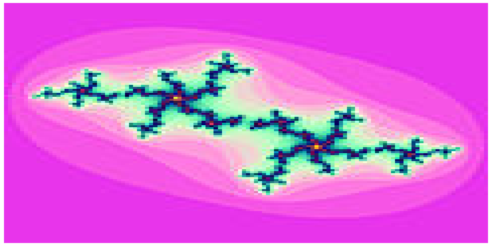
EXPOSICION FINAL HPC

FRACTALES DE JULIA


FRACTAL

Un fractal es un objeto geométrico cuya estructura básica, fragmentada o irregular, se repite a diferentes escalas. El término fue propuesto por el matemático Benoit Mandelbrot en 1975.

Su nombre deriva del latín fractus, que significa quebrado o fracturado. Muchas estructuras naturales son de tipo fractal. La propiedad matemática clave de un objeto genuinamente fractal es que su dimensión métrica fractal es un número no entero, que significa quebrado o fracturado.

En el caso del fractal de Julia es un numero complejo (f_c, $c=(\varphi-2)+(\varphi-1)i = -0.382+0.618i$) con una parte real y una imaginaria**. Muchas estructuras naturales son de tipo fractal y podrían ser representadas matemáticamente por una función llamada función fractal** $f c\,(z) = z^2 + C$

INTRODUCCION

A un objeto geométrico fractal se le atribuyen las siguientes características:

- Es demasiado irregular para ser descrito en términos geométricos tradicionales.
- Es AUTOSIMILAR, su forma es hecha de copias más pequeñas de la misma figura.

Las copias son similares al todo: misma forma pero diferente tamaño.

EJEMPLOS DE AUTOSIMILARIDAD

- **Fractales naturales** son objetos naturales que se pueden representar con muy buena aproximación mediante fractales matemáticos con autosimilaridad estadística. Las nubes, las montañas, el sistema circulatorio, las líneas costeras y los copos de nieve.
- **Conjunto de Mandelbrot** es un fractal auto similar, generado por el conjunto de puntos estables de órbita acotada bajo cierta transformación iterativa no lineal.
- **Paisajes fractales** son un tipo de fractales generados computacionalmente que pueden producir paisajes realistas convincentes.

CONJUNTOS DE JULIA O FRACTALES DE JULIA

CONCEPTO MATEMATICO FRACTALES DE JULIA

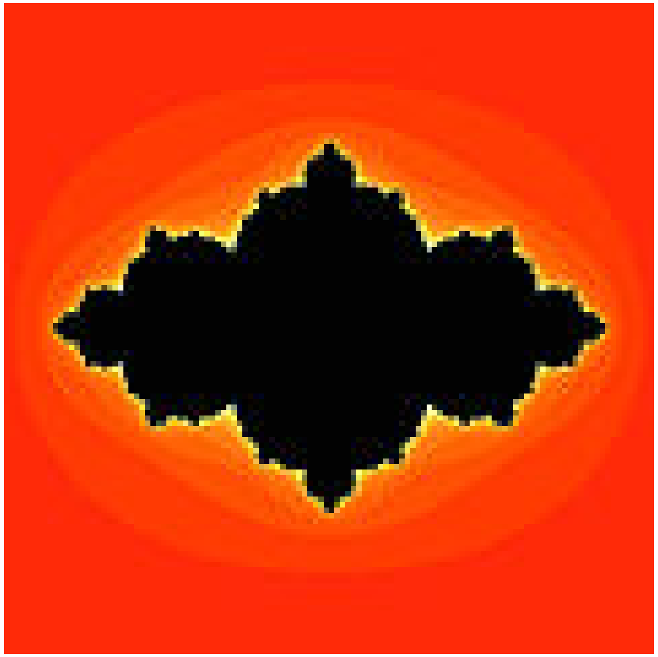
- Estos conjuntos, fruto de los trabajos de Pierre Fatou y Gaston Julia en los años 1920, surgen como resultado de la aplicación reiterada de funciones holomorfas. (Z → ***f*** (Z) → ***f*** (*f* (Z) →……).

- Las **funciones holomorfas** son el principal objeto de estudio del análisis complejo; son funciones que se definen sobre un subconjunto abierto del plano complejo **C** y con valores en **C**, que además son complejo-diferenciables en cada punto.

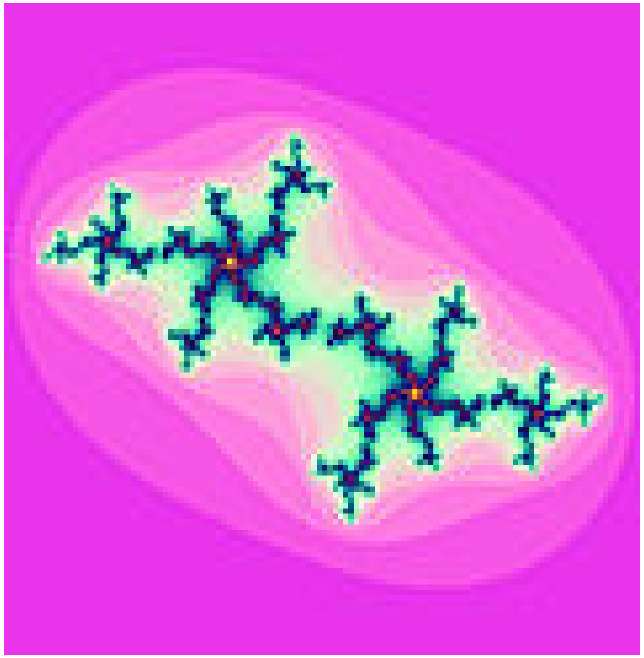
- Analicemos el caso particular de funciones polinómicas de grado mayor que uno. Al aplicar sucesivas veces una función polinómica es muy posible que el resultado tienda a **infinito** ∞ . Al conjunto de valores de que no escapan al infinito mediante esta operación se le denomina conjunto de Julia relleno, y a su frontera, simplemente conjunto de Julia. $f_c(z) = z^2 + C$

Ejemplos de conjuntos de Julia para $f_c(z) = z^2 + C$

En negro, conjunto de Julia relleno asociado a f_c, $c=\varphi-1$, donde φ es el número áureo.



Conjunto de Julia relleno asociado a f_c, $c=(\varphi-2)+(\varphi-1)i = -0.382+0.618i$



Generación de la imagen

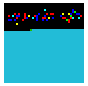
```
Mat image (DIM, DIM, CV_8UC1, Scalar(255));
```

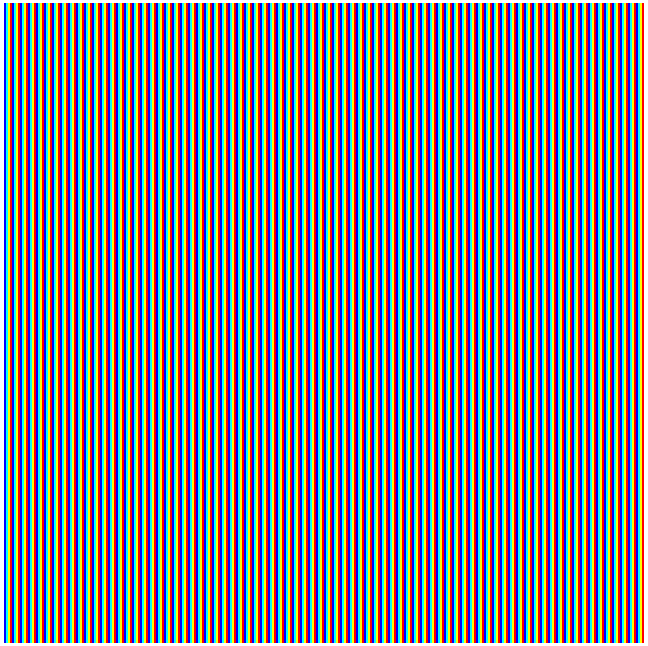
Crea una imagen con las dimensiones DIMxDIM (definidas como variables globales).
CV_8UC1 => escala de blanco o negro
Scalar(255) => Indica que el fondo sea blanco

Primeras pruebas

320x320

40x40





IMPLEMENTACION SECUENCIAL

```
int julia(int x, int y)
{
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    for (int i = 0; i < 200; i++)
    {
        a = a * a + c;
        if (a.magnitude2() > 1000)
        return 0;
    }
    return 1;
}

void juliaCPU(unsigned char *ptr)
{
    for (int i = 0; i < DIM; ++i)
    {
        for (int j = 0; j < DIM; ++j)
        {
            int offset = j + i*DIM;
            int juliaValue = julia (j, i);
            //cout <<offset << ": " <<juliaValue <<endl;
            ptr[offset] = 255 * juliaValue;
        }
    }
}
```

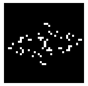
juliaCPU es la función encargada de agregar el color negro o blanco. Para poder generar el fractal se debe llamar a la función **julia**

a = a * a + c

$$fc(z) = z^2 + C$$

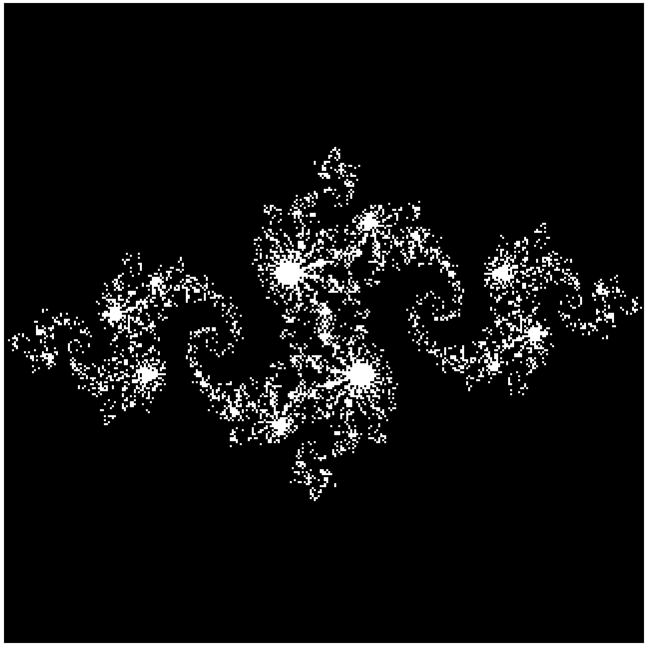
Imágenes

40x40

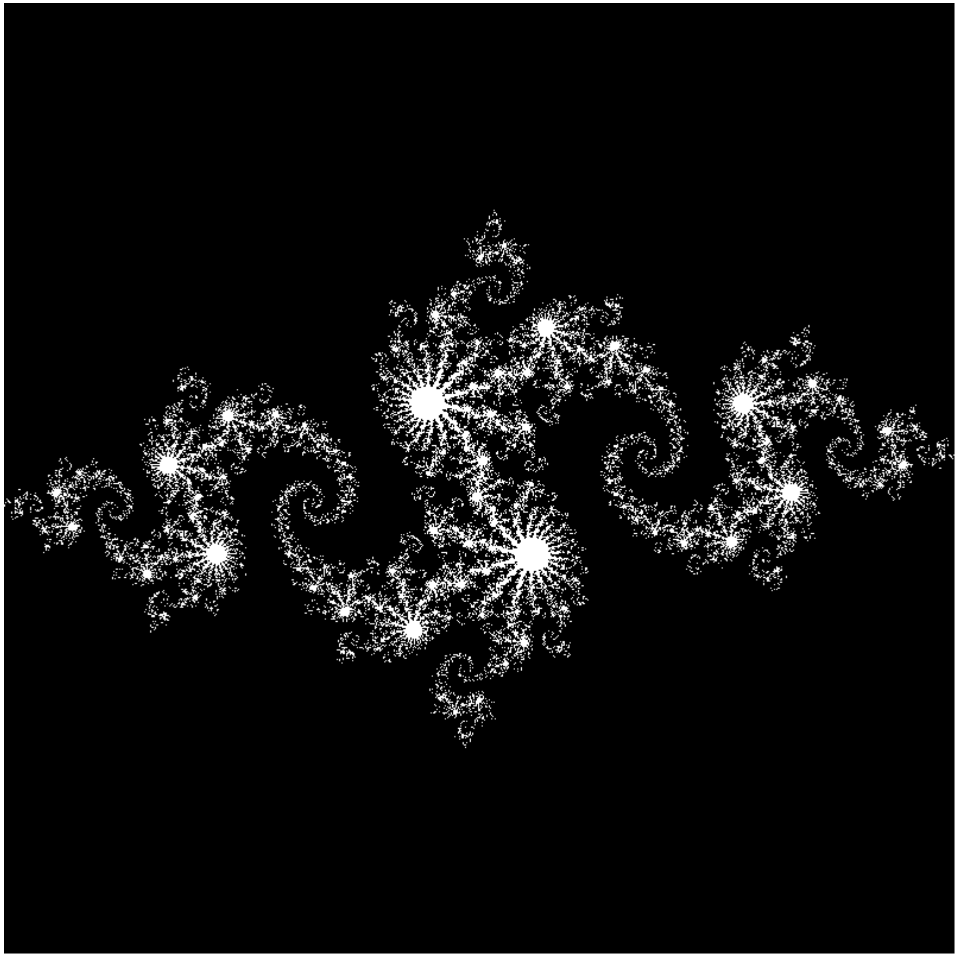


120x120

320x320

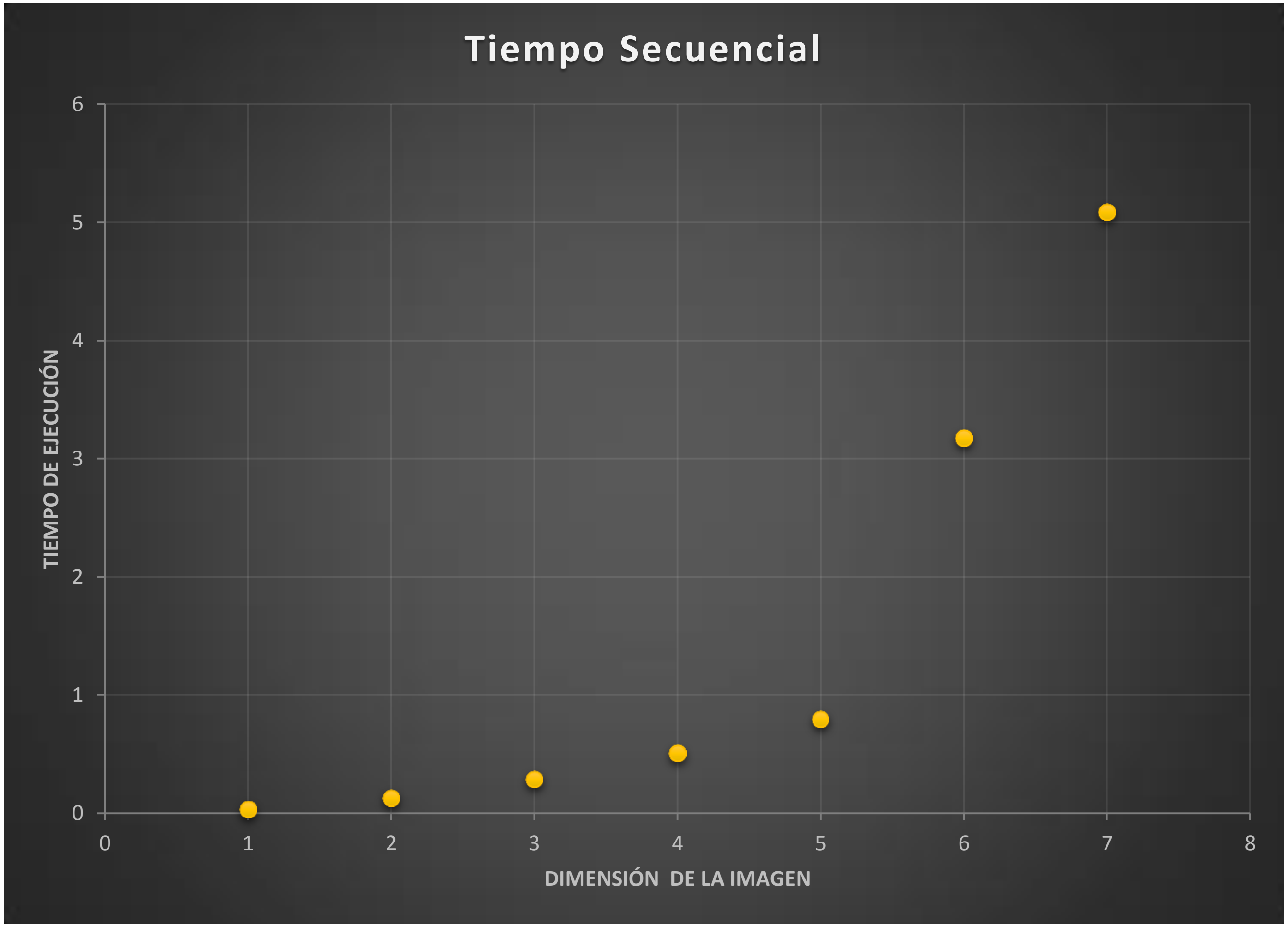


800x800



Graficas

ID	Dimensión (px)
1	200x200
2	400x400
3	600x600
4	800x800
5	1000x1000
6	2000x2000
7	2500x2500



IMPLEMENTACION PARALELA

```
__device__ int juliaGPU(int x, int y)
{
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    for (int i = 0; i < 200; i++)
    {
        a = a * a + c;
        if (a.magnitude2() > 1000)
        return 0;
    }
    return 1;
}


/**************************************************/
/************** SECCION PARALELA ******************/
/**************************************************/
__global__ void KernelGPUJulia(unsigned char *imgIn, int width, int height)
{
    unsigned int row = blockIdx.y*blockDim.y+threadIdx.y;
    unsigned int col = blockIdx.x*blockDim.x+threadIdx.x;

    int offset = col + row * DIM;

    if ((row < DIM) && (col < DIM))
    {
        int juliaValue = juliaGPU(col, row);
        imgIn[offset] = 255 * juliaValue;
    }
}
```

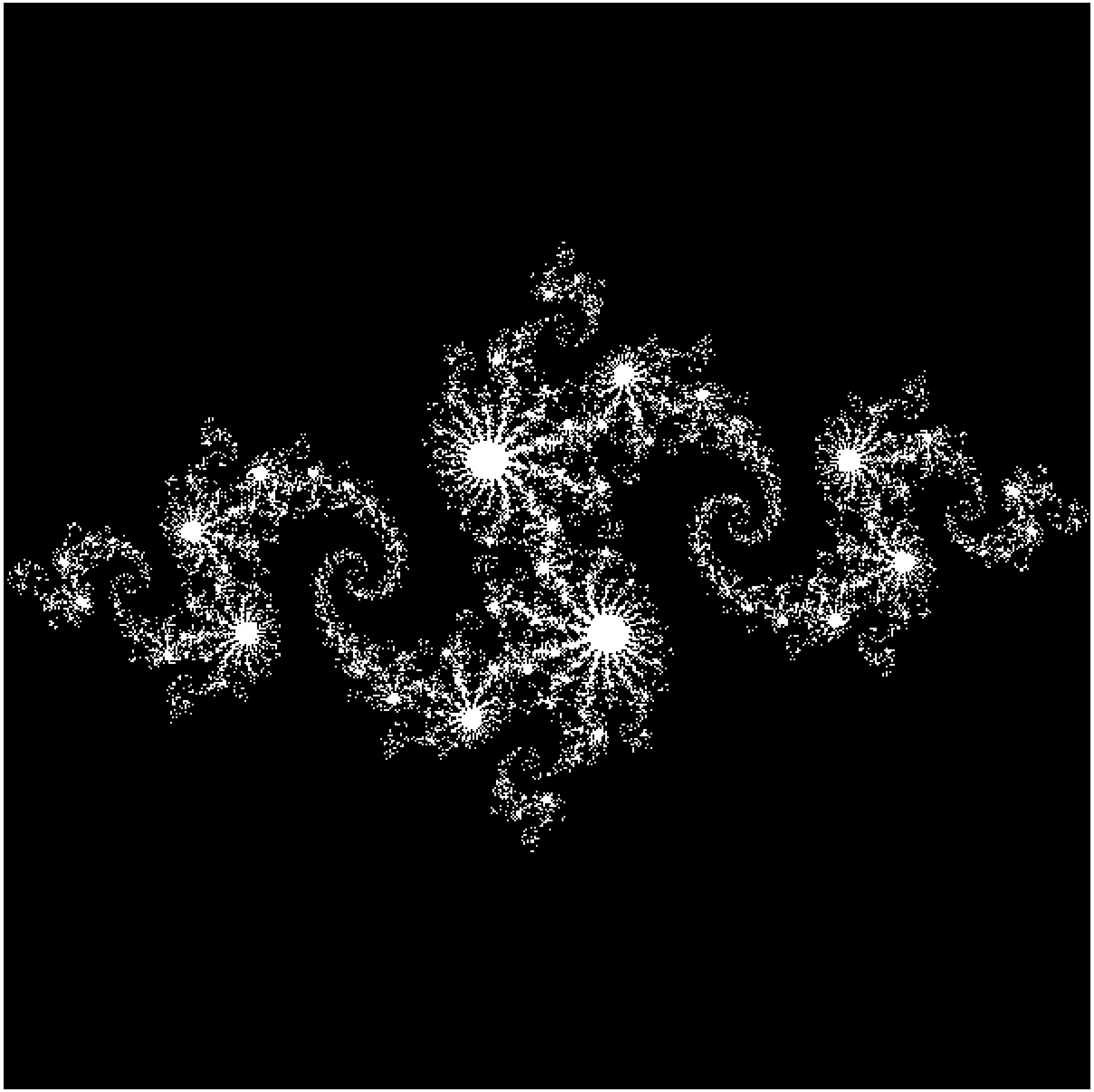
juliaGPU básicamente es la misma de la CPU, la única diferencia es que se ejecuta en el dispositivo.

__device__

a = a * a + c

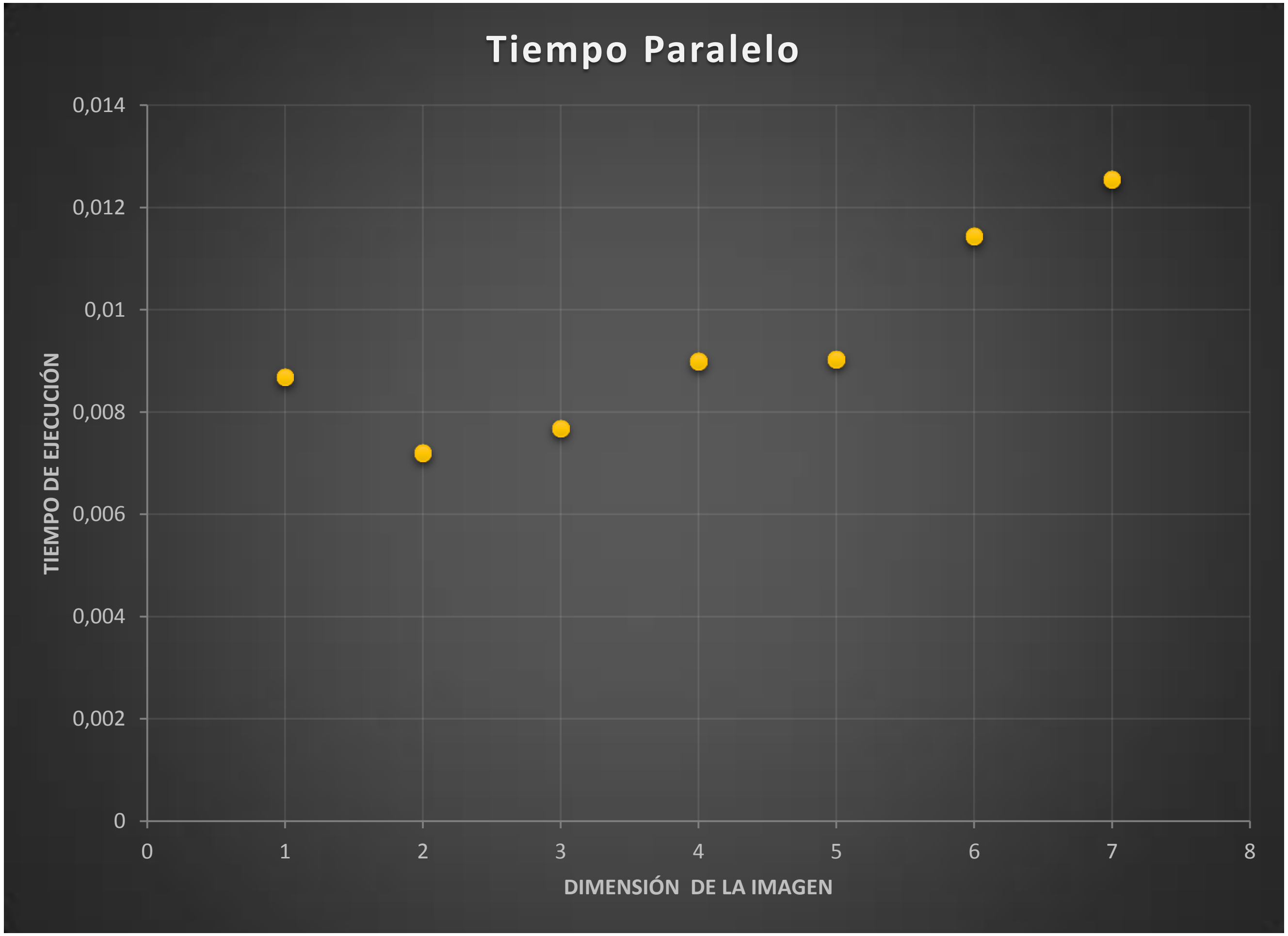
$$fc(z) = z^2 + C$$

Imagen 600x600



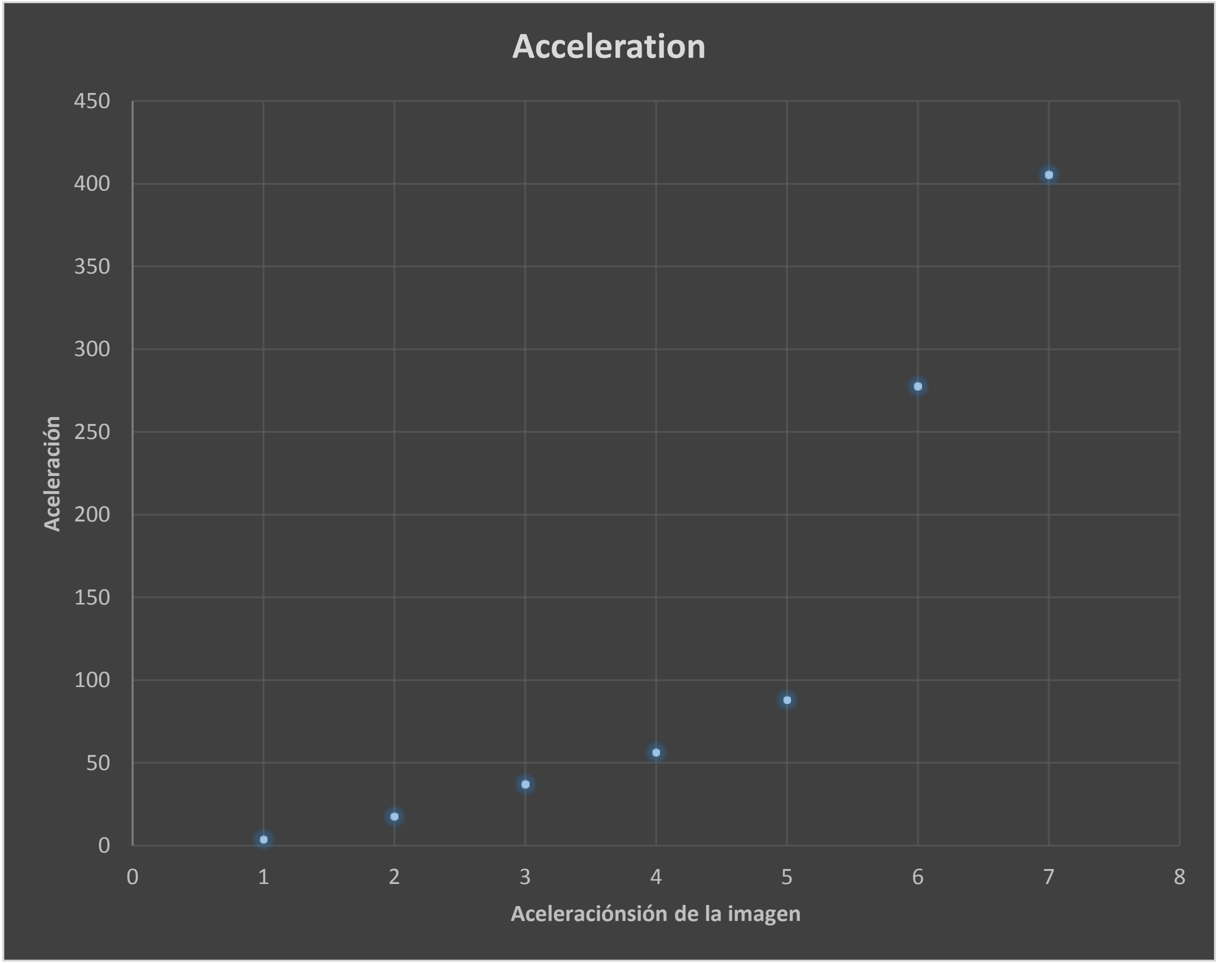
Graficas

ID	Dimensión (px)
1	200x200
2	400x400
3	600x600
4	800x800
5	1000x1000
6	2000x2000
7	2500x2500



Aceleración

ID	Dimensión (px)
1	200x200
2	400x400
3	600x600
4	800x800
5	1000x1000
6	2000x2000
7	2500x2500



Conclusiones

- Para la implementación de manera secuencial hubo un tope en cuanto a la dimensión permitida, a una dimensión de 3000x3000 px, el compilador dejaba de funcionar. Para la dimensión de 2500 px se demoro aproximadamente 1 minuto. En comparación con la paralela, esta arrojaba el resultado casi inmediato.
- Hubo problemas al ejecutarla en GPU debido a que la estructura que se requiere para manejar los números y la función que debe de llamar para generar la imagen, debían de ser ejecutadas en el device.
- Todo lo que se haga en GPU y deba llamar a una función o una estructura de datos, se deben de implementar en el device.
- Es evidente que el procesamiento de datos en paralelo serán de gran beneficio, siempre y cuando estas lo permitan.