**Concurrent and Parallel Programming**

LAB ASSIGNMENT – 6

**CUDA PROGRAMMING**

Name – Ashutosh Soni

Id – 2018ucp1505

I am using Google Colab for running CUDA program….

**Google Colab Setup:**

Step 1: Go to [https://colab.research.google.com](https://colab.research.google.com/) in Browser and Click on New Python 3 Notebook.

Step 2: Click to Runtime > Change > Hardware Accelerator GPU.

Step 3: Refresh the Cloud Instance of CUDA On Server and run the following script...

!apt-get --purge remove cuda nvidia\* libnvidia-\*  
!dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --purge  
!apt-get remove cuda-\*  
!apt autoremove  
!apt-get update

This script removes the cuda script …

Step 4: Install CUDA Version. Here I am installing version 9.2 of it. For doing this run the following script…

!wget https://developer.nvidia.com/compute/cuda/9.2/Prod/local\_installers/cuda-repo-ubuntu1604-9-2-local\_9.2.88-1\_amd64 -O cuda-repo-ubuntu1604-9-2-local\_9.2.88-1\_amd64.deb  
!dpkg -i cuda-repo-ubuntu1604-9-2-local\_9.2.88-1\_amd64.deb  
!apt-key add /var/cuda-repo-9-2-local/7fa2af80.pub  
!apt-get update  
!apt-get install cuda-9.2

This script installs the cuda 9.2 version on the google colab…

Step 5: Check the Version of CUDA by: running the command below to get the following output:

!nvcc --version

This will get the output:

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

Step 6:**Execute the given command to install a small extension to run nvcc from Notebook cells.**

!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git

Step 7**: Load the extension using this code:**

%load\_ext nvcc\_plugin

**Run all the script in the different shell.**

**We can add it by clicking on +code on left hand side.**

**After that setup for running Cuda on google colab is done.**

**To run the cuda program click play button on left …**

**Q1: WAP for Vector addition in CUDA C.**

**Ans:**

**Program:**

%%cu

// Including Header files

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

using namespace std;

// Declaration

int size\_of\_vector=10000;

// CUDA kernel....

// Addition of vector

\_\_global\_\_ void vector\_Add(int \*vector\_a,int \*vector\_b,int \*vector\_c,int vector\_size){

  // Get our Id

  int id = blockIdx.x\*blockDim.x+threadIdx.x;

  if(id < vector\_size){

    vector\_c[id]= vector\_a[id]+vector\_b[id];

  }

}

void memory\_allocating\_error(){

  printf("Error in Allocating memory for vectors\n");

  printf("Aborting\n");

}

int main(int argc,char \*argv[]){

  // declaration of host input vectors

  int \*host\_vector\_a,\*host\_vector\_b;

  // Declaration of device input vectors

  int \*device\_vector\_a,\*device\_vector\_b;

  // declaration of host output vector

  int \*host\_vector\_result;

  // Declaraton of device output vector

  int \*device\_vector\_result;

  // Allocating memory for each vector in host

  host\_vector\_a= (int\*)malloc(size\_of\_vector\*sizeof(int));

  // printing error if any in creation of vector

  if(host\_vector\_a==NULL){

    memory\_allocating\_error();

  }

  host\_vector\_b= (int\*)malloc(size\_of\_vector\*sizeof(int));

  // printing error if any in creation of vector

  if(host\_vector\_a==NULL){

    memory\_allocating\_error();

  }

  host\_vector\_result= (int\*)malloc(size\_of\_vector\*sizeof(int));

  // printing error if any in creation of vector

  if(host\_vector\_a==NULL){

    memory\_allocating\_error();

  }

  // Allocating memory for each vector in device

  cudaMalloc(&device\_vector\_a,size\_of\_vector\*sizeof(int));

  cudaMalloc(&device\_vector\_b,size\_of\_vector\*sizeof(int));

  cudaMalloc(&device\_vector\_result,size\_of\_vector\*sizeof(int));

  // Initialization

  for(int i=0;i<size\_of\_vector;i++){

    // host vector a by random positive value between 0 and 5

    host\_vector\_a[i]=floor(rand()%5);

    // host vector b by random negative value between 0 and 5

    host\_vector\_b[i]=-1\*floor(rand()%5);

  }

  // Copying host vector to device vector

  cudaMemcpy(device\_vector\_a,host\_vector\_a,size\_of\_vector\*sizeof(int),cudaMemcpyHostToDevice);

  cudaMemcpy(device\_vector\_b,host\_vector\_b,size\_of\_vector\*sizeof(int),cudaMemcpyHostToDevice);

  int number\_of\_threads,number\_of\_blocks;

  // Initializing number of threads per block

  number\_of\_threads= 1000;

  // Initializing number of blocks

  number\_of\_blocks=size\_of\_vector/number\_of\_threads;

  // Executing kernel...

  vector\_Add<<<number\_of\_blocks,number\_of\_threads>>> (device\_vector\_a,device\_vector\_b,device\_vector\_result,size\_of\_vector);

  // Copying device result array back to the host result array

  cudaMemcpy(host\_vector\_result,device\_vector\_result,size\_of\_vector\*sizeof(int),cudaMemcpyDeviceToHost);

  // Decalaration of total sum variable

  int total\_sum=0;

  for(int i=0;i<size\_of\_vector;i++){

    total\_sum+=host\_vector\_result[i];

  }

  printf("Final result of both the arrays is : %d\n",total\_sum);

  // Releasing device memory

  cudaFree(device\_vector\_a);

  cudaFree(device\_vector\_b);

  cudaFree(device\_vector\_result);

  // Releasing host memory

  free(host\_vector\_a);

  free(host\_vector\_b);

  free(host\_vector\_result);

  return 0;

}

**Output:**

Final result of both the arrays is: 254

Q2: WAP for Matrix-matrix multiplication in CUDA C.

Ans:

**Program:**

%%cu

// Including Header files

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

using namespace std;

#define tile\_width 2

int width=6;

// Non sharable matrix multiplication

\_\_global\_\_ void square\_matrix\_multiply(int \*matrix\_a ,int \*matrix\_b ,int \*matrix\_result ,int width)

{

  // Calculate row and column

  int col= tile\_width\*blockIdx.x + threadIdx.x;

  int row= tile\_width\*blockIdx.y + threadIdx.y;

  // Multiplying and storing the result

  for(int k=0;k<width;k++){

    matrix\_result[row\*width + col] += matrix\_a[row\*width + k] \* matrix\_b[k\*width + col];

  }

}

int main(int argc,char \*argv[]){

  // declaration of host matrix input

  int host\_matrix\_a[width][width],host\_matrix\_b[width][width];

  // Declaration of device matrix input

  int \*device\_matrix\_a,\*device\_matrix\_b;

  // declaration of host matrix result

  int host\_matrix\_result[width][width];

  // Declaration of device matrix result

  int \*device\_matrix\_result;

  //Allocating memory for device matrix

  cudaMalloc(&device\_matrix\_a,width\*width\*sizeof(int));

  cudaMalloc(&device\_matrix\_b,width\*width\*sizeof(int));

  cudaMalloc(&device\_matrix\_result,width\*width\*sizeof(int));

  // Initialization of host input matrix

  for(int i=0;i<width;i++){

    for(int j=0;j<width;j++){

      host\_matrix\_a[i][j]=1;

      host\_matrix\_b[i][j]=2;

    }

  }

  cudaMemcpy(device\_matrix\_a,host\_matrix\_a,width\*width\*sizeof(int),cudaMemcpyHostToDevice);

  cudaMemcpy(device\_matrix\_b,host\_matrix\_b,width\*width\*sizeof(int),cudaMemcpyHostToDevice);

  dim3 dimGrid(width/tile\_width,width/tile\_width,1);

  dim3 dimBlock(tile\_width,tile\_width,1);

  square\_matrix\_multiply<<<dimGrid,dimBlock>>> (device\_matrix\_a,device\_matrix\_b,device\_matrix\_result,width);

  cudaMemcpy(host\_matrix\_result,device\_matrix\_result,width\*width\*sizeof(int),cudaMemcpyDeviceToHost);

  printf("Matrix After Multiplication is: \n");

  for(int i=0;i<width;i++){

    for(int j=0;j<width;j++){

      printf("%d  ",host\_matrix\_result[i][j]);

    }

    printf("\n");

  }

  cudaFree(device\_matrix\_a);

  cudaFree(device\_matrix\_b);

  cudaFree(device\_matrix\_result);

  return 0;

}

**Output:**

Matrix After Multiplication is:

12 12 12 12 12 12

12 12 12 12 12 12

12 12 12 12 12 12

12 12 12 12 12 12

12 12 12 12 12 12

12 12 12 12 12 12