GPU碰撞检测系统技术方案

目标: 基于CUDA和OpenGL的高性能物理仿真与可视化系统

---

1. 项目概述

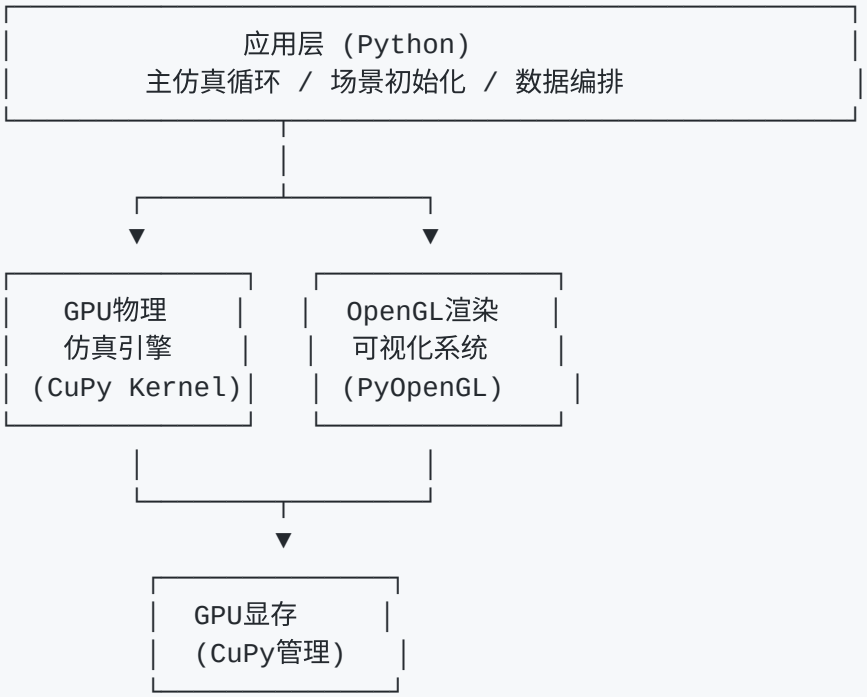
1.1 技术栈

```
物理仿真层 (GPU计算)
├── CuPy 13.6+          # GPU加速数组库 + 自定义CUDA Kernels
├── CUDA 12.x           # 底层GPU编程框架
└── NumPy               # CPU数据处理

可视化层 (实时渲染)
├── PyOpenGL 3.1.10+    # 高质量3D渲染
├── GLUT/GLFW           # 窗口与输入管理
└── imageio-ffmpeg      # 视频导出

应用层 (Python)
├── Python 3.8+
├── SciPy/Matplotlib    # 数据分析
└── OpenCV              # 图像处理
```

---

2. 系统架构设计

2.1 总体框架图



2.2 主仿真循环

```python
# 伪代码表示整体流程
def simulation_main_loop():
    while not finished:
        # 1. 物理积分 (GPU)
        integrate_velocities_and_positions()  # 更新速度、位置

        # 2. 空间分割 (GPU)
        build_uniform_grid()                  # 建立空间网格
        sort_objects_by_grid_cell()           # 排序物体

        # 3. 碰撞检测与响应 (GPU)
        for iteration in range(2):            # 多次迭代以提高稳定性
            detect_collisions()               # 广泛相位检测
            resolve_collisions()              # 冲量响应计算

        # 4. 可视化 (GPU → 屏幕)
        render_frame()                        # OpenGL渲染
        record_video_frame()                  # 可选：视频录制
```

---

3. GPU物理仿真模块

3.1 数据结构设计

刚体物理系统 (RigidBodySystem)

存储所有物体的物理属性在GPU显存中：

```
class RigidBodySystem:
    positions: [N, 3]          # 世界坐标 (float32)
    velocities: [N, 3]         # 速度向量 (float32)
    radii: [N]                 # 球体半径 (float32)
    masses: [N]                # 质量 (float32)
    restitutions: [N]          # 恢复系数/弹性 (float32)
    colors: [N, 3]             # 渲染颜色 (float32)
```

均匀网格 (UniformGrid)

用于空间加速的3D网格数据结构：

```
class UniformGrid:
    cell_size: float           # 网格单元大小
    resolution: [3]            # 网格分辨率 (x, y, z)
    cell_starts: [total_cells]    # 每个网格单元的起始位置
    cell_ends: [total_cells]      # 每个网格单元的结束位置
    sorted_indices: [N]        # 排序后的物体索引映射
```

3.2 CUDA核函数设计 (共5个)

核函数1: 计算网格哈希 (compute_grid_hash_kernel)

输入: 物体位置数组
输出: 对应的网格哈希值 (1D编码)
功能:

- 将3D位置转换为网格坐标 (gx, gy, gz)
- 将网格坐标映射为1D哈希值: `hash = gz*R_y*R_x + gy*R_x + gx`
- 边界夹紧处理

```
输入位置 → 网格坐标 → 1D哈希
(x,y,z) → (gx,gy,gz) → hash值
```

核函数2: 数据重排 (reorder_data_kernel)

输入: 原始数据数组 + 排序后的索引映射
输出: 按网格单元重新排列的数据
功能:

- 根据排序索引重新安排位置、速度、半径等数据
- 使同一网格单元的物体在内存中连续，提升缓存效率

```
原始布局: [A(cell2), B(cell1), C(cell1), D(cell2)]
排序后: [B(cell1), C(cell1), A(cell2), D(cell2)]
```

核函数3: 查找单元边界 (find_cell_start_kernel)

输入: 排序后的哈希值
输出: 每个网格单元的起始和结束索引
功能:

- 识别哈希值变化处为单元边界
- cell_starts[hash] = 该单元第一个物体的索引
- cell_ends[hash] = 该单元最后一个物体之后的索引

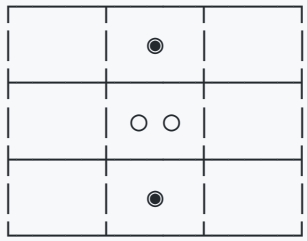
```
排序哈希: [1, 1, 1, 2, 2, 3, 3, 3, 3]
cell_starts: {1:0, 2:3, 3:5}
cell_ends:   {1:3, 2:5, 3:9}
```

核函数4: 广泛碰撞检测 (broad_phase_collision_kernel)

输入: 已排序的位置、半径、网格信息、排序索引
输出: 潜在碰撞对列表
功能:

- 遍历每个物体
- 查询物体所在网格单元周围的27个邻近单元
- 球体AABB/距离测试：`dist < r1 + r2`
- 记录潜在碰撞对

检测范围: 3×3×3邻域



核函数5: 碰撞响应 (collision_response_kernel)

输入: 碰撞对、物体属性
输出: 更新后的速度和位置 (原地修改)
功能:

- 对每个碰撞对进行冲量计算
- 应用冲量响应更新速度
- 位置修正防止穿透

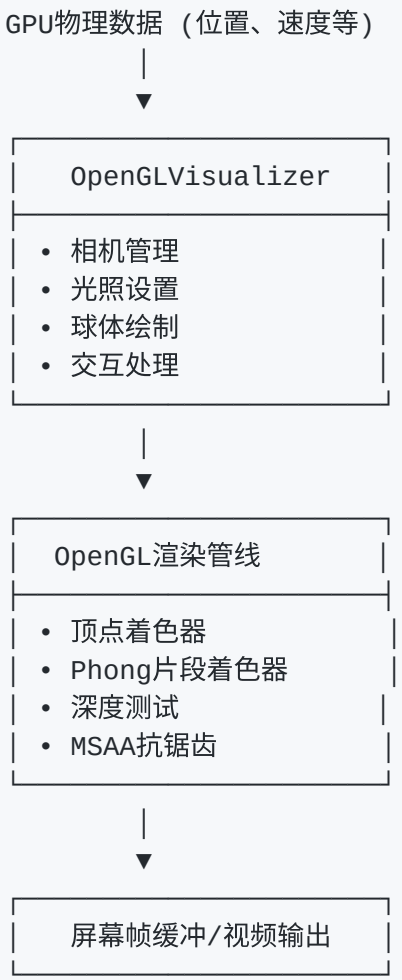
```
碰撞对 (i, j):
  ├─ 计算碰撞法线: n = (pos_j - pos_i) / |...|
  ├─ 相对速度沿法线: v_rel · n
  ├─ 冲量大小: J = -(1+e) * (v_rel · n) / (1/m_i + 1/m_j)
  ├─ 更新速度: v_i -= J·n/m_i, v_j += J·n/m_j
  └─ 位置分离: 沿法线方向修正
```

3.3 PhysicsSimulator 类结构

```
class PhysicsSimulator:
    def __init__(num_objects, world_bounds, cell_size, dt, gravity, damping):
        # 初始化GPU内存
        self.bodies = RigidBodySystem(num_objects)
        self.grid = UniformGrid(world_bounds, cell_size)

        # 碰撞对存储
        self.collision_pairs = GPU_array[max_pairs, 2]
        self.pair_count = GPU_array[1]

    def build_grid(self):
        """构建均匀网格空间结构"""
        1. compute_grid_hash_kernel()       # 计算每个物体的哈希值
        2. argsort(hashes)                  # GPU排序
        3. reorder_data_kernel()            # 按网格重排数据
        4. find_cell_start_kernel()         # 记录单元边界

    def detect_collisions(self):
        """检测碰撞"""
        return broad_phase_collision_kernel()  # 返回碰撞对数量

    def resolve_collisions(num_pairs):
        """处理碰撞响应"""
        collision_response_kernel(collision_pairs[:num_pairs])

    def integrate(self):
        """物理积分：更新速度和位置"""
        # 应用重力加速度
        # 应用阻尼衰减
        # 更新速度 v += a*dt
        # 更新位置 x += v*dt
        # 边界碰撞处理

    def step(self):
        """执行一帧仿真"""
        self.integrate()
        self.build_grid()

        for _ in range(2):  # 多次迭代
            num_pairs = self.detect_collisions()
            self.resolve_collisions(num_pairs)

        return statistics
```

4. OpenGL可视化系统

4.1 渲染架构



4.2 关键组件

球体渲染 (Sphere)

使用GLU Quadric提供高质量的光滑球面：

```
class Sphere:
    def __init__(slices=32, stacks=32):
        # slices: 经度分段数
        # stacks: 纬度分段数
        # 高分辨率保证光滑外观

    def draw(radius):
        gluSphere(quadric, radius, slices, stacks)
```

OpenGL可视化器 (OpenGLVisualizer)

```
class OpenGLVisualizer:
    def __init__(world_bounds, width, height):
        # OpenGL/GLUT初始化
        # 光源配置 (Phong着色)
        # 相机初始化

    def render(positions, radii, colors):
        """渲染单帧"""
        1. 清空帧缓冲
        2. 设置相机视图
        3. 配置Phong光照
        4. 遍历每个物体:
            - 位置变换
            - 材质设置
            - 绘制球体
        5. 绘制参考网格和坐标轴
        6. 交换双缓冲

    def handle_mouse(button, state, x, y):
        # 相机轨道旋转
        # 缩放
        # 平移

    def handle_keyboard(key):
        # 暂停/继续
        # 显示/隐藏网格
        # 重置相机
        # 切换渲染模式
```

4.3 视频录制系统

```
class OpenGLVideoRecorder:
    def __init__(output_path, width, height, fps):
        # 初始化H.264编码器
        # 1920×1080分辨率
        # 60 FPS帧率

    def capture_frame():
        """捕获当前OpenGL帧到MP4"""
        1. 从GPU读取帧数据 (glReadPixels)
        2. 翻转/格式转换
        3. 添加到编码器

    def release():
        # 完成编码、关闭文件
```

---

5. 代码模块化结构

5.1 项目文件组织

```
src/
├── __init__.py              # 公共API导出
├── rigid_body.py            # RigidBodySystem 类
├── spatial_grid.py          # UniformGrid 类
├── kernels.py               # 5个CUDA核函数
├── simulator.py             # PhysicsSimulator 类 (主仿真引擎)
├── opengl_visualizer.py     # OpenGLVisualizer + 视频录制
├── init_helper.py           # 场景初始化工具函数
└── performance.py           # 性能监测工具

examples/
└── gravity_fall.py          # 完整示例：重力下落场景

tests/
├── test_01_head_on.py         # 单元测试：两球直接碰撞
├── test_02_static_overlap.py # 单元测试：多球静止重叠
├── test_03_falling_balls.py  # 单元测试：多球下落
├── test_04_large_scale.py     # 单元测试：大规模球下落
├── test_opengl_basic.py       # 集成测试：OpenGL基本功能
└── test_physics_only.py       # 单元测试：物理纯计算
```

5.2 关键接口定义

PhysicsSimulator 公共接口

```
simulator = PhysicsSimulator(
    num_objects=500,
    world_bounds=((-20, 0, -20), (20, 40, 20)),
    cell_size=2.0,
    device_id=0,
    dt=1.0/60.0,
    gravity=(0, -9.81, 0),
    damping=0.01
)

# 场景初始化
simulator.bodies.positions = GPU_array(positions)
simulator.bodies.velocities = GPU_array(velocities)
simulator.bodies.radii = GPU_array(radii)
simulator.bodies.masses = GPU_array(masses)

# 单步仿真
stats = simulator.step()
# 返回: { 'num_collisions': int, 'frame_time_ms': float }

# 数据获取 (GPU → CPU)
positions = cp.asnumpy(simulator.bodies.positions)
velocities = cp.asnumpy(simulator.bodies.velocities)
```

OpenGLVisualizer 公共接口

```
visualizer = OpenGLVisualizer(
    world_bounds=world_bounds,
    width=1920,
    height=1080,
    title="Simulation"
)

# 设置渲染回调
def render_func():
    positions = cp.asnumpy(simulator.bodies.positions)
    radii = cp.asnumpy(simulator.bodies.radii)
    visualizer.render(positions, radii, colors, info_text)

visualizer.set_render_function(render_func)

# 启动主循环
visualizer.run()  # 阻塞直到窗口关闭
visualizer.close()
```

---

6. 典型使用流程

6.1 初始化阶段

```
# 1. 创建仿真器
sim = PhysicsSimulator(num_objects=500, ...)

# 2. 初始化物体位置（防止重叠）
positions = generate_non_overlapping_positions(...)
sim.bodies.positions = cp.asarray(positions)

# 3. 初始化物理属性
sim.bodies.radii = cp.asarray(radii)
sim.bodies.masses = cp.asarray(masses)
sim.bodies.velocities = cp.asarray(velocities)
sim.bodies.restitutions = cp.asarray(restitution)

# 4. 初始化可视化
visualizer = OpenGLVisualizer(...)
colors = generate_colors(num_objects)
```

6.2 仿真循环

```
for frame in range(num_frames):
    # GPU仿真
    stats = sim.step()

    # CPU读取数据
    positions = cp.asnumpy(sim.bodies.positions)
    radii = cp.asnumpy(sim.bodies.radii)

    # 可视化渲染
    visualizer.render(positions, radii, colors, info)

    # 可选：视频录制
    if recording:
        recorder.capture_frame()

recorder.release()
visualizer.close()
```

---

7. 设计要点与考虑

7.1 性能优化策略

1. **空间加速**：均匀网格降低碰撞检测复杂度 (O(N^2) → O(N*k))
2. **GPU并行化**：充分利用GPU多核特性 (块大小256，适配RTX 3050)
3. **内存局部性**：网格排序提升缓存命中率
4. **最小化数据传输**：尽量保持数据在GPU显存中

7.2 交互设计

- **鼠标**：轨道旋转相机、滚轮缩放、中键平移
- **键盘**：
  - `SPACE`：暂停/继续
  - `W`：线框模式
  - `G`：显示/隐藏网格
  - `R`：重置相机
  - `Esc`：退出