NAME-ANSH GOEL REG. NO-20BCE1798 LAB11-CUDA-THREADS

```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Sun_Feb_14_21:12:58_PST_2021
Cuda compilation tools, release 11.2, V11.2.152
Build cuda_11.2.r11.2/compiler.29618528_0
```

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

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/p
Collecting git+https://github.com/andreinechaev/nvcc4jupyter.git
  Cloning https://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-_5t
  Running command git clone -q https://github.com/andreinechaev/nvcc4jupyter.git /tmp
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=4306
  Stored in directory: /tmp/pip-ephem-wheel-cache-j5qz9yun/wheels/ca/33/8d/3c86eb85e9
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
```

```
%load_ext nvcc_plugin
```

```
created output directory at /content/src
Out bin /content/result.out
```

```
%%cu
#include<stdio.h>
#include<cuda.h>

int main()
{
    cudaDeviceProp p;
    int device_id;
    int major;
    int minor;


    cudaGetDevice(&device_id);
    cudaGetDeviceProperties(&p,device_id);

    major=p.major;
    minor=p.minor;

    printf("Name of GPU on your system is %s\n",p.name);
```

0s completed at 11:27 PM

```
    printf("\n Compute Capability of a current GPU on your system is %d.%d",major,minor);

    return 0;
}
```

```
 Name of GPU on your system is Tesla T4

  Compute Capability of a current GPU on your system is 7.5
```

```
%%cu
#include <stdio.h>
__global__ void Hellokernel()
{

}
main()
{
Hellokernel << <1, 1 >> > ();
printf("Hello World\n");
return 0;
}
```

```
 Hello World
```

```
%%cu
#include <stdio.h>
__global__ void add(int a, int b, int *c)
{
*c = a + b;
}
int main(void)
{
int c;
int *dev_c;
cudaMalloc((void**)&dev_c, sizeof(int));
add << <1, 1 >> > (2, 7, dev_c);
cudaMemcpy(&c, dev_c, sizeof(int),
cudaMemcpyDeviceToHost);
printf("2 + 7 = %d\n", c);
cudaFree(dev_c);
return 0;
}
```

```
 2 + 7 = 9
```

```
%%cu
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 100000;

    // Host input vectors
    double *h_a;
    double *h_b;
    //Host output vector
    double *h_c;

    // Device input vectors
    double *d_a;
    double *d_b;
    //Device output vector
    double *d_c;

    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);

    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);

    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes);
    cudaMalloc(&d_b, bytes);
    cudaMalloc(&d_c, bytes);

    int i;
    // Initialize vectors on host
    for( i = 0; i < n; i++ ) {
        h_a[i] = sin(i)*sin(i);
        h_b[i] = cos(i)*cos(i);
    }
```

```
    // Copy host vectors to device
    cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

    int blockSize, gridSize;

    // Number of threads in each thread block
    blockSize = 1024;

    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n/blockSize);

    // Execute the kernel
    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

    // Copy array back to host
    cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

    // Sum up vector c and print result divided by n, this should equal 1 within error
    double sum = 0;
    for(i=0; i<n; i++)
        sum += h_c[i];
    printf("final result: %f\n", sum/n);

    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

```
 final result: 1.000000
```

```
%%cu
#include <stdio.h>
#include <math.h>
#define TILE_WIDTH 2
__global__ void MatrixMul( float *Md , float *Nd , float *Pd , const int WIDTH )
{
unsigned int col = TILE_WIDTH*blockIdx.x + threadIdx.x ;
unsigned int row = TILE_WIDTH*blockIdx.y + threadIdx.y ;
for (int k = 0 ; k<WIDTH ; k++ )
```

```
{
Pd[row*WIDTH + col]+= Md[row * WIDTH + k ] * Nd[ k * WIDTH + col] ;
}
}
__global__ void MatrixMulSh( float *Md , float *Nd , float *Pd , const int WIDTH )
{
__shared__ float Mds [TILE_WIDTH][TILE_WIDTH] ;
__shared__ float Nds [TILE_WIDTH][TILE_WIDTH] ;
unsigned int col = TILE_WIDTH*blockIdx.x + threadIdx.x ;
unsigned int row = TILE_WIDTH*blockIdx.y + threadIdx.y ;
for (int m = 0 ; m<WIDTH/TILE_WIDTH ; m++ )
{
Mds[threadIdx.y][threadIdx.x] = Md[row*WIDTH + (m*TILE_WIDTH + threadIdx.x)] ;
Nds[threadIdx.y][threadIdx.x] = Nd[ ( m*TILE_WIDTH + threadIdx.y) * WIDTH + col] ;
__syncthreads() ;
for ( int k = 0; k<TILE_WIDTH ; k++ )
Pd[row*WIDTH + col]+= Mds[threadIdx.x][k] * Nds[k][threadIdx.y] ;
__syncthreads() ;
}
}

int main ()
{
const int WIDTH = 6 ;
float array1_h[WIDTH][WIDTH] ,array2_h[WIDTH][WIDTH],
result_array_h[WIDTH][WIDTH] ,M_result_array_h[WIDTH][WIDTH] ;
float *array1_d , *array2_d ,*result_array_d ,*M_result_array_d ;
int i , j ;
for ( i = 0 ; i<WIDTH ; i++ )
{
for (j = 0 ; j<WIDTH ; j++ )
{
array1_h[i][j] = 1 ;
array2_h[i][j] = 2 ;
}
}
cudaMalloc((void **) &array1_d , WIDTH*WIDTH*sizeof (int) ) ;
cudaMalloc((void **) &array2_d , WIDTH*WIDTH*sizeof (int) ) ;
cudaMemcpy ( array1_d , array1_h , WIDTH*WIDTH*sizeof (int) , cudaMemcpyHostToDevi
cudaMemcpy ( array2_d , array2_h , WIDTH*WIDTH*sizeof (int) , cudaMemcpyHostToDevi
cudaMalloc((void **) &result_array_d , WIDTH*WIDTH*sizeof (int) ) ;
cudaMalloc((void **) &M_result_array_d , WIDTH*WIDTH*sizeof (int) ) ;
dim3 dimGrid ( WIDTH/TILE_WIDTH , WIDTH/TILE_WIDTH ,1 ) ;
dim3 dimBlock( TILE_WIDTH, TILE_WIDTH, 1 ) ;
#if 0
MatrixMul <<<dimGrid,dimBlock>>> ( array1_d , array2_d ,M_result_array_d , WIDTH)
#endif
#if 1
MatrixMulSh<<<dimGrid,dimBlock>>> ( array1_d , array2_d ,M_result_array_d , WIDTH)
#endif
```

```
cudaMemcpy(M_result_array_h , M_result_array_d ,                    (    ) ,
for ( i = 0 ; i<WIDTH ; i++ )
{
for ( j = 0 ; j < WIDTH ; j++ )
{
printf ("%f ",M_result_array_h[i][j] ) ;
}
printf ("\n") ;
}
system("pause") ;
}
```

```
sh: 1: pause: not found
12.000000 12.000000 12.000000 12.000000 12.000000 12.000000
12.000000 12.000000 12.000000 12.000000 12.000000 12.000000
12.000000 12.000000 12.000000 12.000000 12.000000 12.000000
12.000000 12.000000 12.000000 12.000000 12.000000 12.000000
12.000000 12.000000 12.000000 12.000000 12.000000 12.000000
12.000000 12.000000 12.000000 12.000000 12.000000 12.000000
```