ARQUITECTURA DE COMPUTADORES

1.CÓDIGO

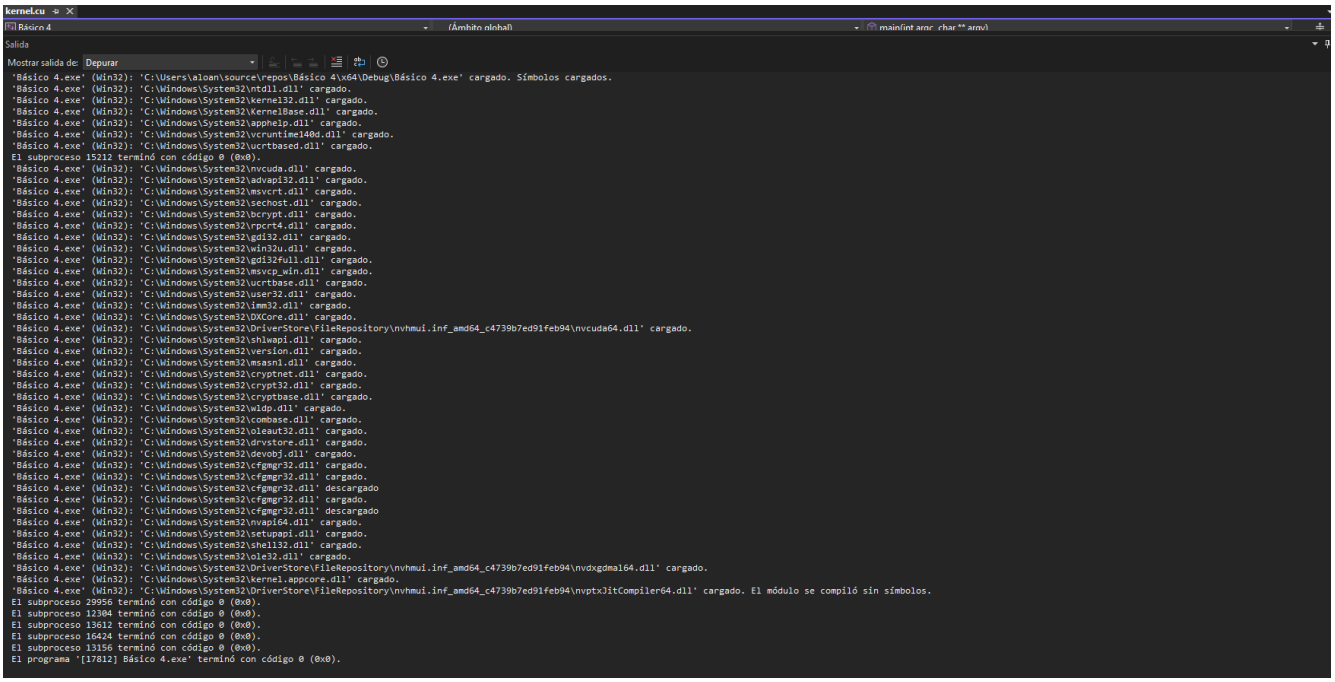
```
/*
 * ARQUITECTURA DE COMPUTADORES
 * 2° Grado en Ingenieria Informatica
 *
 * ENTREGA Básico 4:
 * >> Arrays multidimensionales
 *
 * Alumno: Antonio Alonso Briones
 * Fecha: 13/10/2024
 *
 */


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

// declaración de funciones
__host__ void generar_matriz_random(int* matriz, int filas, int columnas)
{
    for (int i = 0; i < filas * columnas; i++)
    {
        matriz[i] = rand() % 9 + 1; // Valores entre 1 y 9
    }
}

__global__ void procesar_matriz(int* matriz_inicial, int* matriz_final, int
filas, int columnas)
{
    // Cálculo de los índices de fila y columna
    int idxColumna = threadIdx.x;
    int idxFila = threadIdx.y;

    if (idxFila < filas && idxColumna < columnas)
    {
        int idxGlobal = idxColumna + idxFila * columnas;

        // Si la columna es par (idxColumna % 2 == 0), se mantiene el valor
        if (idxColumna % 2 == 0)
        {
            matriz_final[idxGlobal] = matriz_inicial[idxGlobal];
        }
        else
        {
            matriz_final[idxGlobal] = 0;
        }
    }
}

int main(int argc, char** argv)
{
    // Declaración de variables
    int filas, columnas;
    printf("Introduce el número de filas: ");
```

```
    scanf("%d", &filas);
    printf("Introduce el número de columnas: ");
    scanf("%d", &columnas);

    // 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 (usamos el dispositivo 0)
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, 0);

    // Mostrar propiedades del dispositivo
    printf("\n> 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 tamaño de bloque (x, y, z): (%d, %d, %d)\n",
        deviceProp.maxThreadsDim[0], deviceProp.maxThreadsDim[1], device-
Prop.maxThreadsDim[2]);
    printf("Máximo tamaño de la malla (x, y, z): (%d, %d, %d)\n",
        deviceProp.maxGridSize[0], deviceProp.maxGridSize[1], device-
Prop.maxGridSize[2]);

    // Definir el tamaño del bloque bidimensional
    dim3 tamBloque(columnas, filas);

    // Verificar que el tamaño del bloque no exceda los máximos permitidos
    if (tamBloque.x > deviceProp.maxThreadsDim[0] || tamBloque.y > device-
Prop.maxThreadsDim[1])
    {
        printf("El tamaño del bloque excede el máximo permitido.\n");
        // Ajustar el tamaño del bloque si es necesario
        if (tamBloque.x > deviceProp.maxThreadsDim[0])
            tamBloque.x = deviceProp.maxThreadsDim[0];
        if (tamBloque.y > deviceProp.maxThreadsDim[1])
            tamBloque.y = deviceProp.maxThreadsDim[1];
    }

    // Calcular el número de bloques necesarios (en este caso, un solo blo-
que)
    dim3 tamMalla(1, 1);

    // Mostrar el número de hilos lanzados en cada eje
    printf("\n> Configuración del kernel:\n");
    printf("Dimensiones del bloque (hilos por bloque): (%d, %d, %d)\n", tam-
Bloque.x, tamBloque.y, tamBloque.z);
    printf("Dimensiones de la malla (bloques): (%d, %d, %d)\n", tamMalla.x,
tamMalla.y, tamMalla.z);

    // Declaración de punteros para host y device
    int* hst_matriz_inicial, * hst_matriz_final;
    int* dev_matriz_inicial, * dev_matriz_final;
```

```
    int tamMatriz = filas * columnas * sizeof(int);

    // Reserva de memoria en el host
    hst_matriz_inicial = (int*)malloc(tamMatriz);
    hst_matriz_final = (int*)malloc(tamMatriz);

    // Inicialización de la matriz inicial
    printf("\n> Generando la matriz inicial...\n");
    srand(0); // Semilla fija para reproducibilidad
    generar_matriz_random(hst_matriz_inicial, filas, columnas);

    // Reserva de memoria en el device
    cudaMalloc((void**)&dev_matriz_inicial, tamMatriz);
    cudaMalloc((void**)&dev_matriz_final, tamMatriz);

    // Copiar la matriz inicial al device
    cudaMemcpy(dev_matriz_inicial, hst_matriz_inicial, tamMatriz, cu-
daMemcpyHostToDevice);

    // Lanzamiento del kernel
    printf("> Procesando la matriz en el device...\n");
    procesar_matriz << <tamMalla, tamBloque >> > (dev_matriz_inicial,
dev_matriz_final, filas, columnas);

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

    // Copiar datos desde el device al host
    cudaMemcpy(hst_matriz_final, dev_matriz_final, tamMatriz, cudaMemcpyDe-
viceToHost);

    // Impresión de resultados
    printf("\n> MATRIZ INICIAL:\n");
    for (int i = 0; i < filas; i++)
    {
        for (int j = 0; j < columnas; j++)
        {
            printf("%2d ", hst_matriz_inicial[j + i * columnas]);
        }
        printf("\n");
    }

    printf("\n> MATRIZ FINAL:\n");
    for (int i = 0; i < filas; i++)
    {
        for (int j = 0; j < columnas; j++)
        {
            printf("%2d ", hst_matriz_final[j + i * columnas]);
        }
        printf("\n");
    }

    // Liberar memoria
    free(hst_matriz_inicial);
    free(hst_matriz_final);
    cudaFree(dev_matriz_inicial);
    cudaFree(dev_matriz_final);

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

```
}
```

2. RESULTADO DE LA COMPILACIÓN



3. SALIDA POR PANTALLA