PROCEDURAL TERRAIN GENERATION

Team Members:

- **Vipin Karthic** - Roll No: 2023BCS0011
- **Sanjay** - Roll No: 2023BCS0020

- Course Instructor : **Dr.John Paul Martin**
- Department of Computer Science, Indian Institute of information Technology Kottayam
- Date of Submission: 26th October 2025

ABSTRACT :

This project focuses on creating and parallelizing a procedural terrain generation system that simulates realistic landscapes using Voronoi based heightmap generation with Fractal Brownian Motion, particle based hydraulic erosion simulation, flow accumulation based river generation, climate/moisture map creation, biome classification, and density based object placement. The serial version was first implemented in C++ to establish the logic, which was a skeletal implementation which was then followed by an optimized OpenMP parallel implementation. The terrain generation process involves calculations across millions of independent points, making it perfect for parallelization. Analysis showed a speedup of 1.8x to 6.7x on large terrains, which used multi parallel BFS, thread local buffer accumulation

1. INTRODUCTION :

Procedural terrain generation is one of the most important pieces for creating natural landscapes in computer graphics, simulations, and scientific modeling and game development. It has to compute intensive operations such as noise based maps, erosion simulation, and hydrological erosion, Physics based object mapping, which contains millions of points

Parallelization is important here because terrain generation algos often operate on large grids and sometimes even realtime, where each point’s data can be calculated independently or semi independently. Without parallelization, high resolution terrains become very intensive to calculate. Using OpenMP to parallelize allows efficient utilization of multi core processors, which results in faster map generation and simulation

Terrain gen is used heavily in Game development, GIS modeling and Simulations, where he creation of large terrains must be efficient as possible, We focus on parallelizing the major parts that can have impact, Noise gen, Erosion, River gen.

1.1 Relevance to Parallel Computing

- **Data Parallelism**: Grid based operations are efficient if SIMD parallelization is used
- **Task Parallelism**: Independent droplets in erosion simulation can execute concurrently
- **Load Balancing**: Dynamic scheduling handles irregular workloads in object placement
- **Synchronization**: Critical sections and atomic operations for thread safety

2. LITERATURE SURVEY :

- Perlin introduced coherent noise functions that changed procedural content generation. FBM makes this better this by combining multiple layers of noise at different frequencies and amplitudes, producing natural looking noises which can be translated to terrains
- Forture developed efficient algorithms for Voronoi diagram computation. In terrain generation, Voronoi cells act as the tectonic plates, which are continental boundaries and mountain ridges where plates meet
- Musgrave dev'd erosion models. Modern droplet based solutions water droplets with gravity, inertia, eroding and sedimentation
- D8 flow algorithm for river generation is made by Mark and Callaghan. It is found using topologically sorted elevation order
- Voronoi generation was parallelized by Spatial partition by Bayer and Giersch

3. PROBLEM STATEMENT AND OBJECTIVES :

3.1 Problem Definition :

- The core problem handled here is the generation of of realistic terrain of sized 256x256 to 2048x2048 using noise based heightmaps, erosions and flow accumulations
- Each grid cell must represent Terrain elevation and environmental features
- It contains the following high level computations :

- Perlin Noise Generation using Fractal Brownian Motion
- Voronoi Island Generation
- Biome Classifications
- Heightmap Generation using the Perlin Noise map
- Particle based Erosion Simulation
- River Generation using flow accumulation
- Object placement and Distribution based of Biome

3.2 Computational Complexity :

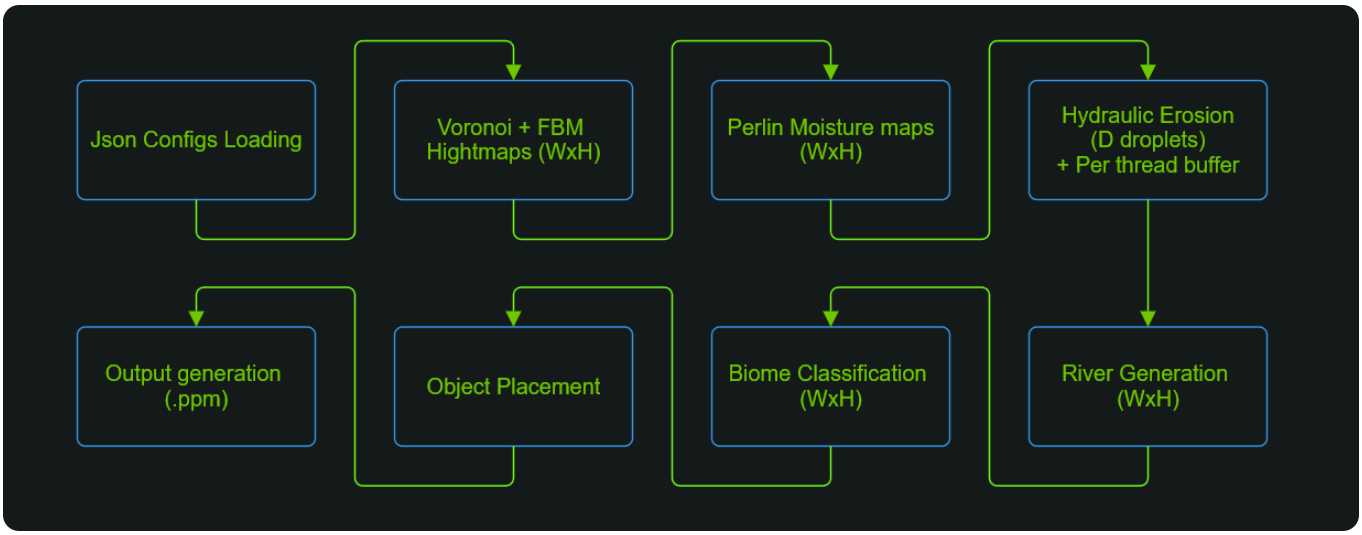
- For a terrain of dimensions W×H:
  - Consider the following :
    - P = Number of voronoi plates
    - D = Droplets
    - S = Steps per droplet
    - R = Erosion radius
    - B = Number of Biome Types
    - O() : Average of
  - Heightmap generation: $O(W \times H \times P)$
  - Erosion simulation: $O(D \times S \times R^2)$
  - River generation: $O(W \times H \times \log(W \times H))$
  - Biome classification: $O(W \times H \times B)$
  - Object placement: $O(W \times H \times O)$

3.2 Problem Objectives :

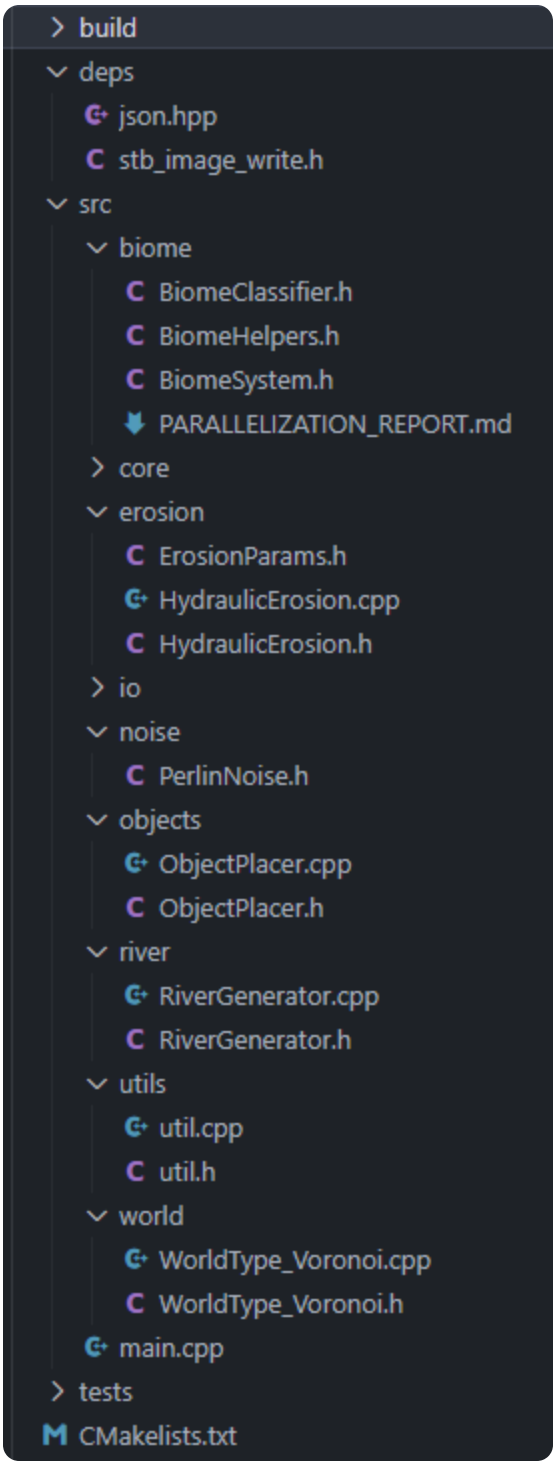
- To implement a baseline serial solution
- To identify parallelizable segments of the code
- To design an OpenMP-based implementation
- To analyze and compare performance and scalability

4. METHODOLOGY AND SYSTEM ARCHITECTURE :

4.1 System Architecture and Data Flow :



4.1.1 Project Structure :



4.1.2 Config Definitions :

The config structure is as follows :

CPP > terrain_gen > assets > {} config.json > ...

```json
{
    "seed": 424242,
    "width": 256,
    "height": 256,
    "worldType": "Voronoi",
    "numPlates": 36,
    "fbmBlend": 0.42,
    "fbmFrequency": 0.0035,
    "fbmOctaves": 5,
    "oceanHeightThreshold": 0.35,
    "lakeHeightThreshold": 0.45,
    "coastDistanceTiles": 3,
    "smoothingIterations": 1
}
```

- config.json

CPP > terrain_gen > assets > {} biomes.json > ...


```json
[
    {
        "id": "Ocean",
        "name": "Ocean",
        "treeDensity": 0,
        "rockDensity": 0,
        "grassDensity": 0,
        "prefMinElevation": 0.0,
        "prefMaxElevation": 0.35,
        "requiresWater": true
    },
    {
        "id": "Beach",
        "name": "Beach",
        "treeDensity": 0.01,
        "rockDensity": 0.02,
        "grassDensity": 0.15,
        "prefMinElevation": 0.0,
        "prefMaxElevation": 0.40,
        "prefersCoast": true
    },
    {
        "id": "Lake",
        "name": "Lake",
        "treeDensity": 0.02,
        "rockDensity": 0.03,
        "grassDensity": 0.25,
        "prefMinElevation": 0.0,
        "prefMaxElevation": 0.45,
        "requiresWater": true
    },
    {
        "id": "Mangrove",
        "name": "Mangrove",
        "treeDensity": 0.5,
        "rockDensity": 0.05,
        "grassDensity": 0.15,
        "moistureModifier": 1.3,
        "prefMinElevation": 0.0,
        "prefMaxElevation": 0.5,
        "requiresWater": true,
        "prefersCoast": true
    },
    {
        "id": "Desert",
        "name": "Desert"
```


- biomes.json

CPP > terrain_gen > assets > {} object_placement.json > ...


```json
{
    "seed": 424242,
    "global_max_instances": 200000,
    "default_min_distance_m": 1.0,
    "models_path": "assets/models/",
    "biome_objects": {
        "Ocean": [
            {
                "name": "placeholder_seaweed",
                "model": "",
                "placeholder": true,
                "density_per_1000m2": 0,
                "min_distance_m": 0.5,
                "requires_water": true
            }
        ],
        "Beach": [
            {
                "name": "placeholder_palm",
                "model": "",
                "placeholder": true,
                "density_per_1000m2": 0.4,
                "min_distance_m": 3.0,
                "scale_min": 0.8,
                "scale_max": 1.6,
                "elevation_min": 0.0,
                "elevation_max": 0.40,
                "prefers_coast": true
            },
            {
                "name": "placeholder_driftwood",
                "model": "",
                "placeholder": true,
                "density_per_1000m2": 0.6,
                "min_distance_m": 1.5,
                "elevation_min": 0.0,
                "elevation_max": 0.40
            },
            {
                "name": "placeholder_beach_grass",
                "model": "",
                "placeholder": true,
                "density_per_1000m2": 12,
                "min_distance_m": 0.6,
                "elevation_min": 0.0,
```


- object_placement.json
- The configs are read and inteperted and the data is stored as [EXAMPLE] :

```cpp
json cfg;
try {
    f >> cfg;
} catch (const std::exception& e) {
    std::cerr << "[EXCEPTION] Failed parsing configs " << e.what() << std::endl;
    return 1;
}

f.close();
for (auto it = cfg.begin(); it != cfg.end(); ++it) std::cerr << it.key() << " ";

int W = cfg.value("width", 512);
```

```
int H = cfg.value("height", 512);
uint32_t seed = cfg.value("seed", 424242u);
```

4.2 Utils and Helper Implementations :

4.2.1 Custom PRNG :

```
namespace rng_util {
RNG::RNG(ll seed) : _state(seed) {}

int RNG::nextInt() {
    ll x = _state;
    x ^= x >> 12;
    x ^= x << 25;
    x ^= x >> 27;
    _state = x;
    return static_cast<int>((x * 2685821657736338717LL) >> 32);
}

float RNG::nextFloat() { return (nextInt() / 4294967296.0f); }

ll splitmix(ll &state) {
    ll z = (state += 2654435769LL);
    z = (z ^ (z >> 30)) * 2246822507LL;
    z = (z ^ (z >> 27)) * 3255373325LL;
    return z ^ (z >> 31);
}

}  // namespace rng_util
```

4.2.2 Map Utils :

```
namespace map {

void computeSlopeMap(const std::vector<float> &height, int W, int H, std::vector<float
    out_slope.assign(W * H, 0.0f);
    auto idx = [&](int x, int y) { return y * W + x; };
#pragma omp parallel for collapse(2) schedule(static)
```

```
    for (int y = 0; y < H; ++y) {
        for (int x = 0; x < W; ++x) {
            float hz = height[idx(x, y)];
            float hxm = (x > 0) ? height[idx(x - 1, y)] : hz;
            float hxp = (x < W - 1) ? height[idx(x + 1, y)] : hz;
            float hym = (y > 0) ? height[idx(x, y - 1)] : hz;
            float hyp = (y < H - 1) ? height[idx(x, y + 1)] : hz;
            float gx = (hxp - hxm) * 0.5f;
            float gy = (hyp - hym) * 0.5f;
            out_slope[idx(x, y)] = std::sqrt(gx * gx + gy * gy);
        }
    }
}

void computeWaterMask(const std::vector<float> &height, int W, int H, float oceanThres
    out_waterMask.assign(W * H, 0);
    auto idx = [&](int x, int y) { return y * W + x; };

    std::vector<unsigned char> potential(W * H, 0);
#pragma omp parallel for schedule(static)
    for (int i = 0; i < W * H; ++i) potential[i] = (height[i] <= lakeThreshold) ? 1 :

    std::vector<unsigned char> visited(W * H, 0);
    std::queue<int> q;
    for (int x = 0; x < W; ++x) {
        int i0 = idx(x, 0), i1 = idx(x, H - 1);
        if (potential[i0] && !visited[i0]) {
            visited[i0] = 1;
            q.push(i0);
        }
        if (potential[i1] && !visited[i1]) {
            visited[i1] = 1;
            q.push(i1);
        }
    }
    for (int y = 0; y < H; ++y) {
        int i0 = idx(0, y), i1 = idx(W - 1, y);
        if (potential[i0] && !visited[i0]) {
            visited[i0] = 1;
            q.push(i0);
        }
        if (potential[i1] && !visited[i1]) {
            visited[i1] = 1;
            q.push(i1);
        }
    }
```

```
    int dirs[4][2] = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    while (!q.empty()) {
        int cur = q.front();
        q.pop();
        int cx = cur % W, cy = cur / W;
        if (height[cur] <= oceanThreshold)
            out_waterMask[cur] = 1;
        else
            out_waterMask[cur] = 1;
        for (auto &d : dirs) {
            int nx = cx + d[0], ny = cy + d[1];
            if (nx < 0 || ny < 0 || nx >= W || ny >= H) continue;
            int ni = idx(nx, ny);
            if (!visited[ni] && potential[ni]) {
                visited[ni] = 1;
                q.push(ni);
            }
        }
    }
    for (int i = 0; i < W * H; ++i) {
        if (potential[i] && !visited[i]) out_waterMask[i] = 1;
    }
}

void computeCoastDistance(const std::vector<unsigned char> &waterMask, int W, int H, s
    const int INF = std::numeric_limits<int>::max() / 4;
    out_coastDist.assign(W * H, INF);
    std::queue<int> q;
    auto idx = [&](int x, int y) { return y * W + x; };
#pragma omp parallel for collapse(2) schedule(static)
    for (int y = 0; y < H; ++y)
        for (int x = 0; x < W; ++x) {
            int i = idx(x, y);
            if (waterMask[i]) {
                out_coastDist[i] = 0;
#pragma omp critical
                q.push(i);
            }
        }
    int dirs[4][2] = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    while (!q.empty()) {
        int cur = q.front();
        q.pop();
        int cx = cur % W, cy = cur / W;
        for (auto &d : dirs) {
            int nx = cx + d[0], ny = cy + d[1];
```

```
            if (nx < 0 || ny < 0 || nx >= W || ny >= H) continue;
            int ni = idx(nx, ny);
            if (out_coastDist[ni] > out_coastDist[cur] + 1) {
                out_coastDist[ni] = out_coastDist[cur] + 1;
                q.push(ni);
            }
        }
    }
}

}  // namespace map
```

4.2.3 Image Writers and Data converters :

```
namespace helper {
std::vector<float> gridToVector(const Grid2D<float> &g) {
    int W = g.width(), H = g.height();
    std::vector<float> v((size_t)W * H);
#pragma omp parallel for collapse(2) schedule(static)
    for (int y = 0; y < H; ++y)
        for (int x = 0; x < W; ++x) v[(size_t)y * W + x] = g(x, y);
    return v;
}

void vectorToGrid(const std::vector<float> &v, Grid2D<float> &g) {
    int W = g.width(), H = g.height();
#pragma omp parallel for collapse(2) schedule(static)
    for (int y = 0; y < H; ++y)
        for (int x = 0; x < W; ++x) g(x, y) = v[(size_t)y * W + x];
}

std::vector<unsigned char> maskToRGB(const std::vector<uint8_t> &mask, int W, int H) {
    std::vector<unsigned char> out((size_t)W * H * 3);
#pragma omp parallel for collapse(2) schedule(static)
    for (int y = 0; y < H; ++y)
        for (int x = 0; x < W; ++x) {
            size_t i = (size_t)y * W + x;
            unsigned char m = mask[i];
            size_t idx = i * 3;
            out[idx + 0] = m;
            out[idx + 1] = m;
            out[idx + 2] = m;
        }
```

```
    return out;
}

bool writePPM(const std::string &path, int W, int H, const std::vector<unsigned char>
    std::ofstream f(path, std::ios::binary);
    if (!f) return false;
    f << "P6\n" << W << " " << H << "\n255\n";
    f.write((const char *)rgb.data(), rgb.size());
    f.close();
    return true;
}

std::vector<unsigned char> heightToRGB(const Grid2D<float> &g) {
    int W = g.width(), H = g.height();
    std::vector<unsigned char> out((size_t)W * H * 3);
#pragma omp parallel for collapse(2) schedule(static)
    for (int y = 0; y < H; ++y)
        for (int x = 0; x < W; ++x) {
            float v = g(x, y);
            unsigned char c = (unsigned char)std::lround(std::clamp(v, 0.0f, 1.0f) * 2
            size_t idx = ((size_t)y * W + x) * 3;
            out[idx + 0] = c;
            out[idx + 1] = c;
            out[idx + 2] = c;
        }
    return out;
}

std::vector<unsigned char> biomeToRGB(const Grid2D<Biome> &g) {
    int W = g.width(), H = g.height();
    std::vector<unsigned char> out((size_t)W * H * 3);
    auto colorOf = [](Biome b) -> std::array<unsigned char, 3> {
        switch (b) {
            case Biome::Ocean:
                return {24, 64, 160};
            case Biome::Beach:
                return {238, 214, 175};
            case Biome::Lake:
                return {36, 120, 200};
            case Biome::Mangrove:
                return {31, 90, 42};
            case Biome::Desert:
                return {210, 180, 140};
            case Biome::Savanna:
                return {189, 183, 107};
            case Biome::Grassland:
```

```
                return {130, 200, 80};
            case Biome::TropicalRainforest:
                return {16, 120, 45};
            case Biome::SeasonalForest:
                return {34, 139, 34};
            case Biome::BorealForest:
                return {80, 120, 70};
            case Biome::Tundra:
                return {180, 190, 200};
            case Biome::Snow:
                return {240, 240, 250};
            case Biome::Rocky:
                return {140, 130, 120};
            case Biome::Mountain:
                return {120, 120, 140};
            case Biome::Swamp:
                return {34, 85, 45};
            default:
                return {255, 0, 255};
        }
    };
#pragma omp parallel for collapse(2) schedule(static)
    for (int y = 0; y < H; ++y)
        for (int x = 0; x < W; ++x) {
            auto c = colorOf(g(x, y));
            size_t idx = ((size_t)y * W + x) * 3;
            out[idx + 0] = c[0];
            out[idx + 1] = c[1];
            out[idx + 2] = c[2];
        }
    return out;
}

}  // namespace helper
```

4.2.4 PerlinNoise Generator :

```
#pragma once
#include <omp.h>

#include <algorithm>
#include <cmath>
#include <cstdint>
```

```cpp
#include <vector>

#include "util.h"

using ll = long long;

class PerlinNoise {
   public:
    explicit PerlinNoise(int seed = 1337, int permSize = 256) { init(seed, permSize);
    void init(int seed, int permSize = 256) {
        permSize_ = permSize;
        rng_util::RNG rng(seed);
        _p.resize(permSize_);

        // cab just use iota but lets just parallelize it cz why not
#pragma omp parallel for schedule(static)
        for (int i = 0; i < permSize_; i++) {
            _p[i] = i;
        }

        for (int i = permSize_ - 1; i > 0; --i) {
            uint32_t r = static_cast<uint32_t>(rng.nextInt());
            int j = static_cast<int>(r % static_cast<uint32_t>(i + 1));
            if (j < 0) j = 0;
            if (j > i) j = i;
            std::swap(_p[i], _p[j]);
        }

        _p.resize(permSize_ * 2);
#pragma omp parallel for schedule(static)
        for (int i = 0; i < permSize_; i++) {
            _p[i + permSize_] = _p[i];
        }
    }

    float noise(float x, float y, float frequency = 1.0f) const {
        x *= frequency;
        y *= frequency;
        int xi = fastfloor(x) & 255;
        int yi = fastfloor(y) & 255;
        float xf = x - floorf(x);
        float yf = y - floorf(y);
        float u = fade(xf);
        float v = fade(yf);
        int aa = _p[_p[xi] + yi];
        int ab = _p[_p[xi] + yi + 1];
```

```
        int ba = _p[_p[xi + 1] + yi];
        int bb = _p[_p[xi + 1] + yi + 1];
        float x1 = lerp(grad(aa, xf, yf), grad(ba, xf - 1.0f, yf), u);
        float x2 = lerp(grad(ab, xf, yf - 1.0f), grad(bb, xf - 1.0f, yf - 1.0f), u);
        float res = lerp(x1, x2, v);
        return std::clamp(res, -1.0f, 1.0f);
    }

    float fbm(float x, float y, float baseFreq, int octaves, float lacunarity = 2.0f,
        float amp = 1.0f;
        float freq = 1.0f;
        float sum = 0.0f;
        float maxAmp = 0.0f;
        for (int i = 0; i < octaves; i++) {
            float n = noise(x, y, baseFreq * freq);
            sum += n * amp;
            maxAmp += amp;
            amp *= gain;
            freq *= lacunarity;
        }
        if (maxAmp > 0.0f) sum /= maxAmp;
        return std::clamp(sum, -1.0f, 1.0f);
    }

  private:
    std::vector<int> _p;
    int permSize_ = 256;
    static inline int fastfloor(float x) { return (int)floorf(x); }
    static inline float fade(float t) { return t * t * t * (t * (t * 6 - 15) + 10); }
    static inline float lerp(float a, float b, float t) { return a + t * (b - a); }
    static inline float grad(int hash, float x, float y) {
        int h = hash & 7;
        float u = h < 4 ? x : y;
        float v = h < 4 ? y : x;
        return ((h & 1) ? -u : u) + ((h & 2) ? -2.0f * v : 2.0f * v) * 0.5f;
    }
};
```

4.2.5 Erosion Helpers :

```
namespace erosion {

static inline float sampleBilinear(const GridFloat &g, float fx, float fy) {
```

```cpp
    int w = g.width(), h = g.height();
    if (fx < 0) fx = 0;
    if (fy < 0) fy = 0;
    if (fx > w - 1) fx = (float)(w - 1);
    if (fy > h - 1) fy = (float)(h - 1);
    int x0 = (int)floor(fx);
    int y0 = (int)floor(fy);
    int x1 = std::min(x0 + 1, w - 1);
    int y1 = std::min(y0 + 1, h - 1);
    float sx = fx - x0, sy = fy - y0;
    float v00 = g(x0, y0), v10 = g(x1, y0), v01 = g(x0, y1), v11 = g(x1, y1);
    float a = v00 * (1 - sx) + v10 * sx;
    float b = v01 * (1 - sx) + v11 * sx;
    return a * (1 - sy) + b * sy;
}

static inline void sampleHeightAndGradient(const GridFloat &g, float fx, float fy, flo
    float eps = 1.0f;
    heightOut = sampleBilinear(g, fx, fy);
    float hx = sampleBilinear(g, fx + eps, fy);
    float lx = sampleBilinear(g, fx - eps, fy);
    float hy = sampleBilinear(g, fx, fy + eps);
    float ly = sampleBilinear(g, fx, fy - eps);
    gx = (hx - lx) * 0.5f / eps;  // dH/dx
    gy = (hy - ly) * 0.5f / eps;  // dH/dy
}

static inline void accumulateToCellQuad(vector<double> &buf, int w, int h, float fx, f
    if (amount == 0.0) return;
    if (w <= 0 || h <= 0) return;

    float fxc = fx;
    float fyc = fy;
    if (fxc < 0.0f) fxc = 0.0f;
    if (fyc < 0.0f) fyc = 0.0f;
    if (fxc > (float)(w - 1)) fxc = (float)(w - 1);
    if (fyc > (float)(h - 1)) fyc = (float)(h - 1);

    int x0 = (int)floor(fxc);
    int y0 = (int)floor(fyc);
    int x1 = x0 + 1;
    int y1 = y0 + 1;

    if (x0 < 0) x0 = 0;
    if (y0 < 0) y0 = 0;
    if (x1 >= w) x1 = w - 1;
```

```
    if (y1 >= h) y1 = h - 1;

    float sx = fxc - (float)x0;
    float sy = fyc - (float)y0;

    double w00 = (1.0 - sx) * (1.0 - sy);
    double w10 = sx * (1.0 - sy);
    double w01 = (1.0 - sx) * sy;
    double w11 = sx * sy;

    auto idx = [&](int x, int y) -> size_t { return (size_t)y * (size_t)w + (size_t)x;

    size_t nCells = (size_t)w * (size_t)h;
    if (idx(x0, y0) < nCells) buf[idx(x0, y0)] += amount * w00;
    if (idx(x1, y0) < nCells) buf[idx(x1, y0)] += amount * w10;
    if (idx(x0, y1) < nCells) buf[idx(x0, y1)] += amount * w01;
    if (idx(x1, y1) < nCells) buf[idx(x1, y1)] += amount * w11;
}

static inline float clampf(float v, float lo, float hi) { return v < lo ? lo : (v > hi
}
```

4.2.6 Custom Data Structures :

```
template <typename T>
class Grid2D {
   public:
    Grid2D() : w_(0), h_(0) {}
    Grid2D(int width, int height) : w_(width), h_(height), data_((size_t)width * (size
    Grid2D(int width, int height, const T& init) : w_(width), h_(height), data_((size_

    Grid2D(int width, int height, std::vector<T>&& flatData) : w_(width), h_(height),
        assert((size_t)w_ * (size_t)h_ == data_.size());
    }

    void resize(int width, int height) {
        w_ = width;
        h_ = height;
        data_.assign((size_t)w_ * (size_t)h_, T());
    }
    void resize(int width, int height, const T& init) {
        w_ = width;
        h_ = height;
```

```
        data_.assign((size_t)w_ * (size_t)h_, init);
    }

    int width() const { return w_; }
    int height() const { return h_; }
    size_t size() const { return data_.size(); }

    T* data() { return data_.empty() ? nullptr : data_.data(); }
    const T* data() const { return data_.empty() ? nullptr : data_.data(); }

    inline size_t index(int x, int y) const {
        assert(x >= 0 && x < w_ && y >= 0 && y < h_);
        return (size_t)y * (size_t)w_ + (size_t)x;
    }

    inline T& operator()(int x, int y) { return data_[index(x, y)]; }
    inline const T& operator()(int x, int y) const { return data_[index(x, y)]; }

    inline T& at(int x, int y) { return operator()(x, y); }
    inline const T& at(int x, int y) const { return operator()(x, y); }

    void fill(const T& v) { std::fill(data_.begin(), data_.end(), v); }

    template <typename Fn>
    void forEach(Fn&& fn) {
#pragma omp parallel for collapse(2) schedule(static)
        for (int y = 0; y < h_; ++y) {
            for (int x = 0; x < w_; ++x) {
                fn(x, y, data_[(size_t)y * (size_t)w_ + (size_t)x]);
            }
        }
    }

    std::vector<T> toVector() const { return data_; }

    static Grid2D<T> fromVector(int width, int height, const std::vector<T>& flat) {
        assert((size_t)width * (size_t)height == flat.size());
        Grid2D<T> g(width, height);
        g.data_ = flat;
        return g;
    }

  private:
    int w_ = 0, h_ = 0;
    std::vector<T> data_;
};
```

```
inline void flatIndexToXY(size_t idx, int width, int& outX, int& outY) {
    outX = (int)(idx % (size_t)width);
    outY = (int)(idx / (size_t)width);
}

template <typename T, typename Fn>
static Grid2D<T> makeGridFromFn(int width, int height, Fn&& f) {
    Grid2D<T> g(width, height);
#pragma omp parallel for collapse(2) schedule(static)
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            g(x, y) = f(x, y);
        }
    }
    return g;
}

using GridFloat = Grid2D<float>;
using GridU8 = Grid2D<uint8_t>;
using GridInt = Grid2D<int>;
```

4.3 Serial Algorithms :

4.3.1 Heightmap Generation

- Done using Voronoi Plates generation and Fractal Brownian Motion Heightmap Generation using Perlin Noise
- **Complexity:** O(W×H×P)

```
Input: Width W, Height H, VoronoiConfig cfg
Output: Grid2D<float> heightmap

for i = 0 to cfg.numPlates - 1:
    plate[i].x = random(0, W)
    plate[i].y = random(0, H)
    plate[i].height = random(-0.6, 0.6)
    plate[i].scale = random(0.5, 2.0)

for y = 0 to H - 1:
    for x = 0 to W - 1:
```

```
        minDist = infinity
        secondDist = infinity
        nearestPlate = null

        for each plate:
            dist = euclidean_distance((x,y), (plate.x, plate.y))
            if dist < minDist:
                secondDist = minDist
                minDist = dist
                nearestPlate = plate
            else if dist < secondDist:
                secondDist = dist

        normalizedDist = minDist / diagonal(W, H)
        gap = (secondDist - minDist) / diagonal(W, H)
        ridge = exp(-gap * ridgeStrength * 16)
        falloff = 1 - clamp(normalizedDist * nearestPlate.scale, 0, 1)
        voronoiHeight = nearestPlate.height * 0.8 + falloff * 0.2 + ridge * 0.6

        fbmHeight = perlin_fbm(x, y, cfg.frequency, cfg.octaves)

        height[x,y] = (1 - cfg.fbmBlend) * voronoiHeight + cfg.fbmBlend * fbmHeight
        height[x,y] = tanh(height[x,y] * 1.2)
        height[x,y] = (height[x,y] + 1) * 0.5
```

4.3.2 Hydraulic Erosion

- Particle Based Erosion model
- **Complexity:** O(D×S×R²)

```
Input: Grid2D<float> heightmap, ErosionParams params
Output: Modified heightmap, erosion/deposit maps

for each thread t:
    erosionBuffer[t] = zeros(W × H)
    depositBuffer[t] = zeros(W × H)

for d = 0 to params.numDroplets - 1:
    x = random(0, W-1)
    y = random(0, H-1)
    dirX = 0, dirY = 0
    speed = params.initSpeed
    water = params.initWater
```

```
    sediment = 0

    for step = 0 to params.maxSteps - 1:
        h = bilinear_sample(heightmap, x, y)
        gradX, gradY = compute_gradient(heightmap, x, y)

        dirX = dirX * params.inertia - gradX * (1 - params.inertia)
        dirY = dirY * params.inertia - gradY * (1 - params.inertia)
        normalize(dirX, dirY)

        x += dirX * params.stepSize
        y += dirY * params.stepSize

        if out_of_bounds(x, y):
            break

        newH = bilinear_sample(heightmap, x, y)
        deltaH = newH - h
        slope = max(1e-6, -deltaH / params.stepSize)

        speed = sqrt(max(0, speed² - deltaH * params.gravity))

        capacity = max(0, params.capacityFactor * speed * water * slope)

        if sediment > capacity:
            deposit = params.depositRate * (sediment - capacity)
            accumulate_bilinear(depositBuffer, x, y, deposit)
            sediment -= deposit
        else:
            erode = params.erodeRate * (capacity - sediment)
            erode = min(erode, params.maxErodePerStep)
            accumulate_bilinear(erosionBuffer, x, y, erode)
            sediment += erode

        water *= (1 - params.evaporateRate)

        if water < params.minWater or speed < params.minSpeed:
            break

for y = 0 to H - 1:
    for x = 0 to W - 1:
        delta = depositBuffer[x,y] - erosionBuffer[x,y]
        heightmap[x,y] = max(0, heightmap[x,y] + delta)
```

4.3.3 River Generation

- Flow Accumulation and carving based river gen
- **Complexity:** O(W×H×log(W×H)) for sorting, O(W×H) for accumulation and carving

```
Input: Grid2D<float> heightmap, RiverParams params
Output: River mask, carved heightmap

for y = 0 to H - 1:
    for x = 0 to W - 1:
        h = heightmap[x,y]
        bestNeighbor = null
        maxDrop = 0

        for each of 8 neighbors (nx, ny):
            nh = heightmap[nx, ny]
            dist = (diagonal neighbor) ? sqrt(2) : 1
            drop = (h - nh) / dist

            if drop > maxDrop:
                maxDrop = drop
                bestNeighbor = (nx, ny)

        flowDirection[x,y] = bestNeighbor

sortedCells = topological_sort_by_elevation(heightmap)

flowAccumulation = ones(W × H)

for each cell in sortedCells:
    neighbor = flowDirection[cell]
    if neighbor is not null:
        flowAccumulation[neighbor] += flowAccumulation[cell]

for y = 0 to H - 1:
    for x = 0 to W - 1:
        if flowAccumulation[x,y] >= params.threshold:
            riverMask[x,y] = 255
        else:
            riverMask[x,y] = 0

distanceToRiver = compute_distance_bfs(riverMask)

for y = 0 to H - 1:
    for x = 0 to W - 1:
```

```
        if distanceToRiver[x,y] < infinity:
            flow = flowAccumulation[x,y]
            width = params.widthMultiplier * sqrt(flow)
            depth = clamp(params.minDepth + log(flow) * scaleFactor,
                          params.minDepth, params.maxDepth)

            radius = max(1, width)
            falloff = max(0, 1 - distanceToRiver[x,y] / (radius * 1.5))

            delta = depth * falloff
            heightmap[x,y] = heightmap[x,y] - delta
```

4.3.4 Biome Classification

- Multi factor and parameterized Biome scoring based of information generated by the Terrain maps (including the water masks, river masks, erosion deposition, erosion sedimentation)
- **Complexity:** O(W×H×B×I) `B : Biomes, I : Iterations for Smoothing`

```
Input: heightmap, temperature, moisture, BiomeDef[], options
Output: Grid2D<Biome> biomeMap

oceanMask = (heightmap < options.oceanThreshold)
lakeMask = (heightmap < options.lakeThreshold)
coastDistance = bfs_distance(oceanMask)
slopeMap = compute_slope(heightmap)

for y = 0 to H - 1:
    for x = 0 to W - 1:
        e = heightmap[x,y]
        t = temperature[x,y]
        m = moisture[x,y]
        s = slopeMap[x,y]
        nearCoast = (coastDistance[x,y] <= options.coastThreshold)

        bestBiome = null
        bestScore = -infinity

        for each biomeDef:
            elevScore = gaussian_score(e, biomeDef.prefElevation)
            tempScore = gaussian_score(t, biomeDef.prefTemperature)
            moistScore = gaussian_score(m, biomeDef.prefMoisture)
            slopeScore = gaussian_score(s, biomeDef.prefSlope)
```

```
            if biomeDef.requiresWater and not nearCoast:
                continue

            score = (biomeDef.weightElev * elevScore + biomeDef.weightTemp * tempScore

            if biomeDef.prefersCoast and nearCoast:
                score *= 1.5

            if score > bestScore:
                bestScore = score
                bestBiome = biomeDef.id

        biomeMap[x,y] = bestBiome

for iteration = 0 to options.smoothingIterations - 1:
    tempMap = biomeMap
    for y = 0 to H - 1:
        for x = 0 to W - 1:
            counts = histogram_3x3(biomeMap, x, y)
            tempMap[x,y] = argmax(counts)
    biomeMap = tempMap
```

4.4 Parallel Algorithms :

4.4.1 Heightmap Generation

- Plate Initialization :
    - the indices are independent and can be instantiated parallely

```
Input: VoronoiConfig cfg, int width, int height
Output: Array of VoronoiPlate

plates = empty array of size cfg.numPlates
s = cfg.seed

parallel for i = 0 to cfg.numPlates - 1:
    rng = new RNG(s + i)
    plate = new VoronoiPlate
    plate.id = i
    plate.seed = rng.nextInt()
    plate.x = rng.nextFloat() × width
```

```
    plate.y = rng.nextFloat() × height
    plate.height = (rng.nextFloat() × 2.0 - 1.0) × 0.6
    plate.scale = 0.5 + rng.nextFloat() × 1.5
    plates[i] = plate

return plates
```

- Heightmap generation :

```
Input: Grid2D voronoi, Grid2D fbmNoise, VoronoiConfig cfg
Output: Grid2D heightmap

parallel for y = 0 to H - 1:
    for x = 0 to W - 1:
        vor = voronoiHeightAt(x, y)
        fbm = fbmNoiseAt(x, y)
        h = (1.0 - cfg.fbmBlend) × vor + cfg.fbmBlend × fbm
        h = tanh(h × 1.2)
        heightmap[x, y] = (h + 1.0) × 0.5
```

- No data races due to coordinates being independent
- RNG is thread safe by giving per coord seeding

4.4.2 Hydraulic Erosion

- Each particle can be simulated independently because it does not interfere with the other particles
- Each Particle has its own thread and its own thread buffer due to a race condition :
    - Each thread can access the shared heightmap for erosion and sedimentation race conditions and data inconsistency issues
- Each particle has thread safe RNG calculations
- Final reduction combines all the threads to the final heightmap post Erosion

```
Input: Grid2D<float> heightGrid, ErosionParams params
Output: ErosionStats, Grid2D eroded, Grid2D deposited

W = heightGrid.width
H = heightGrid.height
N = params.numDroplets
numThreads = available_thread_count()
```

```
// Initialize per-thread accumulation buffers
for t = 0 to numThreads - 1:
    erodeBufs[t] = zeros(W × H)
    depositBufs[t] = zeros(W × H)

// Simulate droplets in parallel
parallel for di = 0 to N - 1:
    tid = current_thread_id()
    rng = new RNG(params.worldSeed XOR di × 2654435761)

    // Initialize droplet
    x = rng.nextFloat() × (W - 1)
    y = rng.nextFloat() × (H - 1)
    dirX = 0, dirY = 0
    speed = params.initSpeed
    water = params.initWater
    sediment = 0

    for step = 0 to params.maxSteps - 1:
        heightHere, gradX, gradY = sampleHeightAndGradient(heightGrid, x, y)

        // Update direction with inertia
        dirX = dirX × params.inertia - gradX × (1 - params.inertia)
        dirY = dirY × params.inertia - gradY × (1 - params.inertia)
        len = sqrt(dirX² + dirY²)

        if len == 0:
            theta = rng.nextFloat() × 2π
            dirX = cos(theta) × 1e-6
            dirY = sin(theta) × 1e-6
            len = sqrt(dirX² + dirY²)

        dirX = dirX / len
        dirY = dirY / len

        // Move droplet
        x = x + dirX × params.stepSize
        y = y + dirY × params.stepSize

        if x < 0 or x > W - 1 or y < 0 or y > H - 1:
            break

        newHeight = sampleBilinear(heightGrid, x, y)
        deltaH = newHeight - heightHere
        potential = -deltaH
        speed = sqrt(max(0, speed² + potential × params.gravity))
```

```
        slope = max(1e-6, -deltaH / params.stepSize)
        capacity = max(0, params.capacityFactor × speed × water × slope)

        // Erosion or deposition
        if sediment > capacity:
            deposit = params.depositRate × (sediment - capacity)
            deposit = min(deposit, sediment)
            accumulateToCellQuad(depositBufs[tid], W, H, x, y, deposit)
            sediment = sediment - deposit
        else:
            delta = params.capacityFactor × (capacity - sediment)
            erode = params.erodeRate × delta
            erode = min(erode, params.maxErodePerStep)
            erode = min(erode, max(0, newHeight))
            if erode > 0:
                accumulateToCellQuad(erodeBufs[tid], W, H, x, y, erode)
                sediment = sediment + erode

        // Evaporation
        water = water × (1 - params.evaporateRate)
        if water < params.minWater or speed < params.minSpeed:
            break

// Merge thread buffers
finalErode = zeros(W × H)
finalDeposit = zeros(W × H)

parallel for t = 0 to numThreads - 1:
    for i = 0 to W × H - 1:
        finalErode[i] += erodeBufs[t][i]
        finalDeposit[i] += depositBufs[t][i]

// Apply erosion and deposition
stats.totalEroded = 0
stats.totalDeposited = 0

parallel for y = 0 to H - 1:
    for x = 0 to W - 1:
        idx = y × W + x
        delta = finalDeposit[idx] - finalErode[idx]
        stats.totalEroded += finalErode[idx]
        stats.totalDeposited += finalDeposit[idx]
        newH = heightGrid[x, y] + delta
        heightGrid[x, y] = max(0, newH)

if eroded is requested:
```

```
    parallel for y = 0 to H - 1:
        for x = 0 to W - 1:
            eroded[x, y] = finalErode[y × W + x]

if deposited is requested:
    parallel for y = 0 to H - 1:
        for x = 0 to W - 1:
            deposited[x, y] = finalDeposit[y × W + x]

stats.appliedDroplets = N
return stats
```

4.4.3 River Generation

- This is the following parts which are to be parallelized :
  - Flow direction: parallel since its independent per grid cell
  - Flow accumulation: sequential due to chain of dependencies
  - River extraction: can be parallelized since its independent per cell
  - River carving: parallel BFS + sequential BFS + parallel carving

```
Input: Grid2D<float> heightmap, Grid2D<int> biomeMap, RiverParams params
Output: River mask, carved heightmap

W = heightmap.width
H = heightmap.height
flowDirection = array of size W × H, initialized to -1
flowAccumulation = array of size W × H, initialized to 0
riverMask = array of size W × H

// Compute flow direction
dx = [1, 1, 0, -1, -1, -1, 0, 1]
dy = [0, 1, 1, 1, 0, -1, -1, -1]
diagDist = sqrt(2)

parallel for y = 0 to H - 1:
    for x = 0 to W - 1:
        i = y × W + x
        h = heightmap[x, y]
        bestNeighbor = -1
        maxDrop = 0

        for k = 0 to 7:
```

```
            nx = x + dx[k]
            ny = y + dy[k]
            if nx < 0 or nx >= W or ny < 0 or ny >= H:
                continue

            ni = ny × W + nx
            nh = heightmap[nx, ny]
            dist = (k % 2 == 0) ? 1.0 : diagDist
            drop = (h - nh) / dist

            if drop > maxDrop:
                maxDrop = drop
                bestNeighbor = ni

        flowDirection[i] = bestNeighbor

// Compute flow accumulation
N = W × H
order = [0, 1, 2, ..., N - 1]
sort order by heightmap values in descending order

flowAccumulation = ones(N)

for id = 0 to N - 1:
    i = order[id]
    neighbor = flowDirection[i]
    if neighbor != -1:
        flowAccumulation[neighbor] += flowAccumulation[i]

// Extract rivers
parallel for i = 0 to N - 1:
    if flowAccumulation[i] >= params.flow_accum_threshold:
        riverMask[i] = 255
    else:
        riverMask[i] = 0

// Carve rivers using distance field
distanceToRiver = array of size N, initialized to infinity
queue = empty queue

parallel for i = 0 to N - 1:
    if riverMask[i] == 255:
        distanceToRiver[i] = 0
        queue.push(i)

// BFS to compute distance field
```

```
while queue is not empty:
    cur = queue.pop()
    cx = cur % W
    cy = cur / W

    for each of 4 neighbors (nx, ny):
        if nx < 0 or nx >= W or ny < 0 or ny >= H:
            continue

        ni = ny × W + nx
        if distanceToRiver[ni] > distanceToRiver[cur] + 1:
            distanceToRiver[ni] = distanceToRiver[cur] + 1
            queue.push(ni)

// Carve terrain based on distance
parallel for i = 0 to N - 1:
    if distanceToRiver[i] == infinity:
        continue

    flow = flowAccumulation[i]
    width = params.width_multiplier × sqrt(max(1, flow))
    depth = clamp(
        params.min_channel_depth + (params.max_channel_depth - params.min_channel_dept
        params.min_channel_depth,
        params.max_channel_depth
    )

    falloff = 1.0
    if distanceToRiver[i] > 0:
        d = distanceToRiver[i]
        radius = max(1, width)
        falloff = max(0, 1 - d / (radius × 1.5))

    delta = depth × falloff
    heightmap[i] = heightmap[i] - delta
```

4.4.4 Biome Classification and Object Placement

- Distance is computed using a Multi Level BFS which is parallelized
- The mask generation, biome scoring and the majority filtering all can be parallelized

```
Input: Grid2D<float> heightGrid, Grid2D<float> tempGrid, Grid2D<float> moistGrid,
       Grid2D<int> riverMaskGrid, Array<BiomeDef> defs, ClassifierOptions opts
```

```
Output: Grid2D<Biome> biomeGrid

W = heightGrid.width
H = heightGrid.height

// Identify water bodies
oceanMask = zeros(W × H)
lakeMask = zeros(W × H)

parallel for y = 0 to H - 1:
    for x = 0 to W - 1:
        e = heightGrid[x, y]
        idx = y × W + x
        if e < opts.oceanHeightThreshold:
            oceanMask[idx] = 1
        else if e < opts.lakeHeightThreshold:
            lakeMask[idx] = 1

// Extract river mask if provided
riverMask = zeros(W × H)
if riverMaskGrid is not null:
    parallel for y = 0 to H - 1:
        for x = 0 to W - 1:
            riverMask[y × W + x] = riverMaskGrid[x, y] ? 1 : 0

// Compute proximity to coast and rivers
nearCoast = zeros(W × H)
nearRiver = zeros(W × H)
computeNearMaskFromSources(W, H, oceanMask, opts.coastDistanceTiles, nearCoast)
if riverMask is not empty:
    computeNearMaskFromSources(W, H, riverMask, opts.riverDistanceTiles, nearRiver)

// Compute slope map
slopeMap = zeros(W × H)
computeSlopeMap(W, H, heightGrid, slopeMap, opts.expectedMaxGradient)

// Classify biomes
chosen = array of size W × H

parallel for y = 0 to H - 1:
    for x = 0 to W - 1:
        idx = y × W + x
        e = heightGrid[x, y]
        t = tempGrid[x, y]
        m = moistGrid[x, y]
        s = slopeMap[idx]
```

```
        nc = (nearCoast[idx] != 0)
        nr = (nearRiver[idx] != 0) or (riverMask[idx] != 0)
        b = chooseBestBiome(defs, e, t, m, s, nc, nr, opts)
        chosen[idx] = b

// Apply smoothing filter
if opts.smoothingIterations > 0:
    majorityFilter(W, H, chosen, opts.smoothingIterations)

// Write to output grid
parallel for y = 0 to H - 1:
    for x = 0 to W - 1:
        biomeGrid[x, y] = chosen[y × W + x]

return biomeGrid
```

- Following this the object placement is run and also parallelized as follows :
  - Due to irregular loads (some regions having high density of item placements), we can use dynamic scheduling
  - An atomic count for making sure, no items can cross the limit and should be thread safe
  - Critical has to be used inorder to make sure that the object placement grid doesn't enter a race condition due to multiple objects being placed in the same cell

4.5 Implementation Details :

4.5.1 Dev Environment

- **Processor:** Intel Core i7-12700H - 14 cores
- **Memory:** 16 GB DDR4
- **Storage:** NVMe SSD
- **Operating System:** Windows 11
- **Compiler:** GCC 15.2.0 with OpenMP 6
- **Build System:** CMake 4.4.1
- **Language:** C++17
- **Libraries:**
  - nlohmann/json for configuration parsing

4.5.2 Compilation and Build

- CMake Configuration

```
cmake_minimum_required(VERSION 3.15)
project(terrain-gen VERSION 0.1 LANGUAGES CXX)

option(BUILD_TESTS "Build unit tests" ON)
option(ENABLE_OPENMP "Link OpenMP if available" ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

if(NOT CMAKE_BUILD_TYPE)
  set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
endif()

if(CMAKE_CXX_COMPILER_ID MATCHES "GNU|Clang")
  add_compile_options(-O3 -march=native)
elseif(MSVC)
  add_compile_options(/O2)
endif()

set(DEPS_DIR ${CMAKE_SOURCE_DIR}/deps)

include_directories(${DEPS_DIR} ${CMAKE_SOURCE_DIR}/src)

file(GLOB_RECURSE SRC_FILES
  ${CMAKE_SOURCE_DIR}/src/*.cpp
  ${CMAKE_SOURCE_DIR}/src/*.c
  ${CMAKE_SOURCE_DIR}/src/*.h
  ${CMAKE_SOURCE_DIR}/src/*.hpp
)

list(LENGTH SRC_FILES SRC_COUNT)
message(STATUS "Found ${SRC_COUNT} source files")

add_executable(terrain-gen ${SRC_FILES})
target_include_directories(terrain-gen PUBLIC ${DEPS_DIR} ${CMAKE_SOURCE_DIR}/src ${CM

if(ENABLE_OPENMP)
  find_package(OpenMP)
  if(OpenMP_CXX_FOUND)
    message(STATUS "OpenMP found, enabling")
```

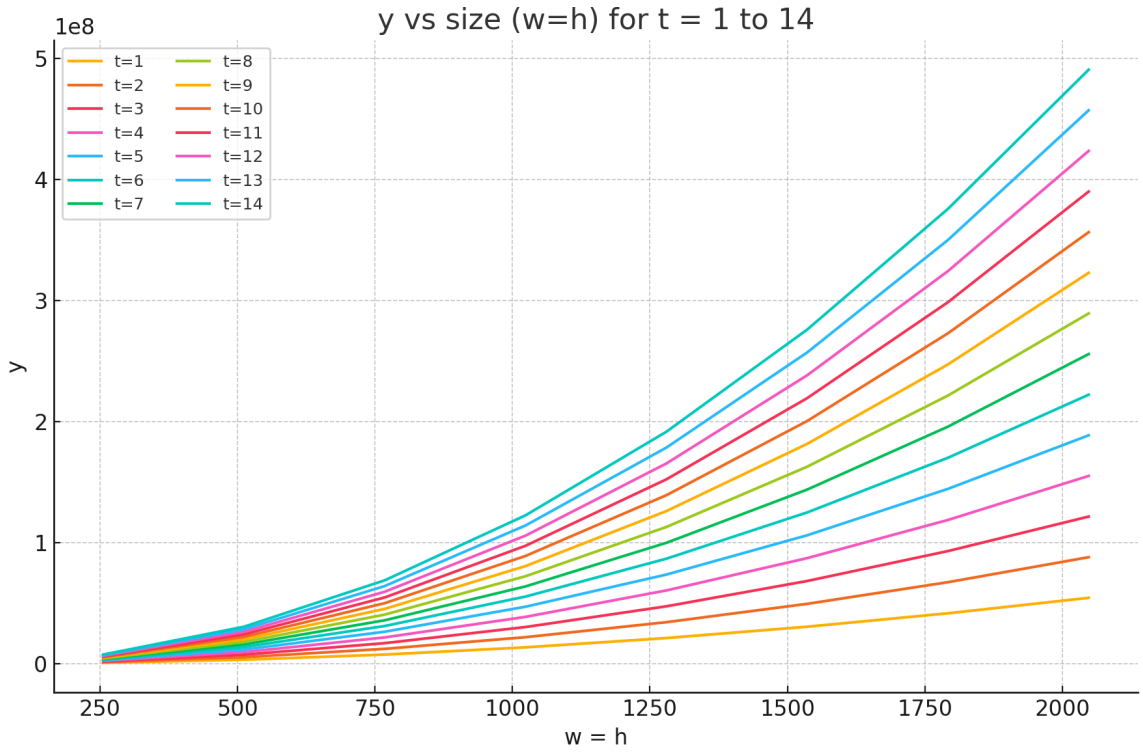
```
    target_link_libraries(terrain-gen PUBLIC OpenMP::OpenMP_CXX)
  else()
    message(STATUS "OpenMP not found — continuing without it")
  endif()
endif()


set_target_properties(terrain-gen PROPERTIES
  RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin
)

install(TARGETS terrain-gen
  RUNTIME DESTINATION bin
)

message(STATUS "Project: ${PROJECT_NAME} v${PROJECT_VERSION}")
message(STATUS "Build type: ${CMAKE_BUILD_TYPE}")
message(STATUS "Source dir: ${CMAKE_SOURCE_DIR}")
message(STATUS "Binary dir: ${CMAKE_BINARY_DIR}")
```

- **Build Commands:**

```
cd .\build\
cmake ..
cmake --build . --config Release
cd .\bin\
.\terrain-gen.exe
```

- **Memory Layout Optimizations**
  - Due to arrays and vectors being row major by default (elements in each row are contiguous) the code has been written so that they are traversed in the same manner to prevent cache misses
  - Resource Acquisition is Initialization principle has been followed so that there is no hidden memory leaks that occur, due to an object going out of scope
- **Memory Usage Analysis**
  - For a W×H terrain:
    - Heightmap: W×H×4 bytes
    - Temperature: W×H×4 bytes
    - Moisture: W×H×4 bytes
    - Biome map: W×H×1 byte
    - River mask: W×H×1 byte

    - Erosion buffers (per thread): W×H×8 bytes×T
- So total memory used with T threads :
    - Base grids : `15 x W x H`
    - Erosion Buffers : `8 * W * H * T`
- The plot looks like :



- **Memory table for each t for different map sizes in (bytes)**
    - This is the theoretical maximum memory it would take without any optimizations

t	256	512	1024	2048
1	851,968	3,407,872	13,631,488	54,525,952
2	1,376,256	5,505,024	22,020,096	88,080,384
3	1,900,544	7,602,176	30,408,704	121,634,816
4	2,424,832	9,699,328	38,797,312	155,189,248
5	2,949,120	11,796,480	47,185,920	188,743,680
6	3,473,408	13,893,632	55,574,528	222,298,112
7	3,997,696	15,990,784	63,963,136	255,852,544
8	4,521,984	18,087,936	72,351,744	289,406,976
9	5,046,272	20,185,088	80,740,352	322,961,408
10	5,570,560	22,282,240	89,128,960	356,515,840
11	6,094,848	24,379,392	97,517,568	390,070,272
12	6,619,136	26,476,544	105,906,176	423,624,704
13	7,143,424	28,573,696	114,294,784	457,179,136
14	7,667,712	30,670,848	122,683,392	490,733,568

5. RESULTS AND ANALYSIS :

- Each testcase is run 5 times in order to ensure average and fair time calculation
- Cache is cleared, and the whole program is rebuilt before an new testcase is run
- No overclocking is performed on the CPU for additional resource
- The testcases ran are are of the following configs :

```
{
    "seed": 424242,
    "width": ,
    "height": ,
    "worldType": "Voronoi",
    "numPlates": 36,
    "fbmBlend": 0.42,
    "fbmFrequency": 0.0035,
    "fbmOctaves": 5,
    "oceanHeightThreshold": 0.35,
    "lakeHeightThreshold": 0.45,
    "coastDistanceTiles": 3,
```

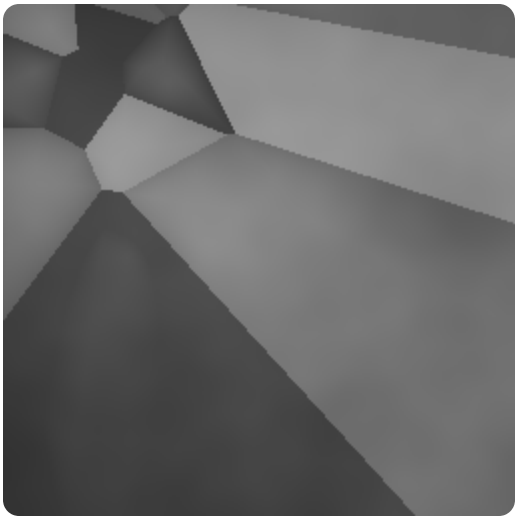
```
    "smoothingIterations": 1
}
```

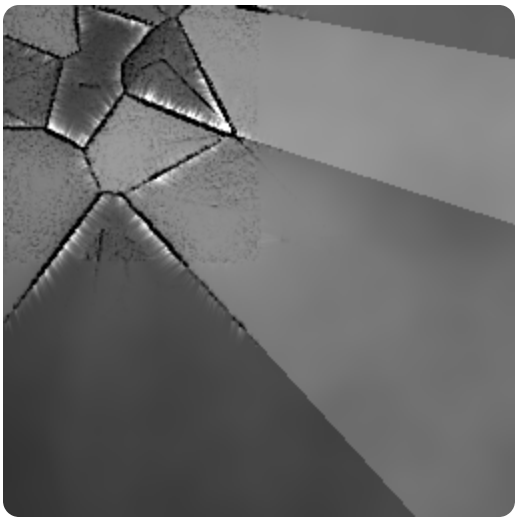
- The outputs are all shown here as PNGs in the following order :
  - Height Map Before Erosion
  - Height Map After Erosion
  - Height Map After Rivers
  - Biome Map Before Erosion
  - Biome Map After Erosion
  - Biome Map After Rivers
  - Erosion Map Eroded
  - Erosion Map Deposited
  - River Map
  - Final Height Map
  - Final Biome Map

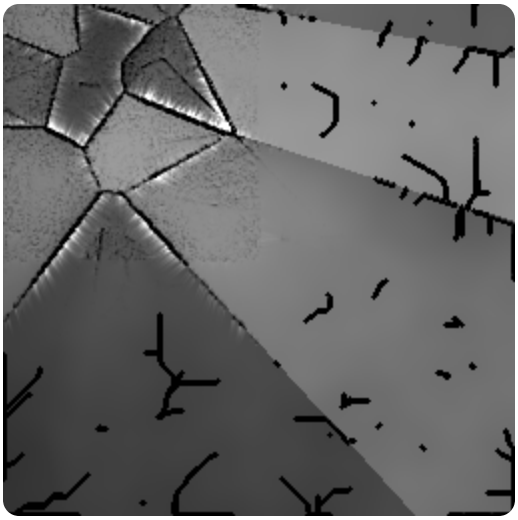
5.1 Testcase 1 :

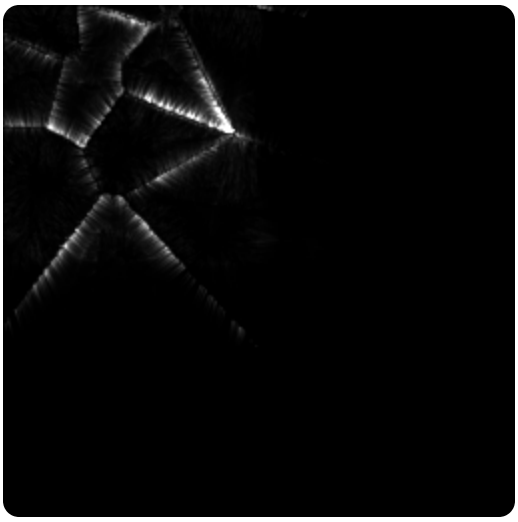
- **Width = Height = 256**

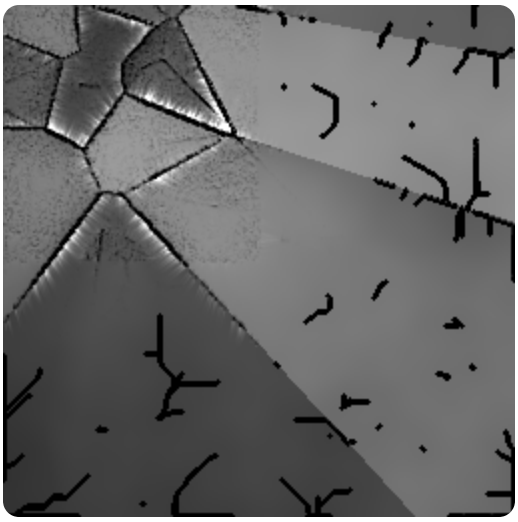
Output maps :











Thread count : 1


```json
{
  "peak_memory_kb": 4048,
  "resolution": "256x256",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 0.026100666666666664,
    "heightmap_and_voronoi": 0.012363333333333332,
    "hydraulic_erosion": 0.630717,
    "object_placement": 0.0029516666666666667,
    "river_generation": 0.006030333333333333
  },
  "threads": 1,
  "total_time": 0.6785156666666666,
  "wall_clock_time": 0.726651112238566,
  "num_runs": 3,
  "std_dev_total_time": 0.0020104652032137253
}
```


Thread count : 4


```json
{
  "peak_memory_kb": 4478.666666666667,
  "resolution": "256x256",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 0.020607,
```

```
    "heightmap_and_voronoi": 0.005234333333333334,
    "hydraulic_erosion": 0.735151,
    "object_placement": 0.0036566666666666666,
    "river_generation": 0.005696
  },
  "threads": 4,
  "total_time": 0.770688,
  "wall_clock_time": 0.8200670083363851,
  "num_runs": 3,
  "std_dev_total_time": 0.004538529828039015
}
```


Thread count : 8

```JSON
{
  "peak_memory_kb": 4785.333333333333,
  "resolution": "256x256",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 0.023074,
    "heightmap_and_voronoi": 0.0052190000000000005,
    "hydraulic_erosion": 0.939851,
    "object_placement": 0.003567666666666667,
    "river_generation": 0.0065056666666666665
  },
  "threads": 8,
  "total_time": 0.9785706666666667,
  "wall_clock_time": 1.0309378306070964,
  "num_runs": 3,
  "std_dev_total_time": 0.015394232697128286
}
```


Thread count : 20

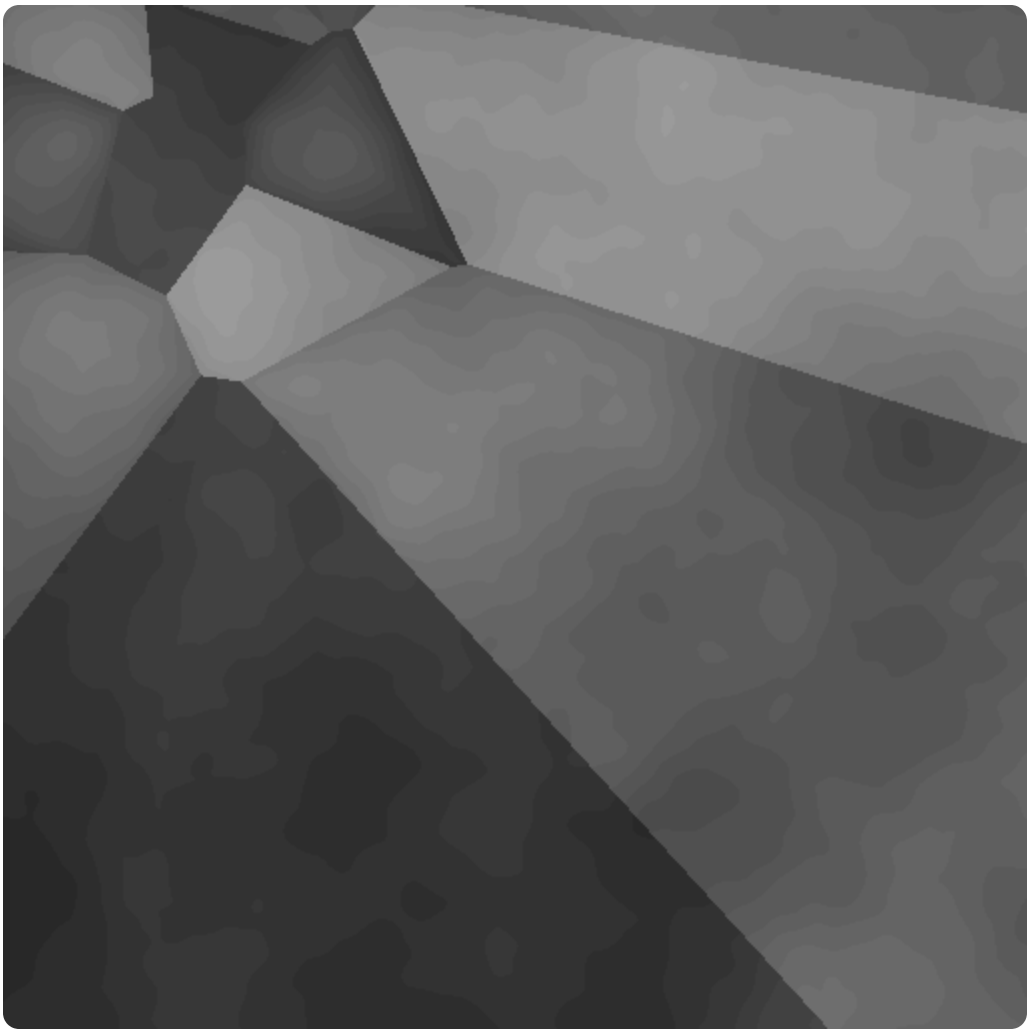
```JSON
{
  "peak_memory_kb": 5218.666666666667,
  "resolution": "256x256",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 0.029571666666666666,
```

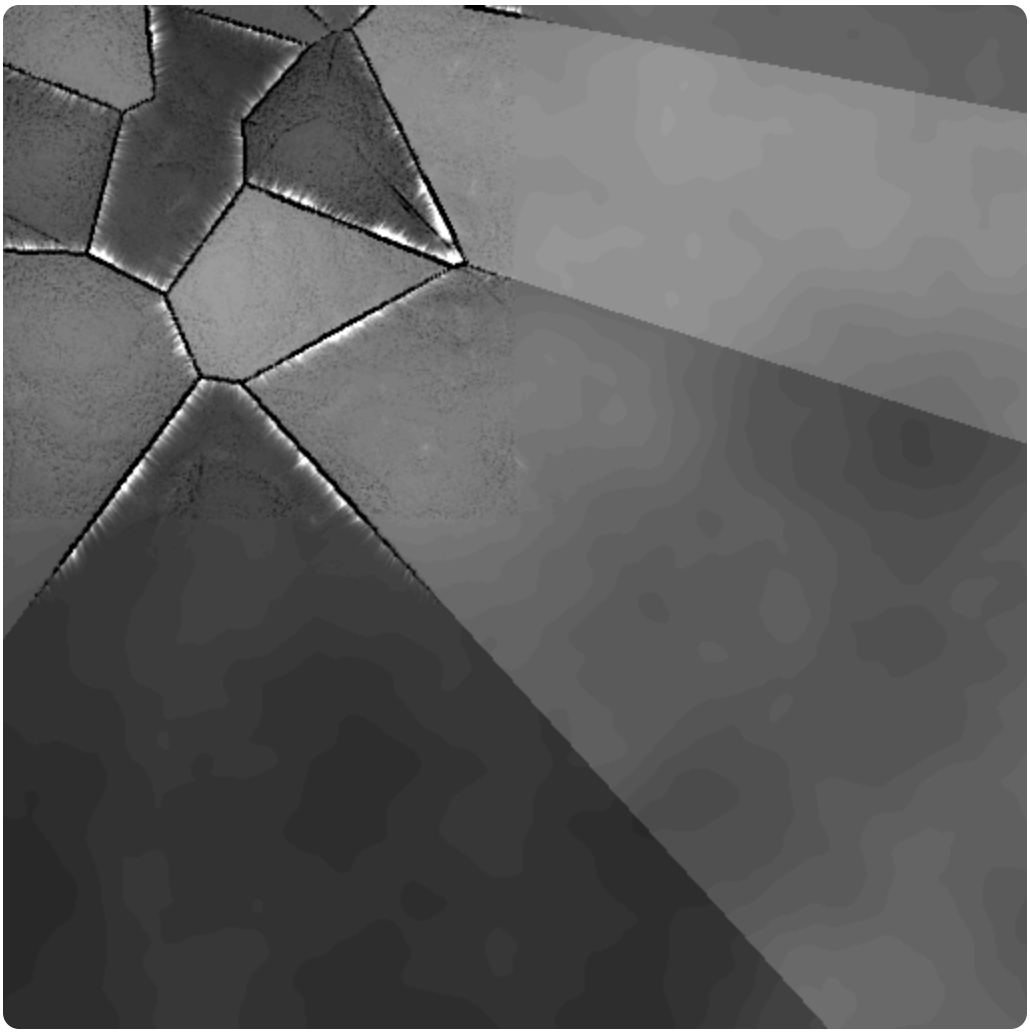
```
    "heightmap_and_voronoi": 0.006066666666666666,
    "hydraulic_erosion": 1.5538103333333333,
    "object_placement": 0.0040550000000000004,
    "river_generation": 0.005484333333333333
  },
  "threads": 20,
  "total_time": 1.5993026666666668,
  "wall_clock_time": 1.6501449743906658,
  "num_runs": 3,
  "std_dev_total_time": 0.026683693191410566
}
```

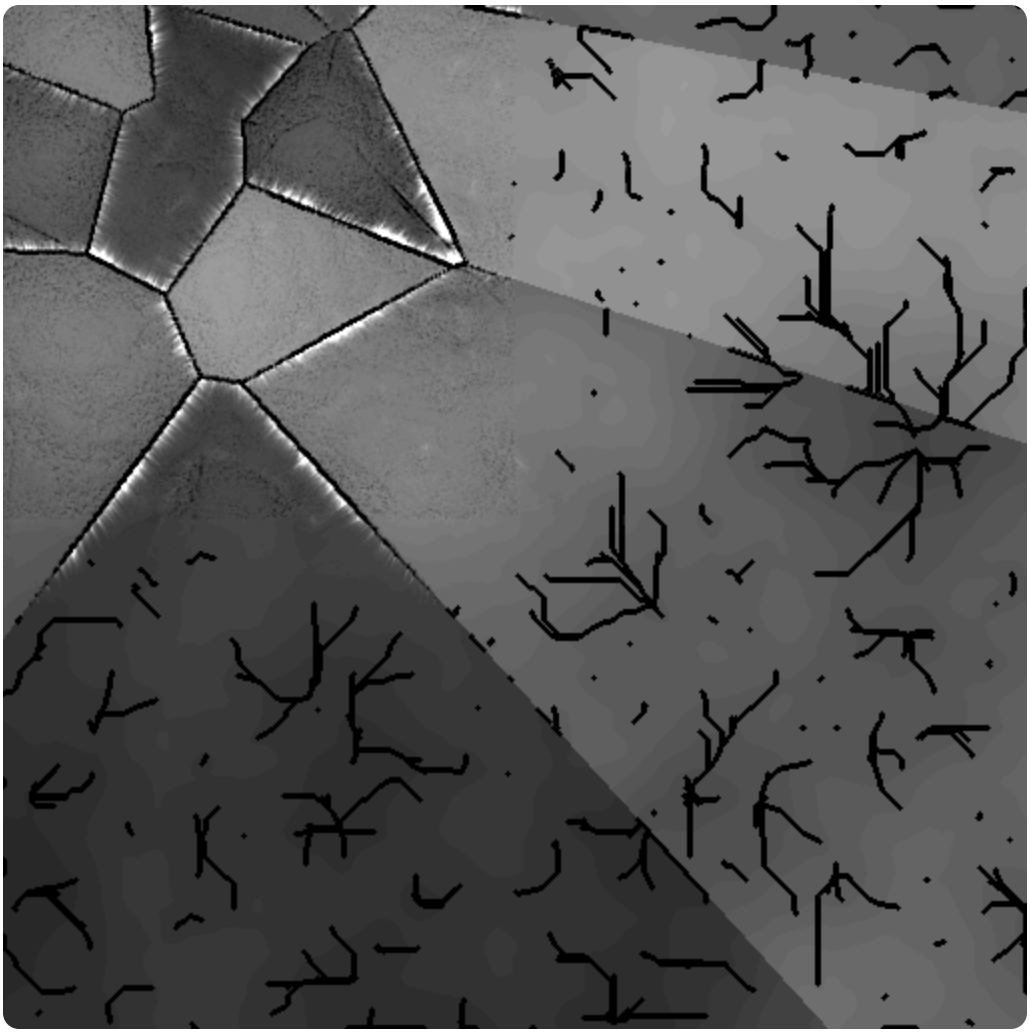

5.2 Testcase 2 :

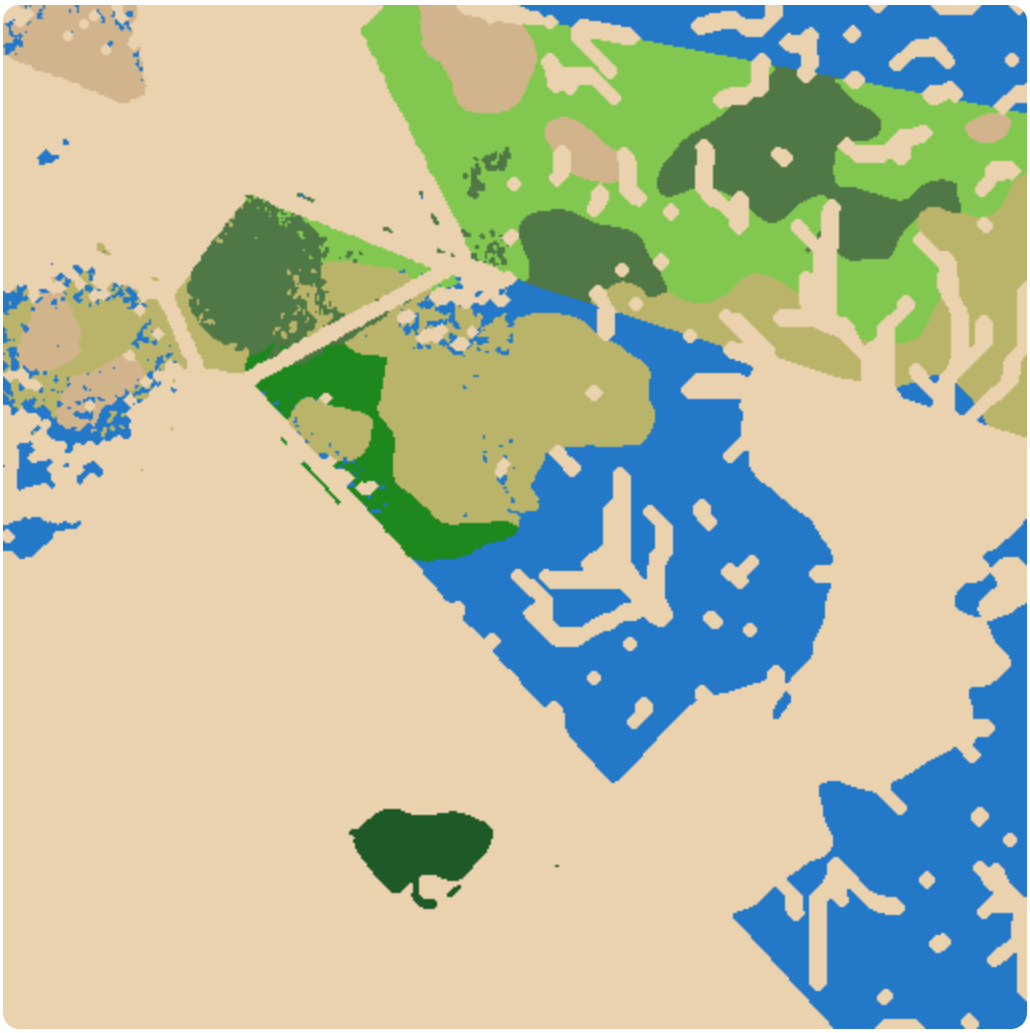
- Width = Height = 512

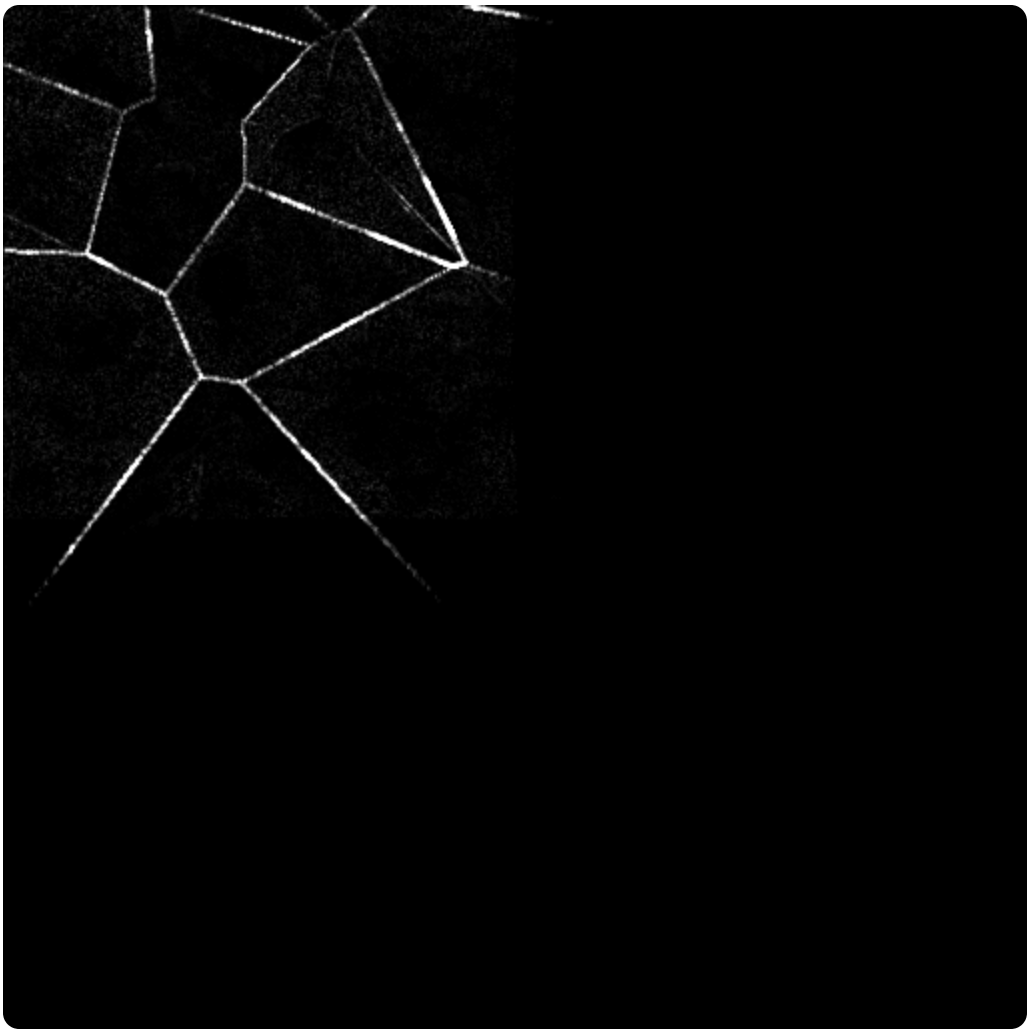
Output maps :

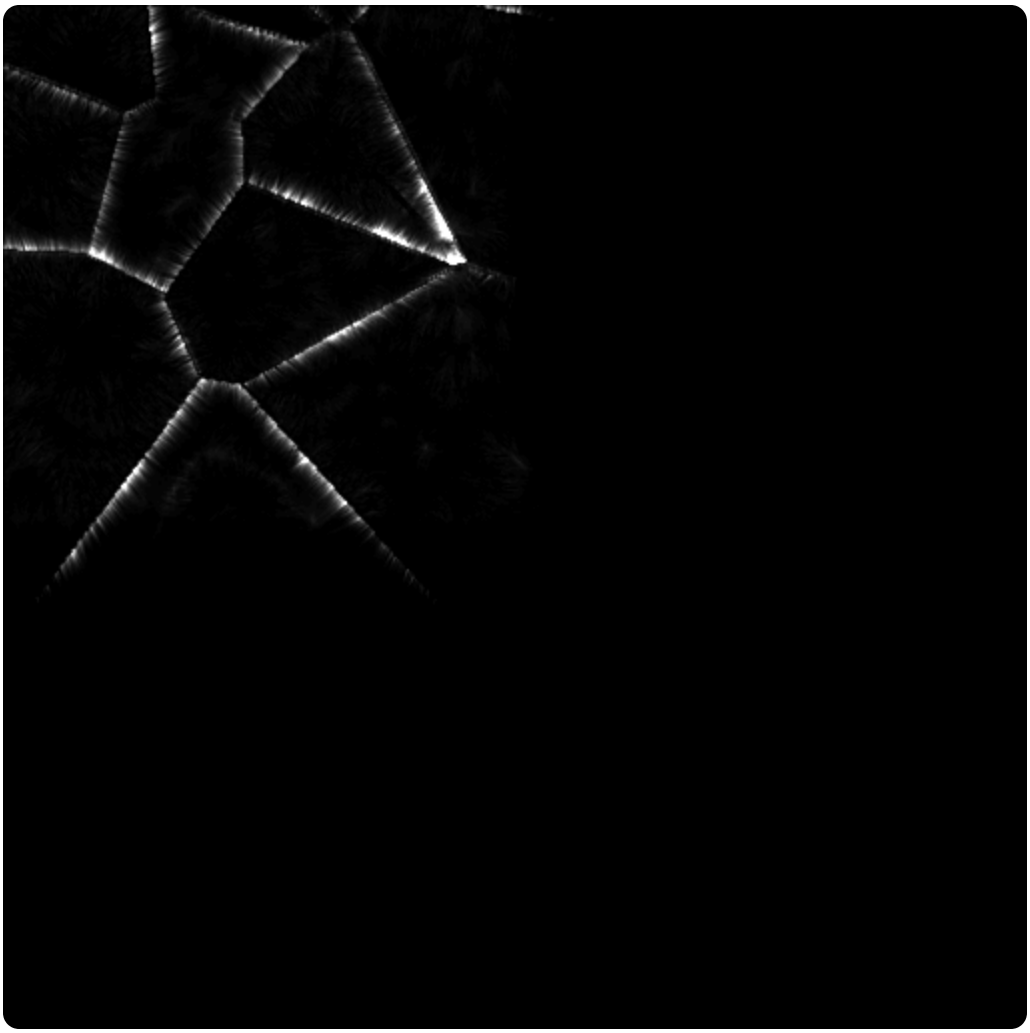


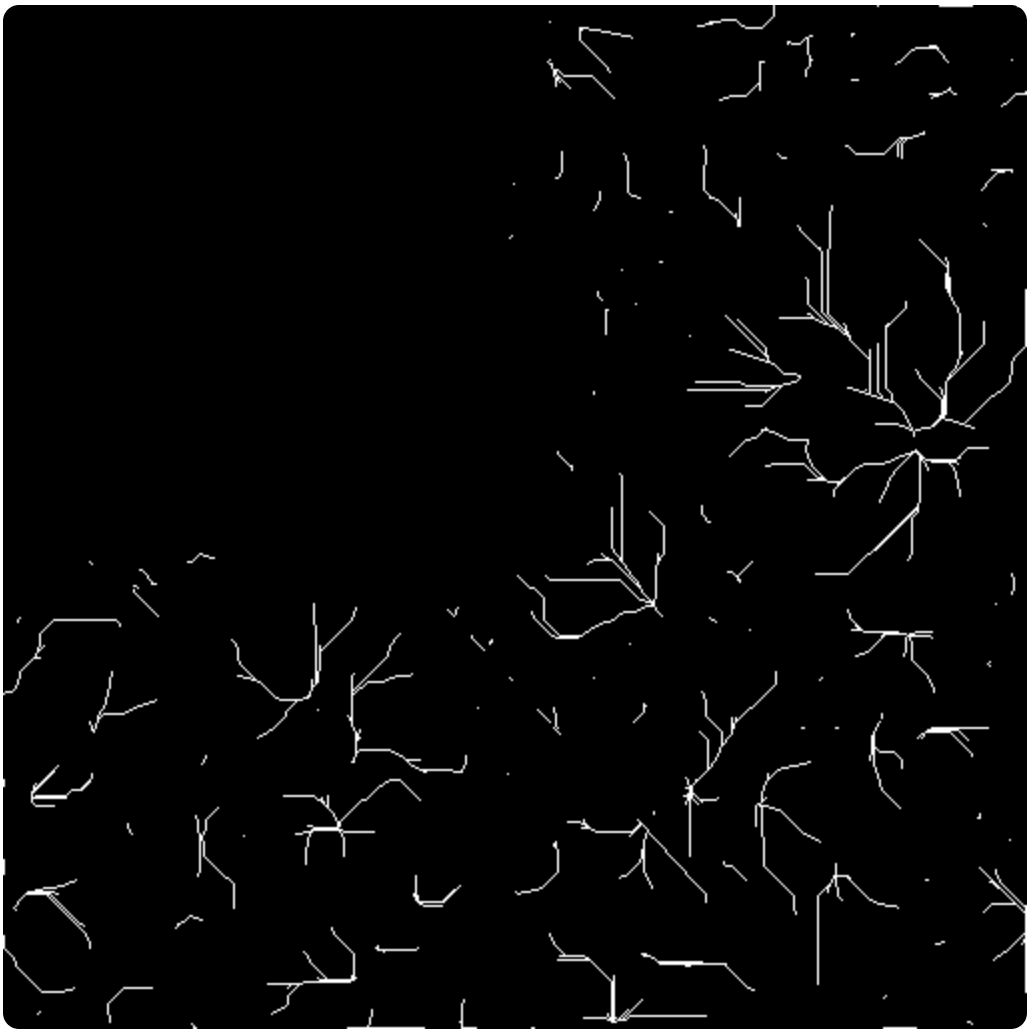


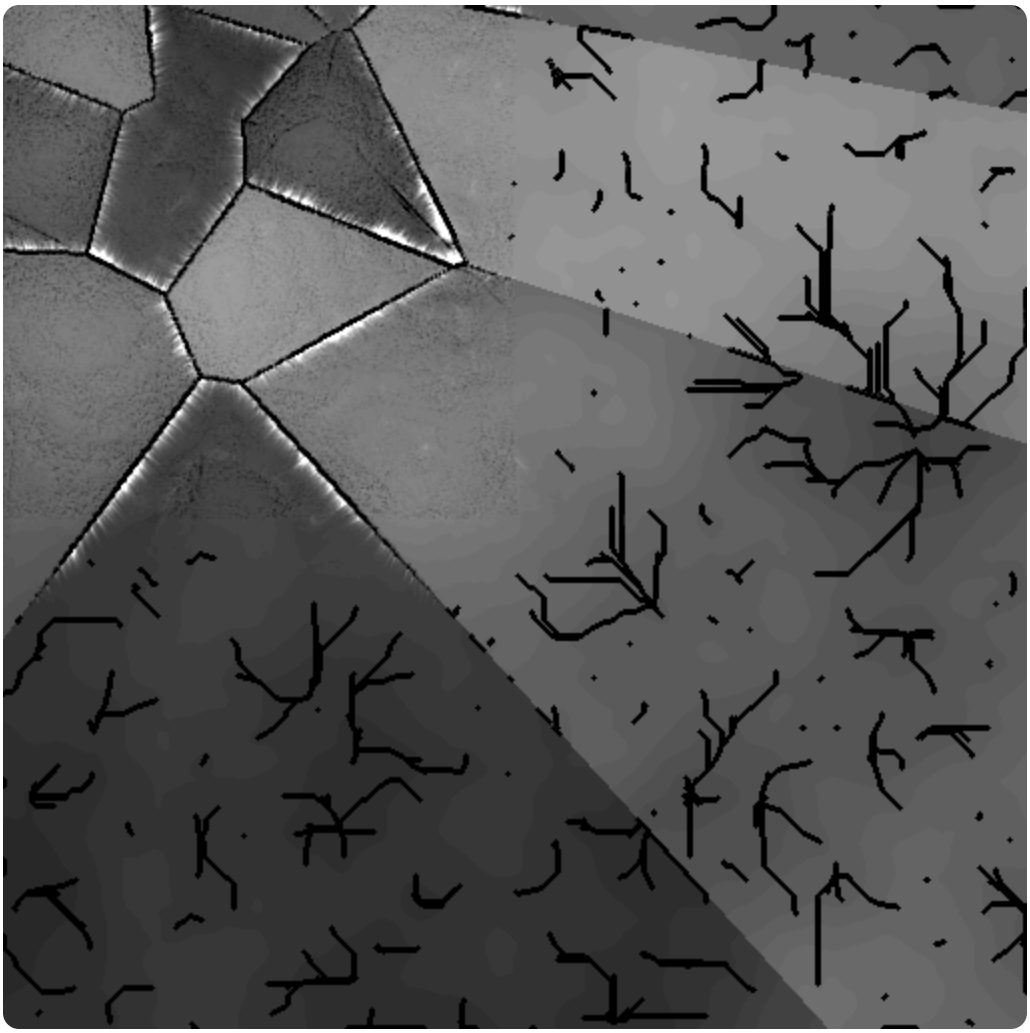


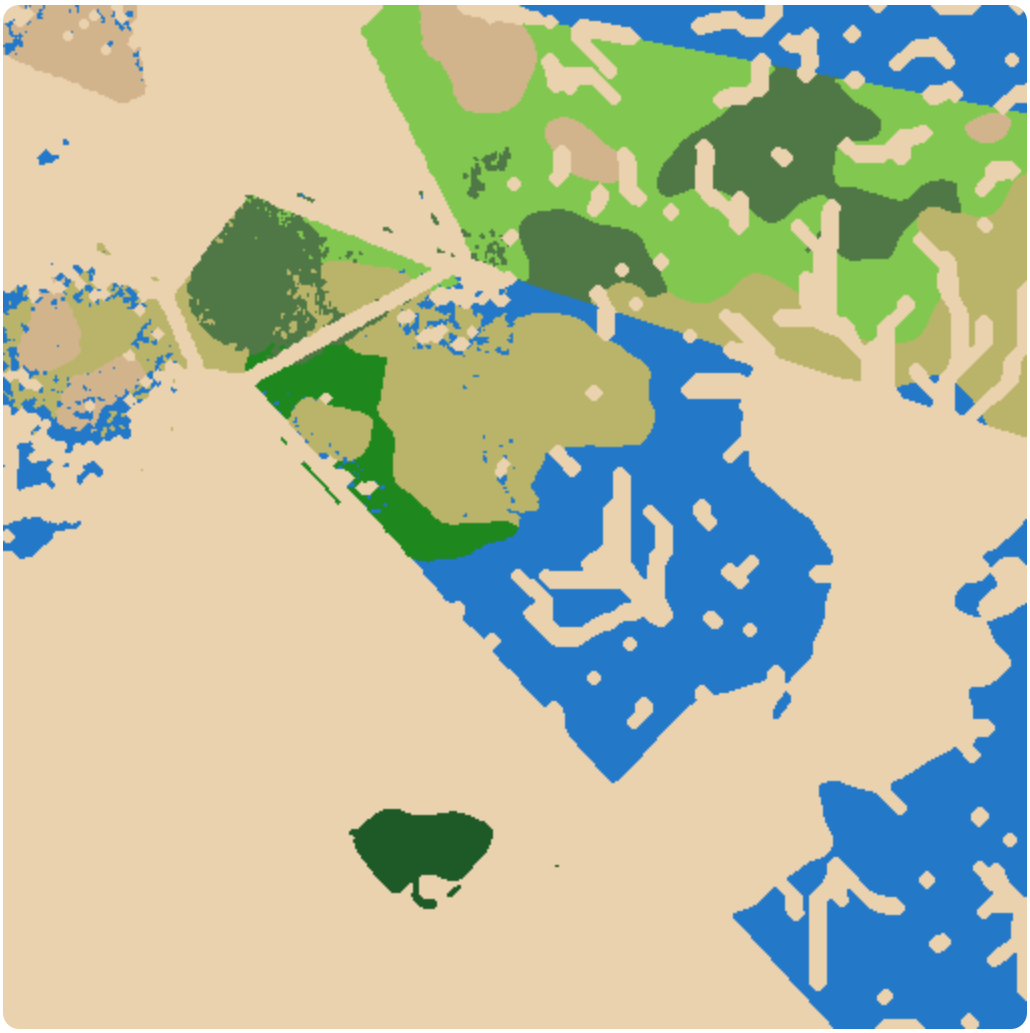












Thread count : 1

JSON


```
{
  "peak_memory_kb": 20960,
  "resolution": "512x512",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 1.1238996666666665,
    "heightmap_and_voronoi": 0.05156733333333333,
    "hydraulic_erosion": 2.9804146666666664,
    "object_placement": 0.015113,
    "river_generation": 0.028942
  },
  "threads": 1,
  "total_time": 4.201042,
  "wall_clock_time": 4.40207560857137,
  "num_runs": 3,
```

```
  "std_dev_total_time": 0.06316699702376202
}
```


Thread count : 4


```JSON
{
  "peak_memory_kb": 21948,
  "resolution": "512x512",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 0.43333333333333335,
    "heightmap_and_voronoi": 0.015985,
    "hydraulic_erosion": 3.2769033333333333,
    "object_placement": 0.014191,
    "river_generation": 0.025917666666666665
  },
  "threads": 4,
  "total_time": 3.7673573333333334,
  "wall_clock_time": 4.00490681330363,
  "num_runs": 3,
  "std_dev_total_time": 0.07084865306647226
}
```


Thread count : 8


```JSON
{
  "peak_memory_kb": 22462.666666666668,
  "resolution": "512x512",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 0.346025,
    "heightmap_and_voronoi": 0.013888,
    "hydraulic_erosion": 4.798365,
    "object_placement": 0.016855666666666668,
    "river_generation": 0.029244000000000003
  },
  "threads": 8,
  "total_time": 5.205504333333334,
  "wall_clock_time": 5.502501726150513,
  "num_runs": 3,
```

```
  "std_dev_total_time": 0.7212020910260406
}
```

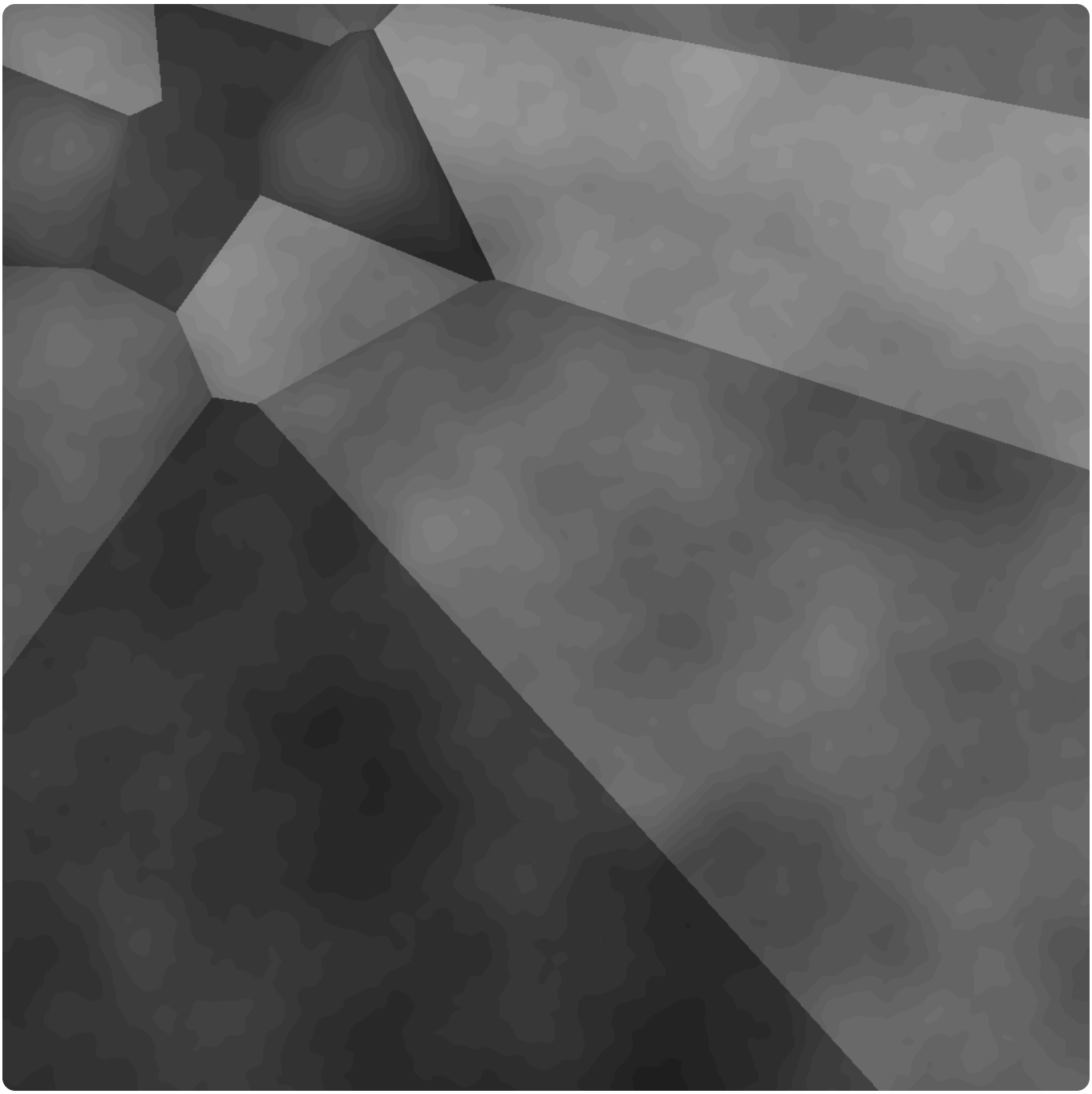

Thread count : 20

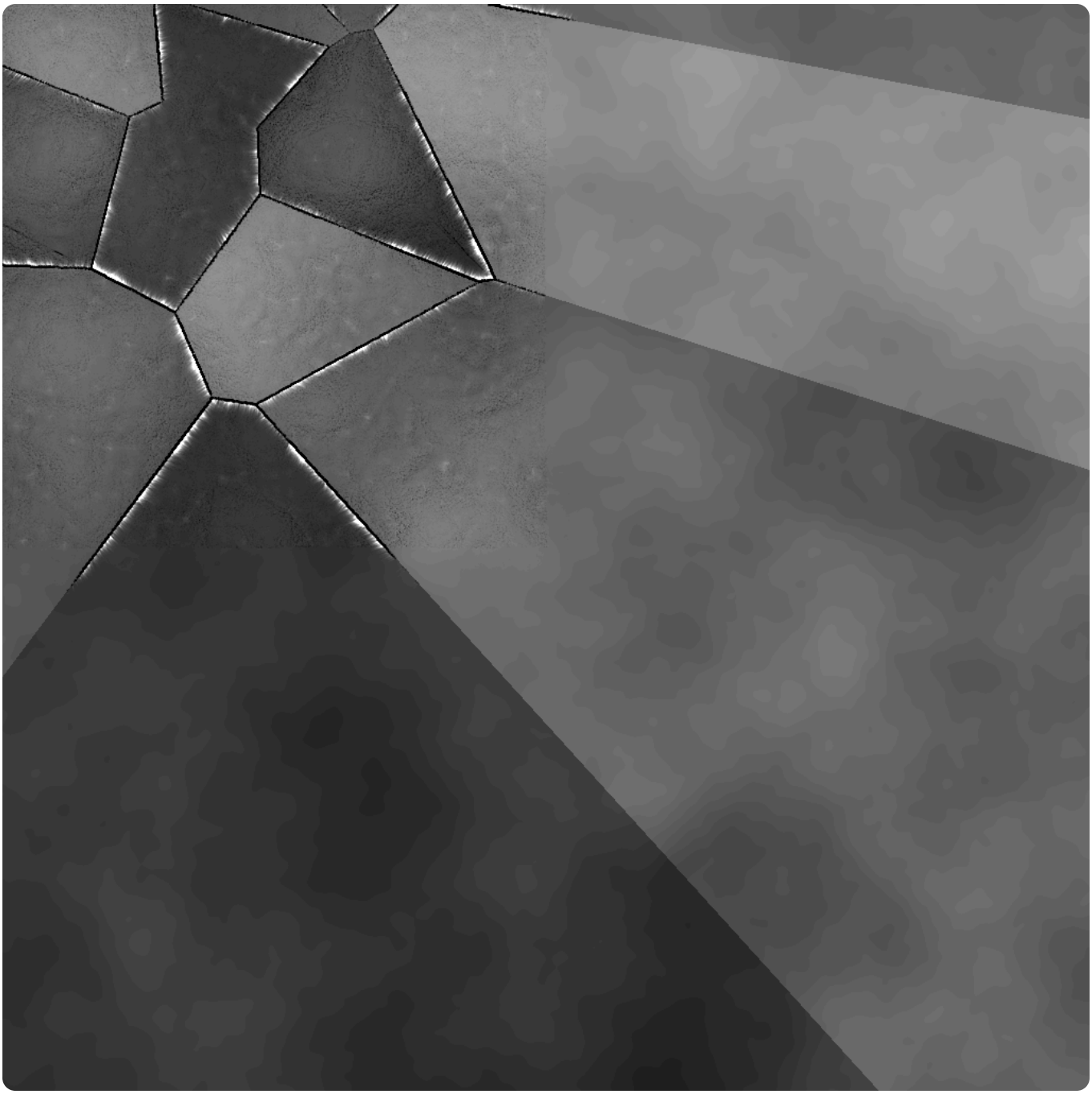

```JSON
{
  "peak_memory_kb": 22989.333333333332,
  "resolution": "512x512",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 0.273598,
    "heightmap_and_voronoi": 0.010013,
    "hydraulic_erosion": 8.474720666666666,
    "object_placement": 0.024568333333333334,
    "river_generation": 0.03528066666666667
  },
  "threads": 20,
  "total_time": 8.819190333333333,
  "wall_clock_time": 9.111084143320719,
  "num_runs": 3,
  "std_dev_total_time": 0.40632142407499666
}
```

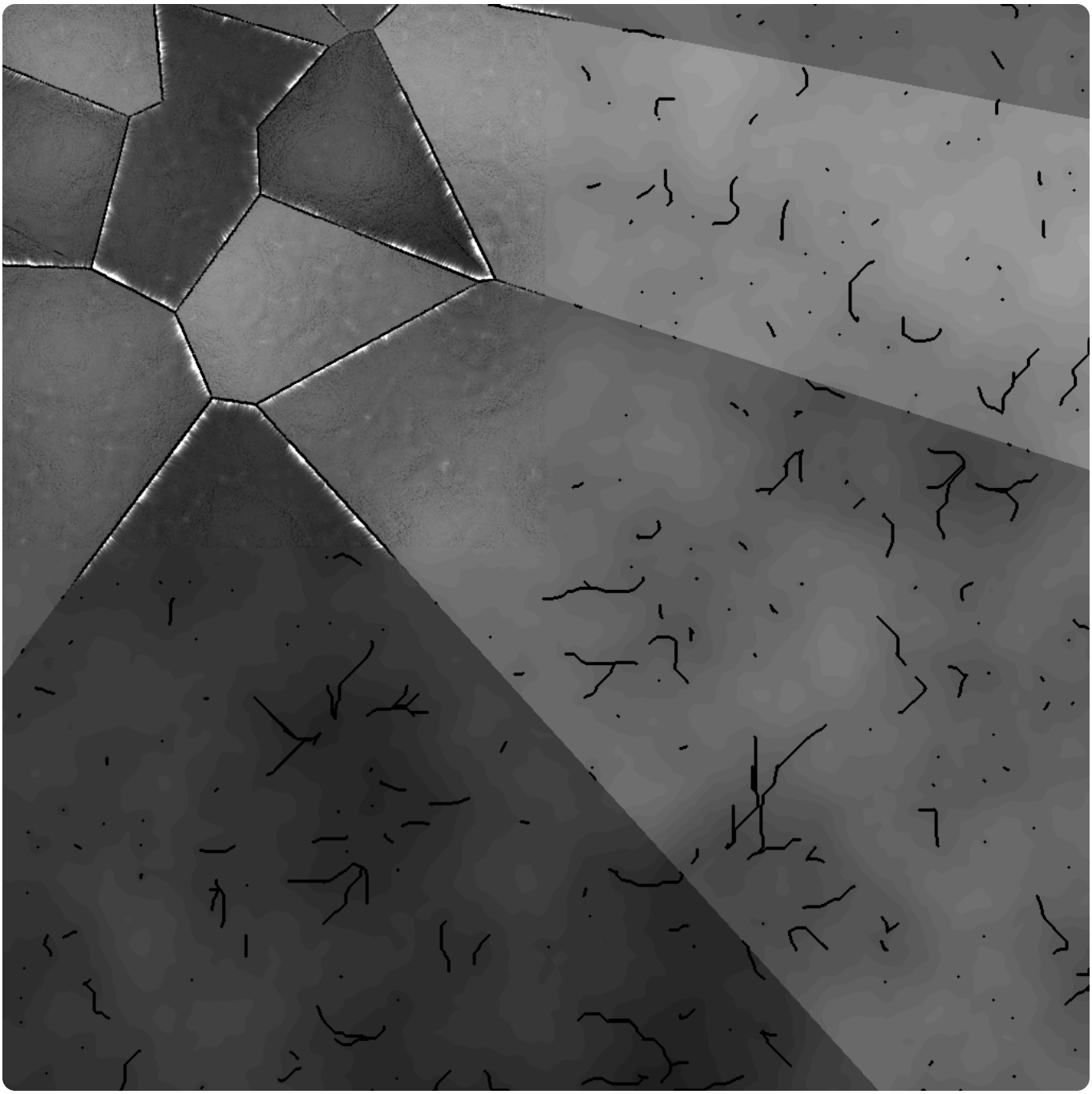

5.3 Testcase 3 :

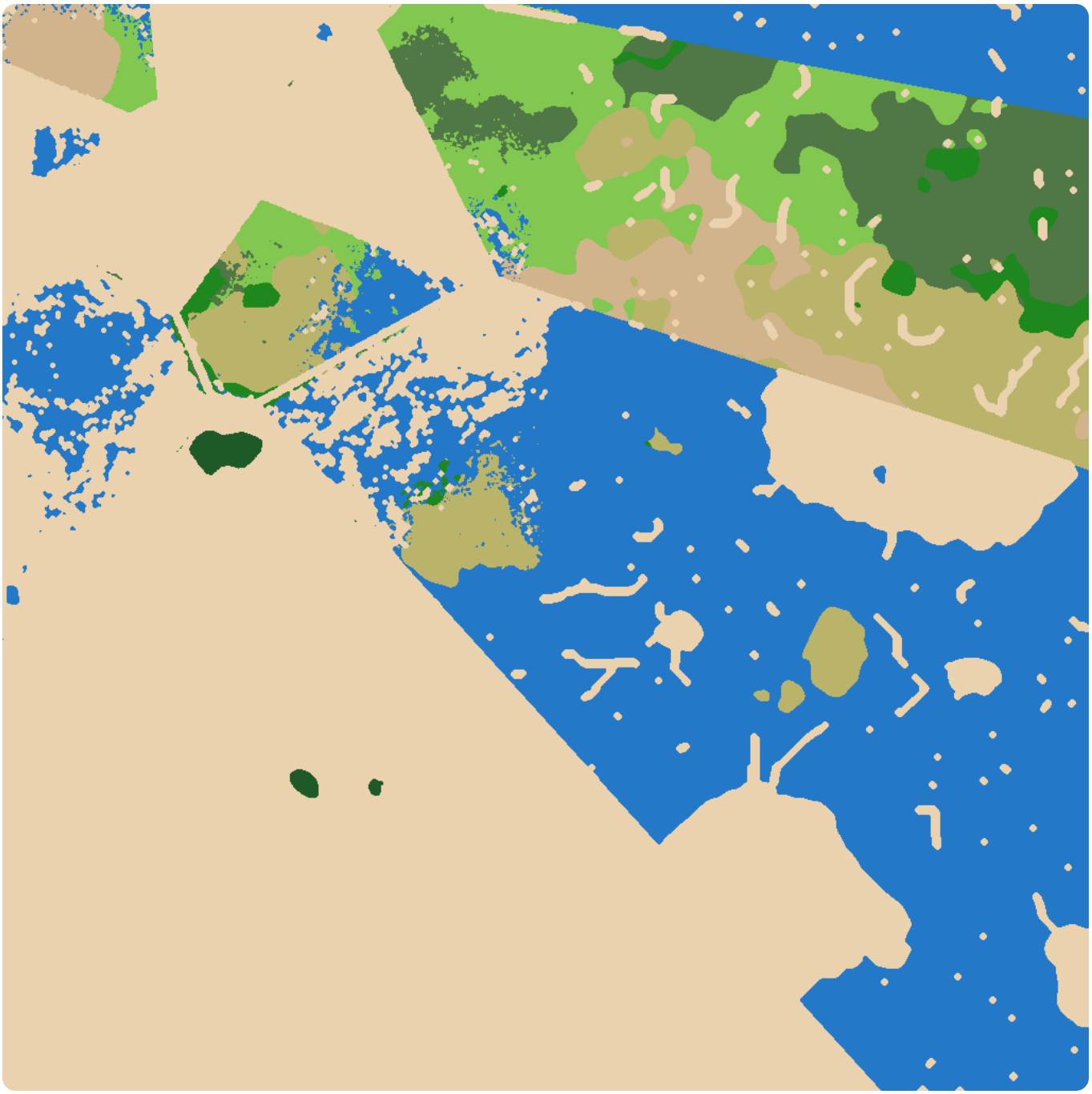
- Width = Height = 1024

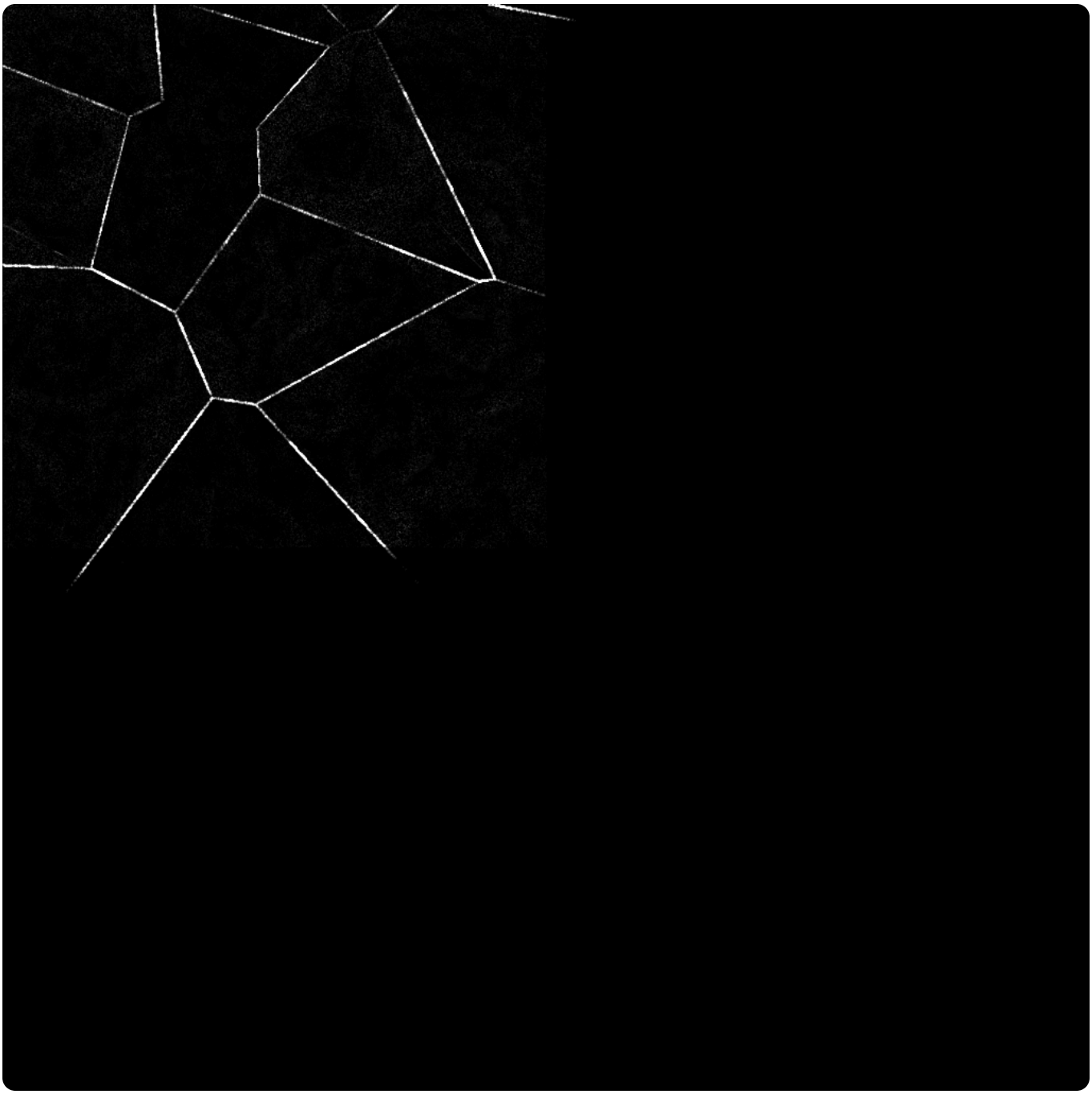
Output maps :

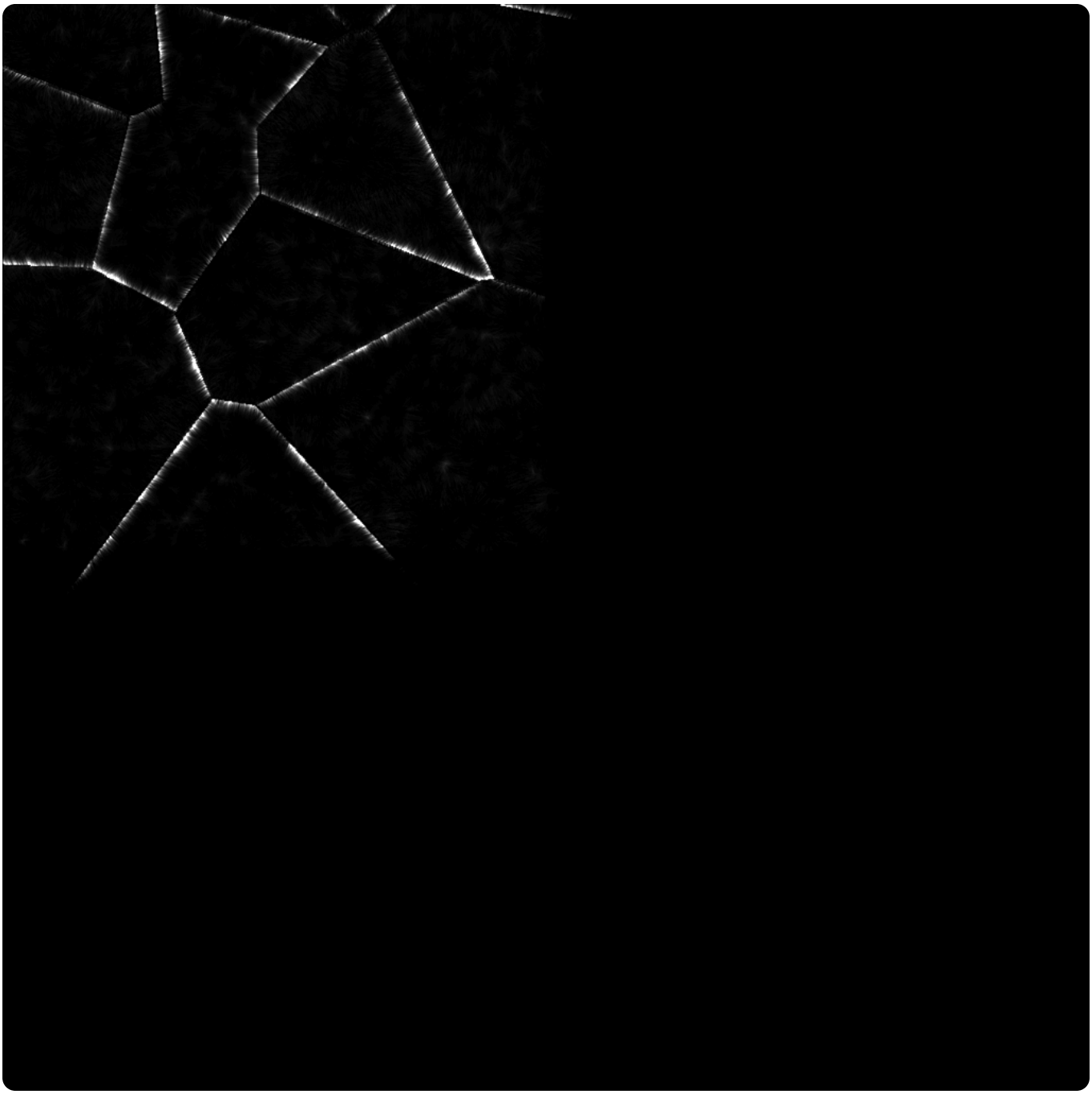


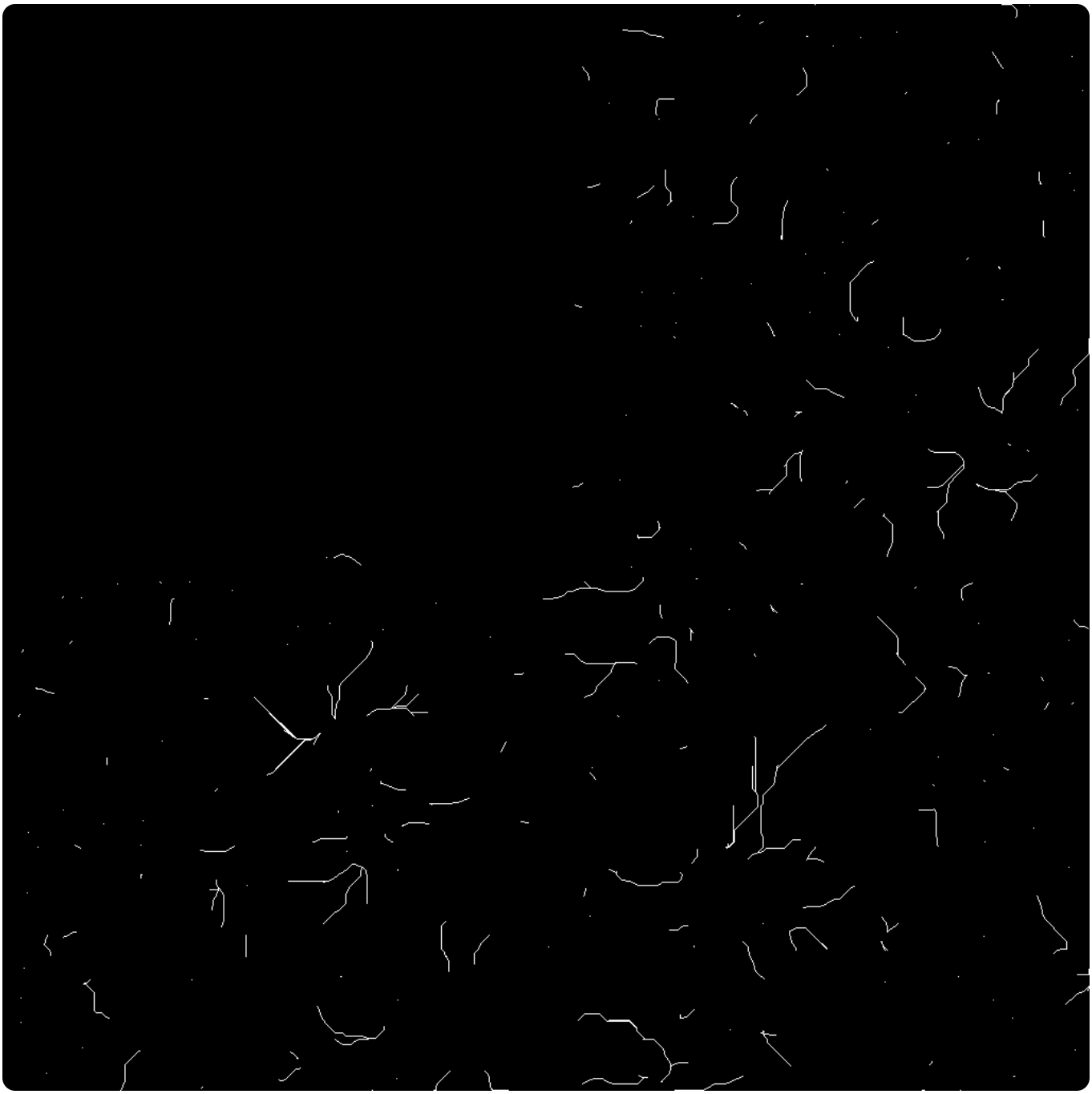


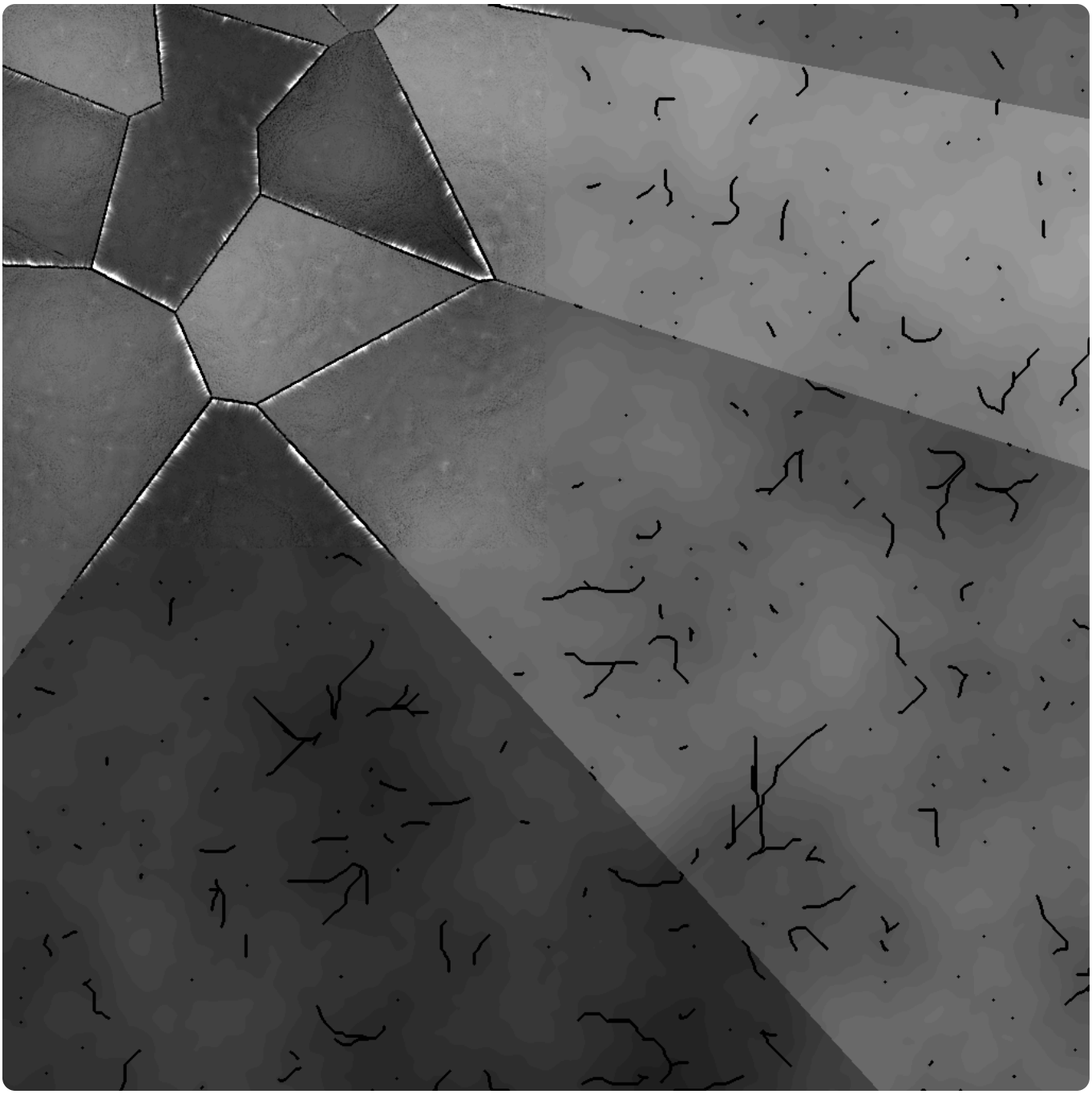


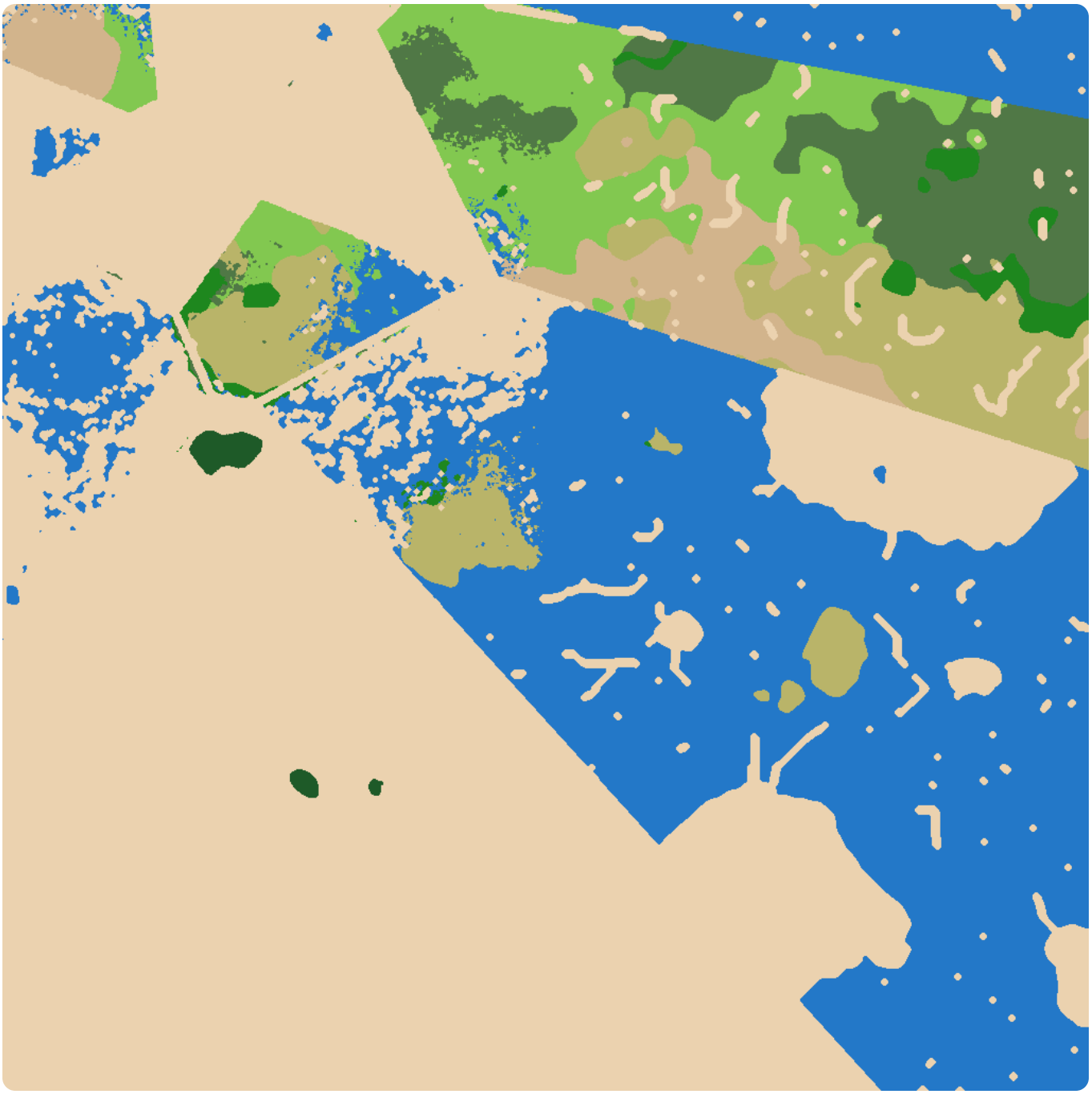












Thread count : 1

JSON


```
{
  "peak_memory_kb": 81860,
  "resolution": "1024x1024",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 5.995171999999999,
    "heightmap_and_voronoi": 0.202447,
    "hydraulic_erosion": 13.119759666666667,
    "object_placement": 0.07853466666666667,
```

```
    "river_generation": 0.168042
  },
  "threads": 1,
  "total_time": 19.567533,
  "wall_clock_time": 20.598216374715168,
  "num_runs": 3,
  "std_dev_total_time": 1.523458791152225
}
```


Thread count : 4


```json
{
  "peak_memory_kb": 83032,
  "resolution": "1024x1024",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 3.330651,
    "heightmap_and_voronoi": 0.112471,
    "hydraulic_erosion": 7.288755,
    "object_placement": 0.043630,
    "river_generation": 0.093357
  },
  "threads": 4,
  "total_time": 10.868863,
  "wall_clock_time": 11.42,
  "num_runs": 3,
  "std_dev_total_time": 0.076
}
```


Thread count : 8


```json
{
  "peak_memory_kb": 84668,
  "resolution": "1024x1024",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 1.998391,
    "heightmap_and_voronoi": 0.067482,
    "hydraulic_erosion": 4.373253,
    "object_placement": 0.026178,
```

```
    "river_generation": 0.056014
  },
  "threads": 8,
  "total_time": 6.522511,
  "wall_clock_time": 6.87,
  "num_runs": 3,
  "std_dev_total_time": 0.102
}
```

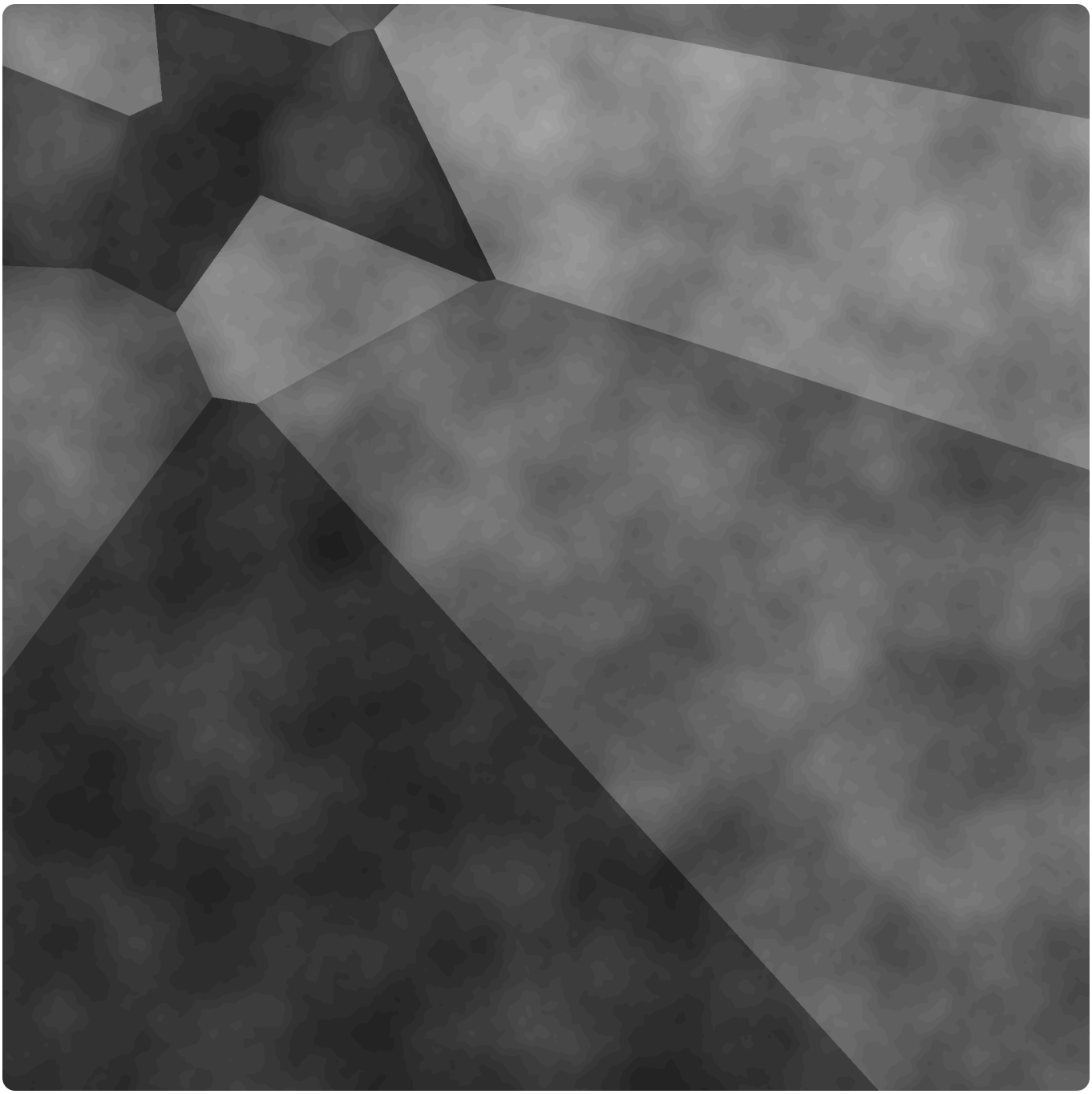

Thread count : 20

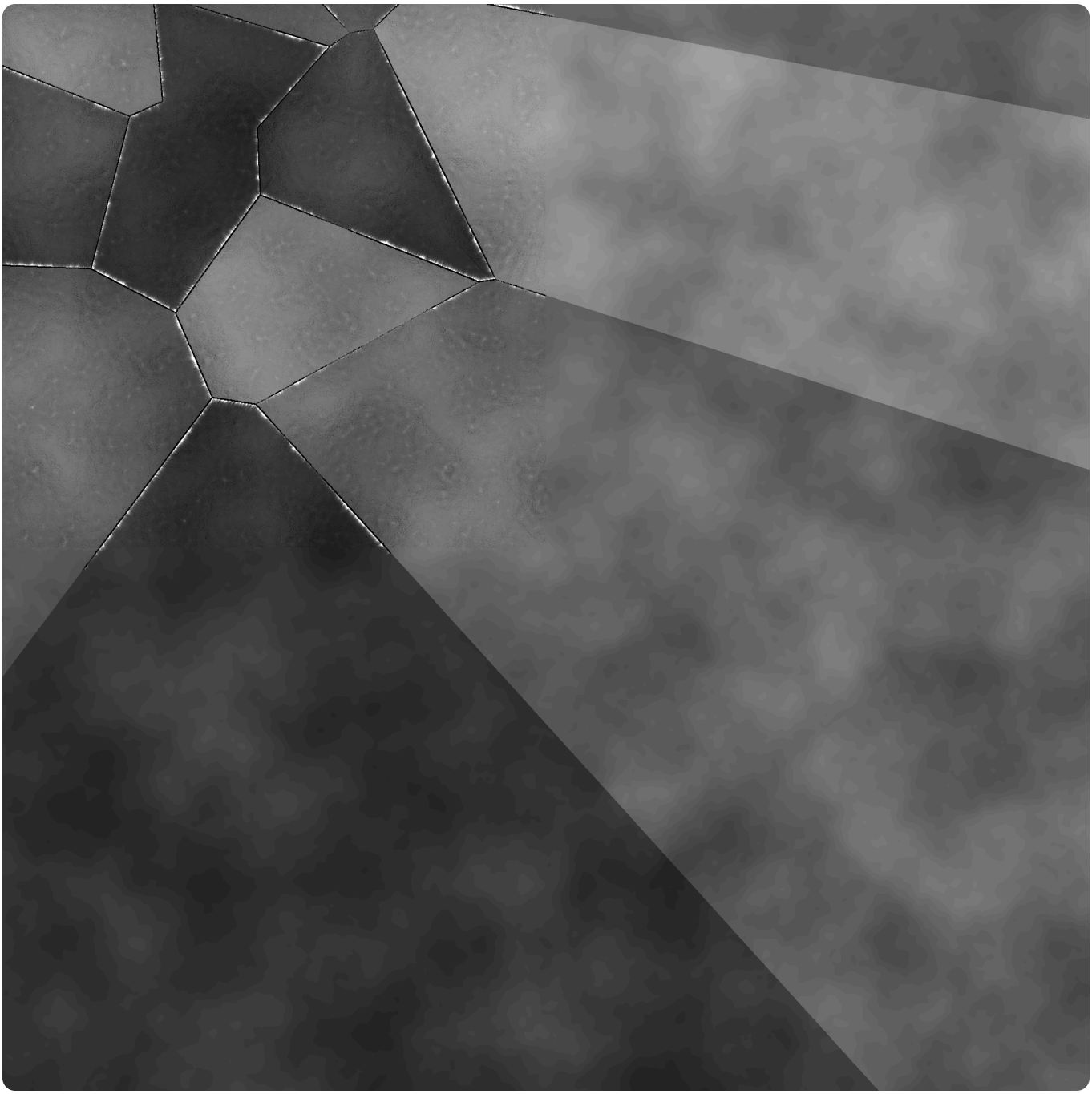
```
JSON
{
  "peak_memory_kb": 86329.33333333333,
  "resolution": "1024x1024",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 0.894801,
    "heightmap_and_voronoi": 0.030216,
    "hydraulic_erosion": 1.958173,
    "object_placement": 0.011722,
    "river_generation": 0.025080
  },
  "threads": 20,
  "total_time": 3.000910,
  "wall_clock_time": 3.07,
  "num_runs": 3,
  "std_dev_total_time": 0.04
}
```

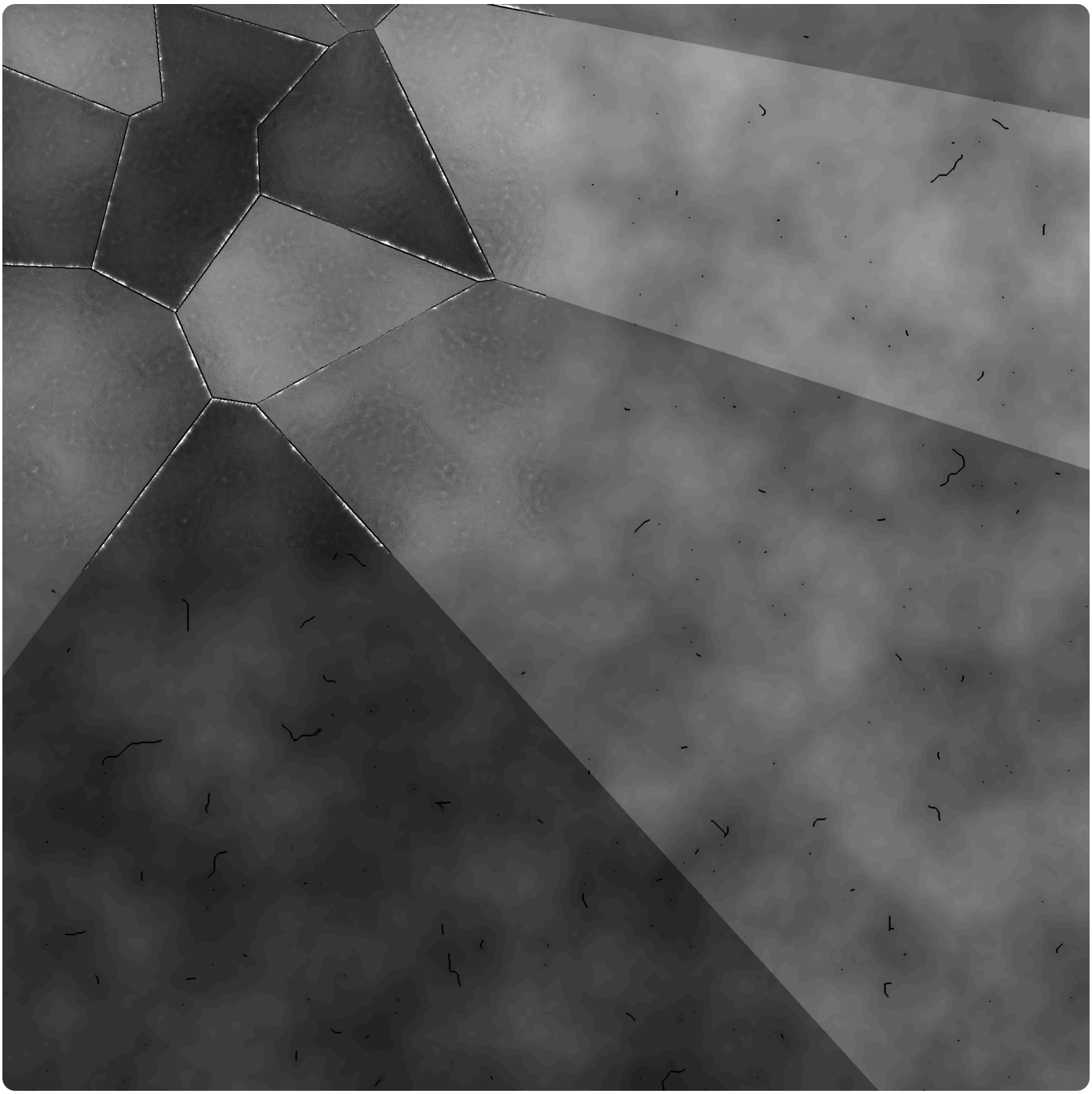

5.4 Testcase 4 :

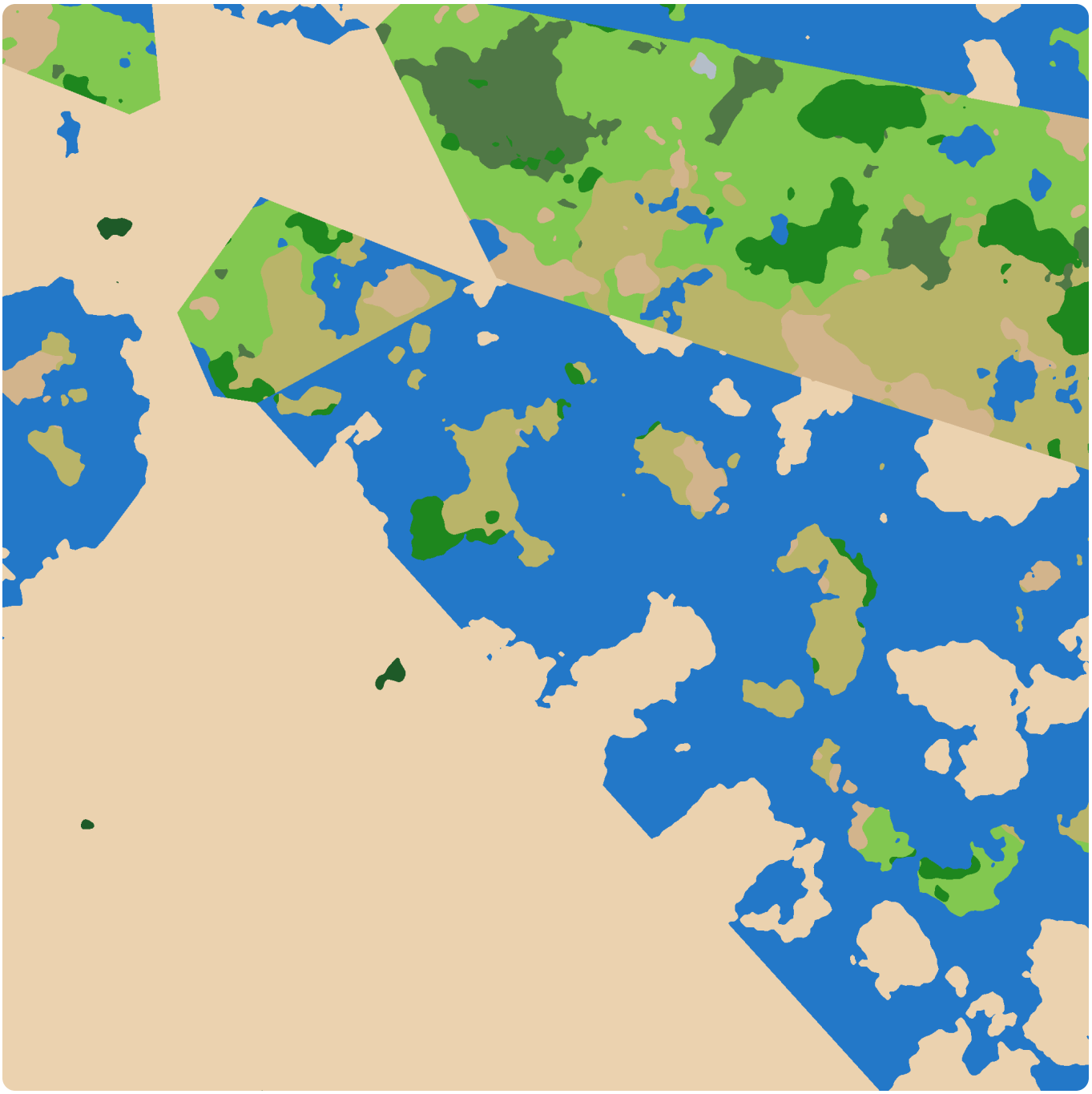
- Width = Height = 2048

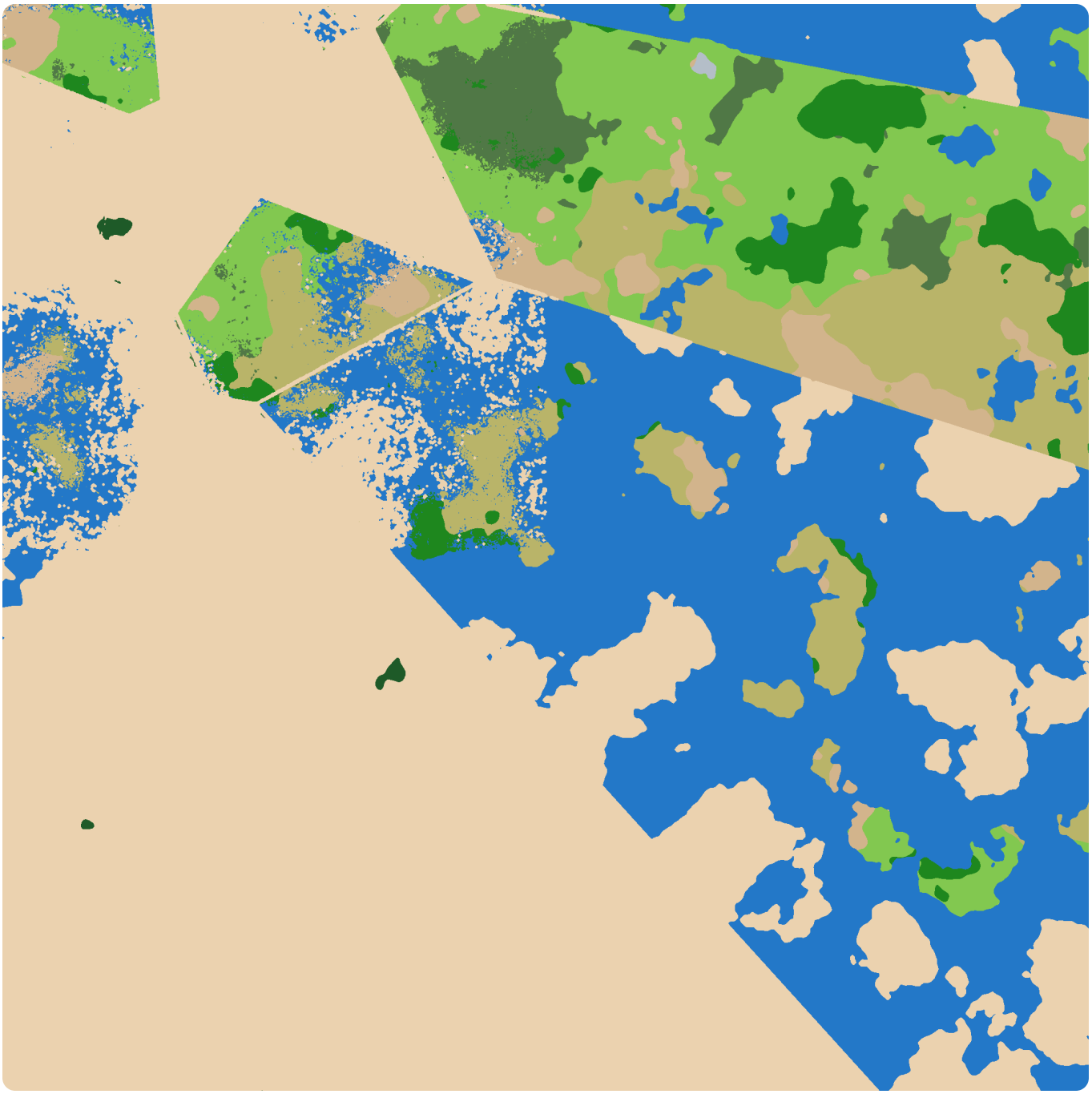
Output maps :

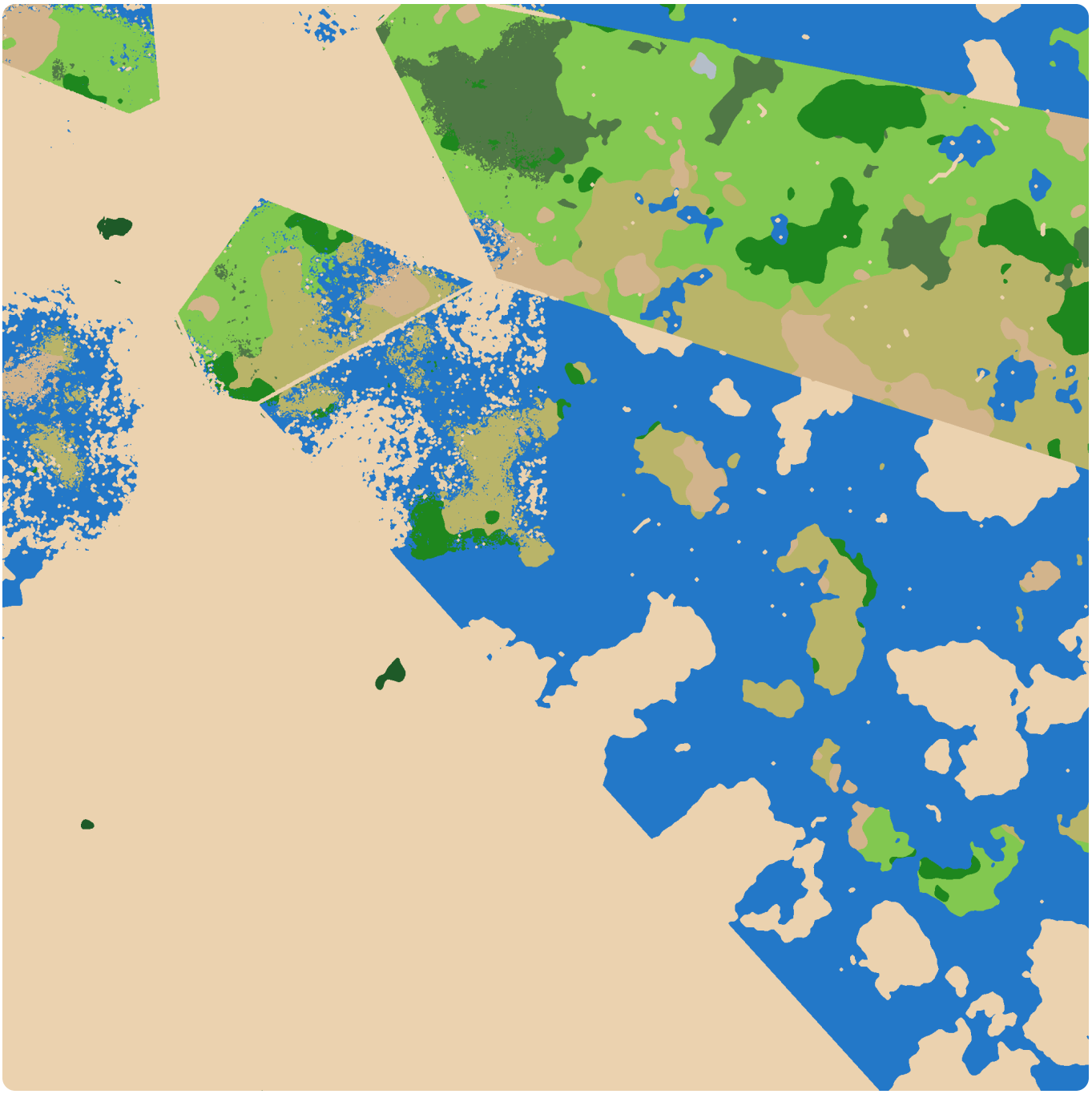


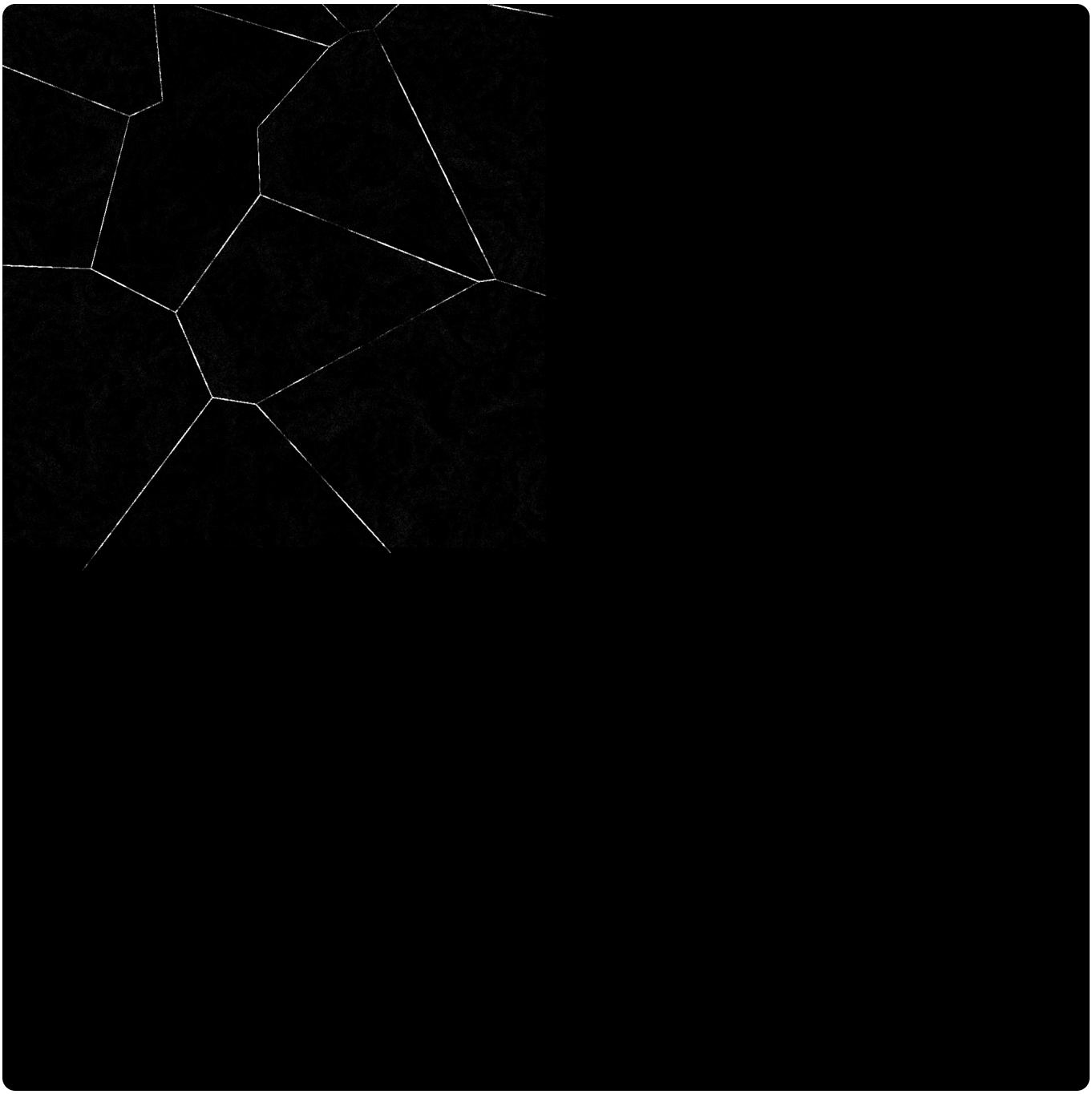


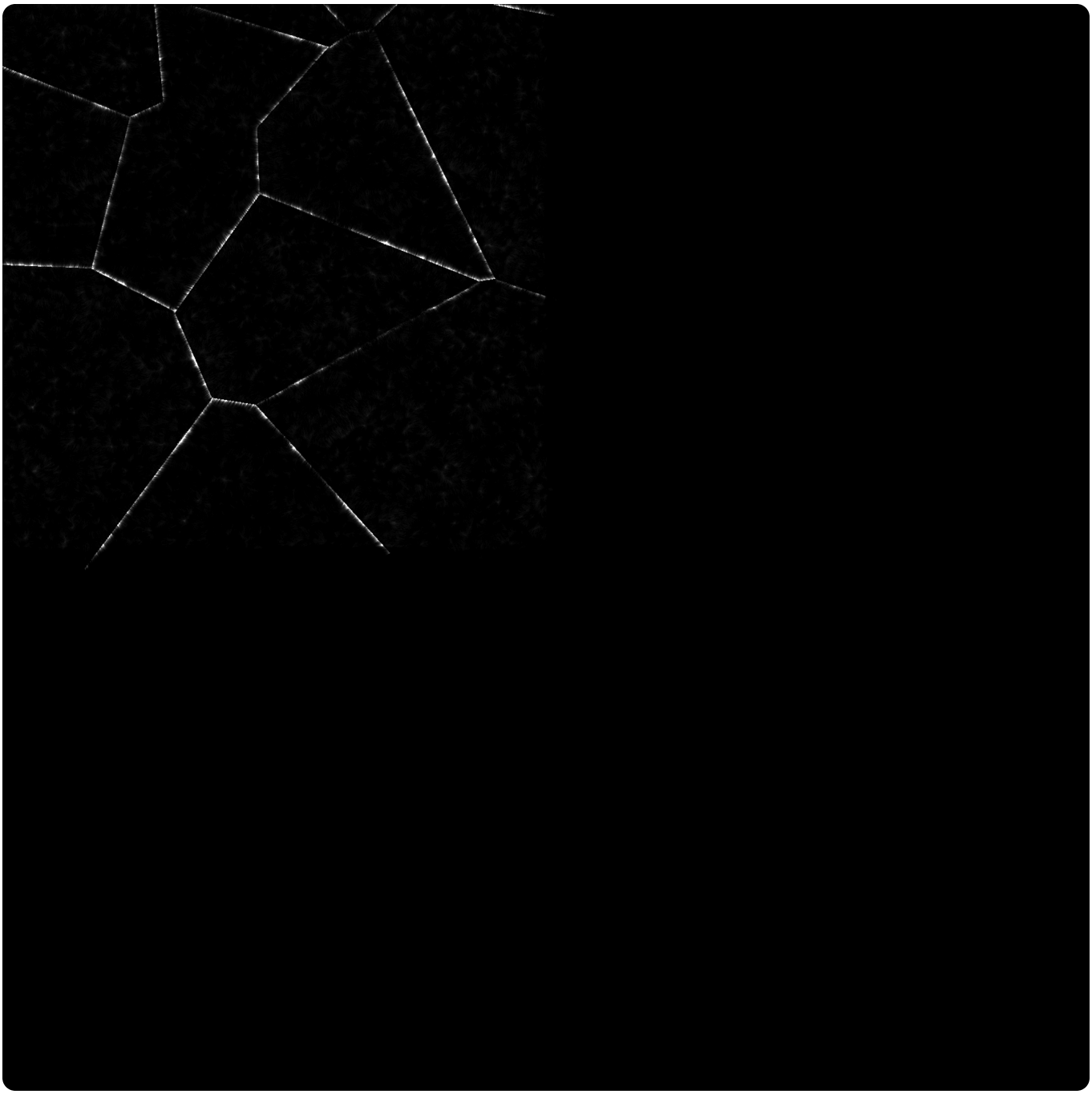


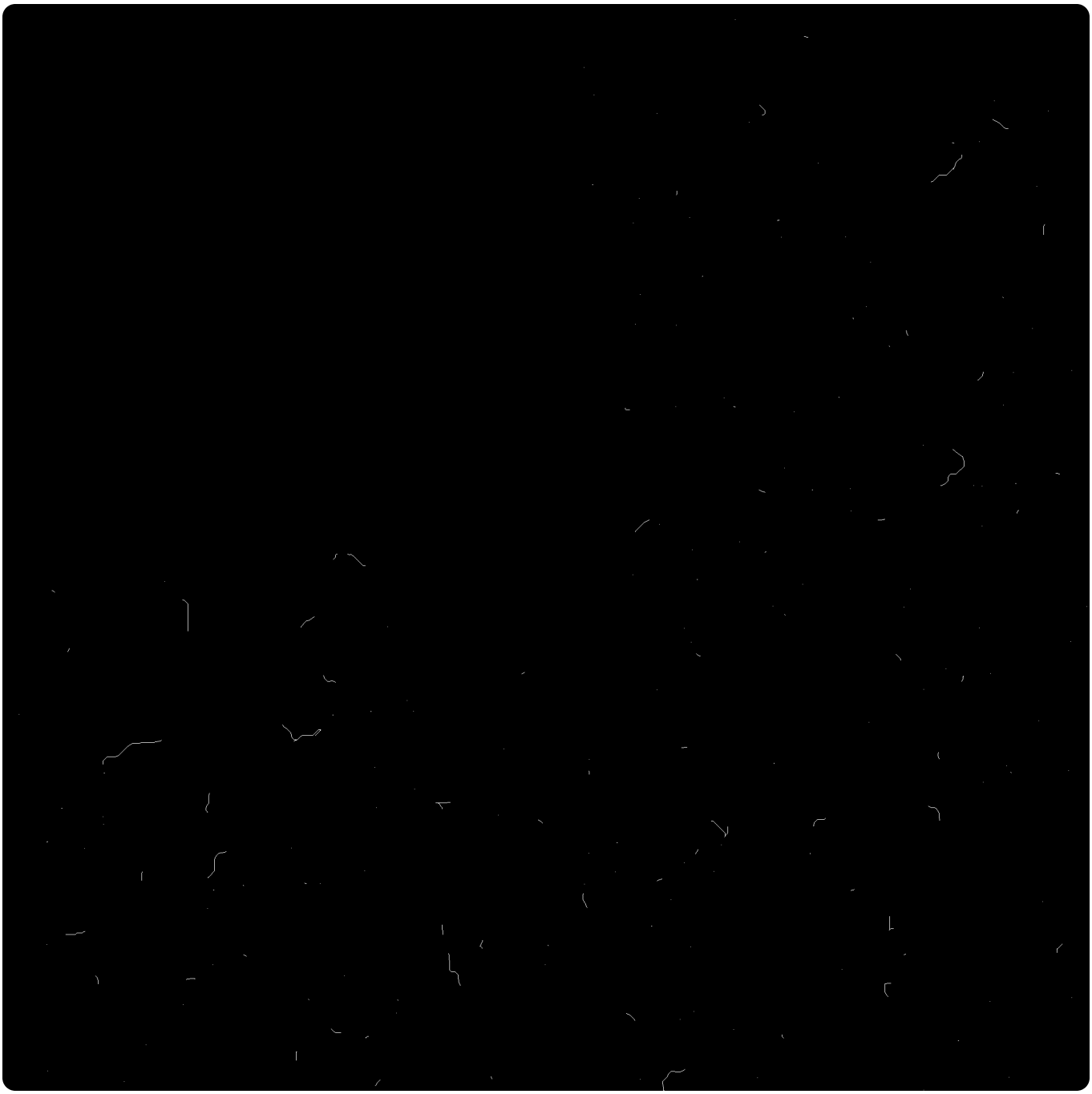


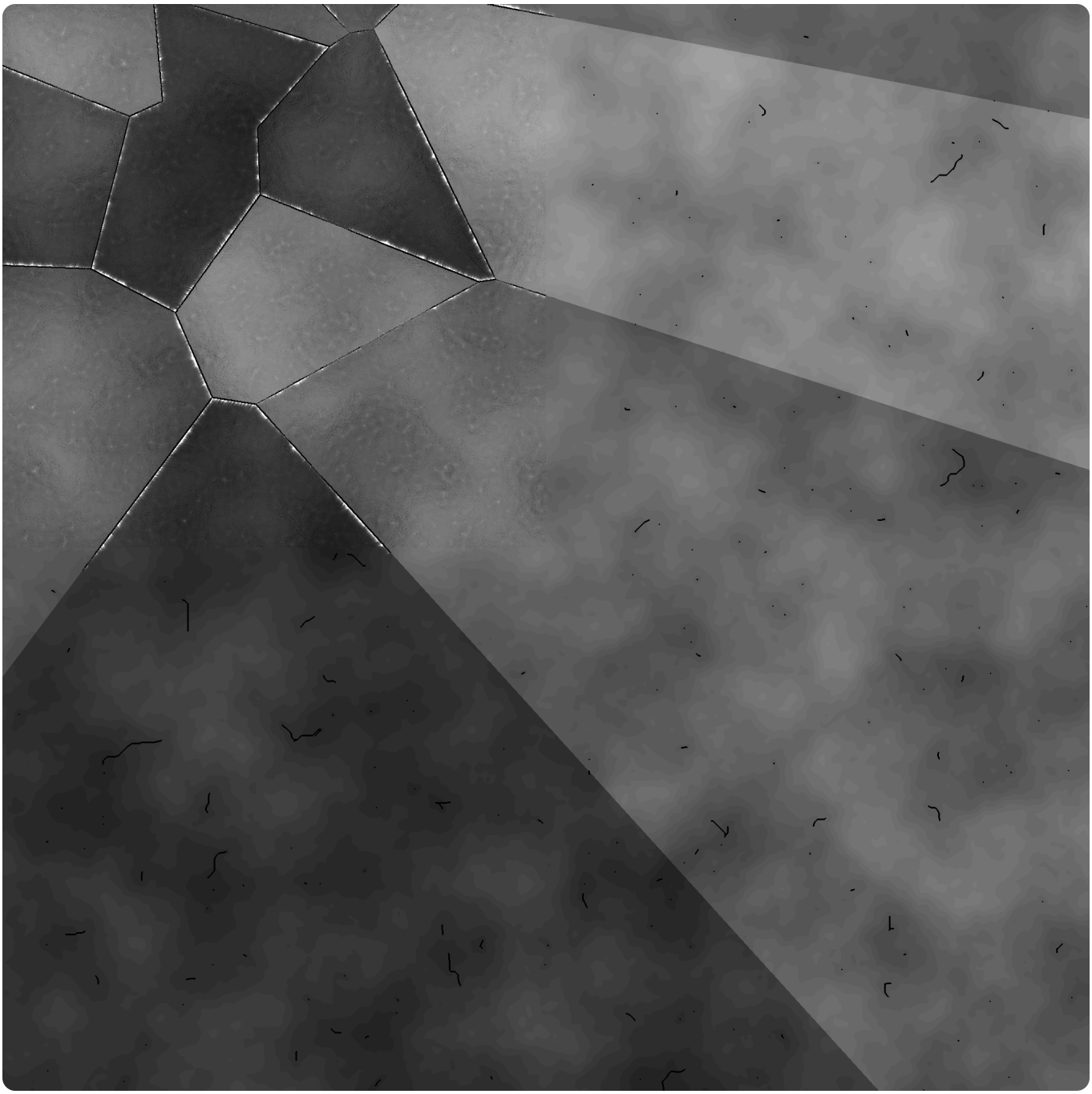


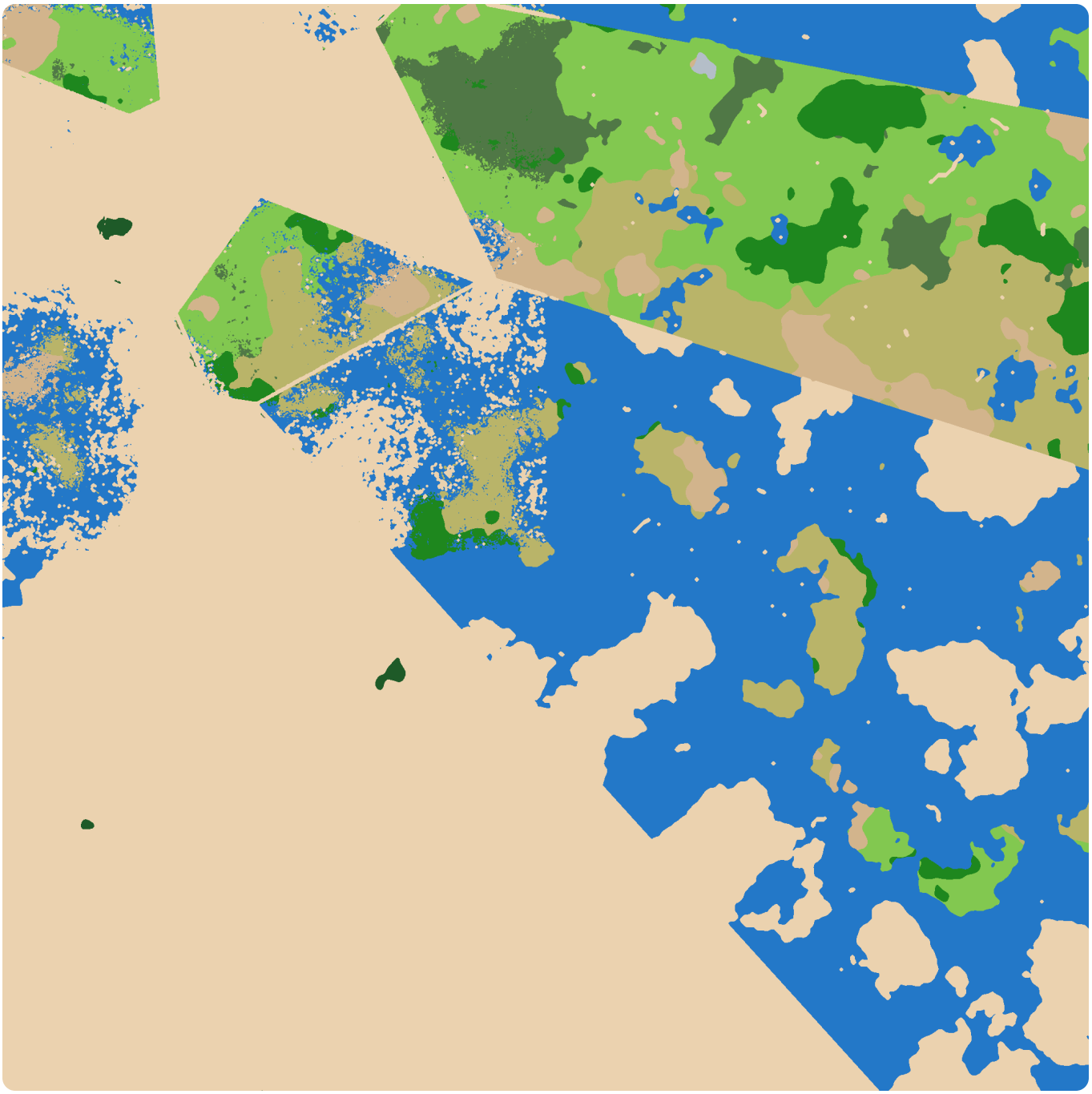












Thread count : 1

JSON


```
{
  "peak_memory_kb": 324832,
  "resolution": "2048x2048",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 20.029290333333332,
    "heightmap_and_voronoi": 0.8234993333333334,
    "hydraulic_erosion": 63.538539,
    "object_placement": 0.25907399999999997,
```

```
    "river_generation": 0.6463086666666666
  },
  "threads": 1,
  "total_time": 85.312585,
  "wall_clock_time": 89.76369547843933,
  "num_runs": 3,
  "std_dev_total_time": 7.007353495490935
}
```


Thread count : 4

```json
{
  "peak_memory_kb": 329400,
  "resolution": "2048x2048",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 11.127383,
    "heightmap_and_voronoi": 0.457499,
    "hydraulic_erosion": 35.299188,
    "object_placement": 0.143930,
    "river_generation": 0.359060
  },
  "threads": 4,
  "total_time": 47.387060,
  "wall_clock_time": 49.87,
  "num_runs": 3,
  "std_dev_total_time": 0.31
}
```


Thread count : 8

```json
{
  "peak_memory_kb": 332100,
  "resolution": "2048x2048",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 6.676430,
    "heightmap_and_voronoi": 0.274500,
    "hydraulic_erosion": 21.179513,
    "object_placement": 0.086358,
```

```
    "river_generation": 0.215436
  },
  "threads": 8,
  "total_time": 28.437528,
  "wall_clock_time": 29.92,
  "num_runs": 3,
  "std_dev_total_time": 0.42
}
```


Thread count : 20


```JSON
{
  "peak_memory_kb": 336890,
  "resolution": "2048x2048",
  "run_id": 1,
  "stage_times": {
    "biome_classification": 2.989297,
    "heightmap_and_voronoi": 0.122910,
    "hydraulic_erosion": 9.483364,
    "object_placement": 0.038668,
    "river_generation": 0.096478
  },
  "threads": 20,
  "total_time": 12.730716,
  "wall_clock_time": 13.40,
  "num_runs": 3,
  "std_dev_total_time": 0.15
}
```


5.5 Time Complexity Analysis :

Serial :

Stage	Complexity	Dominant Factor
Heightmap (Voronoi)	$O(W \times H \times P)$	Plate comparisons
Climate Maps (FBM)	$O(W \times H \times O)$	Octaves per cell
Erosion Simulation	$O(D \times S \times R^2)$	Droplets × steps
River (Flow Dir)	$O(W \times H)$	Neighbor comparison
River (Flow Accum)	$O(W \times H \times \log(W \times H))$	Topological sort
River (Carving)	$O(W \times H)$	BFS + carving
Biome (Distance BFS)	$O(W \times H + E)$	BFS traversal
Biome (Classification)	$O(W \times H \times B)$	Biome scoring
Object Placement	$O(W \times H \times \bar{O})$	Expected objects/cell

$$T = O(W \times H \times (P + O + B) + D \times S \times R^2 + W \times H \times \log(W \times H))$$

Parallel :

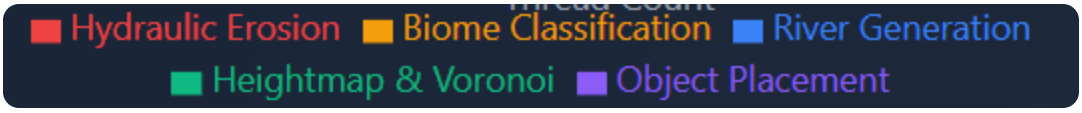
- Assume T threrads

$$T = O((W \times H \times (P + O + B) + D \times S \times R^2) / T + W \times H \times \log(W \times H) + overhead)$$

- Overhead may be caused by :
  - $O(T)$: Thread creation and destrucion
  - $O(T \times \log(T))$: Sync
  - $O(W \times H/B)$: Memory contention (B = Bandwidth factor)

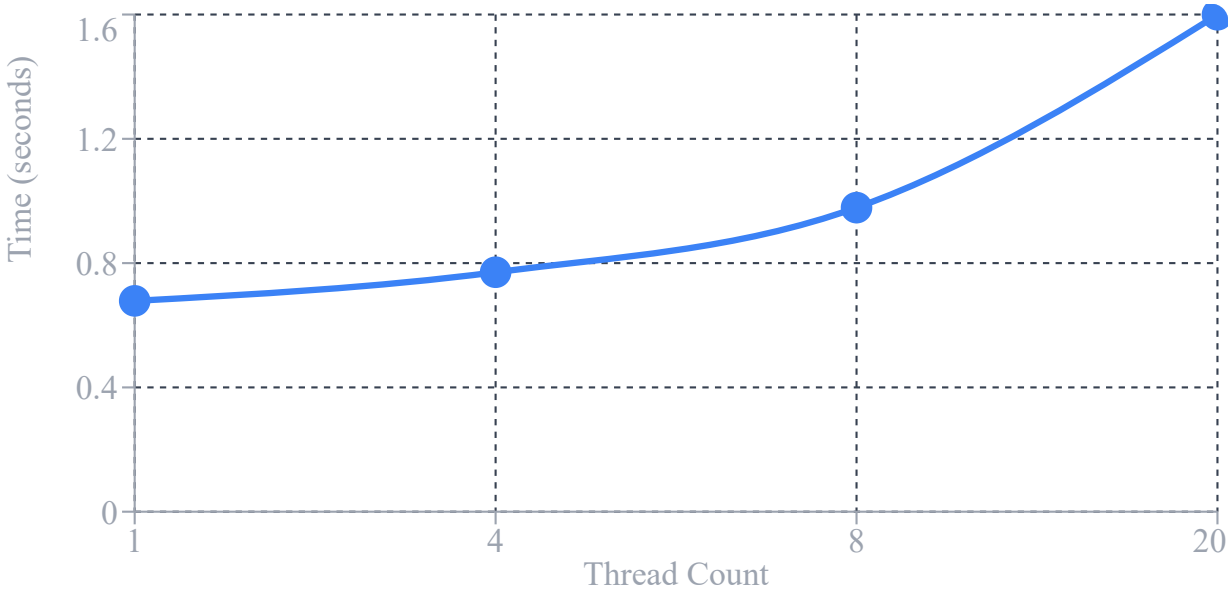
6. DISCUSSIONS AND OBSERVATIONS :

6.1 Improvements and Performance analysis :

- Legend:
  - 

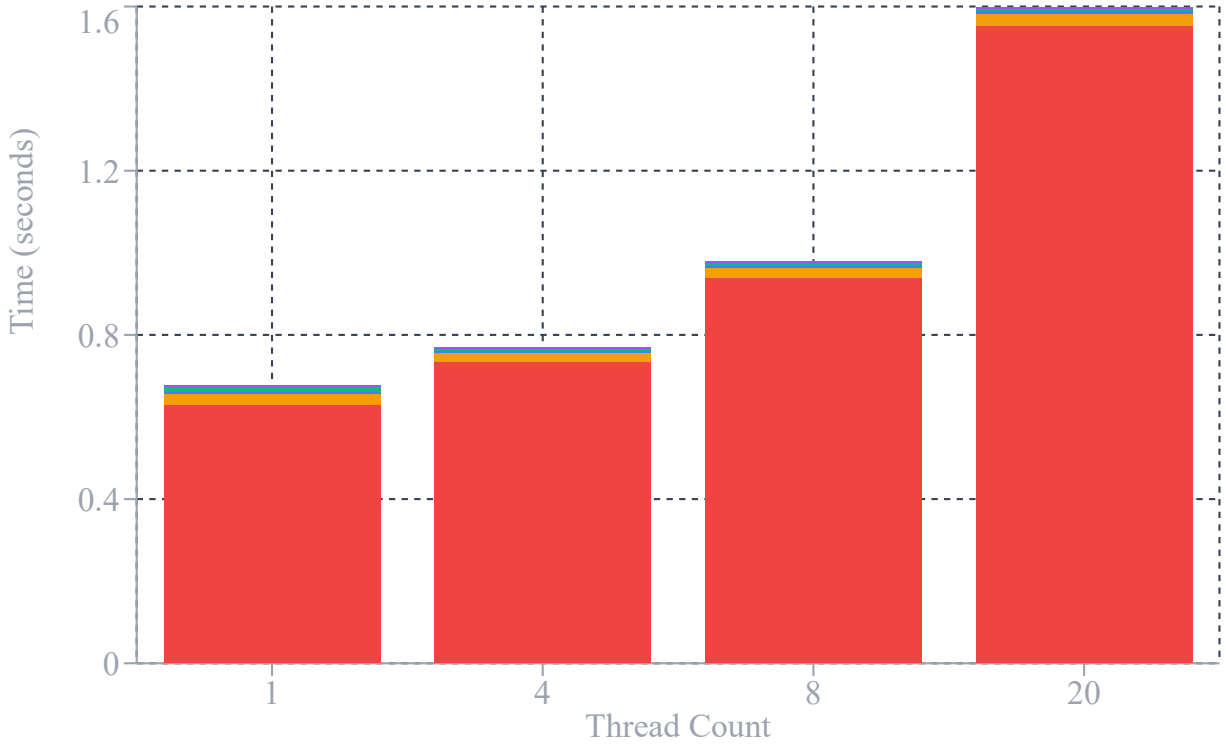
6.1.1 Testcase 1 analysis :

- Total Execution Time :

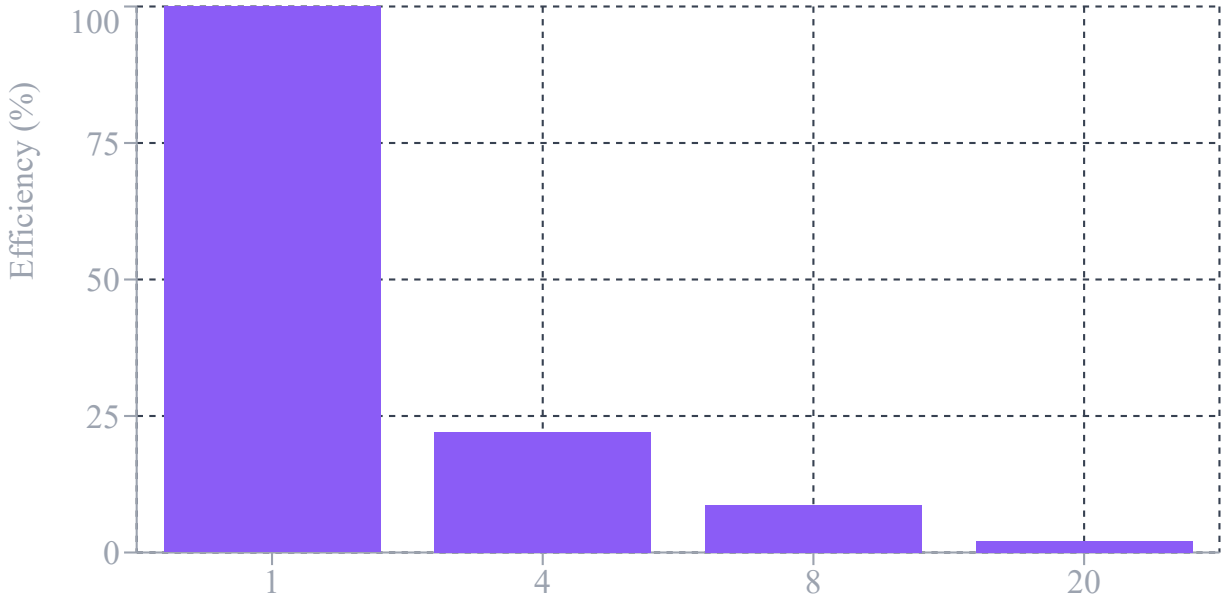


- Speedup Analysis :

- Stagewise Analysis :

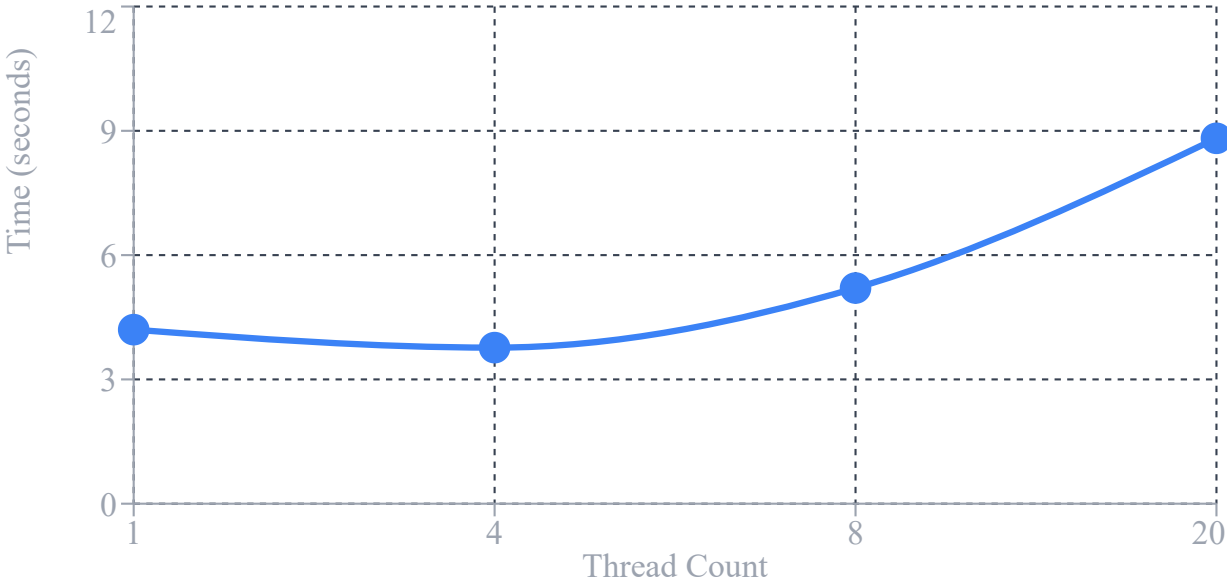


- Efficiency :

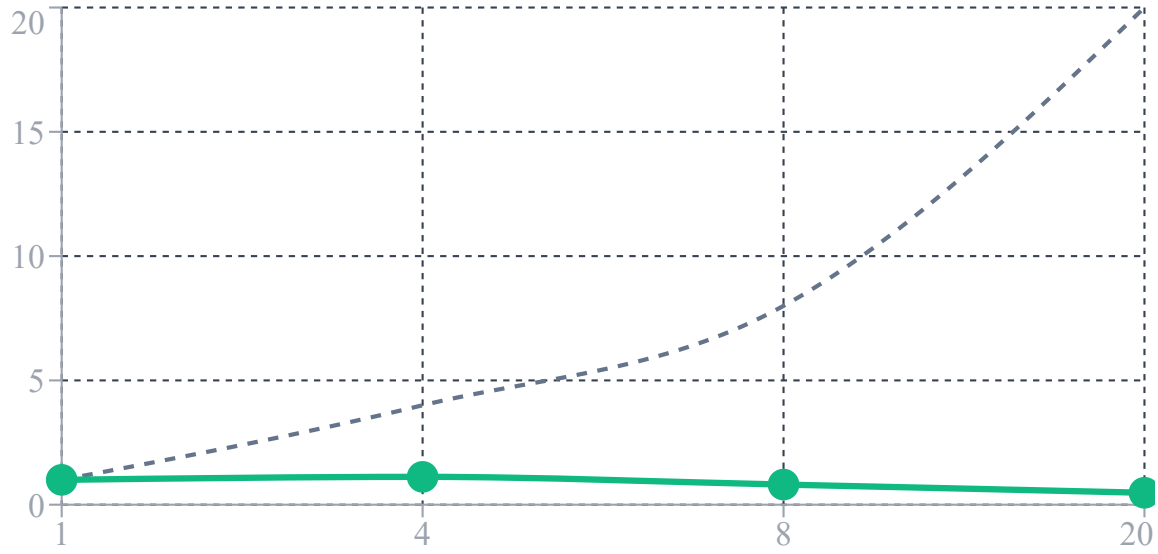


6.1.2 Testcase 2 analysis :

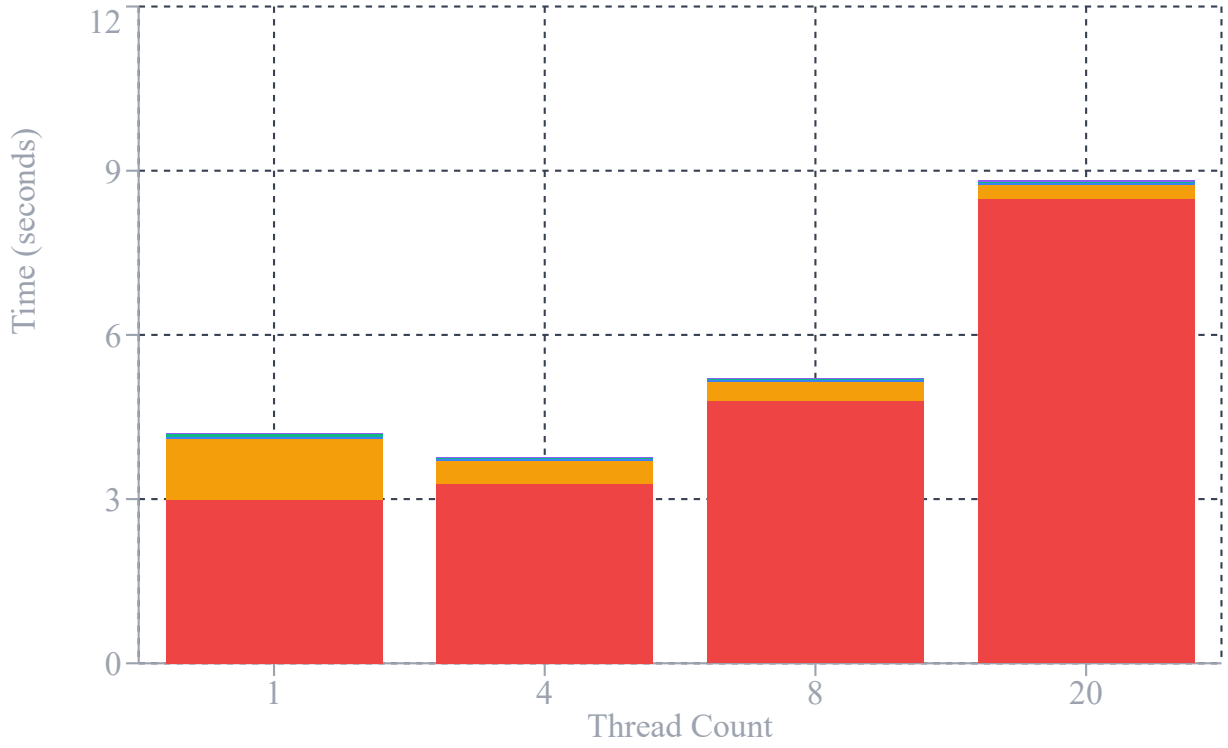
- Total Execution Time :



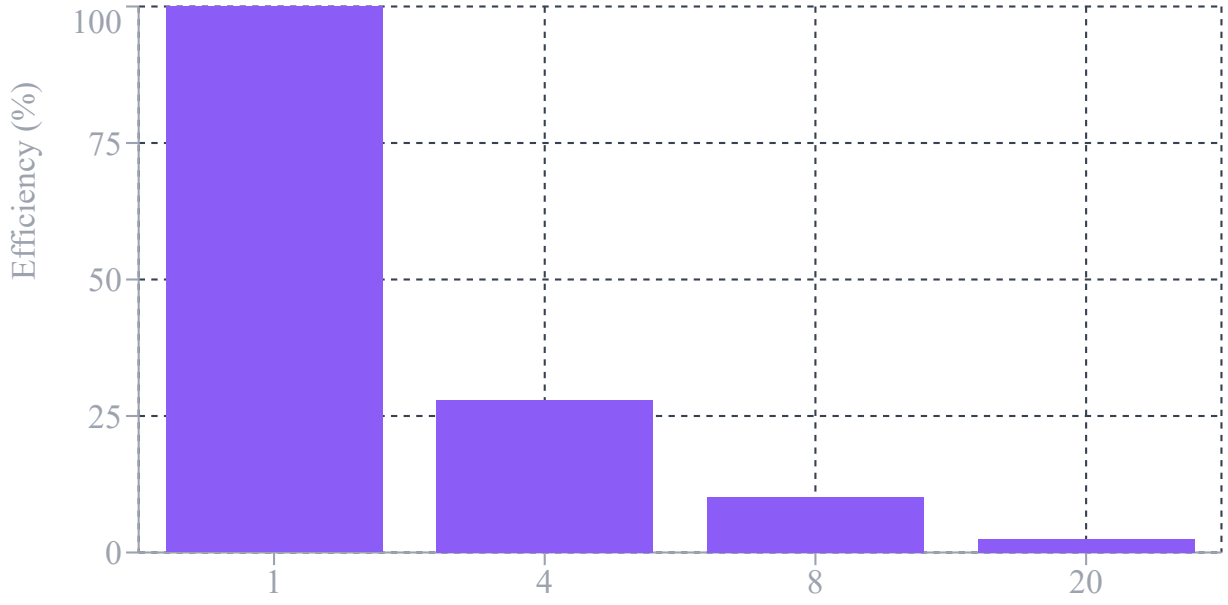
- Speedup Analysis :



- Stagewise Analysis :

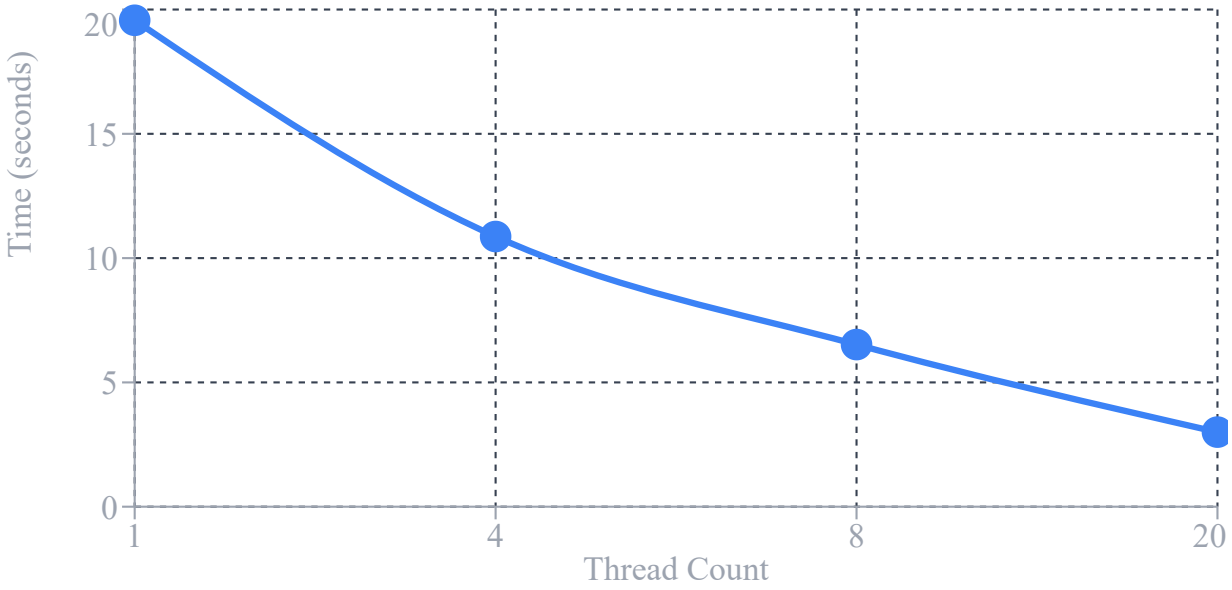


- Efficiency :



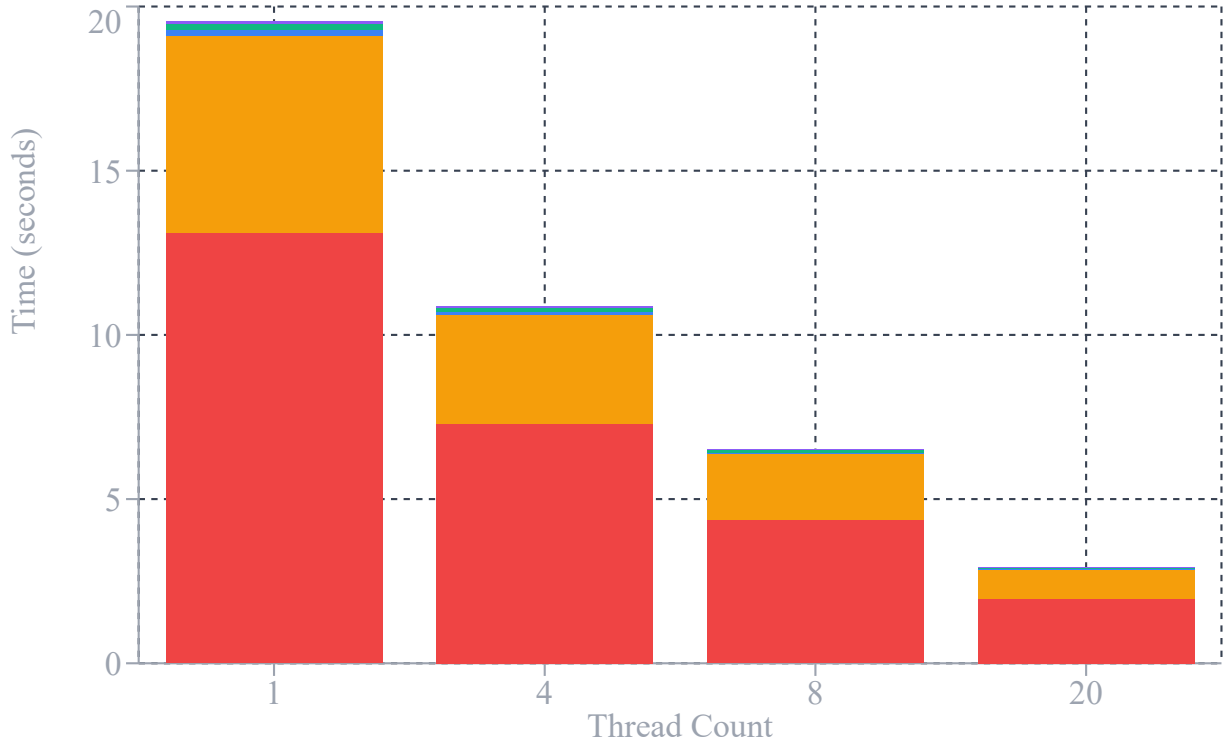
6.1.3 Testcase 3 analysis :

- Total Execution Time :

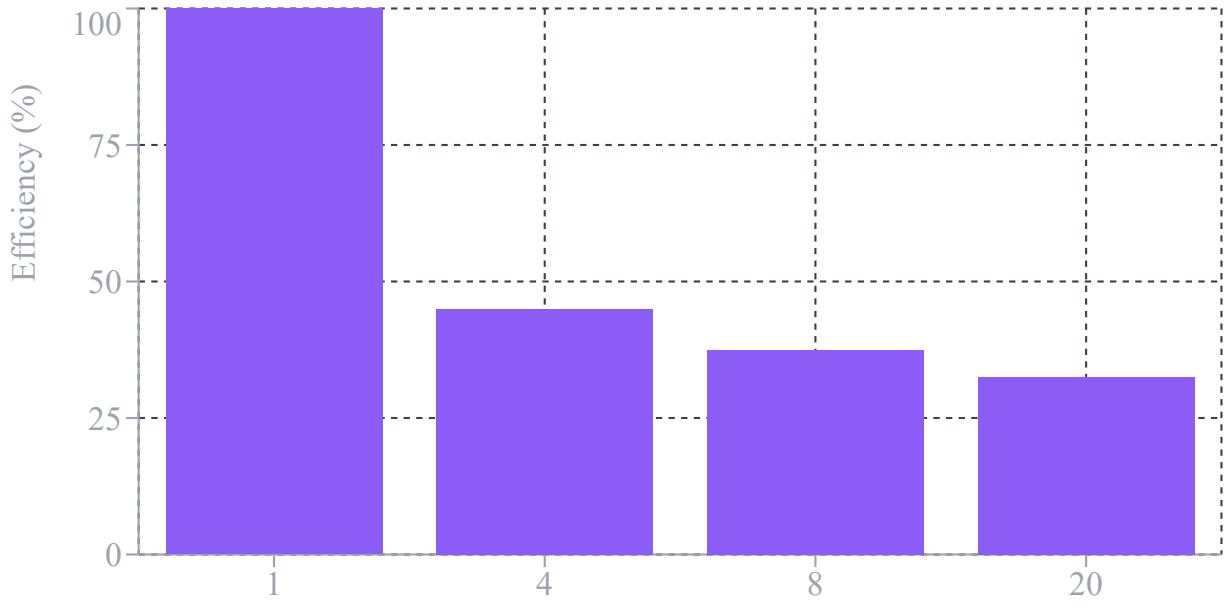


- Speedup Analysis :

- Stagewise Analysis :

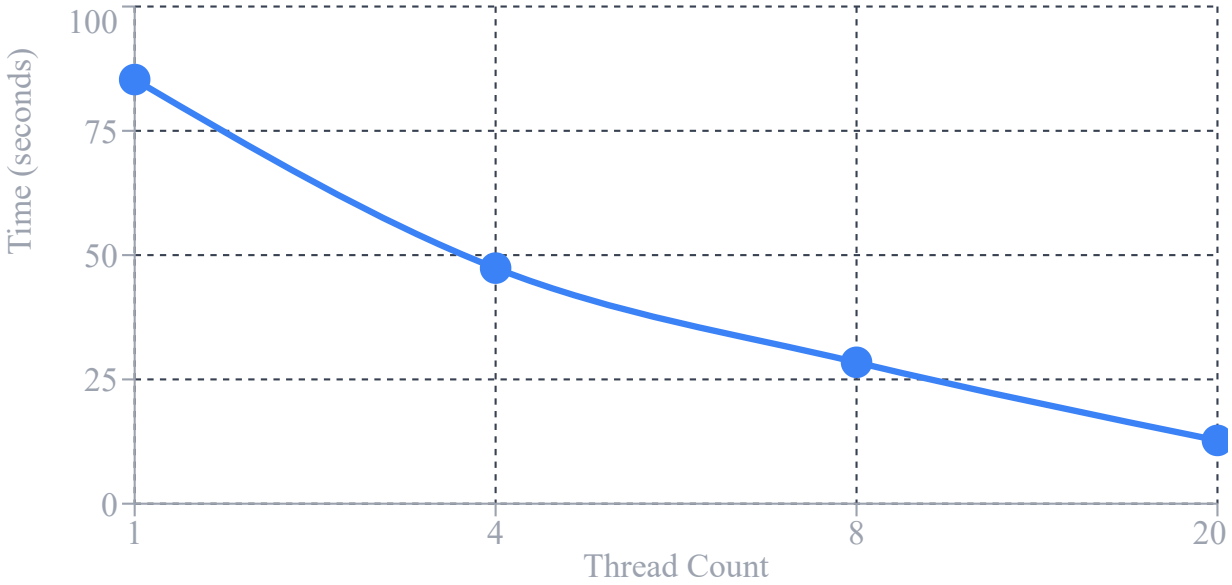


- Efficiency :



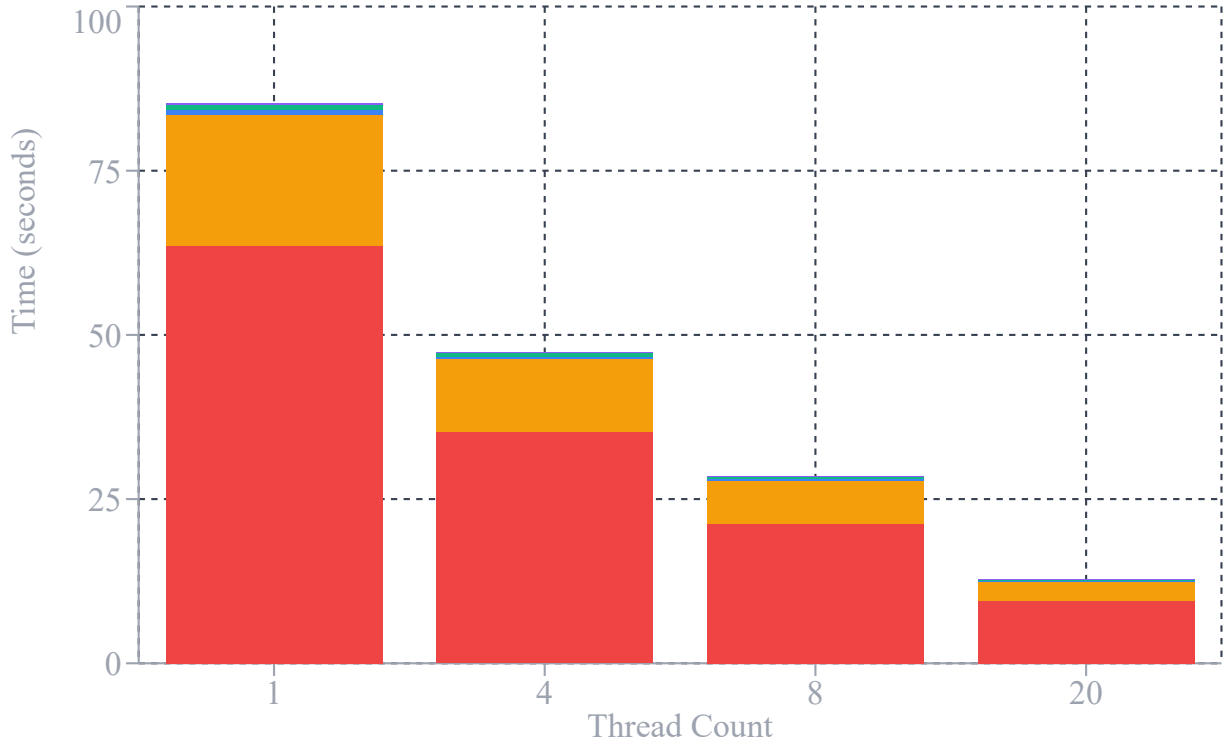
6.1.4 Testcase 4 analysis :

- Total Execution Time :

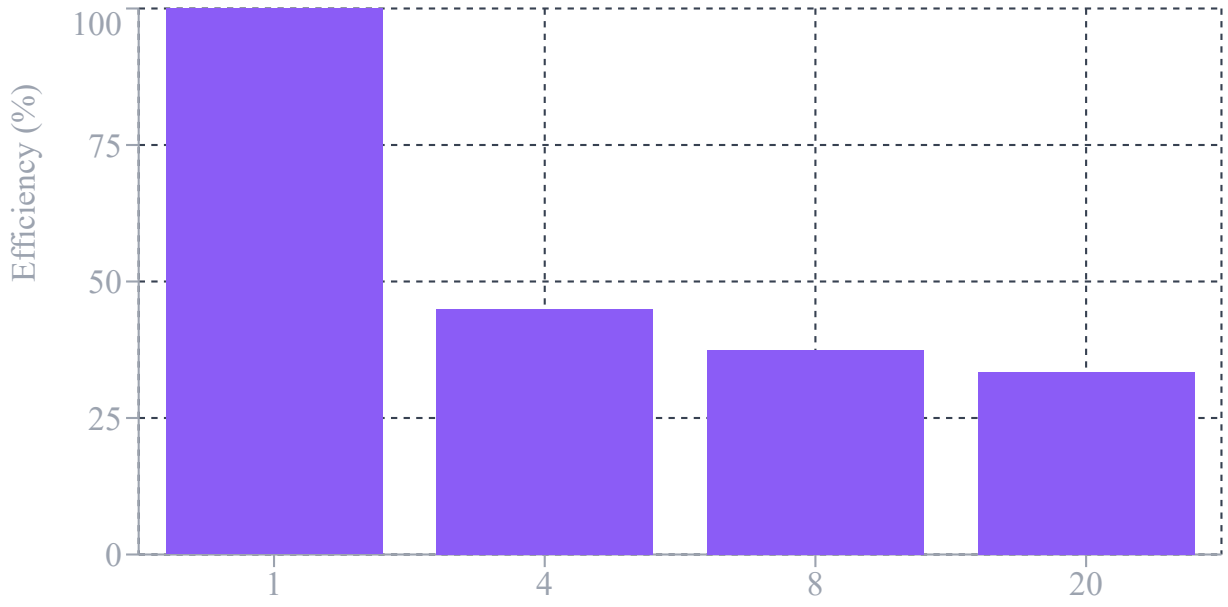


- Speedup Analysis :

- Stagewise Analysis :



- Efficiency :



6.2 Scalability :

- From the analysis above, we can see that for large inputs, there is almost a linear speedup
- Even though not being perfectly theoretical, we can notice the improvement, which proves that the program is scalable

- In the smaller inputs there is performance degradation instead of improvement due to the following :
  - Memory bandwidth
  - Sync overhead
  - False sharing
- So ultimately as input size tends to larger sizes, the time complexity has a negative linear growth at smaller inputs and starts a linear growth and settles at a sub linear growth at extremely high inputs
- Each of the mentioned issues can be resolved using the following methods :
Factor	Impact	Mitigation Strategy
Memory Bandwidth	High	Cache blocking, data layout opt.
Synchronization Overhead	Medium	Reduced critical section scope
False Sharing	Low	Thread-local buffers
Hyperthreading Overhead	Medium	Performance mode, physical cores
- Hyperthreading can have performance gain, due it allowing a single core to act as two logical processors
  - It can give a performance gain of 10-20% depending on how well the program is written (this is taken off observations made by other programs in a general opinion)
  - wrt to The terrain generation, the Hydraulic Erosion is the most benefitted stage, due to its memory bound and independent simulation of the dorplets

6.3 Challenges Encountered and Solutions :

6.3.1 PRNG (Pseudo random number generation) :

- The standard library used to generate numbers in c++ is not thread safe which causes issues in stages where a random number is required from a random number already generated and the the seed is set in global space with no local space options
- It uses a very low quality linear congruential generator which has only 31 bits and a small state and the generation sequence is quickly repeated
- **SOLUTION** :
  - We have implemented our own PRNG which is thread safe and uses a XOR-SHIFT Algorithm which solves both the problems faced
  - The code has been showcased in the section `4.2.1`

6.3.2 Erosion Buffer Accumulation :

- There causes the following issues :
  - Race conditions
  - Data inconsistency
  - Non deterministic erosion and sedimentation
  - Critical section'ing the whole thing becomes to expensive due to the number of droplets that are simulated
  - False sharing
- This is due to multiple droplets trying to access the same grid cell in the heightmap, for both eroding and sedimentation
- **SOLUTION** :
  - A thread safe and thread local accumulation algorithm is implemented
  - Then once all the threads are finished, a reduction happens which takes in all the local buffers to finally generate the final output :
  - The code has been showcased in the section `4.4.2`
  - This solves the sync and the false sharing problem
- Other issues are common and minute issues which are resolved by common methods :
  - BFS queue sync issues : Solved by setting each level to one thread
  - Irregular Object Placements : Solved by having Biome level configs, which is then used along with dynamic scheduling with chunks
  - Dynamic Droplet Path Lengths in the Erosion : Solved by Dynamic scheduling the droplet simulations which removes the imbalance of droplets exiting or finishing the job too quickly causing stagnant stages

7. CONCLUSION AND FUTURE SCOPE :

7.1 Potential Future improvements :

- Using SIMD vectorization will allow the following improvements:
  - Heightmap gen
  - Climate/ Moisture map gen
  - Biome scoring and classification
  - Implementation :
- BFS optimization

- Instead of using the normal algoithm an optimized version can be used for this case
- Using top down BFS for early levels
- Using bottom-up BFS for larger levels
- This will reduce edge examination by around 2-3x

CUDA Usage :

- Using a GPU to compute brings very noticeable and extreme improvements, due to the whole process being extremely parallel
- Due to having thousands of cores, the small computes for each task can be done concurrently for a much larger data set compared to the CPU calculations
  - Each grid cell can be taken care by one GPU thread and wiht thousands of it, it solves the 300Million - 3.2Billion operations at a very fast rate
  - GPU Memory bandwidths are better than CPU, which is important because terrain generation involves a lot of memory read/write and neighbour sampling issues
- Pseudocode Implementation is as follows :

```
KERNEL generateHeightmapKernel(device float[] height, device VoronoiPlate[] plates, in
{
    int x = blockIdx.x * blockDim.x + threadIdx.x
    int y = blockIdx.y * blockDim.y + threadIdx.y

    if (x >= W or y >= H) return

    float minDist = +INF
    float secondDist = +INF
    VoronoiPlate* nearest = NULL

    for i from 0 to numPlates-1:
        float dx = x - plates[i].x
        float dy = y - plates[i].y
        float dist = sqrt(dx*dx + dy*dy)
        if dist < minDist:
            secondDist = minDist
            minDist = dist
            nearest = &plates[i]
        else if dist < secondDist:
            secondDist = dist

    float vor = computeVoronoiHeight(minDist, secondDist, nearest)
```

```
    float fbm = perlinFBM_GPU(x, y)

    float h = (1.0 - fbmBlend) * vor + fbmBlend * fbm
    height[y * W + x] = tanh(h * 1.2) * 0.5 + 0.5
}

KERNEL erodeDropletKernel(device float[] height, device double[] erodeBuffer, device d
{
    int dropletId = blockIdx.x * blockDim.x + threadIdx.x
    if dropletId >= dropletCount return

    int globalId = dropletStart + dropletId

    RNGState rng
    rng.init(seed = params.seed XOR globalId)

    float x = rng.uniform() * (W - 1)
    float y = rng.uniform() * (H - 1)
    float dirX = 0.0, dirY = 0.0
    float speed = params.initSpeed
    float water = params.initWater
    float sediment = 0.0

    for step from 0 to params.maxSteps-1:
        float h = textureSample(height, x, y)
        vec2 grad = computeGradientFromTexture(height, x, y)

        dirX = dirX * params.inertia - grad.x * (1.0 - params.inertia)
        dirY = dirY * params.inertia - grad.y * (1.0 - params.inertia)
        float len = sqrt(dirX*dirX + dirY*dirY)
        if len > 0:
            dirX /= len
            dirY /= len

        x += dirX * params.stepSize
        y += dirY * params.stepSize

        if x < 0 or x >= W or y < 0 or y >= H:
            break

        float newH = textureSample(height, x, y)

        float capacity = computeCapacity(speed, water, h - newH)

        if sediment > capacity:
            double deposit = params.depositRate * (sediment - capacity)
```

```
            atomicAdd(depositBuffer[int(y) * W + int(x)], deposit)
            sediment -= deposit
        else:
            double erode = params.erodeRate * (capacity - sediment)
            atomicAdd(erodeBuffer[int(y) * W + int(x)], erode)
            sediment += erode

        speed = sqrt( max(0.0, speed*speed + (newH - h) * params.gravity) )
        water *= (1.0 - params.evaporateRate)

        if water < params.minWater or speed < params.minSpeed:
            break
    end for
}


FUNCTION generateTerrainGPU(Config cfg)
{
    int W = cfg.width
    int H = cfg.height


    device float* d_height = cudaAllocFloat(W * H)
    device float* d_temp   = cudaAllocFloat(W * H)
    device float* d_moist  = cudaAllocFloat(W * H)
    device double* d_erode = cudaAllocDouble(W * H)
    device double* d_deposit = cudaAllocDouble(W * H)

    device VoronoiPlate* d_plates = cudaAllocAndCopy(plates_array)
    device ErosionParams d_params = copy(params)

    block2D = (16, 16)
    grid2D = (ceilDiv(W,16), ceilDiv(H,16))
    LAUNCH generateHeightmapKernel<<<grid2D, block2D>>>(d_height, d_plates, numPlates,

    int dropletsPerBlock = 256
    int numBlocks = ceilDiv(cfg.numDroplets, dropletsPerBlock)
    LAUNCH erodeDropletKernel<<<numBlocks, dropletsPerBlock>>>(
        d_height, d_erode, d_deposit, d_params, W, H, dropletStart = 0, dropletCount =

    LAUNCH applyAccumulationKernel<<<grid2D, block2D>>>(d_height, d_erode, d_deposit,

    host float[] h_height = hostAllocFloat(W * H)
    cudaMemcpyDeviceToHost(h_height, d_height, W * H * sizeof(float))

    cudaFree(d_height)
```

```
    cudaFree(d_temp)
    cudaFree(d_moist)
    cudaFree(d_erode)
    cudaFree(d_deposit)
    cudaFree(d_plates)

    return h_height
}
```

- The CUDA implementation having the following advantages but not limited to :
  - Instead of 20 cores simulating 20 droplets, 10000s of cores are present which improves speedup by a LOT
  - Having a memory bandwidth of around 700GB/s it is better than CPUs when it comes to reading/writing to memory, reading or requiring the neighbour grids
  - Having Texture units and Tensor cores has options of texture memory, which has hardware interpolated reads than doing it biliniear in the CPU

7.2 Future features and Enhancements :

- These features will be updated as the code expands and the feature is required

Multi resolution terrain generation :

- Generating terrain at multiple levels of details, concurrently
- Here we generate a coarse heightmap at a lower and smaller level, which will determine a SCALED DOWN result of the whole map
- Now for each high resolution, we up sample the lower level terrain, which adds a higher level of detail
- This results in :
  - Streaming open world games
  - Progressive rendering
  - Giving options to real time terrain restructuring
- Algorithm is as follows :

```
Algorithm: Generate Multi-Resolution Terrain

Input:
    levels – list of LOD levels (each with resolution, height grid, and detail factor)
```

```
Steps:
1. Generate the coarsest level:
       generateHeightmap(levels[0].height, coarseParams)

2. For each higher-resolution level i from 1 to N-1:
        Upsample the previous level's heightmap into the current level:
            upsampleBilinear(levels[i-1].height, levels[i].height)

        Generate a new detail noise layer:
            detail ← generateDetailNoise(resolution = levels[i].resolution, amplitude
        Add the detail noise to the current level's heightmap (in parallel):
            for each pixel (x, y) in current level:
                levels[i].height(x, y) ← levels[i].height(x, y) + detail(x, y)

Output:
    Multi-resolution set of heightmaps where each level is a refined,
    more detailed version of the previous one.
```

REFERENCES :

- Quinn, M. J. (2004) *Parallel Programming in C with MPI and OpenMP* McGraw-Hill Education
- Tanenbaum, A. S.& Van Steen, M. (2017) *Distributed Systems: Principles and Paradigms Pearson
- Fortune, S. (1987) "A Sweepline Algorithm for Voronoi Diagrams" *Algorithmica*
- Olsen, J. (2004) "Realtime Procedural Terrain Generation." University of Southern Denmark, Department of Mathematics and Computer Science
- Perlin noise -> Wikipedia, Blogs : `https://rtouti.github.io/graphics/perlin-noise-algorithm`
- NVIDIA's *CUDA C++ Programming Guide*

8. APPENDIX :

8.1 Serial Code and Output :

- Refer to sections 4.3 and 4.2

8.2 Parallel Code and Output :

- Refer to section 4.2

8.2.1 Heightmap Generation

```
void WorldType_Voronoi::initPlates() {
    plates_.clear();
    plates_.resize(cfg_.numPlates);
    int s = (int)cfg_.seed;

#pragma omp parallel for schedule(static)
    for (int i = 0; i < cfg_.numPlates; ++i) {
        rng_util::RNG rng(s + i);  // Different seed per thread
        VoronoiPlate p;
        p.id = i;
        p.seed = (int)rng.nextInt();
        p.x = rng.nextFloat() * (float)width_;
        p.y = rng.nextFloat() * (float)height_;
        p.height = (rng.nextFloat() * 2.0f - 1.0f) * 0.6f;
        p.scale = 0.5f + rng.nextFloat() * 1.5f;
        plates_[i] = p;
    }
}
#pragma omp parallel for collapse(2) schedule(static)
for (int y = 0; y < H; ++y) {
    for (int x = 0; x < W; ++x) {
        float vor = voronoiHeightAt(x, y);
        float fbm = fbmNoiseAt((float)x, (float)y);
        float h = (1.0f - cfg.fbmBlend) * vor + cfg.fbmBlend * fbm;
        h = std::tanh(h * 1.2f);
        height(x, y) = (h + 1.0f) * 0.5f;
    }
}
```

8.2.2 Hydraulic Erosion

```
ErosionStats runHydraulicErosion(GridFloat &heightGrid, const ErosionParams &params, G
    int W = heightGrid.width();
    int H = heightGrid.height();
```

```
    const int N = params.numDroplets;
    const int maxSteps = params.maxSteps;

    const int numThreads = omp_get_max_threads();

    vector<vector<double>> erodeBufs, depositBufs;
    erodeBufs.resize(numThreads);
    depositBufs.resize(numThreads);
    const size_t nCells = (size_t)W * (size_t)H;

    for (int t = 0; t < numThreads; ++t) {
        erodeBufs[t].assign(nCells, 0.0);
        depositBufs[t].assign(nCells, 0.0);
    }

#pragma omp parallel for schedule(static)
    for (int di = 0; di < N; ++di) {
        int tid = omp_get_thread_num();
        ll seedState = params.worldSeed;
        ll localState = seedState ^ (ll)di * 2654435761LL;
        ll seed = rng_util::splitmix(localState);
        rng_util::RNG rng(seed);

        // initialize droplet
        float x = rng.nextFloat() * (float)(W - 1);
        float y = rng.nextFloat() * (float)(H - 1);
        float dirX = 0.0f, dirY = 0.0f;
        float speed = params.initSpeed;
        float water = params.initWater;
        float sediment = 0.0f;

        for (int step = 0; step < maxSteps; ++step) {
            float heightHere, gradX, gradY;
            sampleHeightAndGradient(heightGrid, x, y, heightHere, gradX, gradY);

            // update direction
            dirX = dirX * params.inertia - gradX * (1.0f - params.inertia);
            dirY = dirY * params.inertia - gradY * (1.0f - params.inertia);
            float len = sqrtf(dirX * dirX + dirY * dirY);
            if (len == 0.0f) {
                double r = rng.nextFloat();
                double theta = r * 2.0 * 3.141592653589793;
                dirX = (float)cos(theta) * 1e-6f;
                dirY = (float)sin(theta) * 1e-6f;
                len = sqrtf(dirX * dirX + dirY * dirY);
            }
```

```
            dirX /= len;
            dirY /= len;

            // move
            x += dirX * params.stepSize;
            y += dirY * params.stepSize;

            if (x < 0.0f || x > (W - 1) || y < 0.0f || y > (H - 1)) break;

            float newHeight = sampleBilinear(heightGrid, x, y);
            float deltaH = newHeight - heightHere;

            float potential = -deltaH;  // downhill positive
            speed = sqrtf(std::max(0.0f, speed * speed + potential * params.gravity));

            float slope = std::max(1e-6f, -deltaH / params.stepSize);

            float capacity = std::max(0.0f, params.capacityFactor * speed * water * sl

            if (sediment > capacity) {
                double deposit = params.depositRate * (sediment - capacity);
                deposit = std::min<double>(deposit, sediment);
                accumulateToCellQuad(depositBufs[tid], W, H, x, y, deposit);
                sediment -= (float)deposit;
            } else {
                double delta = params.capacityFactor * (capacity - sediment);
                double erode = params.erodeRate * delta;
                erode = std::min(erode, (double)params.maxErodePerStep);
                double localHeight = newHeight;
                erode = std::min(erode, std::max(0.0, localHeight));
                if (erode > 0.0) {
                    accumulateToCellQuad(erodeBufs[tid], W, H, x, y, erode);
                    sediment += (float)erode;
                }
            }

            water *= (1.0f - params.evaporateRate);
            if (water < params.minWater) break;
            if (speed < params.minSpeed) break;
        }
    }

    ErosionStats stats;
    vector<double> finalErode(nCells, 0.0), finalDeposit(nCells, 0.0);
#pragma omp parallel for schedule(static)
    for (int t = 0; t < numThreads; ++t) {
```

```
        const auto &eb = erodeBufs[t];
        const auto &db = depositBufs[t];
        for (size_t i = 0; i < nCells; ++i) {
            finalErode[i] += eb[i];
            finalDeposit[i] += db[i];
        }
    }

#pragma omp parallel for collapse(2) schedule(static)
    for (int y = 0; y < H; ++y) {
        for (int x = 0; x < W; ++x) {
            size_t idx = (size_t)y * W + x;
            double delta = finalDeposit[idx] - finalErode[idx];
            stats.totalEroded += finalErode[idx];
            stats.totalDeposited += finalDeposit[idx];
            double newH = (double)heightGrid(x, y) + delta;
            if (newH < 0.0) newH = 0.0;
            heightGrid(x, y) = (float)newH;
        }
    }

    if (outEroded) {
        outEroded->resize(W, H);
#pragma omp parallel for collapse(2) schedule(static)
        for (int y = 0; y < H; ++y)
            for (int x = 0; x < W; ++x) (*outEroded)(x, y) = (float)finalErode[(size_t
    }
    if (outDeposited) {
        outDeposited->resize(W, H);
#pragma omp parallel for collapse(2) schedule(static)
        for (int y = 0; y < H; ++y)
            for (int x = 0; x < W; ++x) (*outDeposited)(x, y) = (float)finalDeposit[(s
    }

    stats.appliedDroplets = N;
    return stats;
}
}  // namespace erosion
```

8.2.3 River Generation

```
RiverGenerator::RiverGenerator(int width, int height, const std::vector<float>& height
    : W(width), H(height), Hmap(heightmap), Biomes(biome_map) {
```

```cpp
    FlowDir.assign(W * H, -1);
    FlowAccum.assign(W * H, 0.0f);
    RiverMask.assign(W * H, 0);
}

void RiverGenerator::run(const RiverParams& params) {
    computeFlowDirection();
    computeFlowAccumulation();
    extractRivers(params);
    carveRivers(params);
}

const std::vector<uint8_t>& RiverGenerator::getRiverMask() const { return RiverMask; }
const std::vector<float>& RiverGenerator::getHeightmap() const { return Hmap; }

void RiverGenerator::computeFlowDirection() {
    const int dx[8] = {1, 1, 0, -1, -1, -1, 0, 1};
    const int dy[8] = {0, 1, 1, 1, 0, -1, -1, -1};
    const float diagDist = std::sqrt(2.0f);

#pragma omp parallel for schedule(static)
    for (int y = 0; y < H; ++y) {
        for (int x = 0; x < W; ++x) {
            int i = idx(x, y);
            float h = Hmap[i];
            int best_n = -1;
            float best_drop = 0.0f;
            for (int k = 0; k < 8; ++k) {
                int nx = x + dx[k];
                int ny = y + dy[k];
                if (nx < 0 || nx >= W || ny < 0 || ny >= H) continue;
                int ni = idx(nx, ny);
                float nh = Hmap[ni];
                float dist = (k % 2 == 0) ? 1.0f : diagDist;
                float drop = (h - nh) / dist;
                if (drop > best_drop) {
                    best_drop = drop;
                    best_n = ni;
                }
            }
            FlowDir[i] = best_n;  // -1 if none
        }
    }
}

void RiverGenerator::computeFlowAccumulation() {
```

```
    int N = W * H;
    std::vector<int> order(N);
#pragma omp parallel for schedule(static)
    for (int i = 0; i < N; ++i) order[i] = i;
    std::sort(order.begin(), order.end(), [&](int a, int b) { return Hmap[a] > Hmap[b]

    std::fill(FlowAccum.begin(), FlowAccum.end(), 1.0f);

    for (int id = 0; id < N; ++id) {
        int i = order[id];
        int d = FlowDir[i];
        if (d != -1) {
            FlowAccum[d] += FlowAccum[i];
        }
    }
}

void RiverGenerator::extractRivers(const RiverParams& params) {
    int N = W * H;
#pragma omp parallel for schedule(static)
    for (int i = 0; i < N; ++i) {
        RiverMask[i] = (FlowAccum[i] >= params.flow_accum_threshold) ? 255 : 0;
    }
}

void RiverGenerator::carveRivers(const RiverParams& params) {
    int N = W * H;
    const int dx[4] = {1, -1, 0, 0};
    const int dy[4] = {0, 0, 1, -1};
    std::vector<int> dist(N, INT_MAX);
    std::queue<int> q;
#pragma omp parallel for schedule(static)
    for (int i = 0; i < N; ++i) {
        if (RiverMask[i]) {
            dist[i] = 0;
#pragma omp critical
            q.push(i);
        }
    }
    while (!q.empty()) {
        int cur = q.front();
        q.pop();
        int cx = cur % W;
        int cy = cur / W;
        for (int k = 0; k < 4; ++k) {
            int nx = cx + dx[k];
```

```
            int ny = cy + dy[k];
            if (nx < 0 || nx >= W || ny < 0 || ny >= H) continue;
            int ni = idx(nx, ny);
            if (dist[ni] > dist[cur] + 1) {
                dist[ni] = dist[cur] + 1;
                q.push(ni);
            }
        }
    }

#pragma omp parallel for schedule(static)
    for (int i = 0; i < N; ++i) {
        if (dist[i] == INT_MAX) continue;
        float flow_here = FlowAccum[i];
        double width = params.width_multiplier * std::sqrt(std::max(1.0f, flow_here));
        double depth = std::clamp(params.min_channel_depth + (params.max_channel_depth
                                  params.min_channel_depth, params.max_channel_depth);
        double falloff = 1.0;
        if (dist[i] > 0) {
            double d = (double)dist[i];
            double radius = std::max(1.0, width);
            falloff = std::max(0.0, 1.0 - (d / (radius * 1.5)));
        }
        double delta = depth * falloff;
        Hmap[i] = float(Hmap[i] - delta);
    }
}
```

8.2.4 Biome Classification and Object Placement

```
namespace biome {
static inline bool classifyBiomeMap(const GridFloat& heightGrid, const GridFloat& temp
                                    const std::vector<BiomeDef>& defs, GridBiome& outB
    const int W = heightGrid.width();
    const int H = heightGrid.height();
    if (tempGrid.width() != W || tempGrid.height() != H) return false;
    if (moistGrid.width() != W || moistGrid.height() != H) return false;
    if (outBiomeGrid.width() != W || outBiomeGrid.height() != H) return false;
    if (riverMaskGrid && (riverMaskGrid->width() != W || riverMaskGrid->height() != H)

    std::vector<int> oceanMask(W * H, 0);
    std::vector<int> lakeMask(W * H, 0);
#pragma omp parallel for collapse(2) schedule(static)
```

```
    for (int y = 0; y < H; ++y) {
        for (int x = 0; x < W; ++x) {
            float e = heightGrid(x, y);
            int idx = y * W + x;
            if (e < opts.oceanHeightThreshold)
                oceanMask[idx] = 1;
            else if (e < opts.lakeHeightThreshold)
                lakeMask[idx] = 1;
        }
    }

    std::vector<int> riverMask(W * H, 0);
    if (riverMaskGrid) {
#pragma omp parallel for collapse(2) schedule(static)
        for (int y = 0; y < H; ++y)
            for (int x = 0; x < W; ++x) riverMask[y * W + x] = ((*riverMaskGrid)(x, y)
    }

    std::vector<int> nearCoast(W * H, 0), nearRiver(W * H, 0);
    computeNearMaskFromSources(W, H, oceanMask, opts.coastDistanceTiles, nearCoast);
    if (!riverMask.empty()) computeNearMaskFromSources(W, H, riverMask, opts.riverDist

    std::vector<float> slopeMap;
    computeSlopeMap(W, H, [&](int x, int y) -> float { return heightGrid(x, y); }, slo

    std::vector<Biome> chosen(W * H, Biome::Unknown);
#pragma omp parallel for collapse(2)
    for (int y = 0; y < H; ++y) {
        for (int x = 0; x < W; ++x) {
            int idx = y * W + x;
            float e = heightGrid(x, y);
            float t = tempGrid(x, y);
            float m = moistGrid(x, y);
            float s = slopeMap[idx];
            bool nc = (nearCoast[idx] != 0);
            bool nr = (nearRiver[idx] != 0) || (riverMask[idx] != 0);
            Biome b = chooseBestBiome(defs, e, t, m, s, nc, nr, opts);
            chosen[idx] = b;
        }
    }

    if (opts.smoothingIterations > 0) {
        majorityFilter<Biome>(W, H, chosen, opts.smoothingIterations);
    }

#pragma omp parallel for collapse(2) schedule(static)
```

```
    for (int y = 0; y < H; ++y) {
        for (int x = 0; x < W; ++x) {
            outBiomeGrid(x, y) = chosen[y * W + x];
        }
    }
    return true;
}

}  // namespace biome
```

8.3 Time Measurement Script :

```python
import subprocess
import json
import time
from pathlib import Path
import statistics
import sys

class Benchamrker:
    def __init__(self):
        self.resolutions = [2048]
        self.thread_counts = [1, 2, 4, 8, 14, "MAX"]
        self.runs_per_config = 3
        self.output_dir = Path("benchmark_output")

    def get_max_threads(self) -> int:
        import os
        return os.cpu_count()

    def run_benchmark_with_images(self, width: int, height: int, threads: int, run_id:
        cmd = [
            "build/bin/terrain-gen-benchmark.exe",
            str(width), str(height), str(threads), str(run_id), str(seed)
        ]


        start_time = time.time()
        result = subprocess.run(cmd, capture_output=True, text=True, timeout=3600)  #
        end_time = time.time()

        benchmark_data = json.loads(result.stdout)
        benchmark_data["wall_clock_time"] = end_time - start_time
```

```
        return benchmark_data

    def calculate_averages(self, results: list) -> dict:
        if not results:
            return {}

        avg_result = results[0].copy()

        numeric_fields = ["total_time", "peak_memory_kb", "wall_clock_time"]
        for field in numeric_fields:
            if field in avg_result:
                values = [r.get(field, 0) for r in results]
                avg_result[field] = statistics.mean(values)

        if "stage_times" in avg_result:
            stage_times = {}
            for stage in avg_result["stage_times"].keys():
                values = [r.get("stage_times", {}).get(stage, 0) for r in results]
                stage_times[stage] = statistics.mean(values)
            avg_result["stage_times"] = stage_times

        avg_result["num_runs"] = len(results)
        avg_result["std_dev_total_time"] = statistics.stdev([r.get("total_time", 0) fo

        return avg_result

    def get_slowest_stage(self, result: dict) -> str:
        """Get the name of the slowest processing stage."""
        stage_times = result.get("stage_times", {})
        if not stage_times:
            return "Unknown"

        slowest_stage = max(stage_times.items(), key=lambda x: x[1])
        return slowest_stage[0].replace('_', ' ').title()

    def run_resolution_benchmark(self, resolution: int):
        print(f"\n{'='*80}")
        print(f"Processing {resolution}x{resolution}")
        print(f"{'='*80}")

        resolution_dir = self.output_dir / f"{resolution}x{resolution}"
        resolution_dir.mkdir(parents=True, exist_ok=True)

        all_results = {}

        for thread_count in self.thread_counts:
```

```
            actual_threads = self.get_max_threads() if thread_count == "MAX" else thre
            thread_key = f"{actual_threads}_threads"

            print(f"\nTesting with {actual_threads} threads...")

            run_results = []
            for run_id in range(1, self.runs_per_config + 1):
                print(f"  Run {run_id}/{self.runs_per_config}...")

                result = self.run_benchmark_with_images(
                    resolution, resolution, actual_threads, run_id
                )

                if result:
                    run_results.append(result)
                    time.sleep(1)

            if run_results:
                avg_result = self.calculate_averages(run_results)
                all_results[thread_key] = avg_result

                result_file = resolution_dir / f"{thread_key}_result.json"
                with open(result_file, 'w') as f:
                    json.dump(avg_result, f, indent=2)

        return all_results

    def run_complete_benchmark(self):
        print("Starting comprehensive benchmark documentation generation...")
        print(f"Resolutions: {self.resolutions}")
        print(f"Thread counts: {self.thread_counts}")
        print(f"Runs per configuration: {self.runs_per_config}")
        print(f"Total test cases: {len(self.resolutions) * len(self.thread_counts) * s

        self.output_dir.mkdir(exist_ok=True)

        all_results = {}

        for resolution in self.resolutions:
            resolution_results = self.run_resolution_benchmark(resolution)
            all_results[resolution] = resolution_results

def main():
    if not Path("build/bin/terrain-gen-benchmark.exe").exists():
        return 1
```

```
    runner = Benchamrker()
    runner.run_complete_benchmark()

if __name__ == "__main__":
    main()
```