Programación y Análisis de Algoritmos | Tarea Procesamiento de Imágenes con CUDA y OpenCV

Aguirre Calzadilla César Miguel

10 de diciembre de 2024

Códigos

Todo el código escrito para esta tarea será anexado en un archivo .zip. Dentro se encuentran las rutinas escritas para la tarea así como comentarios sobre las mismas.

El objetivo de la tarea es implementar un código secuencial, y otro paralelo, para resolver los siguientes problemas. Se utilizó OpenCV, librería especializada en la manipulación de imágenes, para completar la tarea.

Problema 1 | Alpha Matting

Este problema nos pide utilizar una máscara binaria para así poder colocar la imagen del baloncito con sombrero, sobre la imagen de la pirámide.

Para lograr dicho propósito, lo que necesitamos es utilizar un proceso llamado Alpha Matting. Esta es una técnica bastante sencilla de entender, veamos cómo funciona. Lo primer es tener un par de imágenes A y B, así como una máscara binaria α.

La ecuación que describe la técnica de Alpha Matting es la siguiente:

$$C = A \cdot \alpha + B \cdot (1 - \alpha) \tag{1}$$

Esta ecuación combina las dos imágenes A y B basándose en la, denominada, máscara α. En resumidas cuentas, se genera una imagen C a partir de la guía de la máscara, pues al ser una imagen binaria (blanco y negro, 0 y 1) entonces va tomando los "pedazos"de cada imagen A o B para generar una nueva, llamada C.

Si aún no queda claro, podemos verlo matemáticamente. Utilizando las imágenes que nos dio el profesor.

Para esta tarea, se utiliza como imagen A una que muestra un balón amarillo con un sombrero, sobre un banco de madera. Como fondo B se utiliza una fotografía de una pirámide (quién sabe qué pirámide sea). La máscara α corresponde a la silueta del

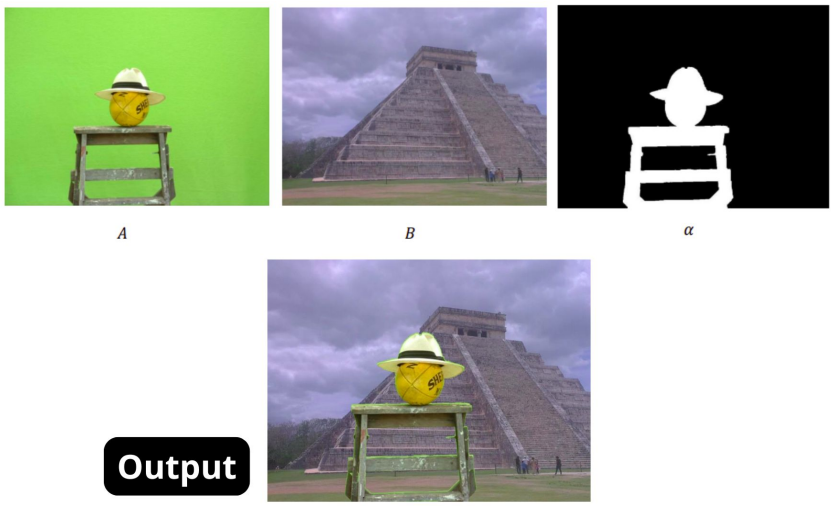

Figura 1: Imágenes a procesar, máscara y salida.

baloncito con sombrero. En otras palabras, es como un recorte"de la imagen A, para así poder poner sobre el fondo B a la imagen A.

Esto puede quedar más claro con un desarrollo matemático sencillo.

Pensemos en que tenemos una función binaria para α, donde $\alpha = 0$ y donde $\alpha = 1$. Entonces, tenemos dos casos.

Caso $\alpha = 1$ Para este caso, nuestra expresión para construir a C sería la siguiente:

$$C = A \cdot 1 + B \cdot (1 - 1) = A$$

Por lo tanto, todo píxel igual a 1 (digamos, blanco, recordemos que es el "molde de A") capturará los píxeles de la imagen A, para generar la imagen C. Ahora, veamos que pasa en el segundo caso.

Caso $\alpha = 0$ Para este apartado, estamos hablando de la región de ceros en la máscara. Es decir, la región en negro de α. Aplicando $\alpha = 0$ para la expresión de C, tendremos algo como lo siguiente:

$$C = A \cdot 0 + B \cdot (1 - 0) = B$$

Por lo tanto, lo que llamamos el fondo, será la región de la nueva imagen C que tomará los píxeles de B, en el área que no tiene píxeles blancos (iguales a 1).

Hablando en términos generales, si estuviésemos trabajando con una α no binaria, tendríamos valores entre 0 y 1. Por lo tanto, la nueva imagen C construida sería una mezcla ponderada entre A y B, acorde con el peso dado por α.

En otras palabras, la máscara α trabaja como una especie de filtro o control de transparencia. Si es blanca, la imagen A será completamente visibles, si es negra, se

reemplazará con la imagen B. De ese modo, el Alpha Matting permite superponer una imagen sobre la otra, de una manera controlada.

0.1. Análisis del código

Aproximación Paralela. Este modelo permite dividir el problema en múltiples hilos con cálclos independientes. Como hemos visto en las tareas pasadas, cada hilo opera sobre un píxel único de las imágenes. La aproximación paralela se sostiene sobre el procesamiento masivo (y, valga la redundancia, paralelo) de los cálculos realizados de manera simultánea para cada píxel de la imagen.

Para ello, se utilizan los núcleos de la GPU de mi computadora. Los hilos en un bloque procesan píxeles de manera paralela. Los bloques, a su vez, dividen la carga entre regiones de la imagen. Para lograr la tarea, necesitamos, primero que nada, cargar las imágenes.

```
cv::Mat imgA = cv::imread("/path/to/Figura-A.png", cv::IMREAD_COLOR);
cv::Mat imgB = cv::imread("/path/to/Figura-B.png", cv::IMREAD_COLOR);
cv::Mat alpha = cv::imread("/path/to/Mascara.png", cv::IMREAD_GRAYSCALE);
```

Listing 1: Carga de las imágenes usando OpenCV

En esta sección de código, las imágenes A, B y la máscara α son cargadas desde el disco. La máscara está definida en escala de grises, i.e. un solo canal, contrario al RGB de las imágenes A y B.

Un paso muy importante antes de todo es validar las dimensiones de nuestras imágenes, para poder trabajar con ellas. Si no tenemos las mismas dimensiones, el código se rompe.

```
if (imgA.size() != imgB.size() || imgA.size() != alpha.size()) {
std::cerr << "Error: Las im genes deben tener el mismo tama o."
<< std::endl;
return -1;
}
```

Ahora, sigue la paralelización en el CUDA Kernel.

```
__global__ void blendImagesKernel(unsigned char* d_A, unsigned char
* d_B, unsigned char* d_alpha, unsigned char* d_C, int width, int
height, int channels) {
int x = blockIdx.x * blockDim.x + threadIdx.x; // Coordenada X
int y = blockIdx.y * blockDim.y + threadIdx.y; // Coordenada Y
int idx = (y * width + x) * channels; //  ndice  en el array plano

if (x < width && y < height) {
    for (int c = 0; c < channels; c++) {
        d_C[idx + c] = d_alpha[y * width + x] / 255.0 * d_A[idx + c
] + (1.0 - d_alpha[y * width + x] / 255.0) * d_B[idx + c];
    }
}
}
```

Cada hilo en CUDA calcula la combinación para un único píxel. Las coordenadas de nuestros hilos (x, y), se determinan a través de: **blockIdx.x** y **blockIdx.y**. Por otra parte, los índices del hilo dentro de un bloque se definen como **threadIdx.x** y **threadIdx.y**.

La parte donde se aplica el Alpha Mating dentro del fragmento anterior es la siguiente:

```
d_C[idx + c] = d_alpha[y * width + x] / 255.0 * d_A[idx + c]
             + (1.0 - d_alpha[y * width + x] / 255.0) * d_B[idx + c];
```

En cuanto a la configuración de la cuadrícula, tenemos lo siguiente:

```
dim3 blockSize(16, 16); // Tama o de bloque
dim3 gridSize((width + blockSize.x - 1) / blockSize.x, (height +
blockSize.y - 1) / blockSize.y);
```

Los bloques están construidos con un tamaño de 16 x 16 hilos. Además, la cuadrícula se calcula para cubrir toda la imagen, ajustándose al tamaño de las mismas.

Para la transferencia de información, tenemos la ayuda de **cudaMalloc** y **cudaMemcpy**. Recordemos que Malloc reserva la memoria en la GPU para los datos (en este caso, imágenes y máscara), mientras que Memcpy transfiere los datos del CPU a la GPU antes de ejecutar el kernel (y viceversa).

```
cudaMalloc(&d_A, imgSize);
cudaMalloc(&d_B, imgSize);
cudaMalloc(&d_alpha, alphaSize);
cudaMalloc(&d_C, imgSize);

cudaMemcpy(d_A, imgA.data, imgSize, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, imgB.data, imgSize, cudaMemcpyHostToDevice);
cudaMemcpy(d_alpha, alpha.data, alphaSize, cudaMemcpyHostToDevice);
```

Finalmente, se libera la memoria con **cudaFree**, aplicado a todas la variables que se utilizaron en el fragmento de código anterior.

Aproximación Secuencial. En este caso, la gran diferencia entre la aproximación paralela y la secuencial está en cómo el código secuencial define ciclos anidados para recorrer cada píxel y cada canal de las imágenes de manera secuencial, valga la redundancia.

```
void blendImagesSequential(const cv::Mat& imgA, const cv::Mat& imgB
, const cv::Mat& alpha, cv::Mat& imgC) {
int width = imgA.cols;
int height = imgA.rows;
int channels = imgA.channels();

for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        float alphaValue = alpha.at<unsigned char>(y, x) / 255.0f;
// Normalizar alpha entre 0 y 1
        for (int c = 0; c < channels; ++c) {
            imgC.at<cv::Vec3b>(y, x)[c] = static_cast<unsigned char
>(
                alphaValue * imgA.at<cv::Vec3b>(y, x)[c] +
                (1.0f - alphaValue) * imgB.at<cv::Vec3b>(y, x)[c]
```

```
13                     );
14                 }
15             }
16     }
17 }
```

La diferencia principal con el enfoque paralelo es la manera en cómo se distribuye el trabajo. Mientras que en el secuencial se usan bucles, en el paralelo se aprovecha la arquitectura de las GPU para realizar cálculos simultáneos en sus núcleos.

Resultado Al final, se logró en ambos casos conseguir una imagen muy similar.

Figura 2: La imagen A se sobre pone en la B, creando una nueva imagen C.

0.2. Tiempos de procesamiento

Figura 3: Tiempos de procesamiento | Paralelo.

Figura 4: Tiempos de procesamiento | Secuencial.

Podemos notar cómo el tiempo total en el paralelo fue de 134.035 ms. Mientras que en el secuencial fue de 47.1623 ms. Lo anterior debido a que en el paralelo se deben transferir datos del CPU a la GPU, y viseversa. Caso contrario al enfoque en CPU.

Sin embargo, el tiempo de procesamiento de las imágenes en el secuencial es de 15.7272 ms, y el tiempo de ejecución del kernel en el paralelo fue de 3.9650 ms. Esto significa que, con un gran conjunto de imágenes, o con una imagen gigante, el tiempo de procesamiento secuencial crecería mucho más que en el paralelo.

De esa manera, aunque el paralelo deba transferir datos de CPU a GPU, es muy probable que el enfoque paralelo nos ahorre bastante tiempo total. Por la naturaleza anidada del enfoque secuencial, el procesamiento podría crecer de manera exponencial.

Problema 2 | Detección de Bordes

En este problema nos piden detectar bordes de una imagen. Se trata de una técnica común en el procesamiento de imágenes e incluso dentro del Machine Learning, y Visión Computacional.

Lo que se nos indica es que apliquemos los filtros, utilizando matrices, K_1 y K_2. El proceso se le conoce como "direccionales Sobel", I_x y I_y, con los que se calculan las derivadas parciales de la intensidad de la imagen en las direcciones (x, y).

Para el caso I_x, se calcula aplicando el filtro K_1, que detecta cambios en la intensidad a lo largo del eje horizontal, X. Este filtro es una matriz definida en el enunciado de la tarea como:

$$K_1 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Para cada píxel, se aplica una convolución que realiza un cálculo ponderado de los valores del píxel y su vecindad.

Por otra parte, el indicador I_y se calcula aplicando el filtro K_2, que trabaja con los cambios de intensidad a lo largo del eje vertical, Y.

$$K_2 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

De manera similar al K_1, este filtro recorre los valeres del píxel y su vecindad a lo largo de la vertical, aplicando la convolución.

Matemáticamente hablando, para cada píxel ubicado en (i, j), se nuestra imagen Ix, la convolución con un filtro K_2 se define de la siguiente manera:

$$I_x(i, j) = \sum_{u=-1}^{1} \sum_{v=-1}^{1} K_1(u, v) \cdot I(i + u, j + v)$$

De manera semejante, para I_y, con K_2, se recorre la vecindad 3×3 del píxel (i, j). Y, una vez que tenemos calculadas las intensidades I_x e I_y, se obtiene la magnitud del gradiente (llamemosle MG). Este mide el cambio total en la intensidad alrededor de cada píxel y se logra hacer con la siguiente expresión:

$$MG(i, j) = \sqrt{\Big(I_x(i, j)\Big)^2 + \Big(I_y(i, j)\Big)^2}$$

En esta expresión, podemos visualizar a $I_x(i, j)$ e $I_y(i, j)$ como las derivadas parciales de la intensidad en las direcciones x y y. Asimismo, el gradiente (I_x, I_y). Como podemos notar, es una ecuación equivalente al de la distancia euclidiana.

Finalmente, se aplica un umbral T, para intentar "binarizar"la imagen y resaltar únicamente los bordes significativos, llamado MGT. Este se define de la siguente manera:

$$MGT(i,j) = \begin{cases} 255 & si\ MG(i,j) > T \\ 0 & cualquier\ otro\ caso \end{cases}$$

En este caso, si MGT es mayor que el umbral, entonces e considera que hay un borde en el píxel (i, j), y se transforma en blanco. Si no se considera que exista borde, se establece como negro, en 0.

0.3. Análisis del código

Aproximación Paralela. En este caso, la aproximación paralela comienza con la definición del filtro Sobel con covolución paralela.

```
__global__ void convolutionKernel(const unsigned char* input, float
* output, const float* kernel, int width, int height, int
kernelSize) {
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;

int halfKernel = kernelSize / 2;
float sum = 0.0;

if (x >= halfKernel && x < (width - halfKernel) && y >= halfKernel
&& y < (height - halfKernel)) {
    for (int ky = -halfKernel; ky <= halfKernel; ky++) {
        for (int kx = -halfKernel; kx <= halfKernel; kx++) {
            int pixel = (y + ky) * width + (x + kx);
            float weight = kernel[(ky + halfKernel) * kernelSize +
(kx + halfKernel)];
            sum += input[pixel] * weight;
        }
    }
    output[y * width + x] = sum;
}
}
```

Listing 2: Kernel convolutivo

Este fragmento del código aplica convoluciones con los filtros Sobel K_1 y K_2, para calcular las parciales I_x e I_y. Además, cada hilo de CUDA procesa un píxel (x,y) independiente. Se introducen un par de ciclos anidados para recorrer la vecindad del píxel y aplicar el kernel. Se intenta paralelizar distribuyendo los píxeles entre los hilos de bloques definidos como **blockIdx** y los hilos **blockIdx**.

En el siguiente fragmento de código, se calcula la magnitud del gradiente con un kernel también.

```
__global__ void gradientMagnitudeKernel(const float* gradX, const
float* gradY, float* magnitude, int width, int height) {
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;

if (x < width && y < height) {
    int idx = y * width + x;
    float scale = 0.5; // Factor de escala
```

```
 8       magnitude[idx] = scale * sqrtf(gradX[idx] * gradX[idx] + gradY[
     idx] * gradY[idx]);
 9   }
}10
```

Listing 3: Kernel de magnitud

Cada hilo procesa un píxel y combina las derivadas correspondientes. Este cálculos e realiza de manera independiente entre píxeles, permitiendo distribuciones eficientes entre nuestros threads.

Posteriormente entra el umbral.

```
 1   __global__ void thresholdKernel(const float* magnitude, unsigned
     char* output, int width, int height, float threshold) {
 2   int x = blockIdx.x * blockDim.x + threadIdx.x;
 3   int y = blockIdx.y * blockDim.y + threadIdx.y;
 4
 5   if (x < width && y < height) {
 6       int idx = y * width + x;
 7       output[idx] = (magnitude[idx] > threshold) ? 255 : 0;
 8   }
}9
```

Listing 4: Kernel del umbral

Se vuelve binario al gradiente, usando al umbral T. Cada hilo evalúa cada píxel para confirmar si su magnitud es mayor que T.

Además, quise colocar un mapa de colores, para no dejarlo solo en blanco y negro. El propósito detrás de esto es añadir un toque más visual. Este mapa se realiza dentro de la CPU, utilizando las funciones de OpenCV.

```
 1   cv::normalize(mgImg, mgImgNormalized, 0, 255, cv::NORM_MINMAX,
     CV_8UC1);
cv2::applyColorMap(mgImgNormalized, colorMap, cv::COLORMAP_JET);
```

Listing 5: Kernel del umbral

Aproximación en Serie. Para este caso, simplemente se definieron las siguientes funciones y se corrió el código.

```
// Funcion para aplicar la convoluci n en la CPU
void convolutionCPU(const cv::Mat& input, cv::Mat& output, const float*
    kernel, int kernelSize) {
    int halfKernel = kernelSize / 2;
    for (int y = halfKernel; y < input.rows - halfKernel; ++y) {
        for (int x = halfKernel; x < input.cols - halfKernel; ++x) {
            float sum = 0.0;
            for (int ky = -halfKernel; ky <= halfKernel; ++ky) {
                for (int kx = -halfKernel; kx <= halfKernel; ++kx) {
                    int pixel = input.at<unsigned char>(y + ky, x + kx)
    ;
                    float weight = kernel[(ky + halfKernel) *
    kernelSize + (kx + halfKernel)];
                    sum += pixel * weight;
                }
            }
            output.at<float>(y, x) = sum;
        }
    }
}


// Funci n para calcular la magnitud del gradiente en la CPU
void gradientMagnitudeCPU(const cv::Mat& gradX, const cv::Mat& gradY,
    cv::Mat& magnitude, float scale) {
    for (int y = 0; y < gradX.rows; ++y) {
        for (int x = 0; x < gradX.cols; ++x) {
            float gx = gradX.at<float>(y, x);
            float gy = gradY.at<float>(y, x);
            magnitude.at<float>(y, x) = scale * sqrtf(gx * gx + gy * gy
    );
        }
    }
}


// Funci n para aplicar un umbral en la CPU
void thresholdCPU(const cv::Mat& magnitude, cv::Mat& output, float
    threshold) {
    for (int y = 0; y < magnitude.rows; ++y) {
        for (int x = 0; x < magnitude.cols; ++x) {
            output.at<unsigned char>(y, x) = (magnitude.at<float>(y, x)
     > threshold) ? 255 : 0;
        }
    }
}
```

Listing 6: Funciones en serie

Resultados Para los resultados, quise utilizar una imagen diferente a las que puso el profesor. Ya que terminó Arcane Season 2.

En ambos, casos, paralelo y secuencial, el output fue muy similar.

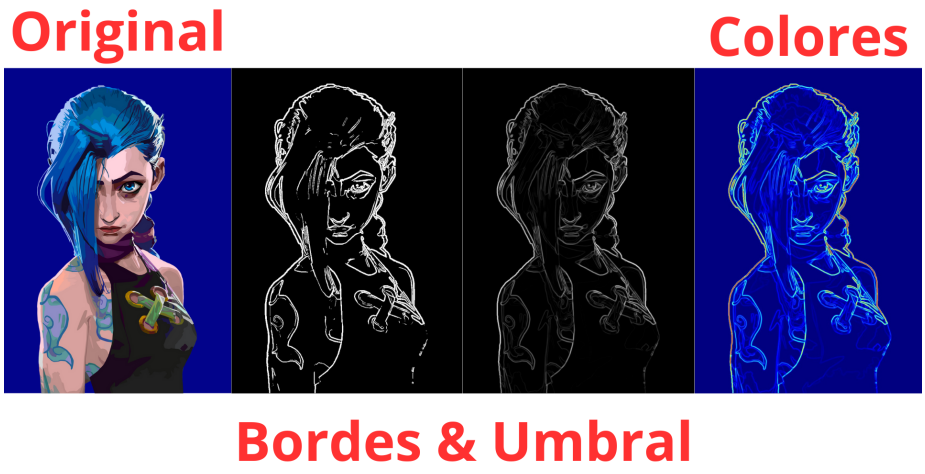

Figura 5: Big fat hero!

0.4. Tiempo de procesamiento

Figura 6: Tiempo de procesamiento en CPU | 85.36 ms

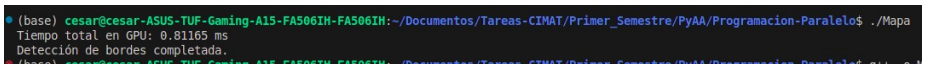

Figura 7: Tiempo de procesamiento en GPU | 0.8116 ms

En este caso, tenemos un speedup de $\frac{T_{CPU}}{T_{GPU}} = \frac{85.36}{0.8116} \approx 105.14$. En teoría, se tiene una mejora (mayor rapidez) de 105 veces la del código secuencial. Algo bastante útil para grandes conjuntos de datos.

Conclusiones

El procesamiento de imágenes puede beneficiarse enormemente de la paralelización de los códigos que dan forma a distintas aplicaciones. Después de todo, las imágenes pueden ser muy grandes, y están formadas por matrices con distintos canales.

Además, si consideramos que los videos son conjuntos de imágenes, entonces es aún más útil tener códigos paralelizados. De esa manera, podemos incluso realizar filtros y detecciones tan rápidos que parezcan en aparecer en tiempo real.