CEN310 Parallel Programming

Week-5

GPU Programming

Download

- PDF
- DOC
- SLIDE
- PPTX

Outline (1/4)

1. Introduction to GPU Computing
    - GPU Architecture Overview
    - CUDA Programming Model
    - GPU Memory Hierarchy
    - Thread Organization
    - Kernel Functions
2. CUDA Programming Basics
    - Memory Management
    - Thread Organization
    - Synchronization
    - Error Handling
    - CUDA Runtime API

Outline (2/4)

3. Memory Management
   - Global Memory
   - Shared Memory
   - Constant Memory
   - Texture Memory
   - Unified Memory
4. Thread Organization
   - Blocks and Grids
   - Warps and Scheduling
   - Thread Synchronization
   - Occupancy
   - Load Balancing

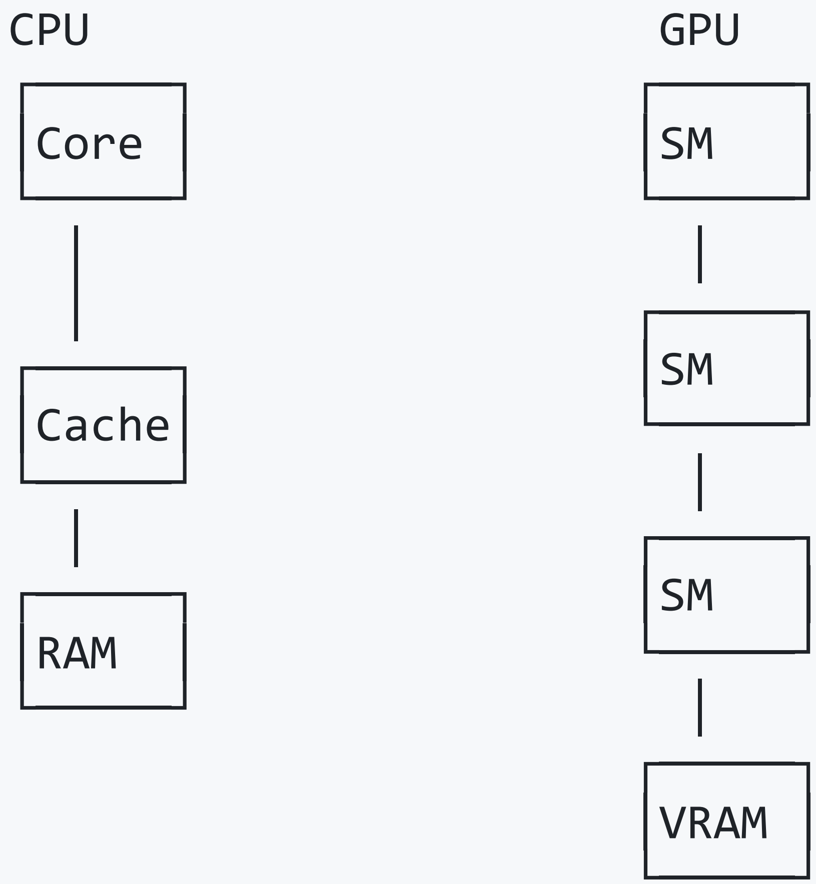
Outline (3/4)

5. Performance Optimization
    - Memory Coalescing
    - Bank Conflicts
    - Divergent Branching
    - Shared Memory Usage
    - Asynchronous Operations
6. Advanced Features
    - Streams and Events
    - Dynamic Parallelism
    - Multi-GPU Programming
    - Unified Memory
    - Cooperative Groups

Outline (4/4)

7. Best Practices
    - Code Organization
    - Error Handling
    - Debugging Techniques
    - Profiling Tools
    - Common Pitfalls
8. Real-World Applications
    - Image Processing
    - Scientific Computing
    - Machine Learning
    - Data Analytics

GPU Architecture (1/4)



Key Components:

- Streaming Multiprocessors (SMs)
- CUDA Cores

GPU Architecture (2/4)

Memory Hierarchy

```
// Example showing different memory types
__device__ __constant__ float device_constant[256];  // Constant memory
__shared__ float shared_array[256];                  // Shared memory

__global__ void memory_example(float* global_input,  // Global memory
                             float* global_output) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    // Register (automatic) variables
    float local_var = global_input[idx];

    // Shared memory usage
    shared_array[threadIdx.x] = local_var;
    __syncthreads();

    // Constant memory usage
    local_var *= device_constant[threadIdx.x];

    // Write back to global memory
    global_output[idx] = local_var;
}
```

GPU Architecture (3/4)

Thread Hierarchy

```
__global__ void thread_hierarchy_example() {
    // Thread identification
    int thread_idx = threadIdx.x;
    int block_idx = blockIdx.x;
    int warp_id = threadIdx.x / 32;

    // Block dimensions
    int block_size = blockDim.x;
    int grid_size = gridDim.x;

    // Global thread ID
    int global_idx = thread_idx + block_idx * block_size;

    // Print thread information
    printf("Thread %d in block %d (warp %d)\n",
           thread_idx, block_idx, warp_id);
}

int main() {
    // Launch configuration
    dim3 block_size(256);
    dim3 grid_size((N + block_size.x - 1) / block_size.x);

    thread_hierarchy_example<<<grid_size, block_size>>>();
    return 0;
}
```

GPU Architecture (4/4)

Basic CUDA Program

```
#include <cuda_runtime.h>

// Kernel definition
__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 block_size = 256;
    int num_blocks = (N + block_size - 1) / block_size;
    vector_add<<<num_blocks, block_size>>>(d_a, d_b, d_c, N);

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

    // Verify results
    for(int i = 0; i < N; i++) {
        if(fabs(h_c[i] - (h_a[i] + h_b[i])) > 1e-5) {
            fprintf(stderr, "Verification failed at %d\n", i);
            break;
        }
    }

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

    return 0;
}
```

2. CUDA Programming Basics

Memory Management (1/4)

```
void memory_management_example() {
    // Host memory allocation
    float* h_data = (float*)malloc(size);

    // Device memory allocation
    float* d_data;
    cudaMalloc(&d_data, size);

    // Pinned memory allocation
    float* h_pinned;
    cudaMallocHost(&h_pinned, size);

    // Unified memory allocation
    float* unified;
    cudaMallocManaged(&unified, size);

    // Memory transfers
    cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);
    cudaMemcpy(h_data, d_data, size, cudaMemcpyDeviceToHost);

    // Asynchronous transfers
    cudaStream_t stream;
    cudaStreamCreate(&stream);
    cudaMemcpyAsync(d_data, h_pinned, size,
                    cudaMemcpyHostToDevice, stream);

    // Cleanup
    free(h_data);
    cudaFree(d_data);
    cudaFreeHost(h_pinned);
    cudaFree(unified);
    cudaStreamDestroy(stream);
}
```

Memory Management (2/4)

Shared Memory Usage

```
__global__ void shared_memory_example(float* input,
                                     float* output,
                                     int n) {
    extern __shared__ float shared[];

    int tid = threadIdx.x;
    int gid = blockIdx.x * blockDim.x + threadIdx.x;

    // Load data into shared memory
    if(gid < n) {
        shared[tid] = input[gid];
    }
    __syncthreads();

    // Process data in shared memory
    if(tid > 0 && tid < blockDim.x-1 && gid < n-1) {
        float result = 0.25f * (shared[tid-1] +
                               2.0f * shared[tid] +
                               shared[tid+1]);
        output[gid] = result;
    }
}

// Kernel launch
int block_size = 256;
int shared_size = block_size * sizeof(float);
shared_memory_example<<<grid_size, block_size, shared_size>>>
    (d_input, d_output, N);
```

Memory Management (3/4)

Constant Memory

```
__constant__ float const_array[256];

void setup_constant_memory() {
    float h_const_array[256];

    // Initialize constant data
    for(int i = 0; i < 256; i++) {
        h_const_array[i] = compute_constant(i);
    }

    // Copy to constant memory
    cudaMemcpyToSymbol(const_array, h_const_array,
                       256 * sizeof(float));
}

__global__ void use_constant_memory(float* input,
                                    float* output,
                                    int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < n) {
        // Use constant memory
        output[idx] = input[idx] * const_array[idx % 256];
    }
}
```

Memory Management (4/4)

Texture Memory

```
texture<float, 2, cudaReadModeElementType> tex_ref;

void texture_memory_example() {
    // Allocate and initialize 2D array
    cudaArray* d_array;
    cudaChannelFormatDesc channel_desc =
        cudaCreateChannelDesc<float>();

    cudaMallocArray(&d_array, &channel_desc,
                    width, height);

    // Copy data to array
    cudaMemcpyToArray(d_array, 0, 0, h_data,
                      width * height * sizeof(float),
                      cudaMemcpyHostToDevice);

    // Bind texture reference
    cudaBindTextureToArray(tex_ref, d_array);

    // Kernel using texture memory
    texture_kernel<<<grid_size, block_size>>>
        (d_output, width, height);

    // Cleanup
    cudaUnbindTexture(tex_ref);
    cudaFreeArray(d_array);
}

__global__ void texture_kernel(float* output,
                               int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if(x < width && y < height) {
        // Read from texture
        float value = tex2D(tex_ref, x, y);
        output[y * width + x] = value;
    }
}
```

3. Thread Organization

Thread Hierarchy (1/4)

```
__global__ void thread_organization_example() {
    // Thread indices
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int tz = threadIdx.z;

    // Block indices
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;

    // Block dimensions
    int bdx = blockDim.x;
    int bdy = blockDim.y;
    int bdz = blockDim.z;

    // Grid dimensions
    int gdx = gridDim.x;
    int gdy = gridDim.y;
    int gdz = gridDim.z;

    // Calculate global indices
    int global_x = bx * bdx + tx;
    int global_y = by * bdy + ty;
    int global_z = bz * bdz + tz;

    // Calculate linear index
    int linear_idx = global_z * gdx * gdy * bdx * bdy +
                     global_y * gdx * bdx +
                     global_x;
}
```

Thread Hierarchy (2/4)

Block Configuration

```
void launch_configuration_example() {
    // 1D configuration
    dim3 block_1d(256);
    dim3 grid_1d((N + block_1d.x - 1) / block_1d.x);
    kernel_1d<<<grid_1d, block_1d>>>();

    // 2D configuration
    dim3 block_2d(16, 16);
    dim3 grid_2d((width + block_2d.x - 1) / block_2d.x,
                 (height + block_2d.y - 1) / block_2d.y);
    kernel_2d<<<grid_2d, block_2d>>>();

    // 3D configuration
    dim3 block_3d(8, 8, 8);
    dim3 grid_3d((width + block_3d.x - 1) / block_3d.x,
                 (height + block_3d.y - 1) / block_3d.y,
                 (depth + block_3d.z - 1) / block_3d.z);
    kernel_3d<<<grid_3d, block_3d>>>();
}

// Kernel examples
__global__ void kernel_1d() {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
}

__global__ void kernel_2d() {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
}

__global__ void kernel_3d() {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int z = blockIdx.z * blockDim.z + threadIdx.z;
}
```

Thread Hierarchy (3/4)

Warp Management

```
__global__ void warp_example() {
    int tid = threadIdx.x;
    int warp_id = tid / 32;
    int lane_id = tid % 32;

    // Warp-level primitives
    int mask = __ballot_sync(__activemask(), tid < 16);
    int value = __shfl_sync(__activemask(), tid, 0);

    // Warp-level reduction
    int sum = tid;
    for(int offset = 16; offset > 0; offset /= 2) {
        sum += __shfl_down_sync(__activemask(), sum, offset);
    }

    // Warp-level synchronization
    __syncwarp();
}
```

Thread Hierarchy (4/4)

Dynamic Parallelism

```
__global__ void child_kernel(int* data, int size) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < size) {
        data[idx] *= 2;
    }
}

__global__ void parent_kernel(int* data,
                            int* sizes,
                            int num_arrays) {
    int array_idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(array_idx < num_arrays) {
        int size = sizes[array_idx];
        int* array_data = &data[array_idx * MAX_ARRAY_SIZE];

        // Launch child kernel
        int block_size = 256;
        int grid_size = (size + block_size - 1) / block_size;
        child_kernel<<<grid_size, block_size>>>
            (array_data, size);
    }
}
```

4. Performance Optimization

Memory Coalescing (1/4)

```
// Bad memory access pattern
__global__ void uncoalesced_access(float* input,
                                   float* output,
                                   int width,
                                   int height) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < width) {
        for(int y = 0; y < height; y++) {
            output[idx + y * width] =
                input[idx + y * width];  // Strided access
        }
    }
}

// Good memory access pattern
__global__ void coalesced_access(float* input,
                                 float* output,
                                 int width,
                                 int height) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if(idx < width && y < height) {
        output[y * width + idx] =
            input[y * width + idx];  // Coalesced access
    }
}
```

Memory Coalescing (2/4)

Bank Conflicts

```
__global__ void bank_conflicts_example(float* data) {
    extern __shared__ float shared[];
    int tid = threadIdx.x;

    // Bad: Bank conflicts
    shared[tid * 32] = data[tid];  // 32-way bank conflict

    // Good: No bank conflicts
    shared[tid] = data[tid];       // Consecutive access

    __syncthreads();

    // Process data
    float result = shared[tid];
    // ...
}
```

Memory Coalescing (3/4)

Shared Memory Optimization

```
template<int BLOCK_SIZE>
__global__ void matrix_multiply(float* A,
                              float* B,
                              float* C,
                              int width) {
    __shared__ float shared_A[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float shared_B[BLOCK_SIZE][BLOCK_SIZE];

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int bx = blockIdx.x;
    int by = blockIdx.y;

    int row = by * BLOCK_SIZE + ty;
    int col = bx * BLOCK_SIZE + tx;

    float sum = 0.0f;

    // Loop over blocks
    for(int block = 0; block < width/BLOCK_SIZE; block++) {
        // Load data into shared memory
        shared_A[ty][tx] = A[row * width +
                            block * BLOCK_SIZE + tx];
        shared_B[ty][tx] = B[(block * BLOCK_SIZE + ty) * width +
                            col];
        __syncthreads();

        // Compute partial dot product
        for(int k = 0; k < BLOCK_SIZE; k++) {
            sum += shared_A[ty][k] * shared_B[k][tx];
        }
        __syncthreads();
    }

    // Store result
    C[row * width + col] = sum;
}
```

Memory Coalescing (4/4)

Memory Access Patterns

```
// Structure of Arrays (SoA)
struct ParticlesSoA {
    float* x;
    float* y;
    float* z;
    float* vx;
    float* vy;
    float* vz;
};

// Array of Structures (AoS)
struct ParticleAoS {
    float x, y, z;
    float vx, vy, vz;
};

// SoA kernel (better coalescing)
__global__ void update_particles_soa(ParticlesSoA particles,
                                     int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < n) {
        particles.x[idx] += particles.vx[idx];
        particles.y[idx] += particles.vy[idx];
        particles.z[idx] += particles.vz[idx];
    }
}

// AoS kernel (worse coalescing)
__global__ void update_particles_aos(ParticleAoS* particles,
                                     int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < n) {
        particles[idx].x += particles[idx].vx;
        particles[idx].y += particles[idx].vy;
        particles[idx].z += particles[idx].vz;
    }
}
```

5. Advanced Features

Streams and Events (1/3)

```
void stream_example() {
    const int num_streams = 4;
    cudaStream_t streams[num_streams];

    // Create streams
    for(int i = 0; i < num_streams; i++) {
        cudaStreamCreate(&streams[i]);
    }

    // Allocate memory
    float *h_input, *d_input, *h_output, *d_output;
    cudaMallocHost(&h_input, size);    // Pinned memory
    cudaMallocHost(&h_output, size);   // Pinned memory
    cudaMalloc(&d_input, size);
    cudaMalloc(&d_output, size);

    // Launch kernels in different streams
    int chunk_size = N / num_streams;
    for(int i = 0; i < num_streams; i++) {
        int offset = i * chunk_size;

        cudaMemcpyAsync(&d_input[offset],
                        &h_input[offset],
                        chunk_size * sizeof(float),
                        cudaMemcpyHostToDevice,
                        streams[i]);

        process_kernel<<<grid_size, block_size, 0, streams[i]>>>
            (&d_input[offset], &d_output[offset], chunk_size);

        cudaMemcpyAsync(&h_output[offset],
                        &d_output[offset],
                        chunk_size * sizeof(float),
                        cudaMemcpyDeviceToHost,
                        streams[i]);
    }

    // Synchronize all streams
    cudaDeviceSynchronize();

    // Cleanup
    for(int i = 0; i < num_streams; i++) {
        cudaStreamDestroy(streams[i]);
    }
    cudaFreeHost(h_input);
    cudaFreeHost(h_output);
    cudaFree(d_input);
    cudaFree(d_output);
}
```

Streams and Events (2/3)

Event Management

```
void event_example() {
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // Record start event
    cudaEventRecord(start);

    // Launch kernel
    process_kernel<<<grid_size, block_size>>>(d_data, N);

    // Record stop event
    cudaEventRecord(stop);

    // Wait for completion
    cudaEventSynchronize(stop);

    // Calculate elapsed time
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("Kernel execution time: %f ms\n", milliseconds);

    // Cleanup
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
}
```

Streams and Events (3/3)

Inter-stream Synchronization

```
void stream_synchronization() {
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1);
    cudaStreamCreate(&stream2);

    cudaEvent_t event;
    cudaEventCreate(&event);

    // Launch work in stream1
    kernel1<<<grid_size, block_size, 0, stream1>>>
        (d_data1, N);
    cudaEventRecord(event, stream1);

    // Make stream2 wait for stream1
    cudaStreamWaitEvent(stream2, event);

    // Launch work in stream2
    kernel2<<<grid_size, block_size, 0, stream2>>>
        (d_data2, N);

    // Cleanup
    cudaEventDestroy(event);
    cudaStreamDestroy(stream1);
    cudaStreamDestroy(stream2);
}
```

6. Best Practices

Error Handling (1/3)

```
#define CUDA_CHECK(call) do {                              \
    cudaError_t error = call;                             \
    if(error != cudaSuccess) {                            \
        fprintf(stderr, "CUDA error at %s:%d: %s\n",      \
                __FILE__, __LINE__,                        \
                cudaGetErrorString(error));                \
        exit(EXIT_FAILURE);                               \
    }                                                      \
} while(0)

void cuda_error_handling() {
    // Allocate memory
    float* d_data;
    CUDA_CHECK(cudaMalloc(&d_data, size));

    // Launch kernel
    process_kernel<<<grid_size, block_size>>>(d_data, N);
    CUDA_CHECK(cudaGetLastError());

    // Synchronize and check for errors
    CUDA_CHECK(cudaDeviceSynchronize());

    // Cleanup
    CUDA_CHECK(cudaFree(d_data));
}
```

Error Handling (2/3)

Debug Tools

```
void debug_example() {
    // Enable device synchronization for debugging
    cudaDeviceSetLimit(cudaLimitDevRuntimeSyncDepth, 1);

    // Print device properties
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    printf("Device: %s\n", prop.name);
    printf("Compute capability: %d.%d\n",
           prop.major, prop.minor);

    // Launch kernel with debug info
    #ifdef DEBUG
        printf("Launching kernel with grid=%d, block=%d\n",
               grid_size.x, block_size.x);
    #endif

    process_kernel<<<grid_size, block_size>>>(d_data, N);

    // Check for kernel errors
    cudaError_t error = cudaGetLastError();
    if(error != cudaSuccess) {
        printf("Kernel error: %s\n",
               cudaGetErrorString(error));
    }
}
```

Error Handling (3/3)

Resource Management

```
class CUDAResource {
private:
    void* ptr;
    size_t size;

public:
    CUDAResource(size_t s) : size(s), ptr(nullptr) {
        CUDA_CHECK(cudaMalloc(&ptr, size));
    }

    ~CUDAResource() {
        if(ptr) {
            cudaFree(ptr);
        }
    }

    void* get() { return ptr; }
    size_t get_size() { return size; }

    // Prevent copying
    CUDAResource(const CUDAResource&) = delete;
    CUDAResource& operator=(const CUDAResource&) = delete;
};

void resource_management_example() {
    try {
        CUDAResource d_input(1024);
        CUDAResource d_output(1024);

        // Use resources
        process_kernel<<<grid_size, block_size>>>
            (d_input.get(), d_output.get(), N);
    }
    catch(const std::exception& e) {
        fprintf(stderr, "Error: %s\n", e.what());
    }
}
```

7. Real-World Applications

Image Processing (1/3)

```
// Image convolution kernel
__global__ void convolution_2d(unsigned char* input,
```