CEN310 Parallel Programming

Week-13 (Real-world Applications II)

Spring Semester, 2024-2025

Overview

Topics

1. Advanced Parallel Patterns
2. N-body Simulations
3. Matrix Computations
4. Big Data Processing

Objectives

- Implement complex parallel patterns
- Optimize scientific simulations
- Perform large-scale matrix operations
- Process big data efficiently

1. Advanced Parallel Patterns

Pipeline Pattern

```cpp
template<typename T>
class ParallelPipeline {
private:
    std::vector<std::thread> stages;
    std::vector<std::queue<T>> queues;
    std::vector<std::mutex> mutexes;
    std::vector<std::condition_variable> cvs;
    bool running;

public:
    ParallelPipeline(int num_stages) {
        queues.resize(num_stages - 1);
        mutexes.resize(num_stages - 1);
        cvs.resize(num_stages - 1);
        running = true;
    }

    void add_stage(std::function<void(T&)> stage_func, int stage_id) {
        stages.emplace_back([this, stage_func, stage_id]() {
            while(running) {
                T data;
                if(stage_id == 0) {
                    // First stage: produce data
                    data = produce_data();
                } else {
                    // Get data from previous stage
                    std::unique_lock<std::mutex> lock(mutexes[stage_id-1]);
                    cvs[stage_id-1].wait(lock,
                        [this, stage_id]() {
                            return !queues[stage_id-1].empty() || !running;
                        });
                    if(!running) break;
                    data = queues[stage_id-1].front();
                    queues[stage_id-1].pop();
                    lock.unlock();
                    cvs[stage_id-1].notify_one();
                }

                // Process data
                stage_func(data);

                if(stage_id < stages.size() - 1) {
                    // Pass to next stage
                    std::unique_lock<std::mutex> lock(mutexes[stage_id]);
                    queues[stage_id].push(data);
                    lock.unlock();
                    cvs[stage_id].notify_one();
                }
            }
        });
    }

    void start() {
        for(auto& stage : stages) {
            stage.join();
        }
    }

    void stop() {
        running = false;
        for(auto& cv : cvs) {
            cv.notify_all();
        }
    }
};
```

2. N-body Simulations

Barnes-Hut Algorithm

```
struct Octree {
    struct Node {
        vec3 center;
        float size;
        float mass;
        vec3 com;
        std::vector<Node*> children;
    };

    Node* root;
    float theta;

    __device__ void compute_force(vec3& pos, vec3& force, Node* node) {
        vec3 diff = node->com - pos;
        float dist = length(diff);

        if(node->size / dist < theta || node->children.empty()) {
            // Use approximation
            float f = G * node->mass / (dist * dist * dist);
            force += diff * f;
        } else {
            // Recurse into children
            for(auto child : node->children) {
                if(child != nullptr) {
                    compute_force(pos, force, child);
                }
            }
        }
    }

    __global__ void update_bodies(vec3* pos, vec3* vel, vec3* acc,
                                float dt, int n) {
        int idx = blockIdx.x * blockDim.x + threadIdx.x;
        if(idx < n) {
            vec3 force(0.0f);
            compute_force(pos[idx], force, root);
            acc[idx] = force;
            vel[idx] += acc[idx] * dt;
            pos[idx] += vel[idx] * dt;
        }
    }
};
```

3. Matrix Computations

Parallel Matrix Factorization

```
__global__ void lu_factorization(float* A, int n, int k) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if(row > k && row < n && col > k && col < n) {
        A[row * n + col] -= A[row * n + k] * A[k * n + col] / A[k * n + k];
    }
}

void parallel_lu(float* A, int n) {
    dim3 block(16, 16);
    dim3 grid((n + block.x - 1) / block.x,
              (n + block.y - 1) / block.y);

    for(int k = 0; k < n-1; k++) {
        lu_factorization<<<grid, block>>>(A, n, k);
        cudaDeviceSynchronize();
    }
}
```

4. Big Data Processing

Parallel Data Analysis

```
template<typename T>
class ParallelDataProcessor {
private:
    std::vector<T> data;
    int num_threads;

public:
    ParallelDataProcessor(const std::vector<T>& input, int threads)
        : data(input), num_threads(threads) {}

    template<typename Func>
    std::vector<T> map(Func f) {
        std::vector<T> result(data.size());
        #pragma omp parallel for num_threads(num_threads)
        for(size_t i = 0; i < data.size(); i++) {
            result[i] = f(data[i]);
        }
        return result;
    }

    template<typename Func>
    T reduce(Func f, T initial) {
        T result = initial;
        #pragma omp parallel num_threads(num_threads)
        {
            T local_sum = initial;
            #pragma omp for nowait
            for(size_t i = 0; i < data.size(); i++) {
                local_sum = f(local_sum, data[i]);
            }
            #pragma omp critical
            {
                result = f(result, local_sum);
            }
        }
        return result;
    }
};
```

Lab Exercise

Tasks

1. Implement Barnes-Hut simulation
2. Develop parallel LU factorization
3. Create big data processing pipeline
4. Analyze performance characteristics

Performance Analysis

- Algorithm complexity
- Memory access patterns
- Load balancing
- Scalability testing

Resources

Documentation

- Advanced CUDA Programming Guide
- Parallel Algorithms Reference
- Scientific Computing Libraries

Tools

- Performance Profilers
- Debugging Tools
- Analysis Frameworks

Questions & Discussion