Name: Muhammad Allah Rakha
Roll No: P19-0006 BCS-6A (PDC)

Assignment: NVIDIA CUDA on Google Colab

GitHub Link Code: https://github.com/aaaastark/NVIDIA-CUDA-Google-Colab.git

CUDA

Kernel, Grid, Block, Threads and Dimensional of a Block/Grid

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<…>>> execution configuration syntax. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable.

Syntax: **Kernel_Name<<< GridSize, BlockSize, SMEMSize, Stream >>> (arg,...);**

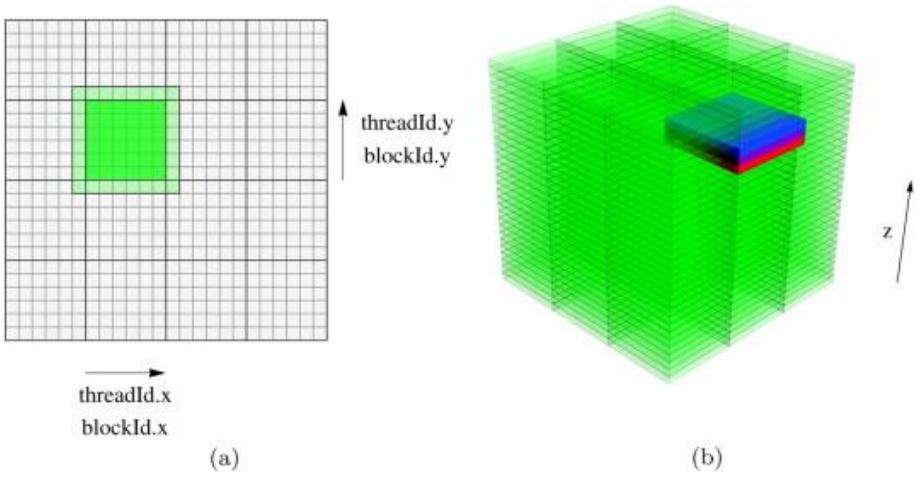
SMEMsize: is the size of Shared Memory at Runtime.
Stream: is a stream on which kernel will execute.

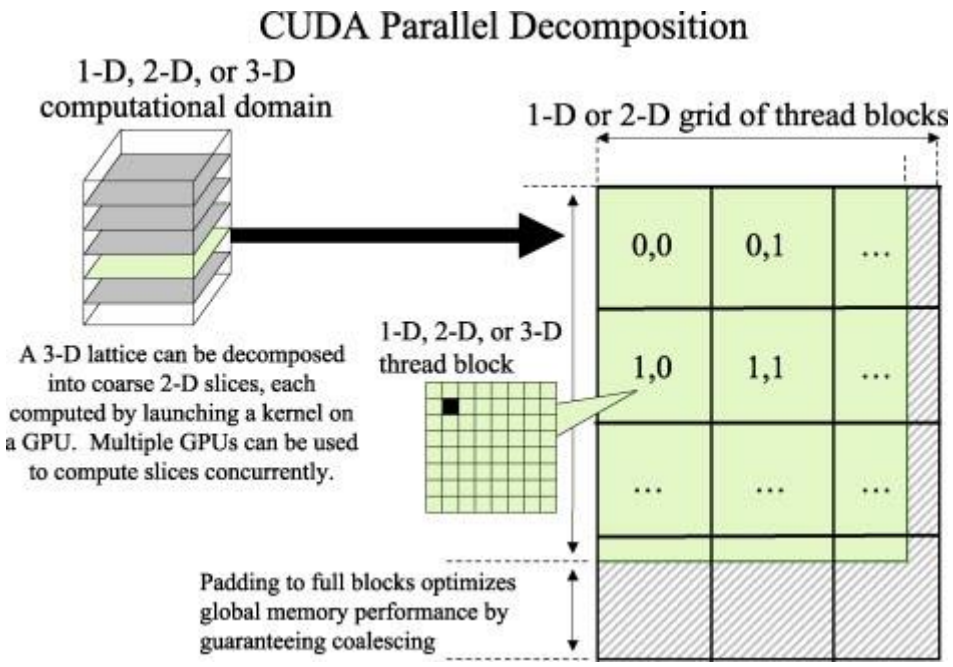
Sample Example:

Here, each of the N threads that execute VecAdd() performs one pair-wise addition.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
        int i = threadIdx.x; C[i] = A[i] + B[i];
}
int main()
{ ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
     ...

}
```

For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.





The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (Dx, Dy), the thread ID of a thread of index (x, y) is (x + y Dx); for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (x + y Dx + z Dx Dy).

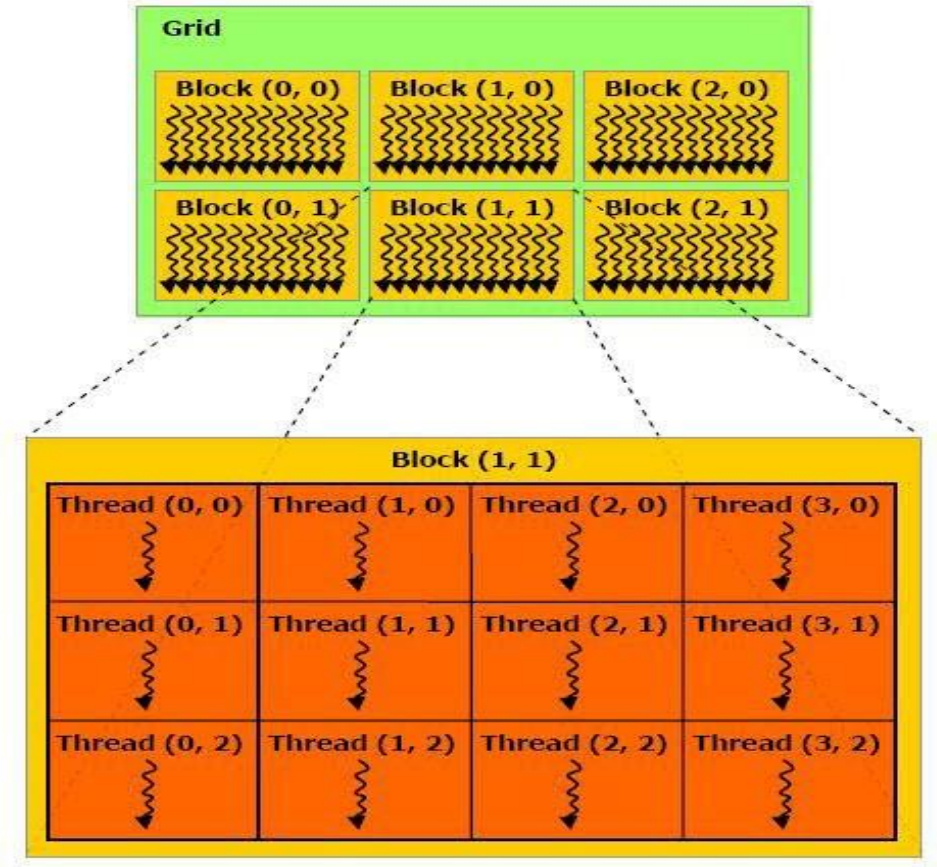
As an example, the following code adds two matrices A and B of size NxN and stores the result into matrix.

C Language:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
int main()
{
 ... // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
 ...
}
```

Threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.
However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.
Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks as illustrated by **Figures** the number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

The number of threads per block and the number of blocks per grid specified in the <<<…>>> syntax can be of type int or dim3. Two-dimensional blocks or grids can be specified as in the example above.
Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.



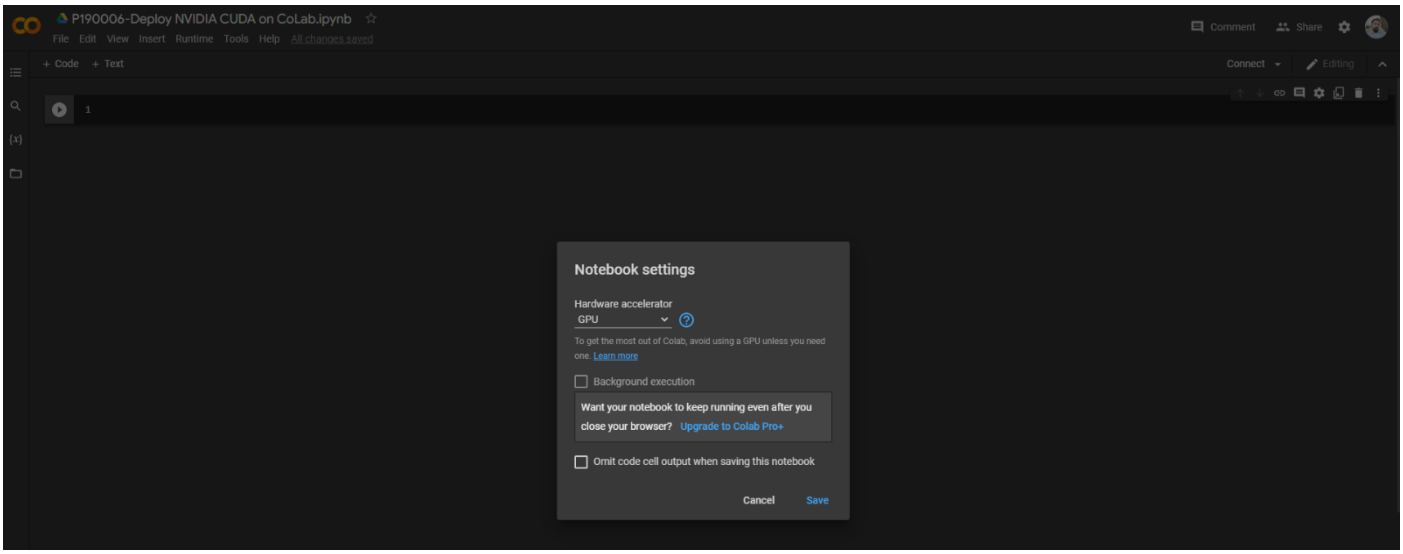
Extending the previous MatAdd() example to handle multiple blocks, the code becomes as follows.

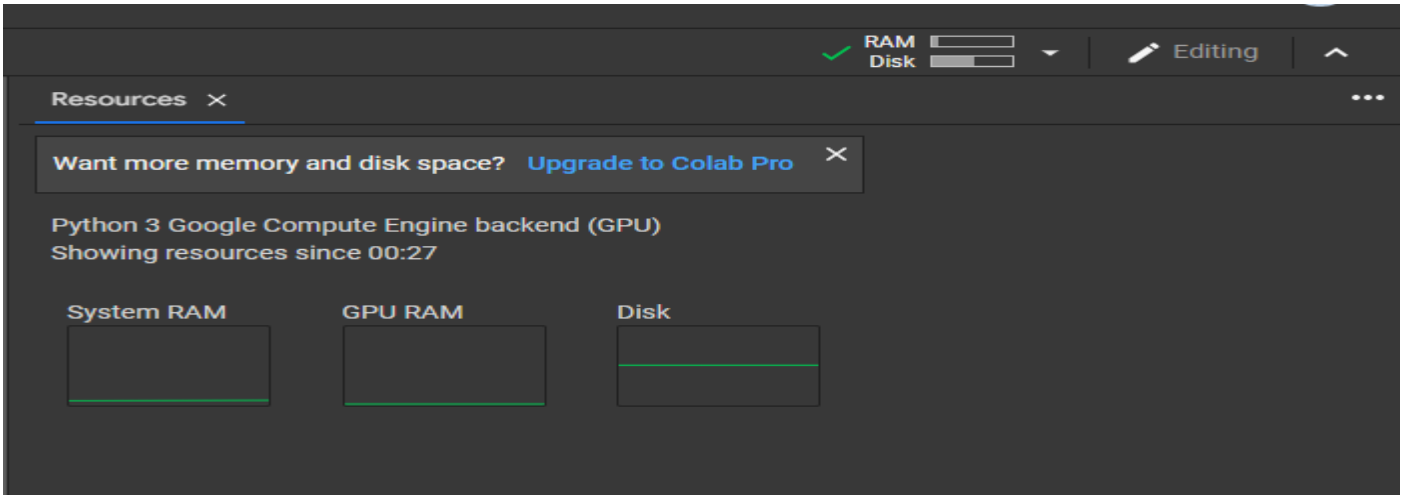
A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.

```
// Kernel definition

__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i < N && j < N)
            C[i][j] = A[i][j] + B[i][j];

}

int main()
{
...
// Kernel invocation
    dim3 threadsPerBlock(16, 16);

   dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);

    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
 ...
}
```

CUDA

Deploy the NVIDIA CUDA on Google Colab (Using GPU)






```
1  # GPU checking purpose run this command.
2  !nvidia-smi
```

```
Sat Apr 23 19:28:42 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| 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  Tesla K80           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   56C    P8    31W / 149W |      0MiB / 11441MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
[2]  1  # Refresh the Cloud Instance of CUDA on Server
     2
     3  !apt-get --purge remove cuda nvidia* libnvidia-*
     4  !dpkg -l | grep cuda- | awk '{print $2}' | xargs -n1 dpkg --purge
     5  !apt-get remove cuda-*
     6  !apt autoremove
     7  !apt-get update
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
Note, selecting 'nvidia-kernel-common-418-server' for glob 'nvidia*'
Note, selecting 'nvidia-325-updates' for glob 'nvidia*'
Note, selecting 'nvidia-346-updates' for glob 'nvidia*'
Note, selecting 'nvidia-driver-binary' for glob 'nvidia*'
```

```
Get:19 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu bionic/main Sources [1,950 kB]
Get:20 http://security.ubuntu.com/ubuntu bionic-security/restricted amd64 Packages [907 kB]
Get:21 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 Packages [1,496 kB]
Get:22 http://security.ubuntu.com/ubuntu bionic-security/main amd64 Packages [2,728 kB]
Get:23 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu bionic/main amd64 Packages [999 kB]
Fetched 14.7 MB in 4s (3,899 kB/s)
Reading package lists... Done
```

```
[5]  1  # Install CUDA Version 9
     2
     3  !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
     4  !dpkg -i cuda-repo-ubuntu1604-9-2-local_9.2.88-1_amd64.deb
     5  !apt-key add /var/cuda-repo-9-2-local/7fa2af80.pub
     6  !apt-get update
     7  !apt-get install cuda-9.2
```

```
depmod...

DKMS: install completed.
Setting up cuda-cuobjdump-9-2 (9.2.88-1) ...
Setting up x11-utils (7.7+3build1) ...
Setting up cuda-cusparse-9-2 (9.2.88-1) ...
Setting up libxfont2:amd64 (1:2.0.3-1) ...
Setting up cuda-nvgraph-9-2 (9.2.88-1) ...
Setting up xfonts-utils (1:7.7+6) ...
Setting up cuda-gpu-library-advisor-9-2 (9.2.88-1) ...
Setting up fakeroot (1.22-2ubuntu1) ...
update-alternatives: using /usr/bin/fakeroot-sysv to provide /usr/bin/fakeroot (fakeroot) in auto mode
Setting up cuda-gdb-9-2 (9.2.88-1) ...
```

```
Processing triggers for dbus (1.12.2-1ubuntu1.2) ...
Processing triggers for hicolor-icon-theme (0.17-2) ...
Processing triggers for fontconfig (2.12.6-0ubuntu2) ...
Processing triggers for mime-support (3.60ubuntu1) ...
Processing triggers for libc-bin (2.27-3ubuntu1.3) ...
/sbin/ldconfig.real: /usr/local/lib/python3.7/dist-packages/ideep4py/lib/libmkldnn.so.0 is not a symbolic link
```

```
     1  # Check the version of CUDA.
     2
     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
```

[8]
```
# Execute the given command to Install a Small Extension to Run NVCC from Notebook cells.

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

```
Collecting git+https://github.com/andreinechaev/nvcc4jupyter.git
  Cloning https://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-7ujcw79x
  Running command git clone -q https://github.com/andreinechaev/nvcc4jupyter.git /tmp/pip-req-build-7ujcw79x
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 sha256=2ef1f55e5495dafe31bd8b629125fb80c1d766dddc7917a4143c64f92c9ec863
  Stored in directory: /tmp/pip-ephem-wheel-cache-s6wvfovl/wheels/ca/33/8d/3c86eb85e97d2b6169d95c6e8f2c297fdec60db6e84cb56f5e
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
```

[10]
```
# Load the Extension using this code.

%load_ext nvcc_plugin
```

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

```
%%cu
#include <stdio.h>
#include <stdlib.h>
__global__ void add(int *a, int *b, int *c) {
*c = *a + *b;
}
int main() {
  int a, b, c;
  // host copies of variables a, b & c
  int *d_a, *d_b, *d_c;
  // device copies of variables a, b & c
  int size = sizeof(int);
  // Allocate space for device copies of a, b, c
  cudaMalloc((void **)&d_a, size);
  cudaMalloc((void **)&d_b, size);
  cudaMalloc((void **)&d_c, size);
  // Setup input values
  c = 0;
  a = 3;
  b = 5;
  // Copy inputs to device
  cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
  // Launch add() kernel on GPU
  add<<<1,1>>>(d_a, d_b, d_c);
  // Copy result back to host
  cudaError err = cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
    if(err!=cudaSuccess) {
        printf("CUDA error copying to Host: %s\n", cudaGetErrorString(err));
    }
  printf("result is %d\n",c);
  // Cleanup
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);
  return 0;
}
```

```
result is 8
```

CUDA

Matrix Multiplicaton

Matrix Multiplication

```
%%cu

#include <stdio.h>
#include <math.h>
#define TILE_WIDTH 2

/*matrix multiplication kernels*/
//non shared

__global__ void MatrixMul( float *Md , float *Nd , float *Pd , const int WIDTH )
{
  // calculate thread id

  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] ;
  }
}

// shared
__global__ void MatrixMulSh( float *Md , float *Nd , float *Pd , const int WIDTH )
{
  //Taking shared array to break the MAtrix in Tile widht and fatch them in that array per ele

  __shared__ float Mds [TILE_WIDTH][TILE_WIDTH] ;
  __shared__ float Nds [TILE_WIDTH][TILE_WIDTH] ;

  // calculate thread id
  unsigned int col = TILE_WIDTH*blockIdx.x + threadIdx.x ;
  unsigned int row = TILE_WIDTH*blockIdx.y + threadIdx.y ;

```

```
34    for (int m = 0 ; m<WIDTH/TILE_WIDTH ; m++ ) // m indicate number of phase
35    {
36        Mds[threadIdx.y][threadIdx.x] =  Md[row*WIDTH + (m*TILE_WIDTH + threadIdx.x)]  ;
37        Nds[threadIdx.y][threadIdx.x] =  Nd[ ( m*TILE_WIDTH + threadIdx.y) * WIDTH + col] ;
38      __syncthreads() ; // for syncronizeing the threads
39
40      // Do for tile
41        for ( int k = 0; k<TILE_WIDTH ; k++ )
42                        Pd[row*WIDTH + col]+= Mds[threadIdx.x][k] * Nds[k][threadIdx.y] ;
43      __syncthreads() ; // for syncronizeing the threads
44
45    }
46  }
47
48  // main routine
49  int main ()
50  {
51     const int WIDTH = 6 ;
52     float array1_h[WIDTH][WIDTH] ,array2_h[WIDTH][WIDTH],
53                        result_array_h[WIDTH][WIDTH] ,M_result_array_h[WIDTH][WIDTH]  ;
54    float *array1_d , *array2_d ,*result_array_d  ,*M_result_array_d ; // device array
55    int i , j ;
56    //input in host array
57    for ( i = 0 ; i<WIDTH ; i++ )
58    {
59        for (j = 0 ; j<WIDTH ; j++ )
60        {
61            array1_h[i][j] = 1 ;
62            array2_h[i][j] = 2 ;
63        }
64    }
65
66    //create device array cudaMalloc ( (void **)&array_name, sizeofmatrixinbytes) ;
67    cudaMalloc((void **) &array1_d , WIDTH*WIDTH*sizeof (int) ) ;
68    cudaMalloc((void **) &array2_d , WIDTH*WIDTH*sizeof (int) ) ;
69
70    //copy host array to device array; cudaMemcpy ( dest , source , WIDTH , direction )
71    cudaMemcpy ( array1_d , array1_h , WIDTH*WIDTH*sizeof (int) , cudaMemcpyHostToDevice ) ;
72    cudaMemcpy ( array2_d , array2_h , WIDTH*WIDTH*sizeof (int) , cudaMemcpyHostToDevice ) ;
73
74    //allocating memory for resultent device array
75    cudaMalloc((void **) &result_array_d , WIDTH*WIDTH*sizeof (int) ) ;
76    cudaMalloc((void **) &M_result_array_d , WIDTH*WIDTH*sizeof (int) ) ;
77
```

```
  //calling kernal
  dim3 dimGrid ( WIDTH/TILE_WIDTH , WIDTH/TILE_WIDTH ,1 ) ;
  dim3 dimBlock( TILE_WIDTH, TILE_WIDTH, 1 ) ;

// Change if 0 to if 1 for running non shared code and make if 0 for shared memory code
#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

  // all gpu function blocked till kernel is working
  //copy back result_array_d to result_array_h

  cudaMemcpy(M_result_array_h , M_result_array_d , WIDTH*WIDTH*sizeof(int) ,
                                                   cudaMemcpyDeviceToHost) ;

  //printf the result array
  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
```

CUDA

Vector Additon

Vector Addition CUDA

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

#define HANDLE_ERROR( err ) ( HandleError( err, __FILE__, __LINE__ ) )

static void HandleError( cudaError_t err, const char *file, int line )
{
    if (err != cudaSuccess)
      {
        printf( "%s in %s at line %d\n", cudaGetErrorString( err ),
                file, line );
        exit( EXIT_FAILURE );
    }
}

const short N = 10 ;

// CUDA Kernel for Vector Addition
__global__ void Vector_Addition ( const int *dev_a , const int *dev_b , int *dev_c)
{
      //Get the id of thread within a block
      unsigned short tid = blockIdx.x ;

      if ( tid < N ) // check the boundry condition for the threads
          dev_c [tid] = dev_a[tid] + dev_b[tid] ;

}
```

```
int main (void)
{
    //Host array
    int Host_a[N], Host_b[N], Host_c[N];

    //Device array
    int *dev_a , *dev_b, *dev_c ;

    //Allocate the memory on the GPU
    HANDLE_ERROR ( cudaMalloc((void **)&dev_a , N*sizeof(int) ) );
    HANDLE_ERROR ( cudaMalloc((void **)&dev_b , N*sizeof(int) ) );
    HANDLE_ERROR ( cudaMalloc((void **)&dev_c , N*sizeof(int) ) );

    //fill the Host array with random elements on the CPU
    for ( int i = 0; i <N ; i++ )
    {
        Host_a[i] = -i ;
        Host_b[i] = i*i ;
    }

    //Copy Host array to Device array
    HANDLE_ERROR (cudaMemcpy (dev_a , Host_a , N*sizeof(int) , cudaMemcpyHostToDevice));
    HANDLE_ERROR (cudaMemcpy (dev_b , Host_b , N*sizeof(int) , cudaMemcpyHostToDevice));

    //Make a call to GPU kernel
    Vector_Addition <<< N, 1  >>> (dev_a , dev_b , dev_c ) ;

    //Copy back to Host array from Device array
    HANDLE_ERROR (cudaMemcpy(Host_c , dev_c , N*sizeof(int) , cudaMemcpyDeviceToHost));

    //Display the result
    for ( int i = 0; i<N; i++ )
        printf ("%d + %d = %d\n", Host_a[i] , Host_b[i] , Host_c[i] ) ;
    //Free the Device array memory
    cudaFree (dev_a) ;
    cudaFree (dev_b) ;
    cudaFree (dev_c) ;

    system("pause");
    return 0 ;
}
```

```
sh: 1: pause: not found
0 + 0 = 0
-1 + 1 = 0
-2 + 4 = 2
-3 + 9 = 6
-4 + 16 = 12
-5 + 25 = 20
-6 + 36 = 30
-7 + 49 = 42
-8 + 64 = 56
-9 + 81 = 72
```