Fractal Image Compression Explained Like You're 5

Imagine you have a big puzzle (your image). Instead of storing every single piece, you find clever ways to **copy, shrink, rotate, or adjust** some pieces to make other pieces.

Step 1: Chop the Image into Small Blocks
- **Range Blocks (R)**: Tiny squares (e.g., 4×4 pixels).
- **Domain Blocks (D)**: Bigger squares (e.g., 8×8 pixels) taken from overlapping parts of the image.

🔹 *Example*:
- **Range Block (R)** → A tiny 4×4 patch of a leaf.
- **Domain Block (D)** → A bigger 8×8 patch from another part of the same leaf.

Step 2: Find Matching Blocks
For each small **R**, search for a big **D** that can be **shrunk + adjusted** to look like **R**.

🔹 *Adjustments include*:
1. **Shrink**: Make **D** smaller (8×8 → 4×4).
2. **Rotate/Flip**: Turn it sideways or upside down if needed.
3. **Brightness/Contrast**: Make it darker, brighter, or inverted.

🔹 *Example*:
- If **R** is a bright leaf, find a darker **D**, then **double its brightness** to match.

Step 3: Save Only the Instructions
Instead of storing all pixels, save:
- **Where R is** (e.g., top-left corner).
- **Which D was used** (e.g., middle-right).
- **How to tweak D** (e.g., "rotate 90°, make 2x brighter").

📝 *Saved as a text file*:
```
RangeX: 0, RangeY: 0
DomainX: 32, DomainY: 64
Rotate: 90°, Flip: None
Contrast: 1.2, Brightness: -10
```

Step 4: Rebuild the Image (Decompress)
1. Start with a blank/random image.
2. For each saved instruction:
   - Find the **D** block.
   - Shrink, rotate, adjust brightness.
   - Paste it into **R**'s position.

3. Repeat 8–16 times → image gets clearer each time!

🎨 *Like tracing over a blurry sketch until it looks sharp.*

Why It’s Cool
✔ **Saves Space**: Stores math formulas, not pixels.
✔ **Fast to Open**: Rebuilding is quick (unlike compression).
✔ **Works Best for Nature**: Trees, clouds, textures repeat a lot!

🚀 **TL;DR**: It’s like solving a puzzle by **copying & tweaking parts of the image** instead of storing every piece!

Absolutely! Let's break this **Fractal Image Compression** code down step-by-step in a simple way. This code uses OpenCV to **compress** a grayscale image using **fractal compression**, and then **decompress** it into an approximate version after a given number of iterations.

✅ Overview of Key Concepts

- **Range blocks**: Small square blocks (e.g., 4x4) that we try to reconstruct.
- **Domain blocks**: Larger blocks (e.g., 8x8) that we downsample and use to approximate range blocks.
- **Affine transform**: A simple linear formula `s * domain + b` is used to map a domain block to a range block.

📌 Step-by-Step Breakdown

---

1. Struct Definition

```
struct Transform {
    int rangeX, rangeY;
    int domainX, domainY;
    double s, b;
};
```

- Holds details of how a **range block** is approximated by a **domain block**:
  - `(rangeX, rangeY)`: top-left corner of range block.
  - `(domainX, domainY)`: top-left of domain block.
  - `s` and `b`: scale and bias for affine transformation.

---

2. Extract a Range Block

```
vector<double> extractBlock(const Mat &img, int x, int y, int size)
```

- Returns a **flattened block** of size `size x size` from the image at position `(x, y)`.

---

3. Extract and Downsample a Domain Block

vector<double> downsampleBlock(const Mat &img, int x, int y, int size)

- Takes a block of size `size x size` (e.g., 8x8), and downsamples it to `size/2 x size/2` (e.g., 4x4).
- Averaging 2×2 pixel values into one.

---

4. Compute Affine Transform (s, b)

void computeAffine(const vector<double> &D, const vector<double> &R, double &s, double &b)

Calculates best `s` and `b` such that:

s * domain_block + b ≈ range_block

-

---

5. Compression Step

void compress(const cv::Mat &img, int rangeSize, std::vector<Transform> &transforms)

💡 Main compression function:

- For each **range block**:
  - Try all possible domain blocks (step size = 4 to speed up).
  - Downsample domain block → make it the same size as range block.
  - Find the **best match** domain block using affine transform.
  - Store the transform (mapping info) in a `Transform` struct.
  - Print progress every 100 range blocks.

⏱ Measures compression time using `clock()`.

---

6. Decompression Step

void decompress(const vector<Transform> &transforms, int width, int height, int rangeSize, int iterations, Mat &outImg)

💡 Main decompression logic:

- Start with a **blank image** (all black).
- For `iterations` number of times:
    - For each transform:
        - Extract the current domain block from the image.
        - Downsample it.
        - Apply the stored `s * D + b` transformation.
        - Put the result into the corresponding range block.
    - Repeat — the image gets **closer and closer** to the original with each iteration.

⏱ Measures decompression time using `clock()`.

---

7. Main Function

int main(int argc, char **argv)

Handles everything:

1. Read input args: image path, output path, iterations.
2. Load grayscale image.
3. Run `compress(...)`.
4. Run `decompress(...)`.
5. Save the final image using `imwrite(...)`.

🔁 Example CLI Run:

./fractal_compress lena.png output.png 10

- Compresses `lena.png`
- Reconstructs it using 10 iterations
- Saves the result as `output.png`

✅ Summary of Core Ideas:

Step	Task	What It Does
1	Range & domain block search	Match small blocks to bigger ones
2	Affine fitting	Compute best `s * D + b ≈ R`
3	Store transforms	Save how to reconstruct
4	Iterative decompression	Rebuild image over time
5	Time measured for both steps	Track performance

Want me to help you visualize how compression & decompression look side-by-side or explain performance tradeoffs too?

Sure! Let’s walk through the code line by line and explain everything in simple terms. This program loads a grayscale image, reduces its resolution using SYCL (a parallel programming model), and saves the result.

Headers & Namespaces

```
#include <sycl/sycl.hpp>
#include <opencv2/opencv.hpp>
#include <vector>
```

- `sycl/sycl.hpp`: SYCL is a C++ abstraction layer for writing code that runs on CPU, GPU, or other accelerators.
- `opencv2/opencv.hpp`: Brings in all OpenCV functionalities for image processing.
- `vector`: For using dynamic arrays.

```
using namespace std;
using namespace cv;
```

- Avoids writing `std::` and `cv::` everywhere. For example, you can write `vector` instead of `std::vector`.

Function to Reduce Image Resolution

```
void reduce_image(sycl::queue& q, const vector<float>& input_image, vector<float>& output_image, int width, int height, int factor)
```

- This function takes an image (`input_image`) and reduces its size by a `factor` using SYCL parallelism.
- `q` is a SYCL queue, which schedules and runs tasks on a device (CPU/GPU).

Inside the function:

```
int output_width = width / factor, output_height = height / factor;
```

- Calculates the dimensions of the output (reduced) image.

```
sycl::buffer<float> input_buf(input_image.data(), sycl::range<1>(input_image.size()));
sycl::buffer<float> output_buf(output_image.data(), sycl::range<1>(output_image.size()));
```

- Wraps input and output image data in SYCL buffers so they can be accessed in kernels.

```
q.submit([&](sycl::handler& cgh) {
```

- Submits a parallel task to the device.

```
auto input_acc = input_buf.get_access<sycl::access::mode::read>(cgh);
auto output_acc = output_buf.get_access<sycl::access::mode::write>(cgh);
```

- Gets access to the buffers: input for reading, output for writing.

```
cgh.parallel_for<class reduce_kernel>(sycl::range<2>(output_height, output_width), [=](sycl::item<2>
item) {
```

- Launches a 2D parallel kernel where each work-item processes one output pixel.

Inside the kernel:

```
int y = item.get_id(0), x = item.get_id(1);
float sum = 0.0f;
```

- `x` and `y` are coordinates of the current output pixel.

```
for (int dy = 0; dy < factor; ++dy)
    for (int dx = 0; dx < factor; ++dx) {
        int input_x = x * factor + dx, input_y = y * factor + dy;
        if (input_x < width && input_y < height)
            sum += input_acc[input_y * width + input_x];
    }
```

- Loops over a `factor × factor` block in the input image and computes the average.
- Makes sure not to go out of bounds.

```
output_acc[y * output_width + x] = sum / (factor * factor);
```

- Stores the average value into the output image at position `(x, y)`.

```
}).wait();
```

- Waits for the computation to finish.

Main Function

```
int main() {

Mat img = imread("Sandesh.jpeg", IMREAD_GRAYSCALE);
if (img.empty()) return -1;
```

- Loads a grayscale image from file. If loading fails, returns -1.

```
int width = img.cols, height = img.rows, factor = 4;
```

- Gets the image size and sets the reduction factor.

```
vector<float> input_img(width * height);
for (int i = 0; i < height; ++i)
    for (int j = 0; j < width; ++j)
        input_img[i * width + j] = img.at<uchar>(i, j);
```

- Converts the OpenCV matrix to a flat `vector<float>` for SYCL processing.

```
vector<float> output_img((width / factor) * (height / factor));
```

- Creates a vector to store the downsampled image.

```
sycl::queue q;
reduce_image(q, input_img, output_img, width, height, factor);
```

- Creates a SYCL queue and calls the function to reduce the image using GPU/CPU.

```
Mat output_mat(height / factor, width / factor, CV_32F, output_img.data());
output_mat.convertTo(output_mat, CV_8U);
```

- Converts the flat `output_img` vector into a proper OpenCV matrix (`Mat`) and changes its type from float to 8-bit unsigned for saving.

```
imwrite("output_image.png", output_mat, {IMWRITE_PNG_COMPRESSION, 3});
```

- Saves the reduced image as PNG with compression level 3.

return 0;
}

What This Program Does in Summary:

1. Loads a grayscale image.
2. Reduces its resolution by averaging blocks of pixels using SYCL parallelism.
3. Saves the smaller version of the image to disk.

Let me know if you want a diagram or visualization for this!