CEN310 Parallel Programming

Week-12 (Real-world Applications I)

Spring Semester, 2024-2025

Overview

Topics

1. Scientific Computing Applications
2. Data Processing Applications
3. Performance Optimization
4. Case Studies

Objectives

- Apply parallel programming to real problems
- Optimize scientific computations
- Process large datasets efficiently
- Analyze real-world performance

1. Scientific Computing Applications

N-Body Simulation

```
__global__ void calculate_forces(float4* pos, float4* vel, float4* forces, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        float4 my_pos = pos[idx];
        float4 force = make_float4(0.0f, 0.0f, 0.0f, 0.0f);

        for(int j = 0; j < n; j++) {
            if(j != idx) {
                float4 other_pos = pos[j];
                float3 r = make_float3(
                    other_pos.x - my_pos.x,
                    other_pos.y - my_pos.y,
                    other_pos.z - my_pos.z
                );
                float dist = sqrtf(r.x*r.x + r.y*r.y + r.z*r.z);
                float f = (G * my_pos.w * other_pos.w) / (dist * dist);
                force.x += f * r.x/dist;
                force.y += f * r.y/dist;
                force.z += f * r.z/dist;
            }
        }
        forces[idx] = force;
    }
}
```

2. Data Processing Applications

Image Processing

```
__global__ void gaussian_blur(
    unsigned char* input,
    unsigned char* output,
    int width,
    int height,
    float* kernel,
    int kernel_size
) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if(x < width && y < height) {
        float sum = 0.0f;
        int k_radius = kernel_size / 2;

        for(int ky = -k_radius; ky <= k_radius; ky++) {
            for(int kx = -k_radius; kx <= k_radius; kx++) {
                int px = min(max(x + kx, 0), width - 1);
                int py = min(max(y + ky, 0), height - 1);
                float kernel_val = kernel[(ky+k_radius)*kernel_size + (kx+k_radius)];
                sum += input[py*width + px] * kernel_val;
            }
        }

        output[y*width + x] = (unsigned char)sum;
    }
}
```

3. Performance Optimization

Memory Access Optimization

```
// Optimize matrix transpose
__global__ void matrix_transpose(float* input, float* output, int width, int height) {
    __shared__ float tile[BLOCK_SIZE][BLOCK_SIZE+1]; // Avoid bank conflicts

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if(x < width && y < height) {
        // Load into shared memory
        tile[threadIdx.y][threadIdx.x] = input[y*width + x];
        __syncthreads();

        // Calculate transposed indices
        int new_x = blockIdx.y * blockDim.y + threadIdx.x;
        int new_y = blockIdx.x * blockDim.x + threadIdx.y;

        if(new_x < height && new_y < width) {
            output[new_y*height + new_x] = tile[threadIdx.x][threadIdx.y];
        }
    }
}
```

4. Case Studies

Monte Carlo Simulation

```
__global__ void monte_carlo_pi(float* points_x, float* points_y, int* inside_circle, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < n) {
        float x = points_x[idx];
        float y = points_y[idx];
        float dist = x*x + y*y;

        if(dist <= 1.0f) {
            atomicAdd(inside_circle, 1);
        }
    }
}

int main() {
    int n = 1000000;
    float *h_x, *h_y, *d_x, *d_y;
    int *h_inside, *d_inside;

    // Allocate and initialize memory
    // ... (memory allocation code)

    // Generate random points
    for(int i = 0; i < n; i++) {
        h_x[i] = (float)rand()/RAND_MAX;
        h_y[i] = (float)rand()/RAND_MAX;
    }

    // Copy data to device and run kernel
    // ... (CUDA memory operations and kernel launch)

    // Calculate pi
    float pi = 4.0f * (*h_inside) / (float)n;
    printf("Estimated Pi: %f\n", pi);

    // Cleanup
    // ... (memory deallocation code)

    return 0;
}
```

Lab Exercise

Tasks

1. Implement N-body simulation
2. Optimize image processing kernel
3. Develop Monte Carlo simulation
4. Compare performance with CPU versions

Performance Analysis

- Execution time
- Memory bandwidth
- GPU utilization
- Scaling behavior

Resources

Documentation

- CUDA Sample Applications
- Scientific Computing Libraries
- Performance Analysis Tools

Tools

- NVIDIA Visual Profiler
- Parallel Computing Toolbox
- Performance Libraries

Questions & Discussion


RECEP TAYYIP
ERDOĞAN
ÜNİVERSİTESİ