CEN310 Parallel Programming

Week-10 (Parallel Algorithm Design & GPU Basics)

Spring Semester, 2024-2025

Overview

Topics

1. Parallel Algorithm Design Strategies
2. Decomposition Techniques
3. GPU Architecture Fundamentals
4. Introduction to CUDA Programming

Objectives

- Understand parallel algorithm design principles
- Learn data decomposition methods
- Explore GPU architecture
- Get started with CUDA programming

- Task parallelism
- Data parallelism
- Pipeline parallelism
- Divide and conquer

Example: Matrix Multiplication

```
// Sequential version
void matrix_multiply(float* A, float* B, float* C, int N) {
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            float sum = 0.0f;
            for(int k = 0; k < N; k++) {
                sum += A[i*N + k] * B[k*N + j];
            }
            C[i*N + j] = sum;
        }
    }
}

// Parallel version
#pragma omp parallel for collapse(2)
void parallel_matrix_multiply(float* A, float* B, float* C, int N) {
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            float sum = 0.0f;
            for(int k = 0; k < N; k++) {
```

2. Decomposition Techniques

Data Decomposition

- Block decomposition
- Cyclic decomposition
- Block-cyclic decomposition

Example: Array Processing

```
// Block decomposition
void block_decomposition(float* data, int size, int num_blocks) {
    int block_size = size / num_blocks;
    #pragma omp parallel for
    for(int b = 0; b < num_blocks; b++) {
        int start = b * block_size;
        int end = (b == num_blocks-1) ? size : (b+1) * block_size;
        for(int i = start; i < end; i++) {
            // Process data[i]
        }
    }
}
```

3. GPU Architecture Fundamentals

Hardware Components

- Streaming Multiprocessors (SMs)
- CUDA Cores
- Memory Hierarchy
- Warp Scheduling

Memory Types

```
CPU (Host)           GPU (Device)
    ↓                      ↓
Host Memory          Global Memory
                          ↓
                    Shared Memory
                          ↓
                     L1 Cache
                          ↓
                    Registers
```

Basic Concepts

- Kernels
- Threads
- Blocks
- Grids

Hello World Example

```
#include <cuda_runtime.h>
#include <stdio.h>

__global__ void hello_kernel() {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    printf("Hello from thread %d\n", idx);
}

int main() {
    // Launch kernel with 1 block of 256 threads
    hello_kernel<<<1, 256>>>();
    cudaDeviceSynchronize();
```

CUDA Memory Management

Memory Operations

```
// Allocate device memory
float *d_data;
cudaMalloc(&d_data, size * sizeof(float));

// Copy data to device
cudaMemcpy(d_data, h_data, size * sizeof(float),
           cudaMemcpyHostToDevice);

// Copy results back
cudaMemcpy(h_result, d_result, size * sizeof(float),
           cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_data);
```

Vector Addition Example

```
__global__ void vector_add(float* a, float* b, float* c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        c[idx] = a[idx] + b[idx];
    }
}

int main() {
    int N = 1000000;
    size_t size = N * sizeof(float);

    // Allocate host memory
    float *h_a = (float*)malloc(size);
    float *h_b = (float*)malloc(size);
    float *h_c = (float*)malloc(size);

    // Initialize arrays
    for(int i = 0; i < N; i++) {
        h_a[i] = rand() / (float)RAND_MAX;
        h_b[i] = rand() / (float)RAND_MAX;
    }

    // Allocate device memory
    float *d_a, *d_b, *d_c;
    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);
    cudaMalloc(&d_c, size);

    // Copy to device
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    // Launch kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    vector_add<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, N);

    // Copy result back
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

Lab Exercise

Tasks

1. Implement matrix multiplication using CUDA
2. Compare performance with CPU version
3. Experiment with different block sizes
4. Analyze memory access patterns

Performance Analysis

- Use nvprof for profiling
- Measure execution time
- Calculate speedup
- Monitor memory transfers

Resources

Documentation

- CUDA Programming Guide
- CUDA Best Practices Guide
- NVIDIA Developer Blog

Tools

- NVIDIA NSight
- CUDA Toolkit
- Visual Profiler

Questions & Discussion