_

Entrega: Reporte Examen Final
Título: Detección de Bordes Utilizando CUDA
Materia: Programación Avanzada en Paralelo
Profesor: Germán Alonso Pinedo Díaz
Integrantes:
Celia Lucía Castañeda Arizaga
Eduardo Sebastián González Ramírez
Jessica Fernanda Isunza López

Introducción

El objetivo de este proyecto fue implementar un pipeline de procesamiento de video utilizando CUDA y OpenCV. El pipeline incluye la conversión a escala de grises, la aplicación de filtros gaussianos (de tamaños 5x5 y 9x9), la detección de bordes y la visualización de los bordes sobre el video original. Este procesamiento se realiza en paralelo utilizando la potencia de cómputo de las GPU, lo que permite un rendimiento superior al procesamiento en CPU. Para poder realizar esto de manera correcta, se implementaron ambas librerías dentro de un proyecto de Visual Studio el cual fue utilizado para obtener los bordes detectados por la cámara de vídeo del dispositivo.

Descripción del Código

El código está organizado en varias funciones que encapsulan los diferentes pasos del procesamiento de video. Cada una de estas funciones realiza una tarea específica y utiliza kernels de CUDA para realizar operaciones en paralelo.

1. Verificación de Errores en CUDA

Se define una macro cudaCheckError para verificar errores de CUDA y asegurar que cualquier fallo se capture y maneje adecuadamente.

```
// Verifica errores de CUDA
#define cudaCheckError(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char* file, int line, bool abort = true) {
    if (code != cudaSuccess) {
        fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}
```

2. Kernel de Escala de Grises

El propósito de este kernel es convertir una imagen en color a escala de grises. Cada pixel de la imagen es procesado para obtener su valor promedio, que se asigna a los tres canales (R, G, B) de la imagen de destino.

```cpp
// Kernel para la escala de grises
__global__ void escalaGrisesKernel(uchar3* img_src, uchar3* img_dst, int rows, int cols) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < cols && y < rows) {
        int idx = y * cols + x;
        uchar3 pixel = img_src[idx];
        unsigned char gray = (pixel.x + pixel.y + pixel.z) / 3;
        img_dst[idx] = make_uchar3(gray, gray, gray);
    }
}
```

- Índices de Hilos: Cada hilo se asocia con un píxel específico basado en su posición (x, y).
- Cálculo de Índice: idx = y * cols + x calcula el índice lineal en la matriz de la imagen.
- Promedio de Canales: El valor de gris se obtiene promediando los valores de los tres canales de color.
- Asignación de Resultado: El valor de gris calculado se asigna a los tres canales de la imagen de destino.

3. Kernel para el Filtro Gaussiano 5x5

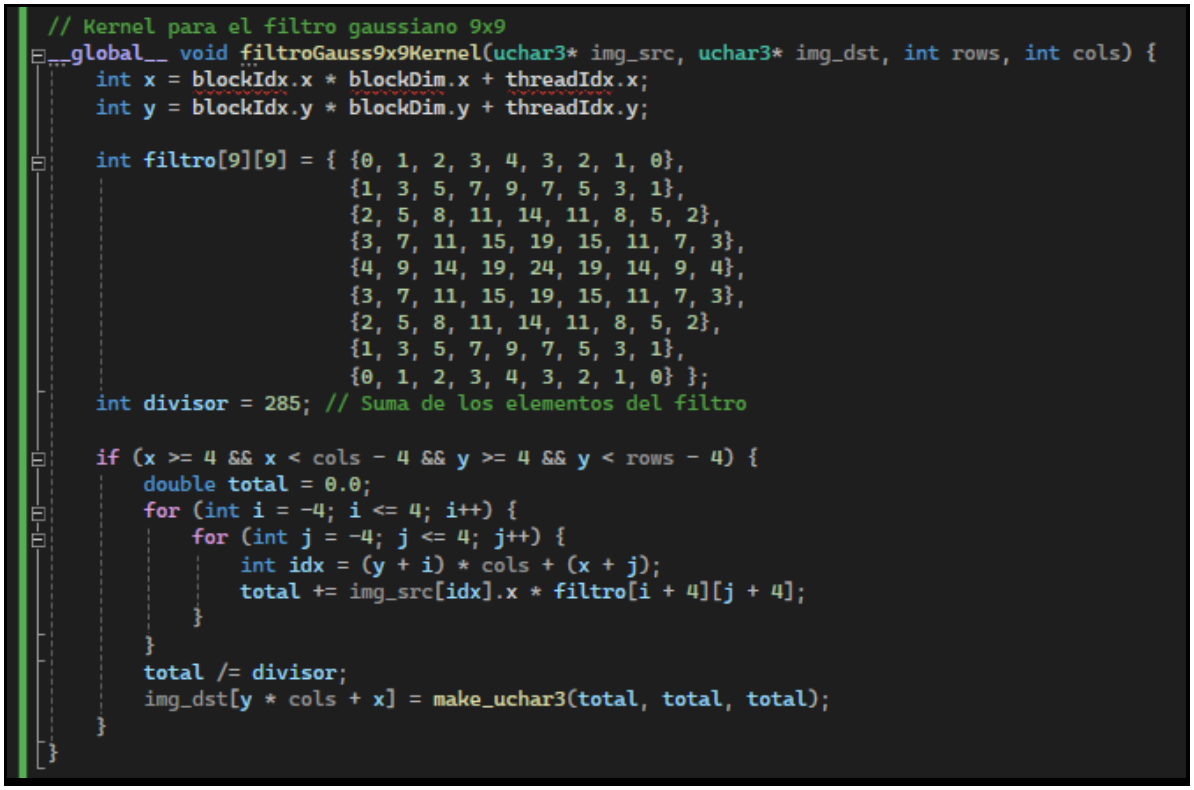
Este kernel aplica un filtro gaussiano 5x5 a la imagen en escala de grises para suavizarla.

```cpp
// Kernel para el filtro gaussiano 5x5
__global__ void filtroGauss5x5Kernel(uchar3* img_src, uchar3* img_dst, int rows, int cols) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int filtro[5][5] = { {0, 1, 2, 1, 0},
                         {1, 3, 5, 3, 1},
                         {2, 5, 9, 5, 2},
                         {1, 3, 5, 3, 1},
                         {0, 1, 2, 1, 0} };
    int divisor = 49; // Suma de los elementos del filtro

    if (x >= 2 && x < cols - 2 && y >= 2 && y < rows - 2) {
        double total = 0.0;
        for (int i = -2; i <= 2; i++) {
            for (int j = -2; j <= 2; j++) {
                int idx = (y + i) * cols + (x + j);
                total += img_src[idx].x * filtro[i + 2][j + 2];
            }
        }
        total /= divisor;
        img_dst[y * cols + x] = make_uchar3(total, total, total);
    }
}
```

- Índices de Hilos: Cada hilo se asocia con un píxel específico basado en su posición (x, y).
- Filtro Gaussiano: Se define una matriz 5x5 que representa el filtro gaussiano.
- Condición de Borde: Solo se procesan píxeles que tienen suficientes vecinos para aplicar el filtro completo.
- Aplicación del Filtro: Se calcula el valor ponderado del píxel considerando sus vecinos según el filtro gaussiano.
- Asignación de Resultado: El valor suavizado se asigna a los tres canales de la imagen de destino.

4. Kernel para el Filtro Gaussiano 9x9

Similar al kernel anterior, este kernel aplica un filtro gaussiano 9x9 para un mayor suavizado.



```
// Kernel para el filtro gaussiano 9x9
__global__ void filtroGauss9x9Kernel(uchar3* img_src, uchar3* img_dst, int rows, int cols) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int filtro[9][9] = { {0, 1, 2, 3, 4, 3, 2, 1, 0},
                         {1, 3, 5, 7, 9, 7, 5, 3, 1},
                         {2, 5, 8, 11, 14, 11, 8, 5, 2},
                         {3, 7, 11, 15, 19, 15, 11, 7, 3},
                         {4, 9, 14, 19, 24, 19, 14, 9, 4},
                         {3, 7, 11, 15, 19, 15, 11, 7, 3},
                         {2, 5, 8, 11, 14, 11, 8, 5, 2},
                         {1, 3, 5, 7, 9, 7, 5, 3, 1},
                         {0, 1, 2, 3, 4, 3, 2, 1, 0} };
    int divisor = 285; // Suma de los elementos del filtro

    if (x >= 4 && x < cols - 4 && y >= 4 && y < rows - 4) {
        double total = 0.0;
        for (int i = -4; i <= 4; i++) {
            for (int j = -4; j <= 4; j++) {
                int idx = (y + i) * cols + (x + j);
                total += img_src[idx].x * filtro[i + 4][j + 4];
            }
        }
        total /= divisor;
        img_dst[y * cols + x] = make_uchar3(total, total, total);
    }
}
```

- Índices de Hilos: Cada hilo se asocia con un píxel específico basado en su posición (x, y).
- Filtro Gaussiano: Se define una matriz 9x9 que representa el filtro gaussiano.
- Condición de Borde: Solo se procesan píxeles que tienen suficientes vecinos para aplicar el filtro completo.
- Aplicación del Filtro: Se calcula el valor ponderado del píxel considerando sus vecinos según el filtro gaussiano.
- Asignación de Resultado: El valor suavizado se asigna a los tres canales de la imagen de destino.

5. Kernel para la Detección de Bordes

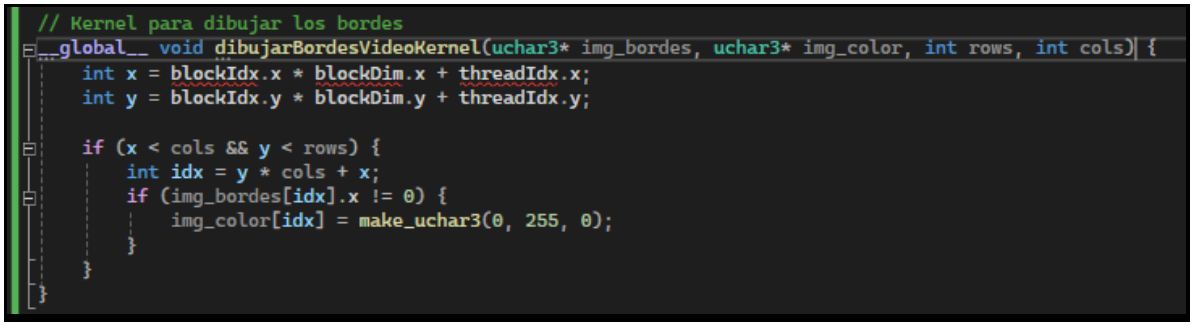
Este kernel aplica un filtro de Sobel para detectar bordes en la imagen suavizada.

```cpp
// Kernel para el análisis de bordes
__global__ void bordesVideoKernel(uchar3* img_src, uchar3* img_dst, int rows, int cols) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int Hx[3][3] = { {0, -1, 0},
                     {-1, 4, -1},
                     {0, -1, 0} };
    int Hy[3][3] = { {0, -1, 0},
                     {-1, 4, -1},
                     {0, -1, 0} };

    if (x > 0 && x < cols - 1 && y > 0 && y < rows - 1) {
        double grad_x = 0.0, grad_y = 0.0;
        for (int i = -1; i <= 1; i++) {
            for (int j = -1; j <= 1; j++) {
                int idx = (y + i) * cols + (x + j);
                grad_x += img_src[idx].x * Hx[i + 1][j + 1];
                grad_y += img_src[idx].x * Hy[i + 1][j + 1];
            }
        }

        double magnitude = sqrt(grad_x * grad_x + grad_y * grad_y);
        if (magnitude > 50) {
            img_dst[y * cols + x] = make_uchar3(255, 255, 255);
        }
        else {
            img_dst[y * cols + x] = make_uchar3(0, 0, 0);
        }
    }
}
```

- Índices de Hilos: Cada hilo se asocia con un píxel específico basado en su posición (x, y).
- Filtros Sobel: Se definen dos matrices 3x3 que representan los filtros Sobel en las direcciones x e y.
- Condición de Borde: Solo se procesan píxeles que tienen suficientes vecinos para aplicar el filtro completo.
- Cálculo del Gradiente: Se calculan los gradientes en las direcciones x e y.
- Magnitud del Gradiente: Se calcula la magnitud del gradiente usando la fórmula sqrt(grad_x * grad_x + grad_y * grad_y).
- Detección de Bordes: Si la magnitud del gradiente es mayor que un umbral (50), el pixel se marca como borde (blanco), de lo contrario, se marca como no borde (negro).

6. Kernel para Dibujar Bordes

Este kernel colorea los bordes detectados sobre la imagen original en color verde.



```
// Kernel para dibujar los bordes
__global__ void dibujarBordesVideoKernel(uchar3* img_bordes, uchar3* img_color, int rows, int cols) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < cols && y < rows) {
        int idx = y * cols + x;
        if (img_bordes[idx].x != 0) {
            img_color[idx] = make_uchar3(0, 255, 0);
        }
    }
}
```

- Índices de Hilos: Cada hilo se asocia con un píxel específico basado en su posición (x, y).
- Condición de Borde: Si el pixel en la imagen de bordes es negro (valor de borde), se dibuja en verde en la imagen original.
- Coloreado de Bordes: El píxel correspondiente en la imagen original se colorea de verde (make_uchar3(0, 255, 0)).

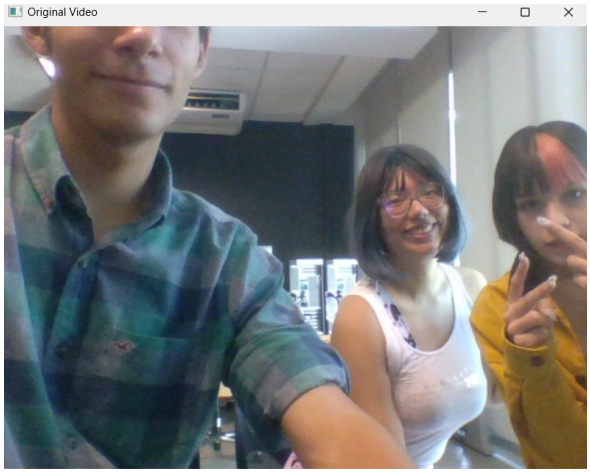
7. Funciones de Procesamiento en Host

Cada kernel tiene una función correspondiente en el host que realiza la configuración de la memoria y la ejecución del kernel. Estas funciones se encargan de la asignación de memoria en la GPU, la transferencia de datos entre el host y el dispositivo, la configuración de los bloques y la grilla de ejecución, y la liberación de memoria.

```
// Escala de Grises
escalaGrises(frame, &gray_frame);

// Filtro Gaussiano 5x5
filtroGauss5x5Video(gray_frame, &gauss5_frame);

// Filtro Gaussiano 9x9
filtroGauss9x9Video(gauss5_frame, &gauss9_frame);

// Análisis de Bordes
bordesVideo(gauss9_frame, &edges_frame);

// Dibujo de Bordes en el Color Original
dibujarBordesVideo(edges_frame, &final_frame);
```

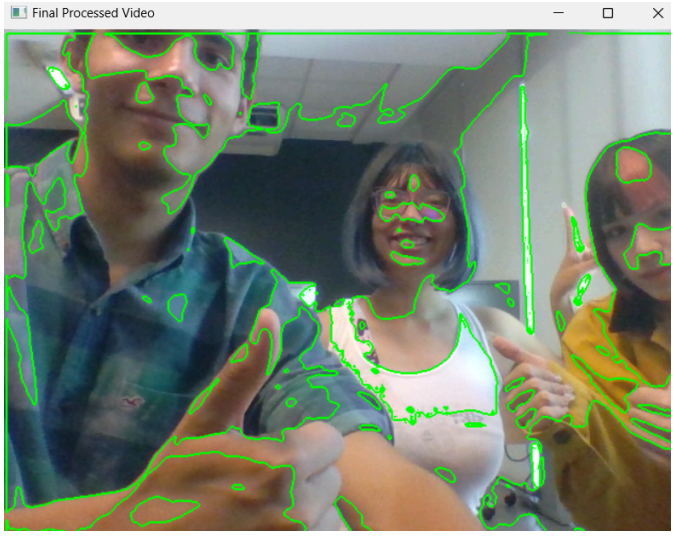
```
void escalaGrises(Mat img_src, Mat* img_dst) {
    int rows = img_src.rows;
    int cols = img_src.cols;
    uchar3* d_img_src;
    uchar3* d_img_dst;

    cudaCheckError(cudaMalloc(&d_img_src, rows * cols * sizeof(uchar3)));
    cudaCheckError(cudaMalloc(&d_img_dst, rows * cols * sizeof(uchar3)));

    cudaCheckError(cudaMemcpy(d_img_src, img_src.ptr(), rows * cols * sizeof(uchar3), cudaMemcpyHostToDevice));

    dim3 blockDim(16, 16);
    dim3 gridDim((cols + blockDim.x - 1) / blockDim.x, (rows + blockDim.y - 1) / blockDim.y);
    escalaGrisesKernel << <gridDim, blockDim >> > (d_img_src, d_img_dst, rows, cols);

    cudaCheckError(cudaMemcpy(img_dst->ptr(), d_img_dst, rows * cols * sizeof(uchar3), cudaMemcpyDeviceToHost));

    cudaCheckError(cudaFree(d_img_src));
    cudaCheckError(cudaFree(d_img_dst));
}
```

Resultados




Gaussian 9x9 Video


Edges Video


Final Processed Video

Conclusión

Cada kernel tiene una función específica en el pipeline de procesamiento de video, y juntos, permiten realizar operaciones complejas en paralelo aprovechando la potencia de cómputo de las GPU. Esta implementación en CUDA mejora significativamente el rendimiento comparado con una implementación secuencial en CPU. Gracias a esto, pudimos obtener una mejor implementación a la original utilizada en CPU, obteniendo una mayor fluidez e imagen más nítida de nuestros bordes. Algunos usos pueden ser detección de placas, filtros en redes sociales, detección de personas en cámaras de seguridad.