# ARQUITECTURA DE COMPUTADORES

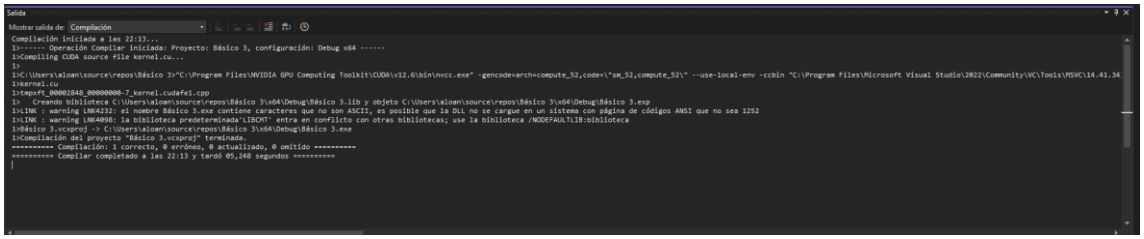
## 1.CÓDIGO

```
/*
 * ARQUITECTURA DE COMPUTADORES
 * 2º Grado en Ingenieria Informatica
 *
 * ENTREGA Básico 3:
 * >> Hilos y bloques.
 *
 * Alumno: Antonio Alonso Briones
 * Fecha: 06/10/2024
 *
 */

// includes
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>

// declaración de funciones
__host__ void generar_vector_random(int* vector, int N)
{
    for (int i = 0; i < N; i++)
    {
        vector[i] = rand() % 10; // Valores entre 0 y 9
    }
}

__global__ void invertir_y_sumar_vectores(int* vector1, int* vector2, int*
resultado, int N)
{
    // Cálculo del índice global del hilo
    int idxGlobal = threadIdx.x + blockDim.x * blockIdx.x;
    if (idxGlobal < N)
    {
        // Invertir el vector
        vector2[idxGlobal] = vector1[N - idxGlobal - 1];
        // Sumar los vectores
        resultado[idxGlobal] = vector1[idxGlobal] + vector2[idxGlobal];
    }
}

int main(int argc, char** argv)
{
    // Declaración de variables
    int N;
    printf("Introduce el tamaño de los vectores: ");
    scanf("%d", &N);

    // Obtener el número de dispositivos CUDA
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);
    if (deviceCount == 0)
    {
        printf("No se han encontrado dispositivos CUDA compatibles.\n");
        return 1;
```

```
    }

    // Obtener propiedades del dispositivo
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, 0);

    // Mostrar propiedades del dispositivo
    printf("> Propiedades del dispositivo seleccionado:\n");
    printf("Nombre: %s\n", deviceProp.name);
    printf("Capacidad de cómputo: %d.%d\n", deviceProp.major, deviceProp.mi-
nor);
    printf("Número de multiprocesadores: %d\n", deviceProp.multiProcessor-
Count);
    printf("Número máximo de hilos por bloque: %d\n", deviceProp.maxTh-
readsPerBlock);
    printf("Máximo número de hilos por dimensión de bloque (x, y, z): (%d,
%d, %d)\n",
        deviceProp.maxThreadsDim[0], deviceProp.maxThreadsDim[1], device-
Prop.maxThreadsDim[2]);
    printf("Máximo número de bloques por dimensión de la malla (x, y, z):
(%d, %d, %d)\n",
        deviceProp.maxGridSize[0], deviceProp.maxGridSize[1], device-
Prop.maxGridSize[2]);
    printf("Tamaño de la memoria global: %.2f MiB\n", deviceProp.totalGlo-
balMem / (1024.0 * 1024.0));

    // Definir el tamaño fijo de los bloques
    int blockSize = 10; // Bloques de 10 hilos

    // Verificar que el blockSize no excede el máximo permitido
    if (blockSize > deviceProp.maxThreadsPerBlock)
    {
        printf("El tamaño del bloque (%d) excede el máximo permitido
(%d).\n", blockSize, deviceProp.maxThreadsPerBlock);
        blockSize = deviceProp.maxThreadsPerBlock;
    }

    // Calcular el número de bloques necesarios
    int numBlocks = (N + blockSize - 1) / blockSize;

    // Verificar que numBlocks no excede el máximo permitido
    if (numBlocks > deviceProp.maxGridSize[0])
    {
        printf("El número de bloques requeridos (%d) excede el máximo permi-
tido (%d).\n", numBlocks, deviceProp.maxGridSize[0]);
        numBlocks = deviceProp.maxGridSize[0];
    }

    // Declaración de punteros para host y device
    int* hst_vector1, * hst_vector2, * hst_resultado;
    int* dev_vector1, * dev_vector2, * dev_resultado;

    // Reserva de memoria en el host
    hst_vector1 = (int*)malloc(N * sizeof(int));
    hst_vector2 = (int*)malloc(N * sizeof(int));
    hst_resultado = (int*)malloc(N * sizeof(int));

    // Reserva de memoria en el device
    cudaMalloc((void**)&dev_vector1, N * sizeof(int));
    cudaMalloc((void**)&dev_vector2, N * sizeof(int));
    cudaMalloc((void**)&dev_resultado, N * sizeof(int));

    // Inicialización del primer vector
```

```
    printf("> Generando el vector 1...\n");
    srand(0); // Inicializar la semilla con un valor fijo
    generar_vector_random(hst_vector1, N);

    // Copiar el vector 1 al device
    cudaMemcpy(dev_vector1, hst_vector1, N * sizeof(int), cudaMemcpyHostTo-
Device);

    // Lanzamiento del kernel
    printf("> Generando el vector 2 y realizando la suma...\n");
    invertir_y_sumar_vectores << <numBlocks, blockSize >> > (dev_vector1,
dev_vector2, dev_resultado, N);

    // Sincronizar para esperar a que el kernel termine
    cudaDeviceSynchronize();

    // Copiar datos desde el device al host
    cudaMemcpy(hst_vector2, dev_vector2, N * sizeof(int), cudaMemcpyDevice-
ToHost);
    cudaMemcpy(hst_resultado, dev_resultado, N * sizeof(int), cudaMemcpyDe-
viceToHost);

    // Impresión de resultados
    printf("> RESULTADOS:\n");
    printf("VECTOR 1:\n");
    for (int i = 0; i < N; i++)
    {
        printf("%2d ", hst_vector1[i]);
    }
    printf("\n");

    printf("VECTOR 2 (Inverso de VECTOR 1):\n");
    for (int i = 0; i < N; i++)
    {
        printf("%2d ", hst_vector2[i]);
    }
    printf("\n");

    printf("SUMA:\n");
    for (int i = 0; i < N; i++)
    {
        printf("%2d ", hst_resultado[i]);
    }
    printf("\n");

    // Liberar memoria
    free(hst_vector1);
    free(hst_vector2);
    free(hst_resultado);
    cudaFree(dev_vector1);
    cudaFree(dev_vector2);
    cudaFree(dev_resultado);

    // Salida
    printf("*************************************************\n");
    printf("<pulsa [INTRO] para finalizar>");
    getchar(); getchar();
    return 0;
}
```

## 2. RESULTADO DE LA COMPILACIÓN




## 3. SALIDA POR PANTALLA