# GRMHD Torus Implementation Plan for Kerr Black Hole Renderer

## Overview

Transform your geometric accretion disk into a dynamic GRMHD torus simulation with realistic plasma physics, magnetic field dynamics, and thermodynamic properties.

## Phase 1: Core GRMHD Framework (C Backend)

### 1.1 Grid Structure

c

```c
typedef struct {
    int nr, ntheta, nphi;  // Grid dimensions (r, θ, φ)
    double r_min, r_max;   // Radial bounds
    double *r, *theta, *phi;  // Coordinate arrays
    double *dr, *dtheta, *dphi;  // Grid spacing
} Grid;

typedef struct {
    // Primitive variables
    double ***rho;      // Rest mass density
    double ***u;        // Internal energy density
    double ***v_r;      // Radial velocity
    double ***v_theta;  // Polar velocity
    double ***v_phi;    // Azimuthal velocity
    double ***B_r;      // Radial magnetic field
    double ***B_theta;  // Polar magnetic field
    double ***B_phi;    // Azimuthal magnetic field

    // Derived quantities
    double ***pressure; // Gas pressure
    double ***temperature; // Temperature
    double ***beta;     // Plasma beta (P_gas/P_mag)
    double ***sigma;    // Magnetization parameter
} FluidState;
```

### 1.2 Metric and Connection Coefficients

c

```c
// Kerr metric in Boyer-Lindquist coordinates
void kerr_metric(double r, double theta, double a, double metric[4][4]);
void kerr_connection(double r, double theta, double a, double gamma[4][4][4]);
void kerr_sqrt_g(double r, double theta, double a, double *sqrt_g);
```

### 1.3 Equation of State

```c
typedef struct {
    double gamma;  // Adiabatic index (typically 4/3 for relativistic gas)
    double K;      // Polytropic constant
} EOS;

double pressure_from_rho_u(double rho, double u, EOS *eos);
double temperature_from_rho_u(double rho, double u, EOS *eos);
double sound_speed(double rho, double u, EOS *eos);
```

## Phase 2: Initial Conditions - Fishbone-Moncrief Torus

### 2.1 Hydrodynamic Equilibrium

```c
typedef struct {
    double r_in;     // Inner edge of torus
    double r_max;    // Radius of pressure maximum
    double rho_max;  // Maximum density
    double beta_min; // Minimum plasma beta
} TorusParams;

// Set up initial torus in hydrostatic equilibrium
void initialize_fishbone_moncrief_torus(FluidState *state, Grid *grid,
                                        TorusParams *params, double a);
```

### 2.2 Magnetic Field Initialization

```c
// Initialize poloidal magnetic field loops
void initialize_magnetic_field(FluidState *state, Grid *grid,
                               double field_strength);

// Ensure div(B) = 0 constraint
void enforce_divergence_free(FluidState *state, Grid *grid);
```

## Phase 3: GRMHD Evolution

### 3.1 Conservative Variables

c

```c
typedef struct {
    double ***D;      // Conserved density
    double ***S_r;    // Conserved momentum (radial)
    double ***S_theta; // Conserved momentum (polar)
    double ***S_phi;  // Conserved momentum (azimuthal)
    double ***tau;    // Conserved energy
    double ***B_r;    // Magnetic field (radial)
    double ***B_theta; // Magnetic field (polar)
    double ***B_phi;  // Magnetic field (azimuthal)
} ConservedVars;

// Convert between primitive and conservative variables
void prim_to_cons(FluidState *prim, ConservedVars *cons, Grid *grid, double a);
void cons_to_prim(ConservedVars *cons, FluidState *prim, Grid *grid, double a);
```

## 3.2 Flux Calculations

c

```c
// Calculate fluxes for GRMHD equations
void calculate_fluxes(FluidState *state, ConservedVars *cons,
                      Grid *grid, double a,
                      double ***flux_r[8], double ***flux_theta[8],
                      double ***flux_phi[8]);

// Riemann solver for shock capturing
void hll_riemann_solver(double *prim_L, double *prim_R, double *flux,
                        int direction, double r, double theta, double a);
```

## 3.3 Time Evolution

c

```c
// Main evolution step using method of lines
void evolve_grmhd(FluidState *state, ConservedVars *cons, Grid *grid,
                  double dt, double a, EOS *eos);

// Adaptive timestep based on CFL condition
double calculate_timestep(FluidState *state, Grid *grid, double cfl_factor);
```

# Phase 4: Radiative Transfer Integration

## 4.1 Emission and Absorption

```c
typedef struct {
    double ***j_nu;     // Emission coefficient
    double ***alpha_nu; // Absorption coefficient
    double ***rho_nu;   // Scattering coefficient
} RadiativeData;

// Synchrotron emission from relativistic electrons
void calculate_synchrotron_emission(FluidState *state, RadiativeData *rad,
                                    double frequency, Grid *grid);

// Thermal bremsstrahlung
void calculate_bremsstrahlung(FluidState *state, RadiativeData *rad,
                              double frequency, Grid *grid);
```

### 4.2 Ray Tracing Integration

```c
// Sample emission along geodesic
double sample_emission_along_ray(vec4 pos, vec4 momentum,
                                FluidState *state, RadiativeData *rad,
                                Grid *grid, double frequency);

// Interpolate fluid quantities to arbitrary position
void interpolate_fluid_state(vec4 pos, FluidState *state, Grid *grid,
                            double *rho, double *u, double *B_mag,
                            vec3 *velocity, vec3 *B_field);
```

## Phase 5: OpenGL Shader Integration (macOS Compatible)

### 5.1 3D Texture Management

c

```c
// macOS-optimized texture creation and updates
typedef struct {
    GLuint density_texture;
    GLuint temperature_texture;
    GLuint magnetic_field_texture;
    GLuint velocity_texture;
    GLuint emission_texture;

    // Texture dimensions
    int nr, ntheta, nphi;

    // Buffer objects for efficient updates
    GLuint pbo_density;
    GLuint pbo_temperature;
    GLuint pbo_magnetic;
    GLuint pbo_velocity;
    GLuint pbo_emission;
} FluidTextures;

// Initialize 3D textures with optimal format for M2 Pro
FluidTextures* init_fluid_textures(int nr, int ntheta, int nphi) {
    FluidTextures* textures = malloc(sizeof(FluidTextures));
    textures->nr = nr; textures->ntheta = ntheta; textures->nphi = nphi;

    // Generate textures
    glGenTextures(1, &textures->density_texture);
    glGenTextures(1, &textures->temperature_texture);
    glGenTextures(1, &textures->magnetic_field_texture);
    glGenTextures(1, &textures->velocity_texture);
    glGenTextures(1, &textures->emission_texture);

    // Generate PBOs for asynchronous updates
    glGenBuffers(1, &textures->pbo_density);
    glGenBuffers(1, &textures->pbo_temperature);
    glGenBuffers(1, &textures->pbo_magnetic);
    glGenBuffers(1, &textures->pbo_velocity);
    glGenBuffers(1, &textures->pbo_emission);

    // Setup 3D textures with R32F format (optimal for M2 Pro)
    setup_3d_texture(textures->density_texture, GL_R32F, nr, ntheta, nphi);
    setup_3d_texture(textures->temperature_texture, GL_R32F, nr, ntheta, nphi);
    setup_3d_texture(textures->magnetic_field_texture, GL_RGB32F, nr, ntheta, nphi);
    setup_3d_texture(textures->velocity_texture, GL_RGB32F, nr, ntheta, nphi);
    setup_3d_texture(textures->emission_texture, GL_RGBA32F, nr, ntheta, nphi);

    return textures;
```

```
}

// Efficient texture update using PBOs
void update_fluid_texture(GLuint texture, GLuint pbo, float* data,
                          int nr, int ntheta, int nphi, GLenum format) {
    size_t data_size = nr * ntheta * nphi * sizeof(float) *
                       (format == GL_RGB32F ? 3 : (format == GL_RGBA32F ? 4 : 1));

    // Bind PBO and upload data
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);
    glBufferData(GL_PIXEL_UNPACK_BUFFER, data_size, data, GL_STREAM_DRAW);

    // Update texture from PBO
    glBindTexture(GL_TEXTURE_3D, texture);
    glTexSubImage3D(GL_TEXTURE_3D, 0, 0, 0, 0, nr, ntheta, nphi,
                    format == GL_RGB32F ? GL_RGB : (format == GL_RGBA32F ? GL_RGBA : G
                    GL_FLOAT, 0);

    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);
}
```

### 5.2 Compute Shader Support (OpenGL 4.3+)

```glsl
#version 430 core

// Compute shader for radiative transfer calculations
layout(local_size_x = 8, local_size_y = 8, local_size_z = 8) in;

layout(binding = 0, r32f) uniform image3D density_image;
layout(binding = 1, r32f) uniform image3D temperature_image;
layout(binding = 2, rgba32f) uniform image3D emission_image;

uniform float u_frequency;
uniform float u_time_step;

void main() {
    ivec3 coords = ivec3(gl_GlobalInvocationID);

    float density = imageLoad(density_image, coords).r;
    float temperature = imageLoad(temperature_image, coords).r;

    // Calculate synchrotron emission
    float emission = calculate_synchrotron_emission(density, temperature, u_frequency)

    // Update emission texture
    imageStore(emission_image, coords, vec4(emission, 0.0, 0.0, 1.0));
}
```

### 5.3 Enhanced Fragment Shader

glsl

```glsl
// Updated fragment shader with fluid sampling
uniform sampler3D u_density_texture;
uniform sampler3D u_temperature_texture;
uniform sampler3D u_magnetic_field_texture;
uniform sampler3D u_velocity_texture;
uniform sampler3D u_emission_texture;

// Grid parameters
uniform vec3 u_grid_min;  // (r_min, theta_min, phi_min)
uniform vec3 u_grid_max;  // (r_max, theta_max, phi_max)
uniform ivec3 u_grid_size; // (nr, ntheta, nphi)
uniform float u_observation_frequency;

// Convert Kerr-Schild position to grid coordinates
vec3 pos_to_grid_coords(vec4 pos) {
    float r = rFromCoords(pos);
    float theta = acos(clamp(pos.w / r, -1.0, 1.0));
    float phi = atan(pos.z, pos.y);

    // Normalize to [0,1] range
    return vec3(
        (r - u_grid_min.x) / (u_grid_max.x - u_grid_min.x),
        (theta - u_grid_min.y) / (u_grid_max.y - u_grid_min.y),
        (phi - u_grid_min.z + PI) / (2.0 * PI)  // phi: [-π,π] -> [0,1]
    );
}

// Sample fluid properties with bounds checking
vec4 safe_sample_3d(sampler3D tex, vec3 coords) {
    if (any(lessThan(coords, vec3(0.0))) || any(greaterThan(coords, vec3(1.0)))) {
        return vec4(0.0);
    }
    return texture(tex, coords);
}

// Enhanced ray tracing with fluid integration
vec3 trace_kerr_ray_with_fluid(vec3 dir, vec4 camPos, mat4 axes) {
    vec4 pos = camPos;
    vec4 dir4D = -axes[0] + vec4(0.0, dir.x, dir.y, dir.z);
    vec4 p = metric(pos) * dir4D;

    vec3 total_emission = vec3(0.0);
    float optical_depth = 0.0;
    bool captured = false;
    vec4 final_pos;
```

```
for (int i = 0; i < u_max_steps; i++) {
    vec4 last_pos = pos;
    transportStep(pos, p);

    // Calculate step length for integration
    float step_length = length(pos.yzw - last_pos.yzw);

    // Sample fluid properties at current position
    vec3 grid_coords = pos_to_grid_coords(pos);
    float density = safe_sample_3d(u_density_texture, grid_coords).r;
    float temperature = safe_sample_3d(u_temperature_texture, grid_coords).r;
    vec3 B_field = safe_sample_3d(u_magnetic_field_texture, grid_coords).rgb;
    vec3 velocity = safe_sample_3d(u_velocity_texture, grid_coords).rgb;
    vec4 emission_data = safe_sample_3d(u_emission_texture, grid_coords);

    if (density > 0.0) {
        // Calculate Doppler factor
        float doppler_factor = calculate_doppler_boost(p, velocity, pos);

        // Emission coefficient (already calculated in compute shader or CPU)
        float j_nu = emission_data.r * doppler_factor * doppler_factor * doppler_f

        // Absorption coefficient (simplified)
        float alpha_nu = density * temperature * 1e-10; // Simplified model

        // Radiative transfer step
        float source_function = j_nu / max(alpha_nu, 1e-10);
        float dtau = alpha_nu * step_length;

        // Formal solution of radiative transfer equation
        if (dtau > 1e-6) {
            float exponential = exp(-dtau);
            total_emission += source_function * (1.0 - exponential) * exp(-optical_
            optical_depth += dtau;
        }
    }

    if (stopCondition(pos)) {
        float r = rFromCoords(pos);
        captured = r < M + sqrt(M*M - a*a);
        break;
    }
    final_pos = pos;
}

if (captured) {
    return total_emission;
```

```
    } else {
        // Sample skybox and add to emission
        vec4 out_dir = inverse(metric(final_pos)) * p;
        vec3 cube_dir = normalize(vec3(-out_dir.y, out_dir.w, -out_dir.z));
        vec3 background = sample_skybox(cube_dir);

        return total_emission + background * exp(-optical_depth);
    }
}

// Calculate relativistic Doppler boost
float calculate_doppler_boost(vec4 photon_momentum, vec3 fluid_velocity, vec4 pos) {
    // Convert 4-momentum to 3-momentum in fluid frame
    mat4 g = metric(pos);
    vec4 u_fluid = vec4(1.0, fluid_velocity) / sqrt(-dot(g * vec4(1.0, fluid_velocity)

    // Doppler factor: nu_observed = nu_emitted * (1 + v·n)
    return 1.0 / max(dot(photon_momentum, u_fluid), 0.1);
}
```

## Phase 6: macOS M2 Pro Optimizations

### 6.1 Metal Performance Shaders Integration

```c
// Metal compute kernels for GRMHD evolution
#include <Metal/Metal.h>
#include <MetalPerformanceShaders/MetalPerformanceShaders.h>

typedef struct {
    id<MTLDevice> device;
    id<MTLCommandQueue> commandQueue;
    id<MTLComputePipelineState> grmhdPipeline;
    id<MTLComputePipelineState> fluxPipeline;
    id<MTLBuffer> primitiveBuffer;
    id<MTLBuffer> conservativeBuffer;
    id<MTLBuffer> fluxBuffer;
} MetalGRMHD;

// Initialize Metal compute pipeline
MetalGRMHD* init_metal_grmhd(int nr, int ntheta, int nphi);

// Execute GRMHD evolution on GPU
void metal_evolve_fluid(MetalGRMHD* metal, FluidState* state, double dt);
```

### 6.2 Accelerate Framework SIMD

```c
#include <Accelerate/Accelerate.h>

// Vectorized operations using Apple's Accelerate framework
void vectorized_flux_calculation(double* prim, double* flux,
                                 int nr, int ntheta, int nphi) {
    // Use vDSP for vectorized arithmetic
    vDSP_vaddD(prim, 1, flux, 1, flux, 1, nr * ntheta * nphi);
    vDSP_vmulD(prim, 1, prim + nr*ntheta*nphi, 1, flux, 1, nr * ntheta * nphi);
}

// Matrix operations for metric calculations
void accelerate_metric_operations(double* positions, double* metrics, int count) {
    // Use BLAS for efficient matrix operations
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                count, 4, 4, 1.0, positions, 4, metrics, 4, 0.0, metrics, 4);
}
```

### 6.3 Grand Central Dispatch Parallelization

```c
#include <dispatch/dispatch.h>

// Parallel grid evolution using GCD
void parallel_grid_evolution(FluidState* state, Grid* grid, double dt) {
    dispatch_queue_t concurrent_queue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIOR

    // Parallel execution across grid zones
    dispatch_apply(grid->nr, concurrent_queue, ^(size_t i) {
        for (int j = 0; j < grid->ntheta; j++) {
            for (int k = 0; k < grid->nphi; k++) {
                evolve_zone(state, i, j, k, dt);
            }
        }
    });
}
```

## Phase 7: Validation and Calibration

### 7.1 Code Verification

- Compare with analytical solutions (Bondi accretion, Michel flow)

- Verify conservation laws (mass, energy, angular momentum)
- Check convergence with resolution

### 7.2 Physical Validation

- Compare with established GRMHD codes (HARM, BHAC, ATHENA++)
- Validate against observational constraints
- Test with different black hole spins and torus configurations

## Implementation Timeline

**Week 1-2**: Set up basic grid structure and metric calculations **Week 3-4**: Implement Fishbone-Moncrief torus initialization **Week 5-6**: Add magnetic field initialization and GRMHD evolution **Week 7-8**: Integrate radiative transfer and emission calculations **Week 9-10**: Update shaders and GPU data transfer **Week 11-12**: Performance optimization and validation

## Key Considerations

1. **Numerical Stability**: Use flux-limited schemes and appropriate artificial viscosity
2. **Constraint Preservation**: Maintain div(B) = 0 and other physical constraints
3. **Boundary Conditions**: Implement outflow boundaries and inner excision
4. **Memory Management**: Efficient storage of 3D grid data
5. **Parallelization**: OpenMP for CPU, CUDA/OpenCL for GPU acceleration

## macOS M2 Pro Specific Implementation

### Build Configuration

makefile

```makefile
# Makefile for macOS M2 Pro
CC = clang
CFLAGS = -O3 -march=native -mtune=native -flto -ffast-math
CFLAGS += -framework OpenGL -framework CoreFoundation -framework Metal
CFLAGS += -framework MetalKit -framework Accelerate

# Libraries
LIBS = -lglfw3 -lm
INCLUDES = -I/opt/homebrew/include -I/usr/local/include

# Apple Silicon optimizations
CFLAGS += -mcpu=apple-m2 -DAPPLE_SILICON

grmhd_renderer: main.c grmhd.c fluid.c renderer.c
	$(CC) $(CFLAGS) $(INCLUDES) -o $@ $^ $(LIBS)
```

## Memory Management

c

```c
#include <sys/mman.h>
#include <libkern/OSAtomic.h>

// Unified memory allocation for GPU/CPU sharing
typedef struct {
    void* cpu_ptr;
    id<MTLBuffer> metal_buffer;
    size_t size;
    bool is_shared;
} UnifiedBuffer;

UnifiedBuffer* create_unified_buffer(size_t size, id<MTLDevice> device) {
    UnifiedBuffer* buffer = malloc(sizeof(UnifiedBuffer));

    // Create Metal buffer with shared storage
    buffer->metal_buffer = [device newBufferWithLength:size
                                               options:MTLResourceStorageModeShared];
    buffer->cpu_ptr = [buffer->metal_buffer contents];
    buffer->size = size;
    buffer->is_shared = true;

    return buffer;
}

// Lock-free atomic operations for thread safety
void atomic_update_fluid_cell(FluidState* state, int i, int j, int k,
                             double new_density) {
    // Use OSAtomic for lock-free updates
    double* target = &state->rho[i][j][k];
    double expected, desired;

    do {
        expected = *target;
        desired = new_density;
    } while (!OSAtomicCompareAndSwapDouble(expected, desired, target));
}
```

## Platform-Specific Optimizations

```c
// Take advantage of M2 Pro's wide vector units
#include <arm_neon.h>

void neon_vectorized_flux(float* input, float* output, int count) {
    for (int i = 0; i < count; i += 4) {
        float32x4_t vec = vld1q_f32(&input[i]);

        // Vectorized fluid calculations
        float32x4_t result = vmulq_f32(vec, vec);  // Example operation
        result = vaddq_f32(result, vec);

        vst1q_f32(&output[i], result);
    }
}

// Efficient cache usage for M2 Pro's cache hierarchy
void cache_optimized_grid_sweep(FluidState* state, Grid* grid) {
    const int cache_block_size = 64;  // M2 Pro cache line size

    for (int ii = 0; ii < grid->nr; ii += cache_block_size) {
        for (int jj = 0; jj < grid->ntheta; jj += cache_block_size) {
            for (int kk = 0; kk < grid->nphi; kk += cache_block_size) {
                // Process cache-friendly blocks
                int i_max = MIN(ii + cache_block_size, grid->nr);
                int j_max = MIN(jj + cache_block_size, grid->ntheta);
                int k_max = MIN(kk + cache_block_size, grid->nphi);

                for (int i = ii; i < i_max; i++) {
                    for (int j = jj; j < j_max; j++) {
                        for (int k = kk; k < k_max; k++) {
                            evolve_zone(state, i, j, k);
                        }
                    }
                }
            }
        }
    }
}
```