UNIVERSIDAD DE ALICANTE

CUDA WORKSHOP 2019

PRÁCTICA 2: El conjunto de Julia

Sergio Orts Escolano (sorts@ua.es)

Alberto García García (agarcia@dtic.ua.es)

José García Rodríguez (jgarcia@dtic.ua.es)

PRÁCTICA 2: EL CONJUNTO DE JULIA

El siguiente ejemplo muestra como computar en CUDA el conjunto de Julia. Para aquellos que desconozcan que es el conjunto de Julia, así llamado por el matemático Gaston Julia, son una familia de conjuntos fractales que se obtienen al estudiar el comportamiento de los números complejos al ser iterados por una función holomorfa. Los cálculos que se utilizan para generar este fractal son bastante simples. Simplemente hay una función iterativa que para cada uno de los puntos del plano complejo evalúa si pertenece o no al conjunto de Julia. Por lo tanto, si en la secuencia de valores producidos en la iteración, la ecuación crece hasta el infinito el punto está considerado fuera del conjunto. De forma contraria si los valores generados por la ecuación permanecen acotados, el punto pertenece al conjunto.

$$Z_{n+1} = Z_n^2 + C$$

figura 1. Ecuación conjunto

En este laboratorio aprenderemos:

- Como utilizar los índices de hilos y bloques para escribir kernels en CUDA para procesar imágenes.
- Como invocar a un kernel en CUDA para el procesamiento de una imagen utilizando una jerarquía de bloques e hilos bidimensional.

El código fuente se encuentra en la carpeta labs/lab1-cuJulia. Inicialmente este código no compila, tendrás que completarlo para ello.

CONJUNTO DE JULIA

Para este ejemplo no es necesario saber cómo funciona el conjunto de Julia, pero para aquellos interesados pueden continuar investigando para obtener más información. Lo que realmente nos interesa a nosotros de este ejemplo, es ver como con CUDA podemos paralelizar este cálculo de forma que cada hilo se encargue de computar un punto del plano complejo de forma independiente. O lo que es lo mismo, cada hilo de procesamiento decida si el pixel a dibujar pertenece o no al conjunto de Julia.

```
void kernel( unsigned char *ptr ){
    for (int y=0; y<DIM; y++) {
        for (int x=0; x<DIM; x++) {
            int offset = x + y * DIM;

            int juliaValue = julia( x, y );
            ptr[offset*4 + 0] = 255 * juliaValue;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
}
```

FIGURA 2 CÁLCULO DEL CONJUNTO EN CPU

Como podemos ver en la *figura 4*, el cálculo del conjunto en la CPU supone iterar sobre todos los píxeles de la imagen que vamos a generar. Para ello se utiliza la función *Julia(x,y)* que nos dice si pertenece o no al conjunto el punto seleccionado. Devolverá 1 en caso de que pertenezca o 0 si no pertenece al conjunto. Para este ejemplo no es necesario comprender el funcionamiento de la función Julia, nos vamos a centrar en como separar este bucle de dos niveles, iteración por columnas y filas, en acceso utiliza una matriz de hilos de computación.

Paso 1: Edita la función *__global__ void kernel(unsigned char *ptr)* en el fichero *"julia_gpu.cu"* para completar la funcionalidad de cálculo del conjunto. Utiliza los comentarios del código para completar las funciones que se piden.

Código 1:

1. Define los índices del elemento a ser computado por cada hilo utilizando la variable especial **blockIdx.x.** En esta primera aproximación utiliza tantos bloques como píxeles para llevar a cabo el cálculo del conjunto. Donde cada bloque solo tenga 1 hilo de computación, por lo que el direccionamiento hacia el píxel a procesar es bastante sencillo.

Paso 2: Edita la función *int main(void)* en el fichero "julia_gpu.cu" para completar la funcionalidad de la invocación del kernel en la GPU.

Código 2:

2. Establece el tamaño del grid y de los bloques, así como utiliza estos parámetros para la invocación del Kernel de cálculo del conjunto.

Una vez completados estos dos pasos, compila tu código y ejecuta para ver los resultados, al ejecutar comprobarás que los tiempos de ejecución son muy parecidos, apenas se obtiene una mejora en tiempo de procesamiento. ¿A qué se debe?

Paso 3: Vamos a modificar el cálculo de índices para dividir el cálculo no solo en bloques, sino en hilos y bloques a su vez. Observa la figura 6, donde se aprecia como una imagen se divide en bloques que a su vez se dividen en hilos de computación.

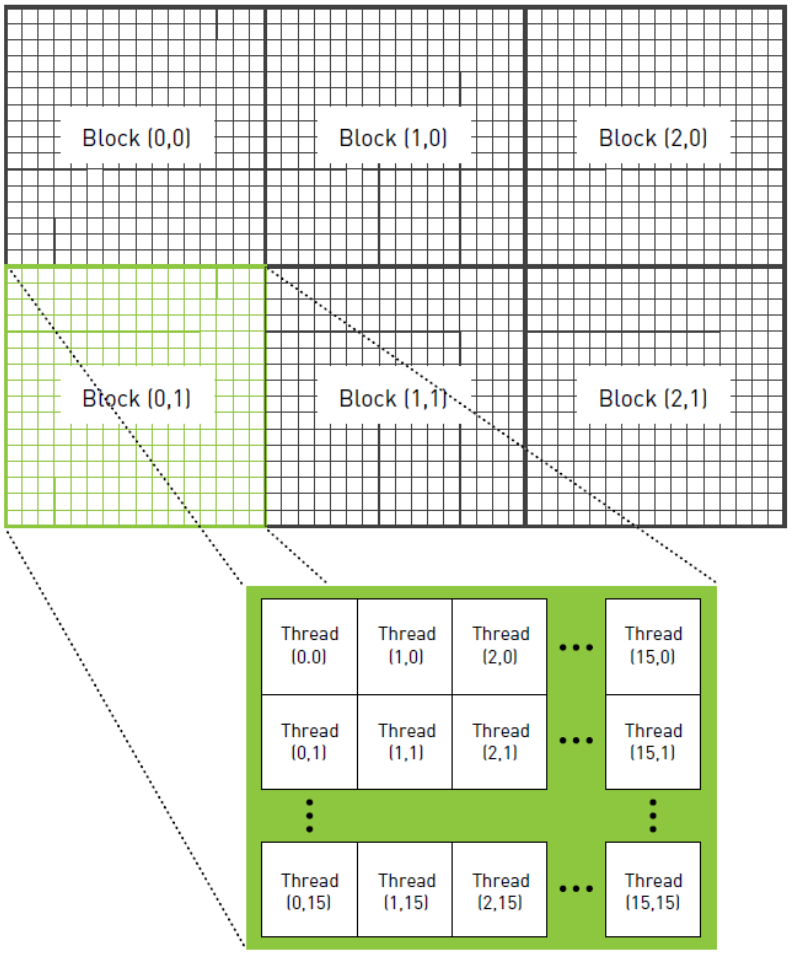

figura 3. Jerarquía 2d de bloques e hilos que puede utilizarse para computar cada uno de los pixeles de una imagen utilizando distintos hilos y bloques.

3. Define los índices del elemento a ser computado por cada hilo utilizando las variables especiales **blockIdx y threadIdx.** En esta segunda aproximación utiliza bloques e hilos para dividir el procesamiento de la imagen. Fíjate en la Figura 5 para el cálculo de las variables x, y, offset.

Una vez completado este último paso y tras ejecutar el código deberías obtener tiempos muy diferentes en la CPU y la GPU, siendo los tiempos de la GPU considerablemente menores que en la CPU. ¿Utilizando la fórmula para el cálculo de la aceleración, sabrías decirnos que aceleración se obtiene con la implementación en GPU respecto a la versión CPU?

$$S_p = \frac{T_1}{T_p}$$

FIGURA 4. Fórmula para el cálculo de la aceleración.

En la figura 7, T_1 hace referencia al tiempo de ejecución del código en secuencial (CPU) y T_p es el tiempo de ejecución de la aplicación sobre el hardware paralelo. Como resultado se obtiene el grado de aceleración frente a ambas implementaciones.

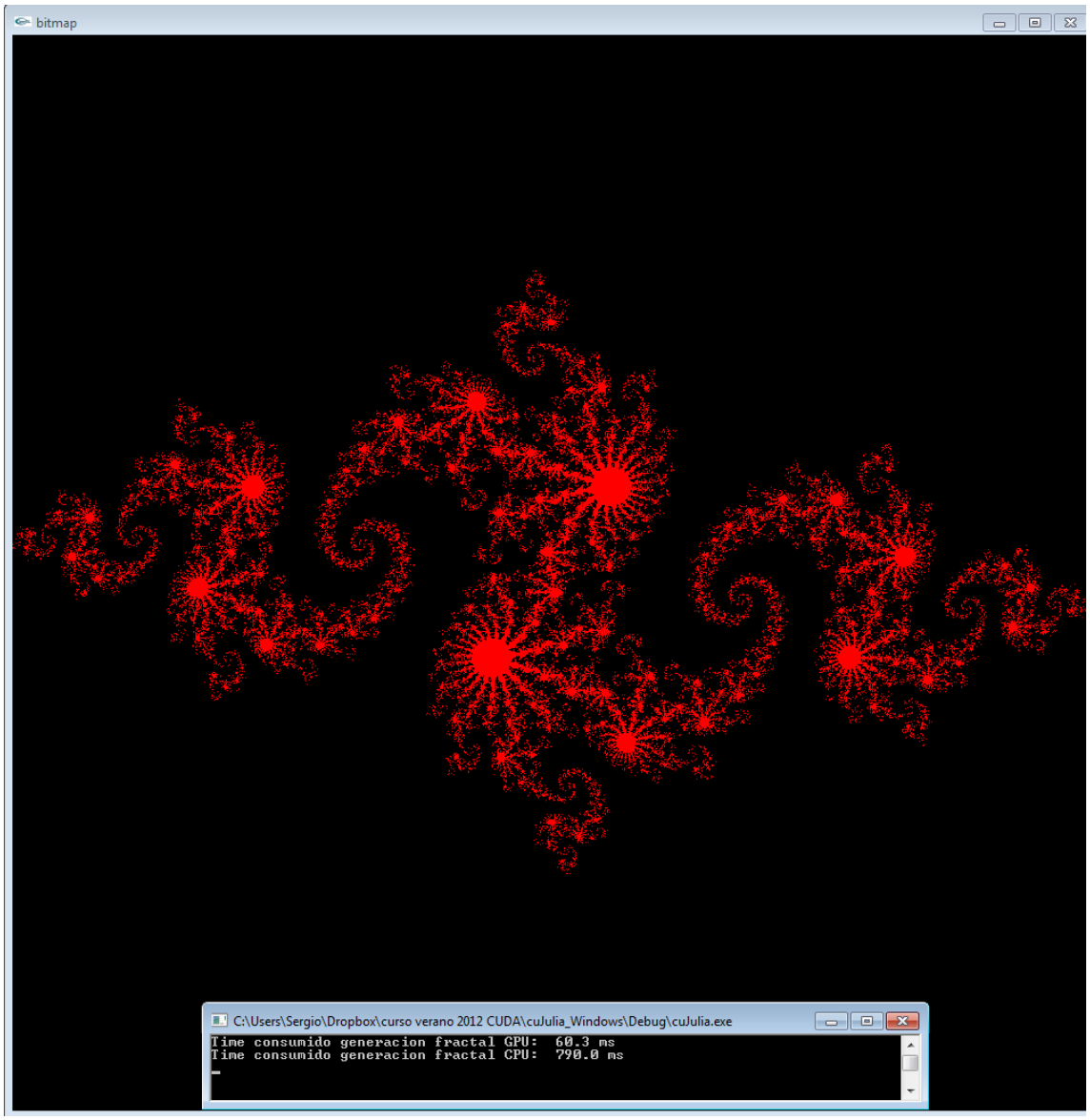

figura 5. Visualización del conjunto de julia windows