Class: Final Year (Computer Science and Engineering)

Year: 2021-22 **Semester:** 1 **Course:** High Performance Computing Lab

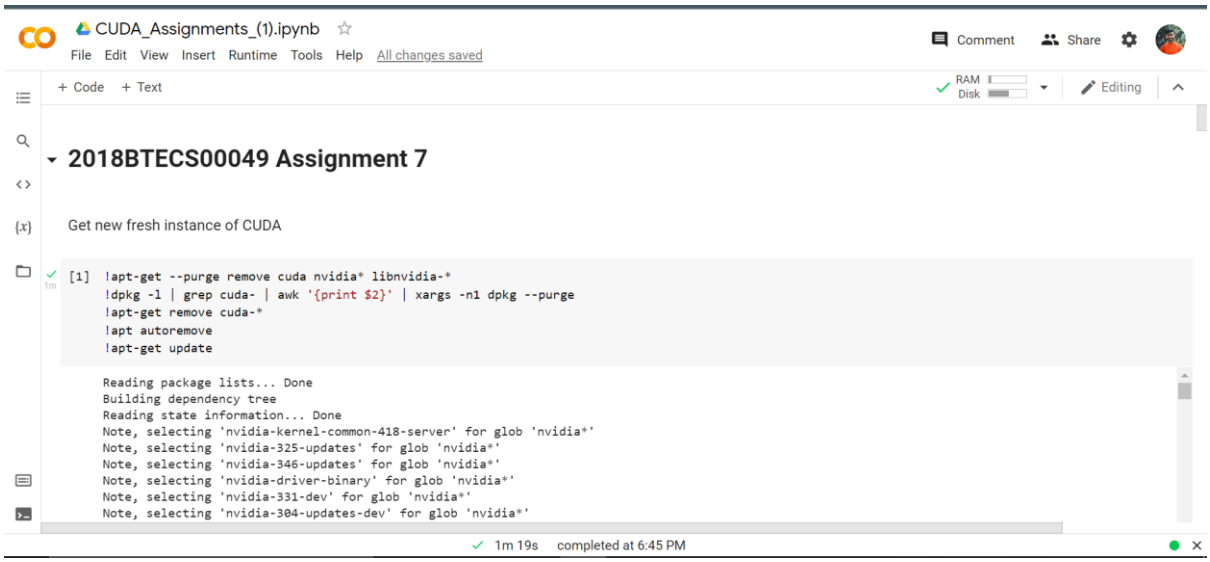
PRN: 2018BTECS00049

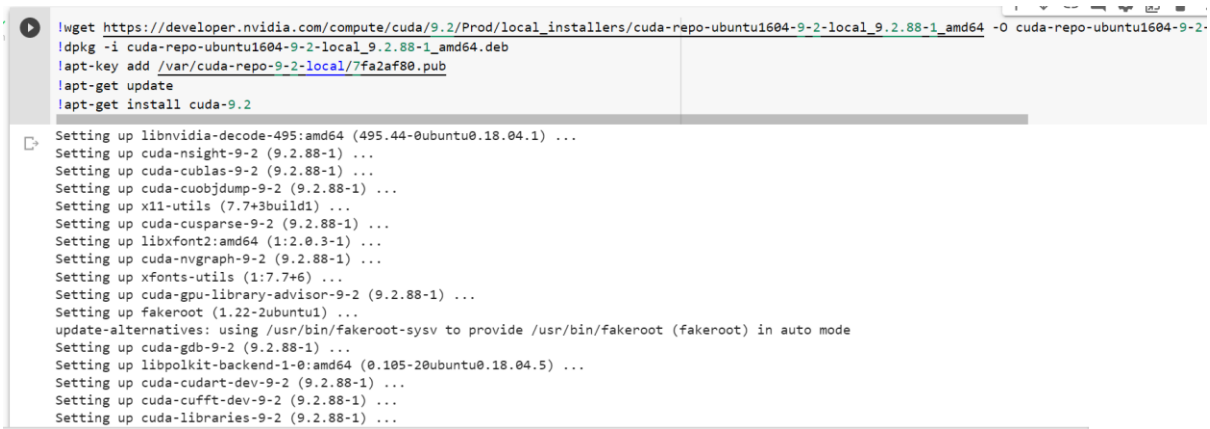
Practical No. 7

Problem Statement 1: Setup the environment requirements, for execution of CUDA C programs.

I have used Google Colab for CUDA pratical as there is no GPU in my Laptop. So there is no setup.

Ans:





CUDA_Assignments_(1).ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

RAM Disk Editing

Check CUDA version

```
[3] !nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2018 NVIDIA Corporation
Built on Wed_Apr_11_23:16:29_CDT_2018
Cuda compilation tools, release 9.2, V9.2.88
```

small extension to run nvcc from the Notebook cells.

```
[4] !pip install git+git://github.com/andreinechaev/nvcc4jupyter.git
```

```
Collecting git+git://github.com/andreinechaev/nvcc4jupyter.git
  Cloning git://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-jwhfe8cs
  Running command git clone -q git://github.com/andreinechaev/nvcc4jupyter.git /tmp/pip-req-build-jwhfe8cs
Building wheels for collected packages: NVCCPlugin
  Building wheel for NVCCPlugin (setup.py) ... done
  Created wheel for NVCCPlugin: filename=NVCCPlugin-0.0.2-py3-none-any.whl size=4305 sha256=230e77c2a19befdb175be04ec56984f36161644efa7400e4a14e5d57db4
  Stored in directory: /tmp/pip-ephem-wheel-cache-gol1vjn2/wheels/c5/2b/c0/87008e795a14bbcdfc7c846a00d06981916331eb980b6c8bdf
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
```

```
%%cu
#include <iostream>
  int
  main()
{
  std::cout << "Welcome To CUDA programming\n";
  return 0;
}
```

```
Welcome To CUDA programming
```

Problem Statement 2: Execute the attached Program 1, and understand the output

```
%%cu
#include <stdio.h>
#include <stdlib.h>
int main() {
      int deviceCount;
      cudaGetDeviceCount(&deviceCount);
      if (deviceCount == 0){
          printf("There is no device supporting CUDA\n");
      }
      int dev;
      for (dev = 0; dev < deviceCount; ++dev) {
          cudaDeviceProp deviceProp;
          cudaGetDeviceProperties(&deviceProp, dev);
          if (dev == 0) {
                if (deviceProp.major < 1){
                  printf("There is no device supporting CUDA.\n");
                }
                else if (deviceCount == 1){
                    printf("There is 1 device supporting CUDA\n");
                }
                else{
                    printf("There are %d devices supporting CUDA\n", deviceCount);
```

```
            }
            else{
                printf("There are %d devices supporting CUDA\n", deviceCount);
            }
        }
        printf("\nDevice %d: \"%s\"\n", dev, deviceProp.name);
        printf("  Major revision number:                         %d\n", deviceProp.major);
        printf("  Minor revision number:                         %d\n", deviceProp.minor);
        printf("  Total amount of global memory:                 %d bytes\n", deviceProp.totalGlobalMem);
        printf("  Total amount of constant memory:               %d bytes\n", deviceProp.totalConstMem);
        printf("  Total amount of shared memory per block:       %d bytes\n", deviceProp.sharedMemPerBlock);
        printf("  Total number of registers available per block: %d\n", deviceProp.regsPerBlock);
        printf("  Warp size:                                     %d\n", deviceProp.warpSize);
        printf("  Multiprocessor count:                          %d\n",deviceProp.multiProcessorCount );
        printf("  Maximum number of threads per block:           %d\n", deviceProp.maxThreadsPerBlock);
        printf("  Maximum sizes of each dimension of a block:    %d x %d x %d\n", deviceProp.maxThreadsDim[0],deviceProp.maxThreadsDim[1], devicePr
        printf("  Maximum sizes of each dimension of a grid:     %d x %d x %d\n", deviceProp.maxGridSize[0], deviceProp.maxGridSize[1],  deviceProp
        printf("  Maximum memory pitch:                          %d bytes\n", deviceProp.memPitch);
        printf("  Texture alignment:                             %d bytes\n", deviceProp.textureAlignment);
        printf("  Clock rate:                                    %d kilohertz\n", deviceProp.clockRate);
    }
}
```

```
There is 1 device supporting CUDA

Device 0: "Tesla K80"
  Major revision number:                         3
  Minor revision number:                         7
  Total amount of global memory:                 -887947264 bytes
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Multiprocessor count:                          13
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     2147483647 x 65535 x 65535
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    823500 kilohertz
```

Problem Statement 3: Write a CUDA C program to perform the addition of two vectors of arbitrary size (Dynamic Array).

```
%%cu
#include<bits/stdc++.h>
using namespace std;

__global__ void addVectors(int *a, int *b, int *c)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    c[id] = a[id] + b[id];
}
int main()
{
    int n = 1e3, i = 0;
    int *host_a, *host_b, *host_c;
    int *devc_a, *devc_b, *devc_c;

    host_a = (int *)malloc(n * sizeof(int));
    host_b = (int *)malloc(n * sizeof(int));
    host_c = (int *)malloc(n * sizeof(int));

    cudaMalloc(&devc_a, n * sizeof(int));
    cudaMalloc(&devc_b, n * sizeof(int));
    cudaMalloc(&devc_c, n * sizeof(int));

    for(i = 0; i < n; i++)
    {
        host_a[i] = i;
        host_b[i] = log(i)/log(2);
    }

    cudaMemcpy(devc_a, host_a, n * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(devc_b, host_b, n * sizeof(int), cudaMemcpyHostToDevice);
```


```
    cudaMemcpy(devc_a, host_a, n * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(devc_b, host_b, n * sizeof(int), cudaMemcpyHostToDevice);
```


```
    int blocks = 1000, threads;
    threads = n / blocks;

    addVectors<<<blocks,threads>>>(devc_a,devc_b,devc_c);

    cudaMemcpy(host_c, devc_c, n * sizeof(int), cudaMemcpyDeviceToHost);

    for(i = 0; i < n; i++)
    {
        cout<<host_a[i]<<" + "<<host_b[i]<<" = "<<host_c[i]<<endl;
    }

    cudaFree(devc_a);
    cudaFree(devc_b);
    cudaFree(devc_c);

    free(host_a);
    free(host_b);
    free(host_c);

    return 0;
}
```