Addition of Vectors using CUDA Code:

```
%%cu
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

__global__ void addKernel(int* c, const int* a, const int* b, int size) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i < size) {
      c[i] = a[i] + b[i];
   }
}

// Helper function for using CUDA to add vectors in parallel.
void addWithCuda(int* c, const int* a, const int* b, int size) {
   int* dev_a = nullptr;
   int* dev_b = nullptr;
   int* dev_c = nullptr;

   // Allocate GPU buffers for three vectors (two input, one output)
   cudaMalloc((void**)&dev_c, size * sizeof(int));
   cudaMalloc((void**)&dev_a, size * sizeof(int));
   cudaMalloc((void**)&dev_b, size * sizeof(int));

   // Copy input vectors from host memory to GPU buffers.
   cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

   // Launch a kernel on the GPU with one thread for each element.
   // 2 is number of computational blocks and (size + 1) / 2 is a number of threads in a block
   addKernel<<<2, (size + 1) / 2>>>(dev_c, dev_a, dev_b, size);

   // cudaDeviceSynchronize waits for the kernel to finish, and returns any errors encountered during
  //the launch.
   cudaDeviceSynchronize();

   // Copy output vector from GPU buffer to host memory.
   cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);

   cudaFree(dev_c);
   cudaFree(dev_a);
   cudaFree(dev_b);
}

int main(int argc, char** argv) {
   const int arraySize = 5;
   const int a[arraySize] = {  1,  2,  3,  4,  5 };
```

```
    const int b[arraySize] = { 10, 20, 30, 40, 50 };
    int c[arraySize] = { 0 };
    addWithCuda(c, a, b, arraySize);
    printf("{1, 2, 3, 4, 5} + {10, 20, 30, 40, 50} = {%d, %d, %d, %d, %d}\n", c[0], c[1], c[2], c[3], c[4]);
    cudaDeviceReset();
    return 0;
}
```

Addition of Vectors using CUDA Output:

```
{1, 2, 3, 4, 5} + {10, 20, 30, 40, 50} = {11, 22, 33, 44, 55}
```

Matrix Multiplication using CUDA C Code:

```
%%cu
#include<stdio.h>
#include<cuda.h>
#define row1 2 /* Number of rows of first matrix */
#define col1 3 /* Number of columns of first matrix */
#define row2 3 /* Number of rows of second matrix */
#define col2 2 /* Number of columns of second matrix */
__global__ void matproduct(int *l, int *m, int *n){
    int x = blockIdx.x;
    int y = blockIdx.y;
    int k;
    n[col2 * y + x] = 0;
    for(k = 0; k < col1; k++){
      n[col2 * y + x] = n[col2 * y + x] + l[col1 * y + k] * m[col2 * k + x];
    }
}

int main(){
    //# int row1 = 2, row2 = 3, col1 = 3, col2 = 2;
    int a[row1][col1] = {{1, 2, 3}, {4, 5, 6}};
    int b[row2][col2] = {{9, 8}, {6, 5}, {3, 2}};
    int c[row1][col2];
    int *d, *e, *f;
    int i, j;
    cudaMalloc((void **)&d, row1*col1*sizeof(int));
    cudaMalloc((void **)&e, row2*col2*sizeof(int));
    cudaMalloc((void **)&f, row1*col2*sizeof(int));
    cudaMemcpy(d, a, row1*col1*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(e, b, row2*col2*sizeof(int), cudaMemcpyHostToDevice);
    dim3 grid(col2, row1);
```

```
   # /* Here we are defining two dimensional Grid(collection of blocks) structure. Syntax is dim3
grid(no. of columns,no. of rows) */
   matproduct<<<grid,1>>>(d,e,f);
   cudaMemcpy(c, f, row1*col2*sizeof(int), cudaMemcpyDeviceToHost);
   printf("\nProduct of two matrices:\n ");

   for(i = 0; i < row1; i++){
     for(j = 0; j < col2; j++){
       printf("%d\t",c[i][j]);
     }
     printf("\n");
   }
   cudaFree(d);
   cudaFree(e);
   cudaFree(f);
   return 0;
}
```

<u>Matrix Multiplication using CUDA C Output:</u>

```
Product of two matrices:
 30     24
84      69
```