CUDA Program for Addition of Two Large Vectors:

```
#include <stdio.h>

#include <stdlib.h>

// CUDA kernel for vector addition

__global__ void vectorAdd(int *a, int *b, int *c, int n) { int i =
   blockIdx.x * blockDim.x + threadIdx.x; if (i < n) {

      c[i] = a[i] + b[i];

   }

}

int main() {

   int n = 1000000; // Vector size

   int *a, *b, *c; // Host vectors

   int *d_a, *d_b, *d_c; // Device vectors

   int size = n * sizeof(int); // Size in bytes

   //    Allocate memory for host vectors a =
   (int*) malloc(size);

   b = (int*) malloc(size); c = (int*)
   malloc(size);

   // Initialize host vectors

   for (int i = 0; i < n; i++) {

      a[i] = i;

      b[i] = i;

   }
```

```
//      Allocate memory for device vectors
cudaMalloc((void**) &d_a, size);

cudaMalloc((void**) &d_b, size);

cudaMalloc((void**) &d_c, size);

//Copy host vectors to device vectors

cudaMemcpy(d_a,      a,      size,      cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

//      Define block size and grid size int
blockSize = 256;

int gridSize = (n + blockSize - 1) / blockSize;

//Launch kernel

vectorAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);

//      Copy device result vector to host result vector
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

//Verify the result

for (int i = 0; i < n; i++) {

  if (c[i] != 2*i) {

    printf("Error: c[%d] = %d\n", i, c[i]);

    break;

  }

}

//      Free device memory
cudaFree(d_a); cudaFree(d_b);
cudaFree(d_c);
```

```
//       Free host memory free(a);

free(b);

free(c);

return 0;}
```

This program uses CUDA to add two large vectors of size 1000000. The vectors are initialized on the host, and then copied to the device memory. A kernel function is defined to perform the vector addition, and then launched on the device. The result is copied back to the host memory andverified. Finally, the device and host memories are freed.

CUDA Program for Matrix Multiplication:

This program multiplies two matrices of size n using CUDA. It first allocates host memory for the matrices and initializes them. Then it allocates device memory and copies the matrices to the device. It sets the kernel launch configuration and launches the kernel function matrix_multiply. The kernel function performs the matrix multiplication and stores the result in matrix c. Finally, it copies the result back to the host and frees the device and host memory.

The kernel function calculates the row and column indices of the output matrix using the block index and thread index. It then uses a for loop to calculate the sum of the products of the corresponding elements in the input matrices. The result is stored in the output matrix.

Note that in this program, we use CUDA events to measure the elapsed time of the kernel function. This is because the kernel function runs asynchronously on the GPU, so we need to use events to synchronize the host and device and measure the time accurately.

```
#include <stdio.h>

#define BLOCK_SIZE 16

__global__void matrix_multiply(float *a, float *b, float *c, int n)

{

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

   int col = blockIdx.x * blockDim.x + threadIdx.x;

   float sum = 0;
```

```
    if (row < n && col < n) {

        for (int i = 0; i < n; ++i) {

            sum += a[row * n + i] * b[i * n + col];

        }

        c[row * n + col] = sum;}
}

int main()

{

    int n = 1024;

    size_t size = n * n * sizeof(float);

    float *a, *b, *c;

    float *d_a, *d_b, *d_c;

    cudaEvent_t start, stop;

    float elapsed_time;

    //      Allocate host memory a = (float
    *)malloc(size);        b        =        (float
    *)malloc(size);        c        =        (float
    *)malloc(size);

    //Initialize matrices

    for (int i = 0; i < n * n; ++i) {

        a[i] = i % n;

        b[i] = i % n;

    }

    //        Allocate device memory
    cudaMalloc(&d_a, size);
```

```
cudaMalloc(&d_b, size);
cudaMalloc(&d_c, size);

//Copy input data to device

cudaMemcpy(d_a,      a,      size,      cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Set kernel launch configuration
 dim3 threads(BLOCK_SIZE, BLOCK_SIZE);

dim3 blocks((n + threads.x - 1) / threads.x, (n + threads.y - 1) / threads.y);

//       Launch kernel
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
matrix_multiply<<<blocks, threads>>>(d_a, d_b, d_c, n);
cudaEventRecord(stop); cudaEventSynchronize(stop);
cudaEventElapsedTime(&elapsed_time, start, stop);

  // Copy output data to host

  cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

  // Print elapsed time

  printf("Elapsed time: %f ms\n", elapsed_time);

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

  //       Free host memory free(a);

  free(b);

  free(c);

  return 0;

}
```