Centro de Investigación en Matemáticas (CIMAT).
Unidad Monterrey
Programación y Análisis de Algoritmos

Tarea 3 de Cómputo Paralelo

Gustavo Hernández Angeles

10 de diciembre de 2024

1 Combinación de dos imagenes usando una máscara

Teniendo las imagenes A, B y la máscara α, la tarea es combinar las imagenes A y B con la siguiente expresión, considerando que α es una imagen binaria:

$$C = A\alpha + (1 - \alpha)B \tag{1.1}$$

(a) Imagen original A (b) Fondo B (c) Máscara α

Figura 1.1: Imágenes utilizadas en la combinación con máscara.

SOLUCIÓN

Código secuencial

De forma secuencial, OpenCV ofrece funciones muy útiles para hacer el trabajo mucho más sencillo. En este caso utilizamos las funciones `multiply` y `add` para realizar la multiplicación diádica de imagenes, y la suma de imagenes como matrices. El resultado de la operación se guarda en la variable que se encuentr en el tercer argumento de la función. Para $1-\alpha$ utilizamos la función `all` que crea una matriz del mismo tamaño que α llena de 1's, para después hacer la resta. Cabe destacar que las variables de tipo `Mat` sobreescriben los operadores básicos como la suma y la resta para hacer estas operaciones más sencillas. Para finalizar se suman tanto el resultante de $A \cdot \alpha$ y $B \cdot (1 - \alpha)$.

```
void secuencial(Mat A, Mat B, Mat alpha, Mat &salida)
{
    multiply(alpha, A, A);                          // A*alpha
    multiply(Scalar::all(1.0) - alpha, B, B); // B*(1-alpha)
    add(A, B, salida);
}
```

Código paralelo

Hacemos algo similar a lo que se hacían con las imagenes, asignamos el tamaño de los bloques y de la malla de acuerdo a el tamaño de la imagen tratándola como una matriz. Damos como entrada las imagenes A, B y α como vectores de flotantes, fácilmente accedibles mediante el atributo `.ptr` de los objetos `Mat` y casteandolos como flotantes, también damos como entrada las dimensiones de la imagen junto al número de canales que tiene.

Una vez que nos aseguramos que no nos salimos de la imagen con el condicional `if`, podemos realizar la operación definida por el problema, realizando un ciclo sobre los canales para que las operaciones sean independientes entre los canales.

```
__global__ void paralelo(const float *A, const float *B,
                         const float *alpha, float *salida,
                         int ancho, int alto, int canales)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < ancho && y < alto)
    {
        int idx = (y * ancho + x) * canales;

        for (int c = 0; c < canales; ++c)
        {
            salida[idx + c] = alpha[idx + c] * A[idx + c] +
                              (1.0f - alpha[idx + c]) * B[idx + c];
        }
    }
}
```

Secuencial vs Paralelo

Para ambas funciones se mide el tiempo de ejecución de forma similar a las anteriores tareas; la función secuencial se hace mediante la librería `chrono` y la función en paralelo se hace mediante los `cudaEvents`. El número de repeticiones de cada función se determina mediante la variable `n_iteraciones = 100`, para asegurar una medición más precisa del tiempo de ejecución.

(a) Resultado en secuencial.

(b) Resultado en paralelo.

Por alguna razón que no entiendo, aparecen unos extraños bordes negros en la imagen de salida tanto para la implementación en secuencial como en paralelo, agradecería el feedback. Los tiempos de ejecución y el speedup también lo calcula el programa. Podemos observar un speedup de hasta 24x!!!

```
tmpxft_00004548_00000000-10_problema1.cudafe1.cpp
   Creando biblioteca problema1.lib y objeto problema1.exp
PS C:\Users\Gus\Documents\CIMAT\Primer Semestre\Programacion\Tareas\Tarea3_CUDA\Codigo> .\problema1.exe
Tiempo en secuencial (segundos):        0.00416247
Tiempo en paralelo (segundos):          0.000168673     Speedup: 24.6777
El programa se ejecuto exitosamente...
```

2 Detección de bordes en la imagen

Dada la imagen I realice los siguiente pasos para obtener la detección de bordes.

1. Aplicar los siguientes filtros:
   $I_x = I \otimes K_1$, con $K_1 = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$
   $I_y = I \otimes K_2$, con $K_2 = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$
2. Luego, calcular para cada pixel (i, j):
   $\text{MG}(i, j) = \sqrt{(I_x(i, j))^2 + (I_y(i, j))^2}$
3. Finalmente, para cada pixel (i, j), aplicar un umbral T sobre la imagen MG:

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

2.1 Código secuencial

Aquí puede ser un poco más complicado, en estas implementaciones no pude realizar el programa para que calcule en los 3 canales, por lo que únicamente tomé el ejemplo donde es un canal, la imagen que está en escala de grises: pinzas_gray.png. Al igual que en el problema pasado, utilicé funciones de OpenCV para realizar ciertas partes de la tarea, declaré las dos matrices de correlación K_1 y K_2 como objetos `Mat` para posteriormente utilizar la función `filter2D` para realizar la correlación, guardando el resultado en la variable I_x. Después realizamos un ciclo sobre cada pixel de la imagen A para calcular el gradiente mediante las imagenes I_x e I_y, y la magnitud de este gradiente se almacena en el mismo pixel sobre el que iteramos en una variable MG, la imagen de salida se crea justo después aplicando un umbral a la imagen de la magnitud de gradientes.

```
void p2_secuencial(Mat A, Mat &salida, int umbral)
{

    Mat Ix;
    Mat Iy;
    Mat MG = Mat::zeros(A.size(), CV_32F);

    // Para aplicar los filtros rapidamente con la funcion
    // filter2D, necesitamos matrices del tipo Mat_
    Mat K1 = (Mat_<int>(3, 3) << -1, 0, 1,
              -2, 0, 2,
              -1, 0, 1);

    Mat K2 = (Mat_<int>(3, 3) << -1, -2, -1,
              0, 0, 0,
              1, 2, 1);

    // Calculamos Ix e Iy
    filter2D(A, Ix, CV_32F, K1);
```

```
        filter2D(A, Iy, CV_32F, K2);

        // Calculamos MG y al aplicar el umbral, guardamos en salida.
        for (int i = 0; i < A.rows; ++i)
        {
            for (int j = 0; j < A.cols; ++j)
            {
                float ix_ij = Ix.at<float>(i, j);
                float iy_ij = Iy.at<float>(i, j);
                MG.at<float>(i, j) = sqrt(ix_ij * ix_ij + iy_ij * iy_ij);
                // Aplicamos umbral a MG y lo guardamos en salida
                salida.at<uchar>(i, j) = (MG.at<float>(i, j) > umbral) ? 255 : 0;
            }
        }
    }
```

2.2 Código paralelo

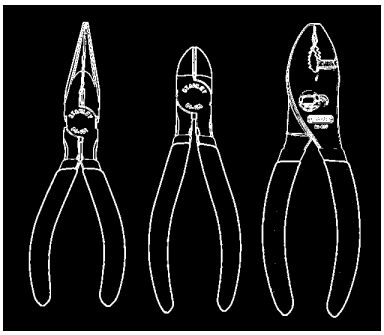
Aquí de igual forma asignaremos a cada hilo un pixel de la imagen mediante los id's tanto del bloque como del hilo correspondiente. En las orillas de la imagen asignamos automáticamente el valor de 0, asumiendo que no hay nada en los bordes de la imagen. Para la parte interior de la imagen el orden de los pasos se mantienen iguales, sin embargo, aquí calcularemos los gradientes I_x e I_y para el mismo pixel sobre el que nos colocamos con el hilo, realizando la correlación en el mismo pixel a través de dos ciclos `for`, obteniendo el indice del pixel vecino en cada iteración y guardando el valor de la suma en cada variable de gradiente correspondiente. Los kernels para esta ocasión se acceden por memoria global al declararlos al inicio del programa como `__const__ int K1;` y lo mismo para K_2. Una vez que obtenemos los gradientes, podemos encontrar la magnitud en todos los pixeles utilizando la función `sqrtf()`, la cual realiza la operación de raíz cuadrada de forma vectorizada. Finalmente, la salida se calcula aplicando únicamente el umbral a este vector de magnitudes.

```
    __global__ void p2_paralelo(const float *d_A, float *salida,
                                int ancho, int alto, int umbral)
    {
        int x = blockIdx.x * blockDim.x + threadIdx.x;
        int y = blockIdx.y * blockDim.y + threadIdx.y;

        if (x > 0 && x < ancho - 1 && y > 0 && y < alto - 1)
        {
            int idx = y * ancho + x;

            // Ix e Iy
            float ix = 0.0f;
            float iy = 0.0f;

            for (int i = -1; i <= 1; ++i)
            {
                for (int j = -1; j <= 1; ++j)
                {
                    int idx_vecino = (y + i) * ancho + (x + j);
                    float valor_pixel = d_A[idx_vecino];

                    ix += valor_pixel * k_1[i + 1][j + 1];
                    iy += valor_pixel * k_2[i + 1][j + 1];
                }
```

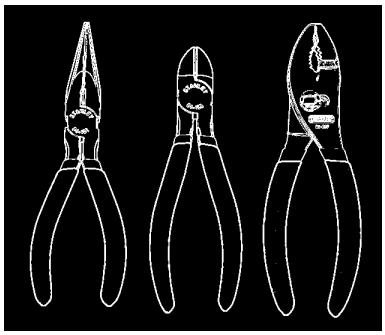
```
            }

            // Magnitud
            float magnitud = sqrtf(ix * ix + iy * iy);

            // Aplicamos el umbral.
            salida[idx] = (magnitud > umbral) ? 255.0f : 0.0f;
        }
        else
        {
            int idx = y * ancho + x;
            salida[idx] = 0.0;
        }
    }
```

2.3 Secuencial vs Paralelo

Se hace la medición de tiempos de igual forma que en el ejemplo pasado, esta vez haciendo `n_iteraciones = 50`.



(a) Resultado en secuencial.



(b) Resultado en paralelo.

Los resultados se muestran con un umbral = 100. Este valor se puede modificar en cada ejecución del código, pero pienso que este es un buen valor para ver los bordes. Los resultados muestran que el código paralelo tiene un speedup de hasta 14x respecto al secuencial.

```
tmpxft_00000d08_00000000-10_problema2.cudafe1.cpp
   Creando biblioteca problema2.lib y objeto problema2.exp
PS C:\Users\Gus\Documents\CIMAT\Primer Semestre\Programacion\Tareas\Tarea3_CUDA\Codigo> .\problema2.exe
Umbral: 100
Tiempo en secuencial (segundos):        0.00014882
Tiempo en paralelo (segundos):          1.1616e-05      Speedup: 12.8116
```