e-02, avg = 1.366389e-02
```

The original command line argument takes,

n = number of Steps that represents number of trapezoids used in the integration
a = The left endpoint of the interval for the integration
b = The right endpoint of the interval for the integration
blk_ct = block count
th_per_blk = threads per block

The blk_ct multiplied by th_per_blk is used by CUDA to evenly distribute the kernel in the GPU to achieve parallelism.

The command line arguments have been modified to take only **steps** and then calculates the **threads per block** and **block counts**:

Threads per Block (th_per_blk): The maximum number of threads per block for a given GPU is a fixed property. The code used to find out maximum is:

```
*n_p = strtol(argv[1], NULL, 10);
*a_p = strtod(argv[2], NULL);
*b_p = strtod(argv[3], NULL);
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, 0);
*th_per_blk_p = deviceProp.maxThreadsPerBlock;
```

Block Count (blk_ct): The block count is computed based on the total number of trapezoids (n_p) and the threads per block (th_per_blk) by

```
*blk_ct_p = (*n_p + *th_per_blk_p - 1) / *th_per_blk_p;
```

Speed-up and Parallel Percentage:

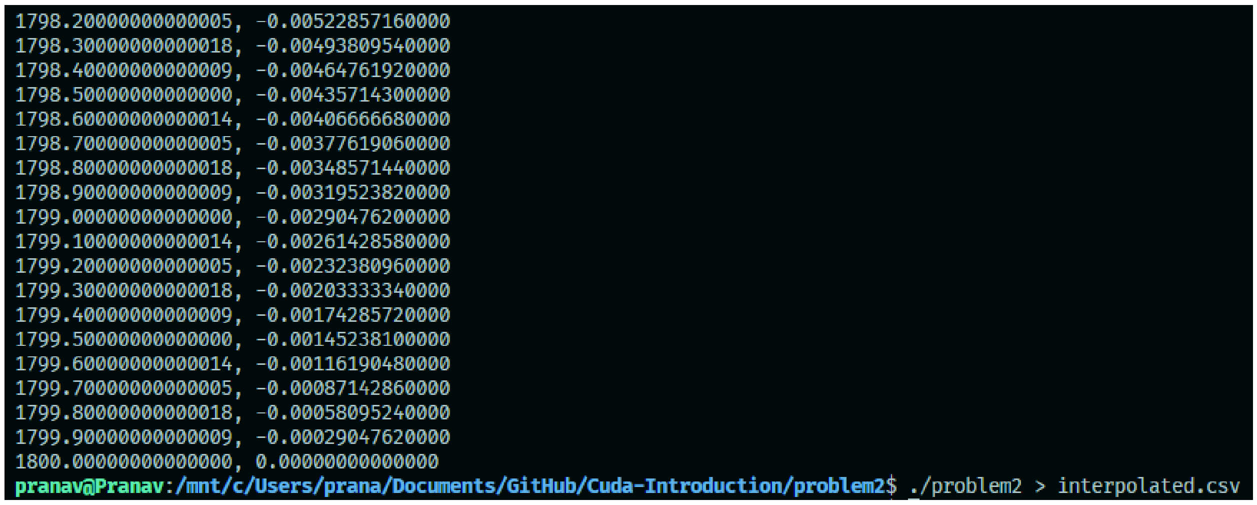
The **speed-up (SU) achieved** by modified_trap1 with 1,000,000 steps from sequential cuda_traps1.c is **5.64** (= 0.01366389/0.002421293) and the **parallel percentage (P)** is,

SU = 1 / (1 - P + P/S)
5.64 = 1 / (1 - P + P/ (3456 * 2048))
P = 0.822695151695546

Parallel percentage (P) is **82.3%**

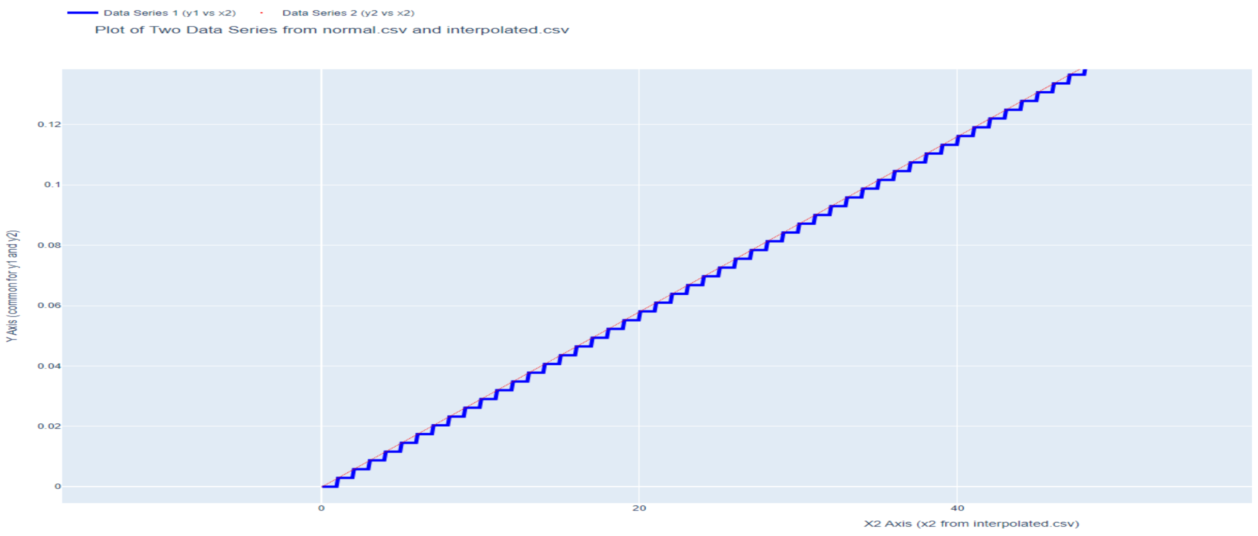
2) Interpolate acceleration for 10 times per second

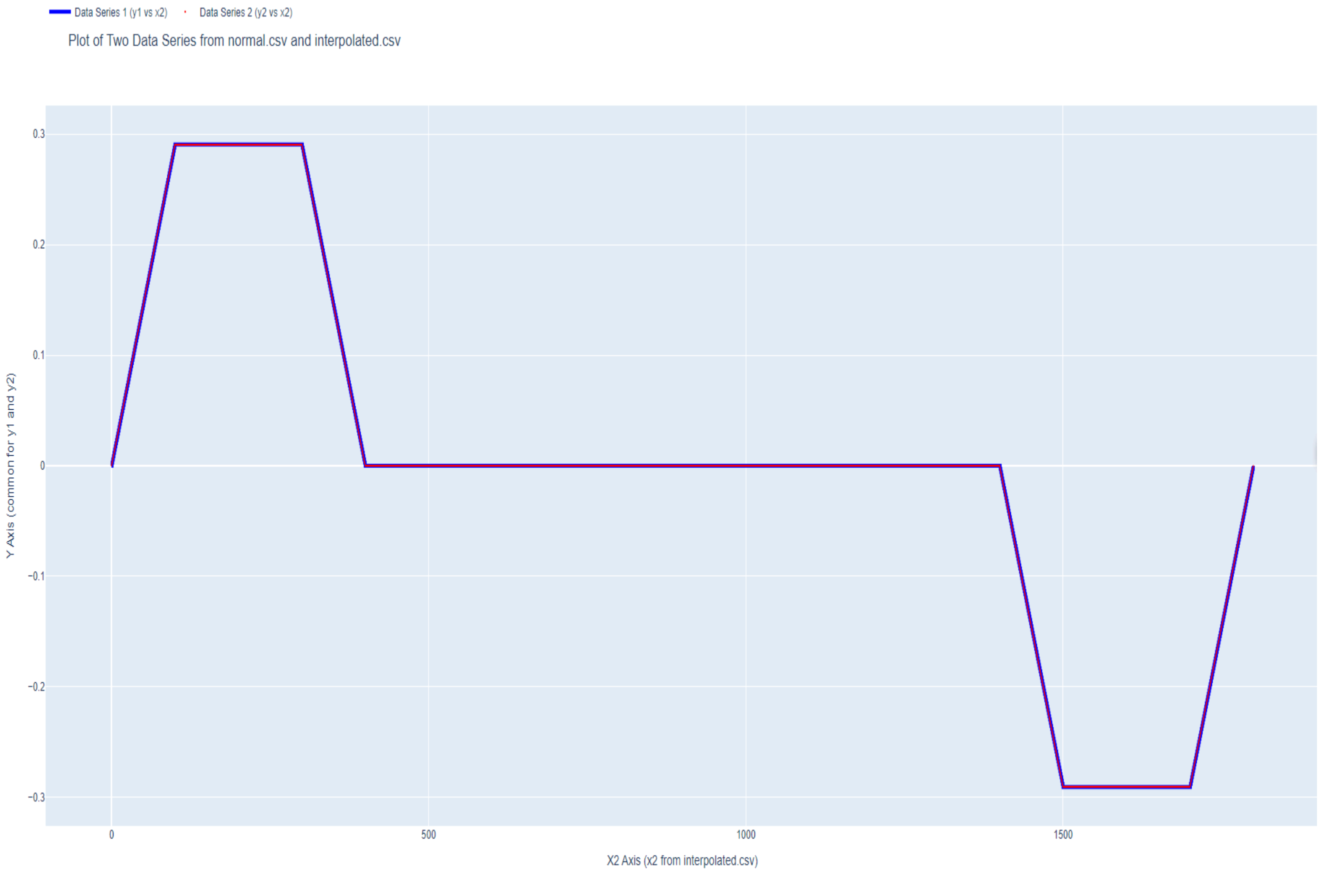
The **problem2.c** reads acceleration profile from ex4accel.h and **loads to DefaultProfile**. Every entry in the DefaultProfile contains acceleration value at a specific second. Linear interpolation method is used to compute intermediate acceleration values using the number of steps per second. The **generated interpolated data** has been **written into an interpolated.csv** file.



```
1798.20000000000005, -0.00522857160000
1798.30000000000018, -0.00493809540000
1798.40000000000009, -0.00464761920000
1798.50000000000000, -0.00435714300000
1798.60000000000014, -0.00406666680000
1798.70000000000005, -0.00377619060000
1798.80000000000018, -0.00348571440000
1798.90000000000009, -0.00319523820000
1799.00000000000000, -0.00290476200000
1799.10000000000014, -0.00261428580000
1799.20000000000005, -0.00232380960000
1799.30000000000018, -0.00203333340000
1799.40000000000009, -0.00174285720000
1799.50000000000000, -0.00145238100000
1799.60000000000014, -0.00116190480000
1799.70000000000005, -0.00087142860000
1799.80000000000018, -0.00058095240000
1799.90000000000009, -0.00029047620000
1800.00000000000000, 0.00000000000000
pranav@Pranav:/mnt/c/Users/prana/Documents/GitHub/Cuda-Introduction/problem2$ ./problem2 > interpolated.csv
```

It has been a challenge to create a chart using excel where I could not plot original and interpolated data with each having its own x-axis values. So I have created a python program to create a chart to plot both the data sets. The following provides the chart created by overlaying the 1 second original data from **normal.csv** with the problem2.c generated interpolated data written in **interpolated.csv**. As we closely observe the chart, we are able to notice the staircase view of **original data (blue color)** compared to a smoother curve of **interpolated data (red color)**. Depending on the type of integration (trapezoidal, riemann, simpson), the numerical integration could introduce error when integrating area under the function with the original data when compared to the interpolated data.



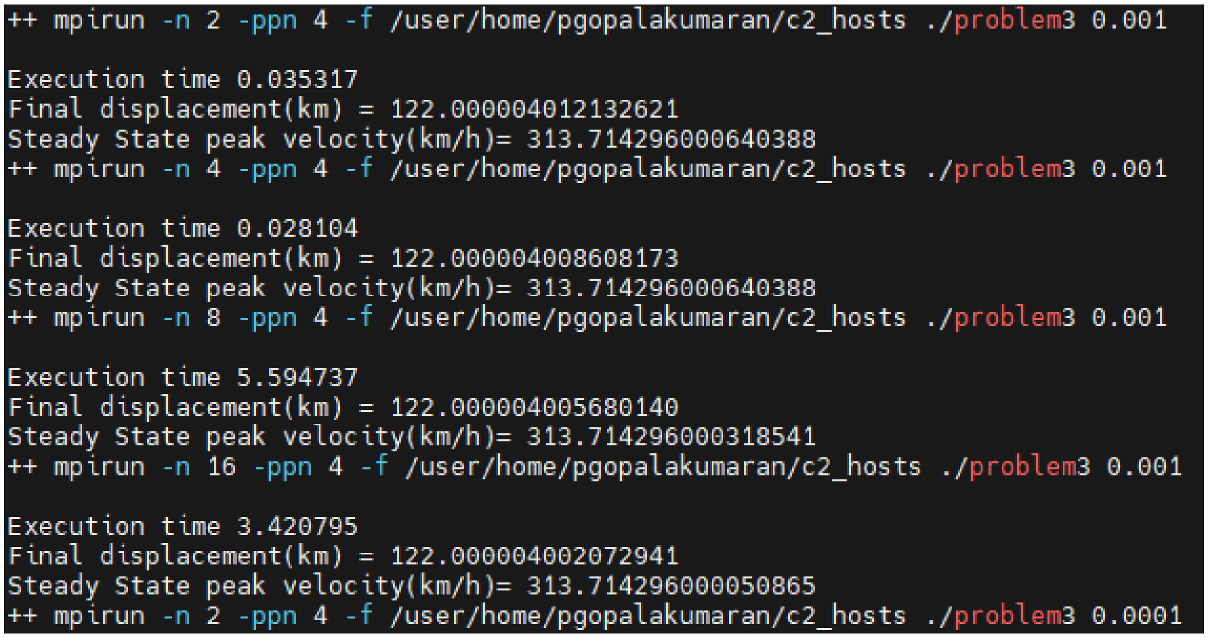


3) Integrate Train's Acceleration Profile

The problem3.c implements Simpson method that accepts an optional command-line argument for step_size that is used to distribute work to each MPI process(i.e. rank). Each rank calculates its portion of the local velocity by integrating the acceleration profile. The local velocities array (local_velocity) is passed using MPI_send/MPI_recv from each non zero rank to the zeroth rank after calculation, to resolve loop carried dependencies. After this rank 0 passes the calculated velocity array back to all the ranks using MPI_Bcast. The second part of the program then uses the calculated velocity array in local_velocity for the second integration. The final value after integration is the total displacement at the end point. Finally the program outputs total displacement, steady-state peak velocity, and execution time as provided below:

a) Determine velocity and position over time using dt=0.001 seconds

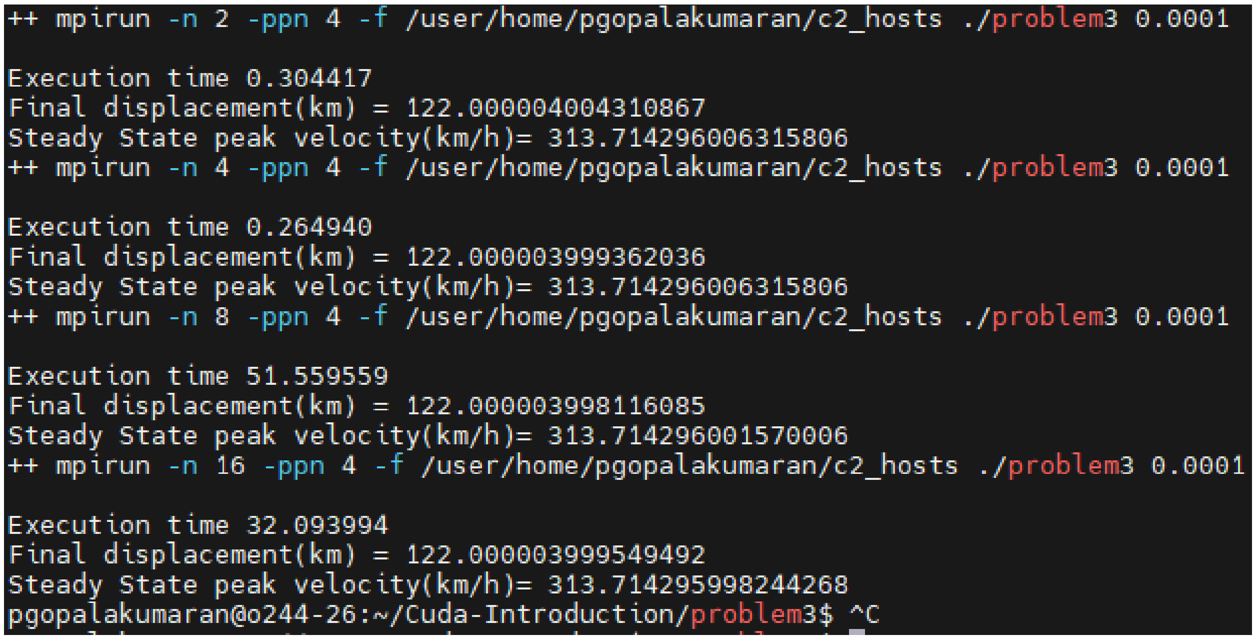
The program3.c has been run using a script for dt=0.001 and the execution time, final displacement and Steady state peak velocity has been determined. As can be observed, the final displacement (122 km) and steady state peak velocity (313.7 km/h) has been calculated correctly.



```
++ mpirun -n 2 -ppn 4 -f /user/home/pgopalakumaran/c2_hosts ./problem3 0.001

Execution time 0.035317
Final displacement(km) = 122.000004012132621
Steady State peak velocity(km/h)= 313.714296000640388
++ mpirun -n 4 -ppn 4 -f /user/home/pgopalakumaran/c2_hosts ./problem3 0.001

Execution time 0.028104
Final displacement(km) = 122.000004008608173
Steady State peak velocity(km/h)= 313.714296000640388
++ mpirun -n 8 -ppn 4 -f /user/home/pgopalakumaran/c2_hosts ./problem3 0.001

Execution time 5.594737
Final displacement(km) = 122.000004005680140
Steady State peak velocity(km/h)= 313.714296000318541
++ mpirun -n 16 -ppn 4 -f /user/home/pgopalakumaran/c2_hosts ./problem3 0.001

Execution time 3.420795
Final displacement(km) = 122.000004002072941
Steady State peak velocity(km/h)= 313.714296000050865
++ mpirun -n 2 -ppn 4 -f /user/home/pgopalakumaran/c2_hosts ./problem3 0.0001
```

b) Determine velocity and position over time using dt=0.0001 seconds and compare with first profile

The program3.c has been run using a script for dt=0.0001. As can be observed, the final displacement (122 km) and steady state peak velocity (313.7 km/h) has been calculated correctly and this value remains close to the values for dt=0.001.



```
++ mpirun -n 2 -ppn 4 -f /user/home/pgopalakumaran/c2_hosts ./problem3 0.0001

Execution time 0.304417
Final displacement(km) = 122.000004004310867
Steady State peak velocity(km/h)= 313.714296006315806
++ mpirun -n 4 -ppn 4 -f /user/home/pgopalakumaran/c2_hosts ./problem3 0.0001

Execution time 0.264940
Final displacement(km) = 122.000003999362036
Steady State peak velocity(km/h)= 313.714296006315806
++ mpirun -n 8 -ppn 4 -f /user/home/pgopalakumaran/c2_hosts ./problem3 0.0001

Execution time 51.559559
Final displacement(km) = 122.000003998116085
Steady State peak velocity(km/h)= 313.714296001570006
++ mpirun -n 16 -ppn 4 -f /user/home/pgopalakumaran/c2_hosts ./problem3 0.0001

Execution time 32.093994
Final displacement(km) = 122.000003999549492
Steady State peak velocity(km/h)= 313.714295998244268
pgopalakumaran@o244-26:~/Cuda-Introduction/problem3$ ^C
```

The final displacement and Steady State peak velocity for dt=0.001 has been compared with dt=0.0001 for the 16 node execution. The impact of the small step size is summarized below.

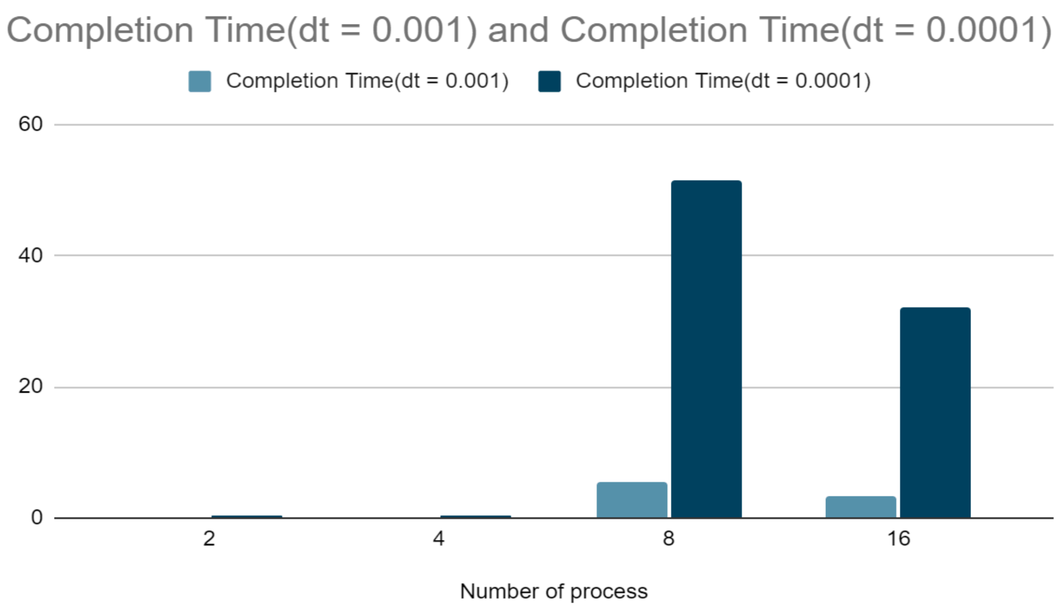
	Final Displacement (km)	**Steady State Peak Velocity (km/h)**
dt=0.001	122.000004002072941	313.714296000050865
dt=0.0001	122.000003999549492	313.714295998244268
Difference (dt=0.001 - dt=0.0001)	0.000000002523009	0.000000001806029

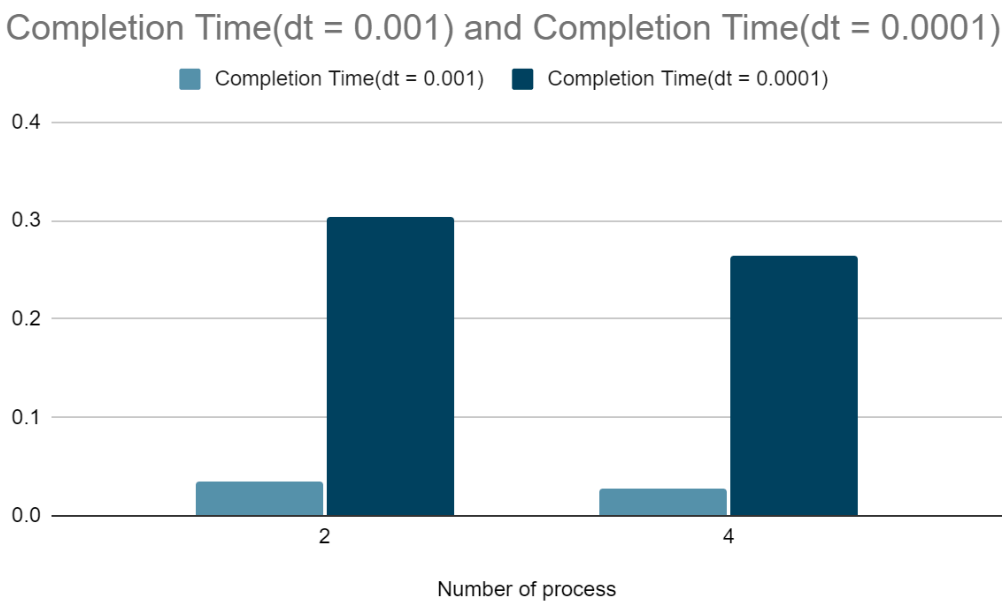
c) Plot the completion time for dt = 0.0001 for 2, 4, 8 and 16 processes on ECC

The code has been executed for 2, 4, 8 and 16 processes through the command line arguments and the execution time has been summarized in the table below:

Number of Process	**Completion Time (dt = 0.001)**	**Completion Time (dt = 0.0001)**
2	0.035317	0.304417
4	0.028104	0.264940

8	5.594737	51.559559
16	3.420795	32.093994


Completion Time(dt = 0.001) and Completion Time(dt = 0.0001)
Completion Time(dt = 0.001)
Completion Time(dt = 0.0001)
Number of process


Completion Time(dt = 0.001) and Completion Time(dt = 0.0001)
Completion Time(dt = 0.001)
Completion Time(dt = 0.0001)
Number of process