Accelerating Fractal Computations: A CUDA-Based Julia Set

- **Project Overview:** Exploring the acceleration of Julia set fractal computations using CUDA for high-performance computing.

- **Comparison Across Platforms:** Implementations in CUDA C++, CPU, and Python to analyze performance improvements.

Github

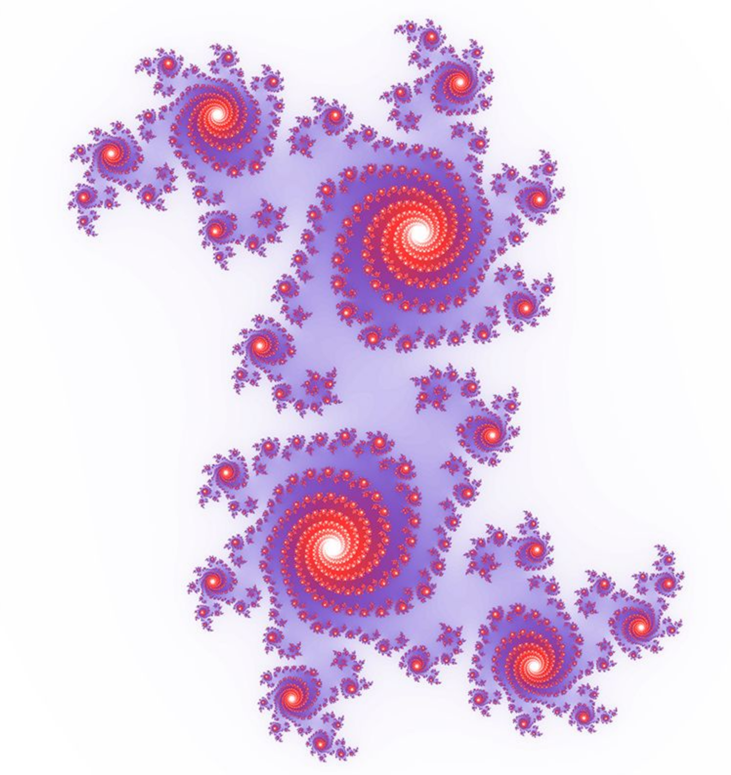


Fig.1. Julia set, c=0.355 + 0.355i

Source: Julia Set Fractal (2D)

Overview of Julia Sets

- **Definition:** Julia sets are fractals derived from complex polynomials. Named after French mathematician Gaston Julia in the early 20th century.
- **Significance:** Visualize complex dynamics and chaos. Important in fields like mathematics, physics, and computer graphics.

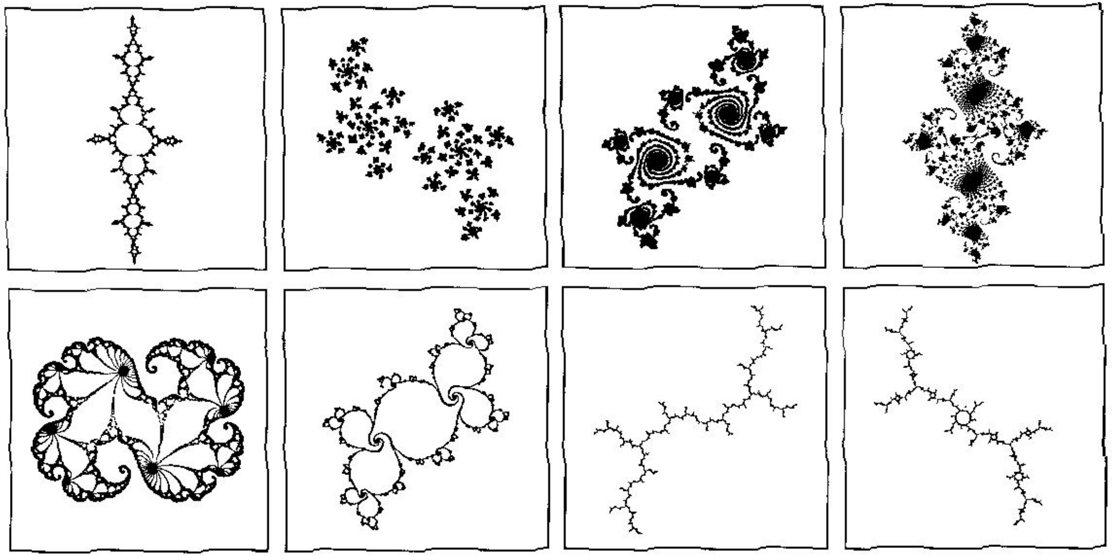


Fig.2. Julia Sets

Source: https://www.karlsims.com/julia.html

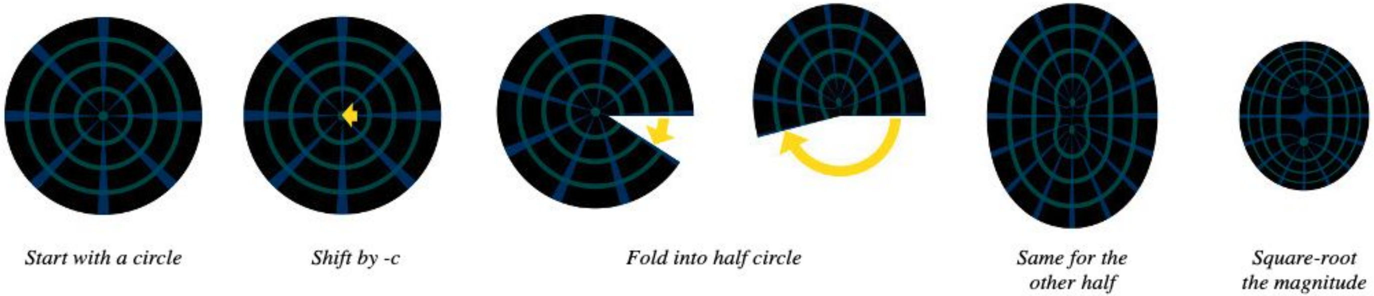
Maths of Julia Sets

Check video: [Folding a Circle into a Julia Set]

Pipeline:

- initialize a complex number $z = x + yi$, x and y are image pixel coordinates in the range of about -2 to 2.
- z is repeatedly updated using $z := z^2 + c$, where c is another complex number that gives a specific Julia set.

This process can be better understood visually by repeatedly transforming a shape using the inverse equation $z = \sqrt{z - c}$. The square root of a complex number halves its angle and square-roots its magnitude: $\sqrt[2]{z} = \sqrt[2]{|z| * e^{i\phi}} = \sqrt[2]{|z|} * e^{i\frac{\phi}{2}}$

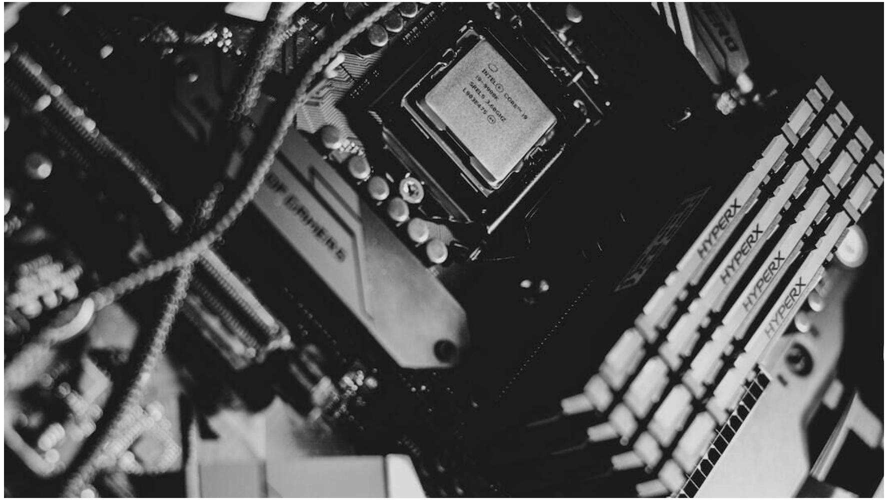


Source: Understanding Julia and Mandelbrot Sets

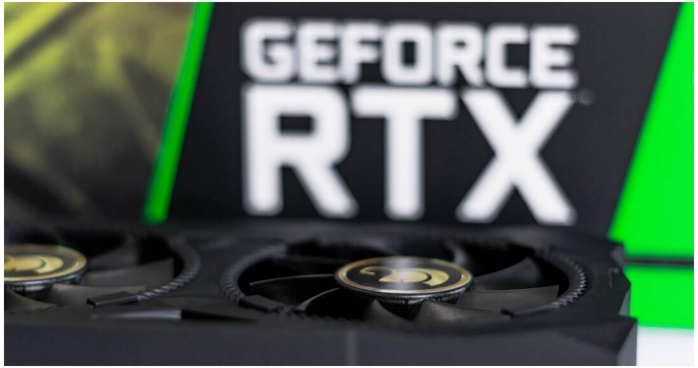
Fig.3. Scheme of 1 iteration

Computational Approaches

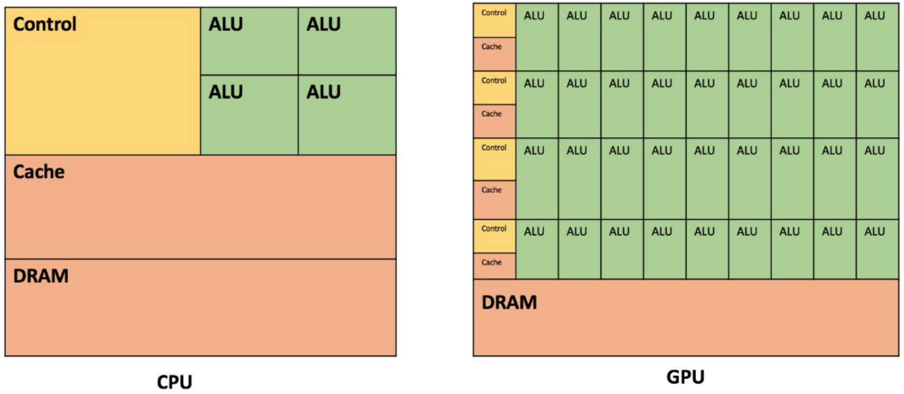
- **CPU Implementation:** Traditional approach using single-threaded or multithreaded computation.
- **CUDA C++ Implementation:** Utilizes GPU parallel processing capabilities for accelerated computations.
- **Python Implementation:** Leverages high-level language simplicity with libraries like NumPy and CuPy for performance.



GPU Parallelism on Generating Julia Sets



- **Pixel Distribution:** Each pixel computation in the Julia set is handled by a separate GPU thread, enabling massive parallelism.
- **Thread Efficiency:** GPU threads work concurrently, significantly speeding up the overall computation process.
- **Scalability:** The approach scales with the number of available GPU cores, improving performance with more powerful GPUs.

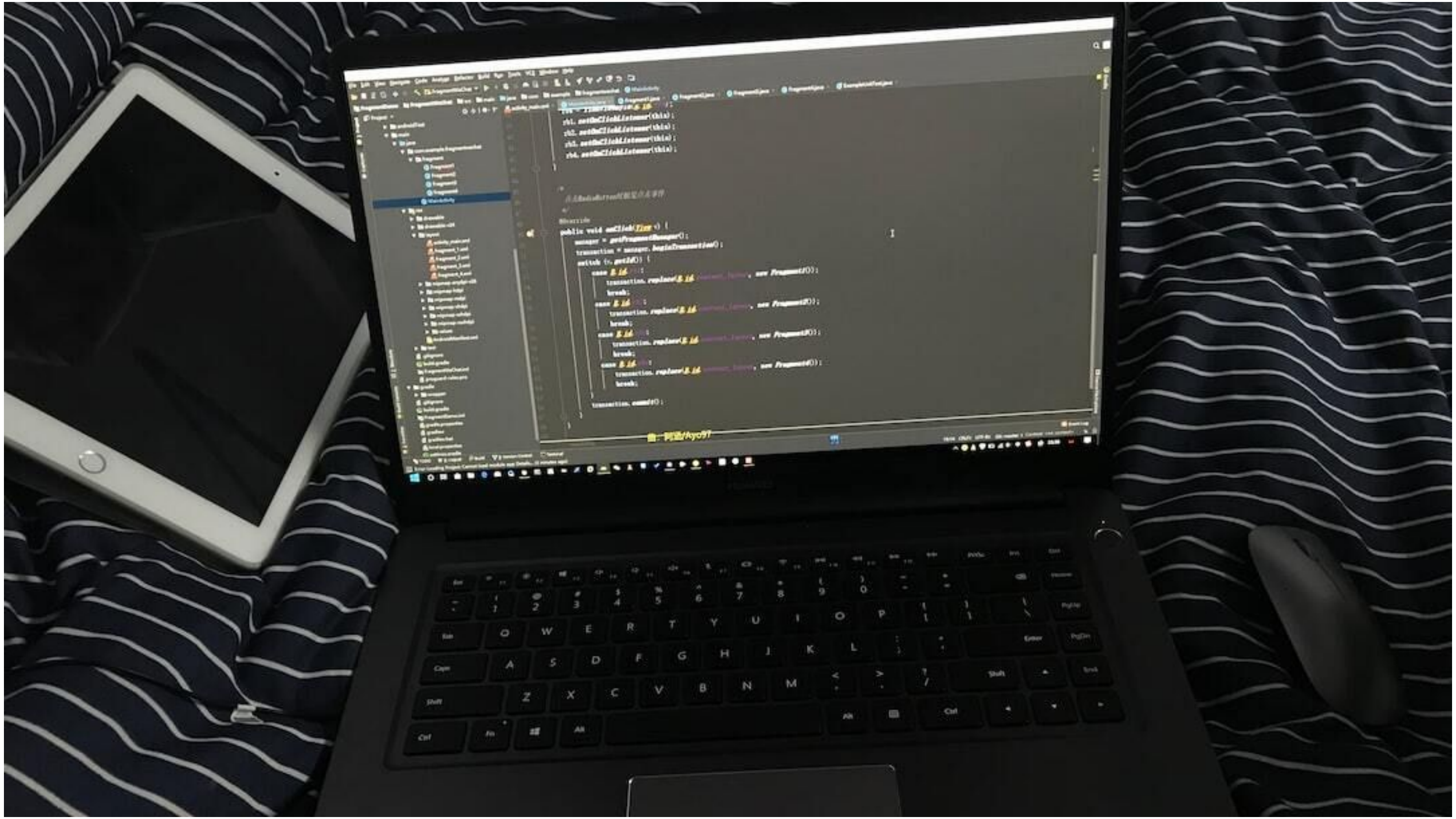


A GPU has more Arithmetic Logic Units (ALU) than a typical CPU.

- Increased ability to process simple operations in parallel

Source: [CPU vs GPU in Machine Learning]

Implementation Details



C++ Implementation

```
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;
        }
    }
}
```

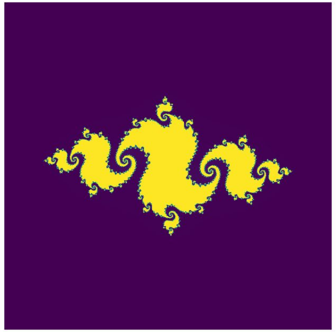
CUDA C++ Implementation

```
__global__ void kernel(unsigned char *ptr) {
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    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;
}

int main(void) {
    unsigned char *dev_bitmap;
    unsigned char *bitmap = new unsigned char[DIM * DIM * 4];

    cudaMalloc((void**)&dev_bitmap, DIM * DIM * 4);

    dim3 grid(DIM, DIM);
    kernel<<<grid, 1>>>(dev_bitmap);

    cudaMemcpy(bitmap, dev_bitmap, DIM * DIM * 4, cudaMemcpyDeviceToHost);
    cudaFree(dev_bitmap);

    return 0;
}
```

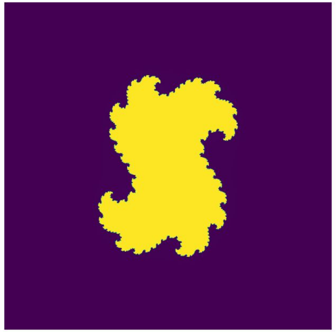
Python Implementation

```
@cuda.jit(device=True)
def julia_kernel(z, c, max_iterations):
    for i in range(max_iterations):
        if abs(z) > 2:
            return i
        z = z**2 + c
    return max_iterations

@cuda.jit
def compute_julia_set_kernel(julia_set, c, max_iterations, width, height, zoom):
    x, y = cuda.grid(2)
    if x < width and y < height:
        zx = -2 * zoom + 4 * zoom * x / (width - 1)
        zy = -2 * zoom + 4 * zoom * y / (height - 1)
        z = complex(zx, zy)
        julia_set[y, x] = julia_kernel(z, c, max_iterations)

def compute_julia_set(c, max_iterations=1000, width=800, height=800, zoom=1):
    julia_set = np.zeros((height, width), dtype=np.uint16)
    threadsperblock = (16, 16)
    blockspergrid_x = int(np.ceil(width / threadsperblock[0]))
    blockspergrid_y = int(np.ceil(height / threadsperblock[1]))
    blockspergrid = (blockspergrid_x, blockspergrid_y)
    compute_julia_set_kernel[blockspergrid, threadsperblock](julia_set, c, max_iterations, width, height, zoom)
    return julia_set
```

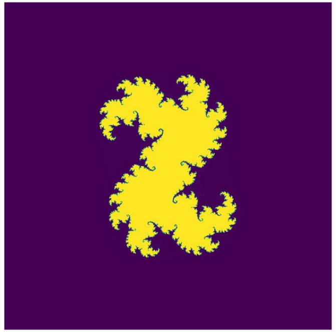
Visualizations of Generated Julia Sets



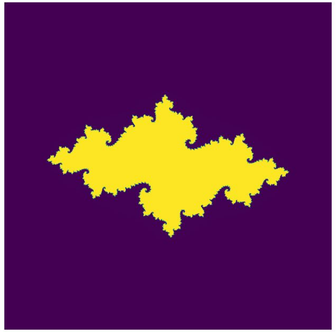
(-0.8, 0.156)



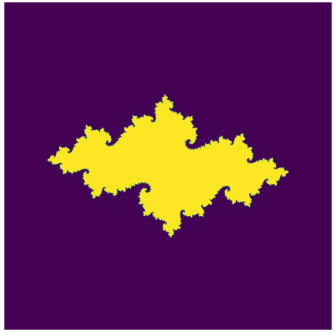
(0.355, 0.355)



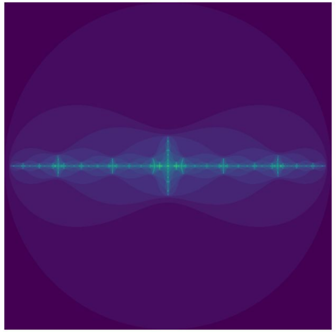
(0.3698, -0.2913)



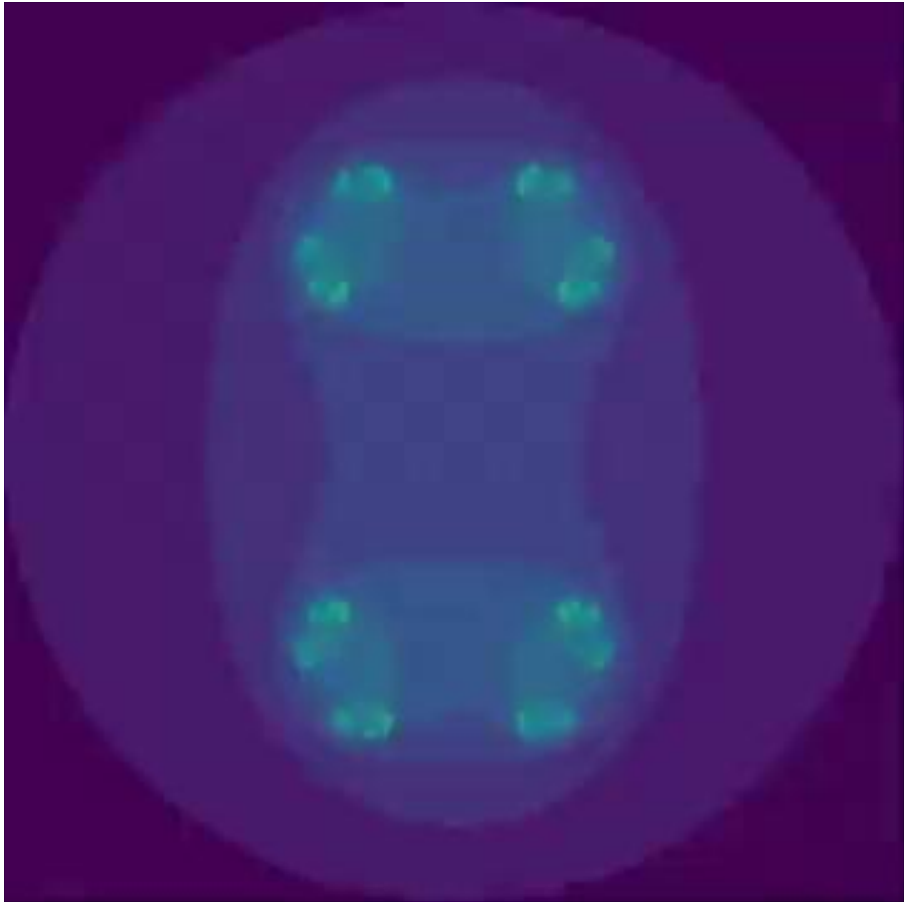
(-0.7269, 0.1889)



(-0.74543, 0.11301)

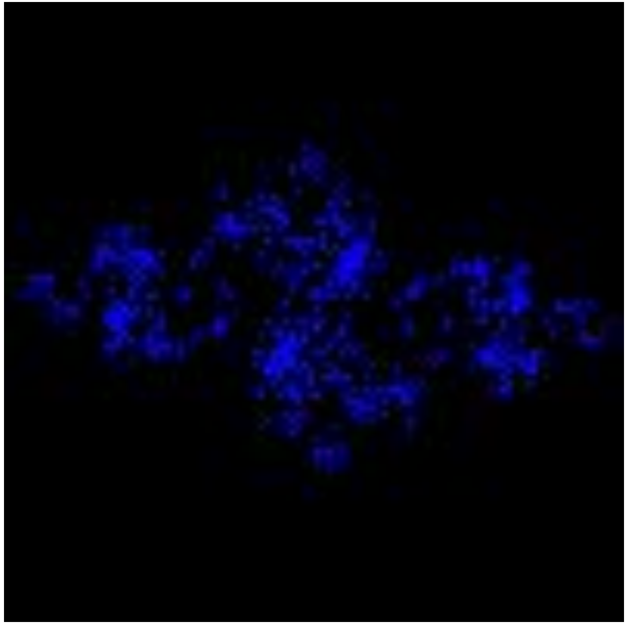


(-1.8, 0.0)

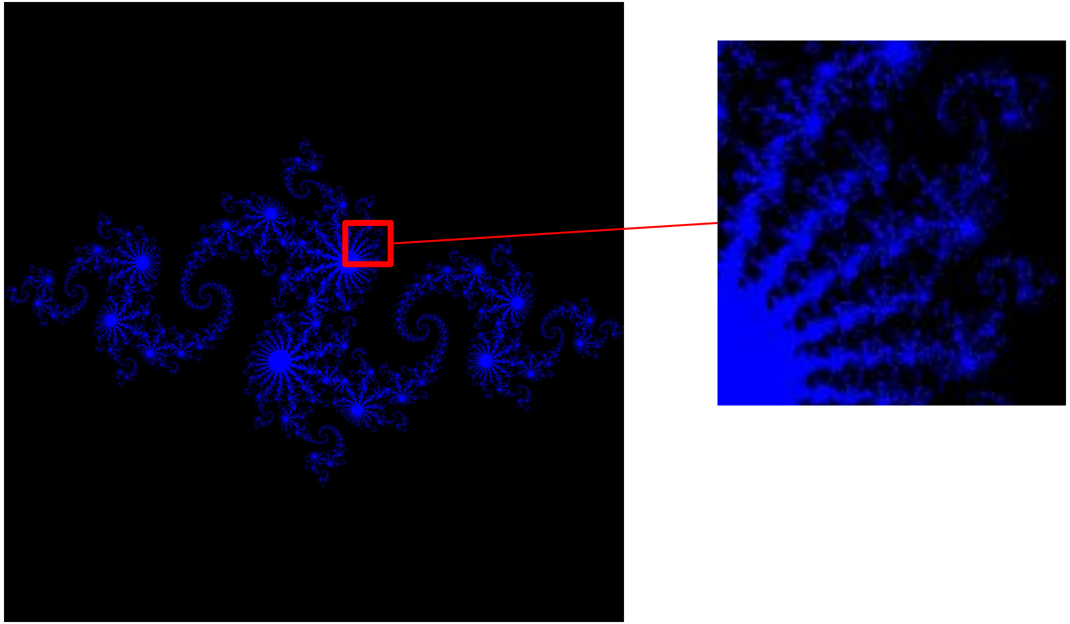


Julia fractals. Feel the difference!

DIM = 100

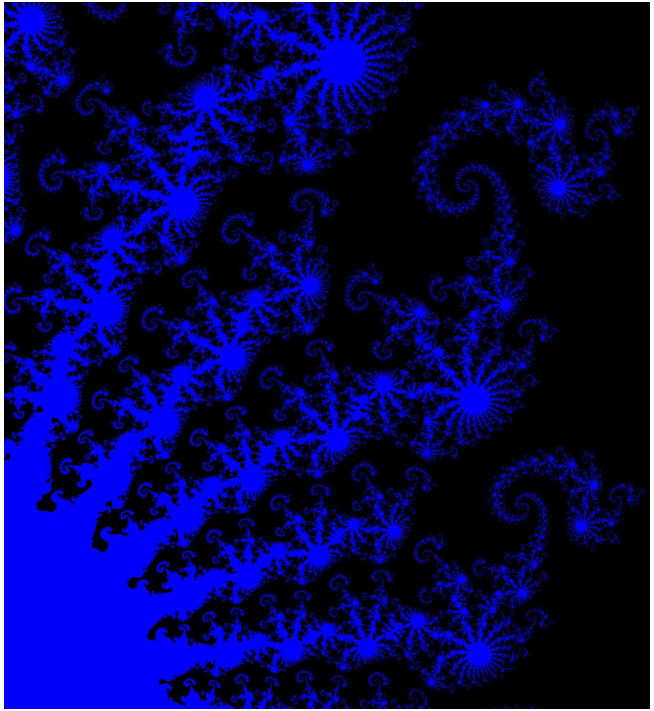


DIM = 3000

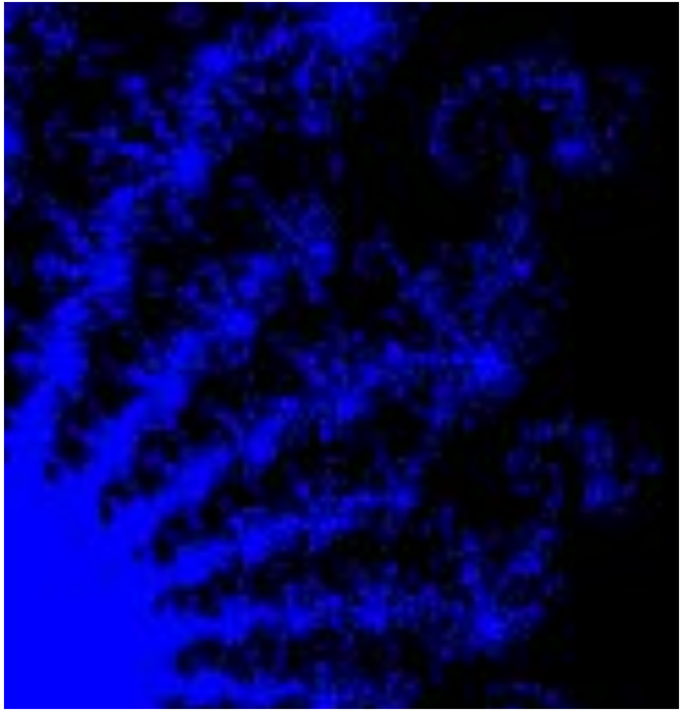


Julia fractals. Feel the difference!

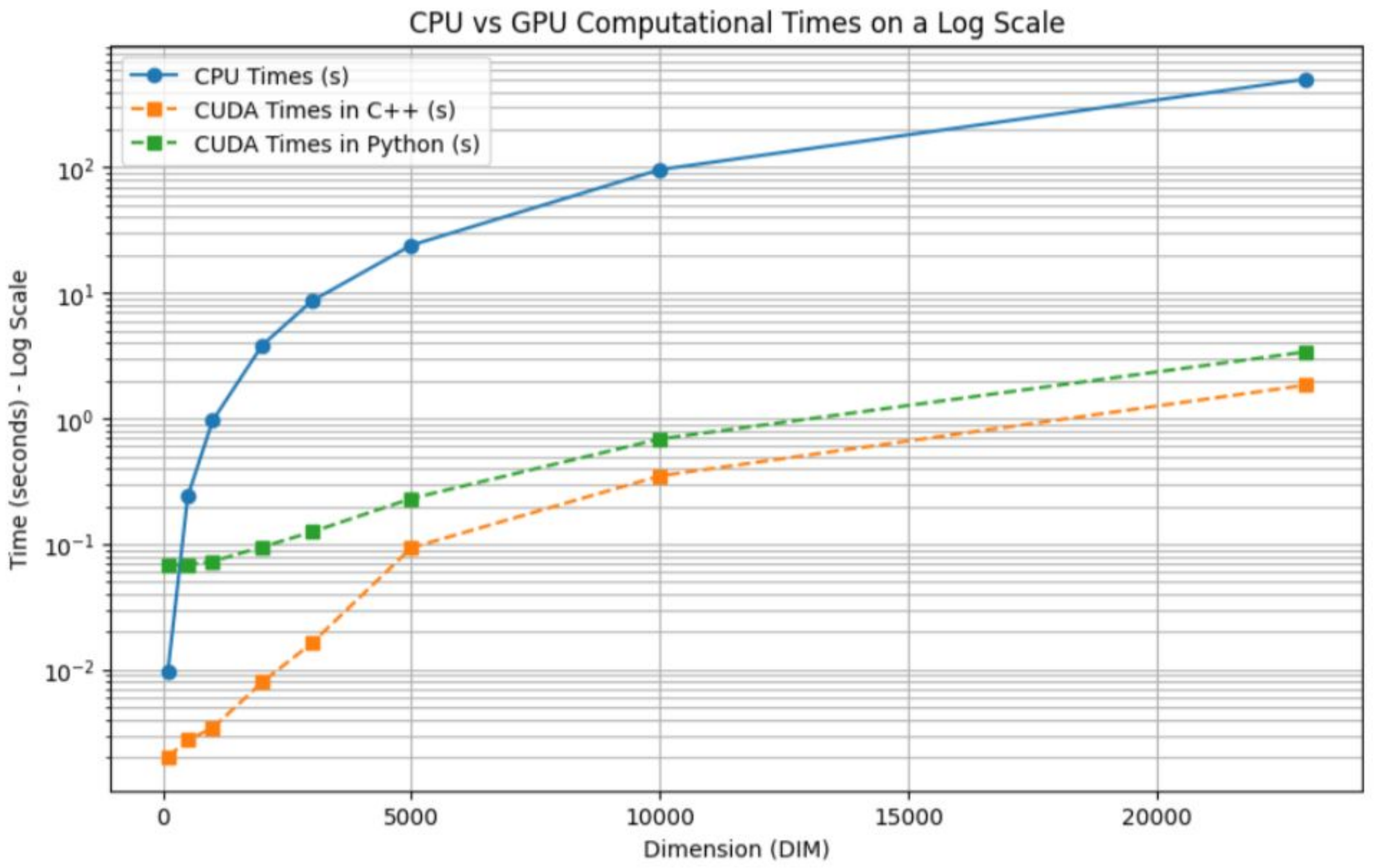
DIM = 23000



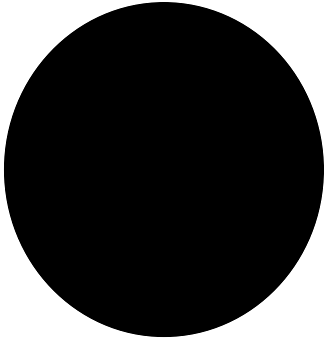
DIM = 3000



Performance Comparison

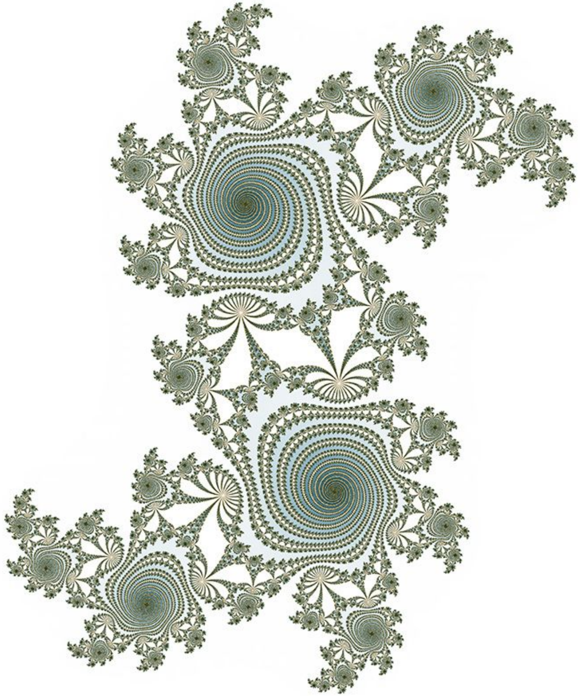


Project Team



Folu Obidare

Maksim Komiakov



Julia set, c=0.355534 - 0.337292i

Source: [Julia Set Fractal (2D)]

Inna Larina

Altair Toleugazinov

THANK YOU