Access Cluster

To access the Markov HPC cluster, use the following SSH command:

```
ssh -X pxd222@markov.case.edu
ssh -Y pxd222@markov.case.edu
```

- `-X`: Enables X11 forwarding, which allows you to run graphical applications remotely.

Allocating GPU Nodes

1. **Allocate a GPU Node**:
    - Use `srun` to allocate a GPU node and ensure `DISPLAY` is passed correctly:

```
srun --x11 -p markov_gpu --gres=gpu:1 --mem=8gb --pty /bin/bash
```

Build Environment

```
module load CUDA/12.1.1
module load cuDNN/8.9.2.26-CUDA-12.1.1
```

Code Setup

Setup:

- Create directories `Basic` and `Flash` for both the implementations below.
- Create scripts `basic_attention.cu` and `flash_attention.cu` and copy code segments below into their respective scripts.
- Compile and run both scripts then compare performance.

Basic Attention Kernel

basic_attention.cu

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <curand.h>
#include <curand_kernel.h>
#include <stdio.h>
#include <math.h>

// Configuration
#define BLOCK_SIZE 16
#define MAX_THREADS 1024

// Error checking macro
#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)

// Kernel for matrix multiplication
__global__ void matmul(float* A, float* B, float* C, int M, int N, int K) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < K) {
        float sum = 0.0f;
        for (int i = 0; i < N; i++) {
            sum += A[row * N + i] * B[i * K + col];
        }
        C[row * K + col] = sum;
    }
}

// Kernel for softmax operation
__global__ void softmax(float* input, float* output, int seq_len) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < seq_len) {
        float max_val = input[row * seq_len];

        // Find max value in row
        for (int i = 1; i < seq_len; i++) {
            max_val = fmaxf(max_val, input[row * seq_len + i]);
        }

        // Compute exponentials and sum
        float sum = 0.0f;
        for (int i = 0; i < seq_len; i++) {
            float exp_val = expf(input[row * seq_len + i] - max_val);
            output[row * seq_len + i] = exp_val;
            sum += exp_val;
        }

        // Normalize
        for (int i = 0; i < seq_len; i++) {
            output[row * seq_len + i] /= sum;
        }
    }
}

// Kernel for attention scores computation
__global__ void attention_scores(float* Q, float* K, float* scores,
                                 int seq_len, int head_dim) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < seq_len && col < seq_len) {
        float sum = 0.0f;
        float scale = 1.0f / sqrtf(head_dim);

        for (int i = 0; i < head_dim; i++) {
            sum += Q[row * head_dim + i] * K[col * head_dim + i];
        }
        scores[row * seq_len + col] = sum * scale;
    }
}
```

```
// Initialize random matrix
__global__ void initialize_random_matrix(float* matrix, int rows, int cols,
                                         unsigned long long seed) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    curandState state;

    for (int i = idx; i < rows * cols; i += stride) {
        curand_init(seed, i, 0, &state);
        matrix[i] = 0.1f * curand_uniform(&state);  // Scale down values for numerical
stability
    }
}

// Function to print matrix dimensions and a few sample values
void print_matrix_info(const char* name, float* matrix, int rows, int cols) {
    printf("%s dimensions: %d x %d\n", name, rows, cols);
    printf("First few values: ");
    for (int i = 0; i < min(5, rows * cols); i++) {
        printf("%.4f ", matrix[i]);
    }
    printf("\n");
}

int main() {
    // Problem dimensions
    const int batch_size = 1;
    const int num_heads = 8;
    const int seq_len = 512;
    const int head_dim = 64;
    const int d_model = head_dim * num_heads;

    printf("\nProblem Configuration:\n");
    printf("Batch size: %d\n", batch_size);
    printf("Number of attention heads: %d\n", num_heads);
    printf("Sequence length: %d\n", seq_len);
    printf("Head dimension: %d\n", head_dim);
    printf("Model dimension: %d\n", d_model);

    // Calculate sizes
    size_t input_size = batch_size * seq_len * d_model * sizeof(float);
    size_t qkv_size = batch_size * num_heads * seq_len * head_dim * sizeof(float);
    size_t score_size = batch_size * num_heads * seq_len * seq_len * sizeof(float);

    // Host memory allocation
    float *h_input = (float*)malloc(input_size);
    float *h_W_Q = (float*)malloc(d_model * d_model * sizeof(float));
    float *h_W_K = (float*)malloc(d_model * d_model * sizeof(float));
    float *h_W_V = (float*)malloc(d_model * d_model * sizeof(float));
    float *h_output = (float*)malloc(input_size);

    // Device memory allocation
    float *d_input, *d_W_Q, *d_W_K, *d_W_V;
    float *d_Q, *d_K, *d_V, *d_scores, *d_attn_output;

    CUDA_CHECK(cudaMalloc(&d_input, input_size));
    CUDA_CHECK(cudaMalloc(&d_W_Q, d_model * d_model * sizeof(float)));
    CUDA_CHECK(cudaMalloc(&d_W_K, d_model * d_model * sizeof(float)));
    CUDA_CHECK(cudaMalloc(&d_W_V, d_model * d_model * sizeof(float)));
    CUDA_CHECK(cudaMalloc(&d_Q, qkv_size));
    CUDA_CHECK(cudaMalloc(&d_K, qkv_size));
    CUDA_CHECK(cudaMalloc(&d_V, qkv_size));
```

```
    CUDA_CHECK(cudaMalloc(&d_scores, score_size));
    CUDA_CHECK(cudaMalloc(&d_attn_output, qkv_size));

    // Initialize random values
    int blockSize = 256;
    int numBlocks = (seq_len * d_model + blockSize - 1) / blockSize;

    initialize_random_matrix<<<numBlocks, blockSize>>>(d_input, seq_len, d_model, 1234ULL);
    initialize_random_matrix<<<numBlocks, blockSize>>>(d_W_Q, d_model, d_model, 1235ULL);
    initialize_random_matrix<<<numBlocks, blockSize>>>(d_W_K, d_model, d_model, 1236ULL);
    initialize_random_matrix<<<numBlocks, blockSize>>>(d_W_V, d_model, d_model, 1237ULL);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Create CUDA events for timing
    cudaEvent_t start, stop;
    cudaEvent_t proj_start, proj_stop;
    cudaEvent_t attn_start, attn_stop;
    CUDA_CHECK(cudaEventCreate(&start));
    CUDA_CHECK(cudaEventCreate(&stop));
    CUDA_CHECK(cudaEventCreate(&proj_start));
    CUDA_CHECK(cudaEventCreate(&proj_stop));
    CUDA_CHECK(cudaEventCreate(&attn_start));
    CUDA_CHECK(cudaEventCreate(&attn_stop));

    // Start timing
    CUDA_CHECK(cudaEventRecord(start));

    // Linear projections timing
    CUDA_CHECK(cudaEventRecord(proj_start));

    dim3 projBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 projGrid((d_model + BLOCK_SIZE - 1) / BLOCK_SIZE,
                  (seq_len + BLOCK_SIZE - 1) / BLOCK_SIZE);

    for (int b = 0; b < batch_size; b++) {
        for (int h = 0; h < num_heads; h++) {
            float* Q_head = d_Q + (b * num_heads + h) * seq_len * head_dim;
            float* K_head = d_K + (b * num_heads + h) * seq_len * head_dim;
            float* V_head = d_V + (b * num_heads + h) * seq_len * head_dim;

            matmul<<<projGrid, projBlock>>>(d_input, d_W_Q, Q_head, seq_len, d_model,
head_dim);
            matmul<<<projGrid, projBlock>>>(d_input, d_W_K, K_head, seq_len, d_model,
head_dim);
            matmul<<<projGrid, projBlock>>>(d_input, d_W_V, V_head, seq_len, d_model,
head_dim);
        }
    }

    CUDA_CHECK(cudaEventRecord(proj_stop));

    // Attention computation timing
    CUDA_CHECK(cudaEventRecord(attn_start));

    dim3 attnBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 attnGrid((seq_len + BLOCK_SIZE - 1) / BLOCK_SIZE,
                  (seq_len + BLOCK_SIZE - 1) / BLOCK_SIZE);

    for (int b = 0; b < batch_size; b++) {
        for (int h = 0; h < num_heads; h++) {
            float* Q_head = d_Q + (b * num_heads + h) * seq_len * head_dim;
            float* K_head = d_K + (b * num_heads + h) * seq_len * head_dim;
            float* V_head = d_V + (b * num_heads + h) * seq_len * head_dim;
```

```
            float* scores_head = d_scores + (b * num_heads + h) * seq_len * seq_len;
            float* output_head = d_attn_output + (b * num_heads + h) * seq_len * head_dim;

            // Compute attention scores
            attention_scores<<<attnGrid, attnBlock>>>(Q_head, K_head, scores_head,
                                                     seq_len, head_dim);

            // Apply softmax
            softmax<<<seq_len, min(seq_len, MAX_THREADS)>>>(scores_head, scores_head,
seq_len);

            // Multiply with values
            matmul<<<projGrid, projBlock>>>(scores_head, V_head, output_head,
                                           seq_len, seq_len, head_dim);
        }
    }

    CUDA_CHECK(cudaEventRecord(attn_stop));
    CUDA_CHECK(cudaEventRecord(stop));
    CUDA_CHECK(cudaEventSynchronize(stop));

    // Calculate timing results
    float total_time = 0;
    float proj_time = 0;
    float attn_time = 0;
    CUDA_CHECK(cudaEventElapsedTime(&total_time, start, stop));
    CUDA_CHECK(cudaEventElapsedTime(&proj_time, proj_start, proj_stop));
    CUDA_CHECK(cudaEventElapsedTime(&attn_time, attn_start, attn_stop));

    printf("\nTiming Results:\n");
    printf("Linear Projections time: %.3f ms\n", proj_time);
    printf("Attention Computation time: %.3f ms\n", attn_time);
    printf("Total execution time: %.3f ms\n", total_time);

    // Copy results back for verification
    CUDA_CHECK(cudaMemcpy(h_output, d_attn_output, qkv_size, cudaMemcpyDeviceToHost));

    // Print a few output values for verification
    printf("\nOutput verification (first few values):\n");
    for (int i = 0; i < 5; i++) {
        printf("%.4f ", h_output[i]);
    }
    printf("\n");

    // Print some intermediate values for debugging
    float *h_Q = (float*)malloc(qkv_size);
    CUDA_CHECK(cudaMemcpy(h_Q, d_Q, qkv_size, cudaMemcpyDeviceToHost));
    printf("\nQ matrix verification (first few values):\n");
    for (int i = 0; i < 5; i++) {
        printf("%.4f ", h_Q[i]);
    }
    printf("\n");

    // Cleanup
    free(h_input);
    free(h_W_Q);
    free(h_W_K);
    free(h_W_V);
    free(h_output);
    free(h_Q);

    CUDA_CHECK(cudaFree(d_input));
    CUDA_CHECK(cudaFree(d_W_Q));
```

```
    CUDA_CHECK(cudaFree(d_W_K));
    CUDA_CHECK(cudaFree(d_W_V));
    CUDA_CHECK(cudaFree(d_Q));
    CUDA_CHECK(cudaFree(d_K));
    CUDA_CHECK(cudaFree(d_V));
    CUDA_CHECK(cudaFree(d_scores));
    CUDA_CHECK(cudaFree(d_attn_output));

    CUDA_CHECK(cudaEventDestroy(start));
    CUDA_CHECK(cudaEventDestroy(stop));
    CUDA_CHECK(cudaEventDestroy(proj_start));
    CUDA_CHECK(cudaEventDestroy(proj_stop));
    CUDA_CHECK(cudaEventDestroy(attn_start));
    CUDA_CHECK(cudaEventDestroy(attn_stop));

    return 0;
}
```

Compilation Command

```
nvcc -O3 -arch=sm_60 -o basic_attention basic_attention.cu
```

Output of Script

```
[pxd222@classt01 Basic]$ ./basic_attention

Problem Configuration:
Batch size: 1
Number of attention heads: 8
Sequence length: 512
Head dimension: 64
Model dimension: 512

Timing Results:
Linear Projections time: 1.757 ms
Attention Computation time: 12.571 ms
Total execution time: 14.334 ms

Output verification (first few values):
1.2131 1.2870 1.2420 1.3345 1.3415

Q matrix verification (first few values):
1.2950 1.2912 1.2722 1.3086 1.2938
```

Flash Attention Kernel

flash_attention.cu

```
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <curand.h>
#include <curand_kernel.h>
#include <stdio.h>
#include <math.h>

// Configuration
#define TILE_SIZE 32        // Reduced tile size for better occupancy
#define HEAD_DIM 64
#define THREADS_PER_BLOCK 256
```

```
#define WARP_SIZE 32

// Error checking macro
#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)

// Shared memory structure
struct SharedMemory {
    float q[TILE_SIZE][HEAD_DIM];
    float k[TILE_SIZE][HEAD_DIM];
    float v[TILE_SIZE][HEAD_DIM];
};

__global__ void matmul(float* A, float* B, float* C, int M, int N, int K) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < K) {
        float sum = 0.0f;
        for (int i = 0; i < N; i++) {
            sum += A[row * N + i] * B[i * K + col];
        }
        C[row * K + col] = sum;
    }
}

__global__ void initialize_random_matrix(float* matrix, int rows, int cols,
                                         unsigned long long seed) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    curandState state;

    for (int i = idx; i < rows * cols; i += stride) {
        curand_init(seed, i, 0, &state);
        matrix[i] = 0.1f * curand_uniform(&state);
    }
}

__global__ void flash_attention_kernel(
    const float* __restrict__ Q,
    const float* __restrict__ K,
    const float* __restrict__ V,
    float* __restrict__ O,
    float* __restrict__ M,
    float* __restrict__ L,
    const int seq_len) {

    extern __shared__ char shared_mem[];
    SharedMemory* shared = reinterpret_cast<SharedMemory*>(shared_mem);

    const int row_block = blockIdx.x * TILE_SIZE;
    const int tid = threadIdx.x;
    const float scale = 1.0f / sqrtf(HEAD_DIM);

    // Each thread handles one or more rows
    for (int row = tid; row < TILE_SIZE && row_block + row < seq_len; row += blockDim.x) {
```

```
        float max_val = -INFINITY;
        float sum_exp = 0.0f;
        float acc[HEAD_DIM] = {0.0f};

        // Load Q row into shared memory
        for (int k = 0; k < HEAD_DIM; k++) {
            shared->q[row][k] = Q[(row_block + row) * HEAD_DIM + k];
        }

        // Process K,V tiles
        for (int tile = 0; tile < seq_len; tile += TILE_SIZE) {
            __syncthreads();

            // Load K,V tile
            for (int i = tid; i < TILE_SIZE && tile + i < seq_len; i += blockDim.x) {
                for (int k = 0; k < HEAD_DIM; k++) {
                    shared->k[i][k] = K[(tile + i) * HEAD_DIM + k];
                    shared->v[i][k] = V[(tile + i) * HEAD_DIM + k];
                }
            }
            __syncthreads();

            // Compute attention scores and update running max
            float local_max = -INFINITY;
            float scores[TILE_SIZE];
            int valid_cols = min(TILE_SIZE, seq_len - tile);

            for (int j = 0; j < valid_cols; j++) {
                float qk_sum = 0.0f;
                for (int k = 0; k < HEAD_DIM; k++) {
                    qk_sum += shared->q[row][k] * shared->k[j][k];
                }
                scores[j] = qk_sum * scale;
                local_max = fmaxf(local_max, scores[j]);
            }

            // Update global max and rescale previous terms if needed
            if (local_max > max_val) {
                float scale_factor = expf(max_val - local_max);
                for (int k = 0; k < HEAD_DIM; k++) {
                    acc[k] *= scale_factor;
                }
                sum_exp *= scale_factor;
                max_val = local_max;
            }

            // Compute attention and update accumulators
            float local_sum = 0.0f;
            for (int j = 0; j < valid_cols; j++) {
                float exp_val = expf(scores[j] - max_val);
                local_sum += exp_val;
                for (int k = 0; k < HEAD_DIM; k++) {
                    acc[k] += exp_val * shared->v[j][k];
                }
            }
            sum_exp += local_sum;
        }

        // Write outputs
        if (row_block + row < seq_len) {
            M[row_block + row] = max_val;
            L[row_block + row] = sum_exp;
            float inv_sum = 1.0f / sum_exp;
```

```
            for (int k = 0; k < HEAD_DIM; k++) {
                O[(row_block + row) * HEAD_DIM + k] = acc[k] * inv_sum;
            }
        }
    }
}

int main() {
    // Problem dimensions
    const int batch_size = 1;
    const int num_heads = 8;
    const int seq_len = 512;
    const int head_dim = 64;
    const int d_model = head_dim * num_heads;

    printf("\nProblem Configuration:\n");
    printf("Batch size: %d\n", batch_size);
    printf("Number of attention heads: %d\n", num_heads);
    printf("Sequence length: %d\n", seq_len);
    printf("Head dimension: %d\n", head_dim);
    printf("Model dimension: %d\n", d_model);

    // Calculate sizes
    size_t input_size = batch_size * seq_len * d_model * sizeof(float);
    size_t qkv_size = batch_size * num_heads * seq_len * head_dim * sizeof(float);
    size_t softmax_stats_size = batch_size * num_heads * seq_len * sizeof(float);

    // Host memory allocation
    float *h_input = (float*)malloc(input_size);
    float *h_W_Q = (float*)malloc(d_model * d_model * sizeof(float));
    float *h_W_K = (float*)malloc(d_model * d_model * sizeof(float));
    float *h_W_V = (float*)malloc(d_model * d_model * sizeof(float));
    float *h_output = (float*)malloc(input_size);

    // Device memory allocation
    float *d_input, *d_W_Q, *d_W_K, *d_W_V;
    float *d_Q, *d_K, *d_V, *d_output;
    float *d_m, *d_l;

    CUDA_CHECK(cudaMalloc(&d_input, input_size));
    CUDA_CHECK(cudaMalloc(&d_W_Q, d_model * d_model * sizeof(float)));
    CUDA_CHECK(cudaMalloc(&d_W_K, d_model * d_model * sizeof(float)));
    CUDA_CHECK(cudaMalloc(&d_W_V, d_model * d_model * sizeof(float)));
    CUDA_CHECK(cudaMalloc(&d_Q, qkv_size));
    CUDA_CHECK(cudaMalloc(&d_K, qkv_size));
    CUDA_CHECK(cudaMalloc(&d_V, qkv_size));
    CUDA_CHECK(cudaMalloc(&d_output, qkv_size));
    CUDA_CHECK(cudaMalloc(&d_m, softmax_stats_size));
    CUDA_CHECK(cudaMalloc(&d_l, softmax_stats_size));

    // Initialize matrices
    int blockSize = 256;
    int numBlocks = (seq_len * d_model + blockSize - 1) / blockSize;
    initialize_random_matrix<<<numBlocks, blockSize>>>(d_input, seq_len, d_model, 1234ULL);
    initialize_random_matrix<<<numBlocks, blockSize>>>(d_W_Q, d_model, d_model, 1235ULL);
    initialize_random_matrix<<<numBlocks, blockSize>>>(d_W_K, d_model, d_model, 1236ULL);
    initialize_random_matrix<<<numBlocks, blockSize>>>(d_W_V, d_model, d_model, 1237ULL);
    CUDA_CHECK(cudaDeviceSynchronize());

    // Create events for timing
    cudaEvent_t start, stop, proj_start, proj_stop, attn_start, attn_stop;
    CUDA_CHECK(cudaEventCreate(&start));
    CUDA_CHECK(cudaEventCreate(&stop));
```

```
    CUDA_CHECK(cudaEventCreate(&proj_start));
    CUDA_CHECK(cudaEventCreate(&proj_stop));
    CUDA_CHECK(cudaEventCreate(&attn_start));
    CUDA_CHECK(cudaEventCreate(&attn_stop));

    // Start timing
    CUDA_CHECK(cudaEventRecord(start));

    // Linear projections
    CUDA_CHECK(cudaEventRecord(proj_start));
    dim3 proj_block(16, 16);
    dim3 proj_grid((d_model + 16 - 1) / 16, (seq_len + 16 - 1) / 16);

    for (int b = 0; b < batch_size; b++) {
        for (int h = 0; h < num_heads; h++) {
            float* Q_head = d_Q + (b * num_heads + h) * seq_len * head_dim;
            float* K_head = d_K + (b * num_heads + h) * seq_len * head_dim;
            float* V_head = d_V + (b * num_heads + h) * seq_len * head_dim;

            matmul<<<proj_grid, proj_block>>>(d_input, d_W_Q, Q_head, seq_len, d_model,
head_dim);
            matmul<<<proj_grid, proj_block>>>(d_input, d_W_K, K_head, seq_len, d_model,
head_dim);
            matmul<<<proj_grid, proj_block>>>(d_input, d_W_V, V_head, seq_len, d_model,
head_dim);
        }
    }
    CUDA_CHECK(cudaEventRecord(proj_stop));

    // Flash attention computation
    CUDA_CHECK(cudaEventRecord(attn_start));

    size_t shared_mem_size = sizeof(SharedMemory);
    int num_blocks = (seq_len + TILE_SIZE - 1) / TILE_SIZE;

    printf("\nLaunching flash attention with configuration:\n");
    printf("Tile size: %d\n", TILE_SIZE);
    printf("Threads per block: %d\n", THREADS_PER_BLOCK);
    printf("Number of blocks: %d\n", num_blocks);
    printf("Shared memory size: %zu bytes\n", shared_mem_size);

    for (int b = 0; b < batch_size; b++) {
        for (int h = 0; h < num_heads; h++) {
            float* Q_head = d_Q + (b * num_heads + h) * seq_len * head_dim;
            float* K_head = d_K + (b * num_heads + h) * seq_len * head_dim;
            float* V_head = d_V + (b * num_heads + h) * seq_len * head_dim;
            float* O_head = d_output + (b * num_heads + h) * seq_len * head_dim;
            float* M_head = d_m + (b * num_heads + h) * seq_len;
            float* L_head = d_l + (b * num_heads + h) * seq_len;

            flash_attention_kernel<<<num_blocks, THREADS_PER_BLOCK, shared_mem_size>>>(
                Q_head, K_head, V_head, O_head, M_head, L_head, seq_len
            );
            CUDA_CHECK(cudaGetLastError());
        }
    }

    CUDA_CHECK(cudaEventRecord(attn_stop));
    CUDA_CHECK(cudaEventRecord(stop));
    CUDA_CHECK(cudaEventSynchronize(stop));

    // Calculate timings
    float total_time = 0;
```

```
    float proj_time = 0;
    float attn_time = 0;
    CUDA_CHECK(cudaEventElapsedTime(&total_time, start, stop));
    CUDA_CHECK(cudaEventElapsedTime(&proj_time, proj_start, proj_stop));
    CUDA_CHECK(cudaEventElapsedTime(&attn_time, attn_start, attn_stop));

    printf("\nTiming Results:\n");
    printf("Linear Projections time: %.3f ms\n", proj_time);
    printf("Flash Attention time: %.3f ms\n", attn_time);
    printf("Total execution time: %.3f ms\n", total_time);

    // Verify results
    CUDA_CHECK(cudaMemcpy(h_output, d_output, qkv_size, cudaMemcpyDeviceToHost));
    printf("\nOutput verification (first few values):\n");
    for (int i = 0; i < 5; i++) {
        printf("%.4f ", h_output[i]);
    }
    printf("\n");

    float *h_Q = (float*)malloc(qkv_size);
    float *h_m = (float*)malloc(softmax_stats_size);
    float *h_l = (float*)malloc(softmax_stats_size);

    CUDA_CHECK(cudaMemcpy(h_Q, d_Q, qkv_size, cudaMemcpyDeviceToHost));
    CUDA_CHECK(cudaMemcpy(h_m, d_m, softmax_stats_size, cudaMemcpyDeviceToHost));
    CUDA_CHECK(cudaMemcpy(h_l, d_l, softmax_stats_size, cudaMemcpyDeviceToHost));


    printf("\nQ matrix verification (first few values):\n");
    for (int i = 0; i < 5; i++) {
        printf("%.4f ", h_Q[i]);
    }
    printf("\n");

    printf("\nSoftmax statistics for first row:\n");
    printf("Max value (m): %.4f\n", h_m[0]);
    printf("Sum value (l): %.4f\n", h_l[0]);

    // Validate results
    bool valid_output = false;
    for (int i = 0; i < seq_len * head_dim; i++) {
        if (h_output[i] != 0.0f) {
            valid_output = true;
            break;
        }
    }

    if (!valid_output) {
        printf("\nWarning: Output appears to be all zeros. This might indicate a kernel
execution problem.\n");
    }

    // Cleanup
    free(h_input);
    free(h_W_Q);
    free(h_W_K);
    free(h_W_V);
    free(h_output);
    free(h_Q);
    free(h_m);
    free(h_l);

    CUDA_CHECK(cudaFree(d_input));
```

```
    CUDA_CHECK(cudaFree(d_W_Q));
    CUDA_CHECK(cudaFree(d_W_K));
    CUDA_CHECK(cudaFree(d_W_V));
    CUDA_CHECK(cudaFree(d_Q));
    CUDA_CHECK(cudaFree(d_K));
    CUDA_CHECK(cudaFree(d_V));
    CUDA_CHECK(cudaFree(d_output));
    CUDA_CHECK(cudaFree(d_m));
    CUDA_CHECK(cudaFree(d_l));

    CUDA_CHECK(cudaEventDestroy(start));
    CUDA_CHECK(cudaEventDestroy(stop));
    CUDA_CHECK(cudaEventDestroy(proj_start));
    CUDA_CHECK(cudaEventDestroy(proj_stop));
    CUDA_CHECK(cudaEventDestroy(attn_start));
    CUDA_CHECK(cudaEventDestroy(attn_stop));

    return 0;
}
```

Compilation Command

```
nvcc -O3 -arch=sm_60 -o flash_attention flash_attention.cu
```

Output of Script

```
[pxd222@classt01 Flash]$ ./flash_attention

Problem Configuration:
Batch size: 1
Number of attention heads: 8
Sequence length: 512
Head dimension: 64
Model dimension: 512

Launching flash attention with configuration:
Tile size: 32
Threads per block: 256
Number of blocks: 16
Shared memory size: 24576 bytes

Timing Results:
Linear Projections time: 1.735 ms
Flash Attention time: 2.560 ms
Total execution time: 4.299 ms

Output verification (first few values):
1.2131 1.2870 1.2420 1.3345 1.3415

Q matrix verification (first few values):
1.2950 1.2912 1.2722 1.3086 1.2938

Softmax statistics for first row:
Max value (m): 14.3975
Sum value (l): 156.8147
```

→ This confirms that flash attention maintains numerical accuracy while improving performance.

Performance Measurements

Basic Attention Implementation

Operation	Time (ms)
Linear Projections	1.743
Attention Computation	12.549
Total Time	14.297

Flash Attention Implementation

Operation	Time (ms)
Linear Projections	1.727
Attention Computation	2.559
Total Time	4.293

Speed Improvements

Component	Improvement
Linear Projections	~1% faster (1.743ms → 1.727ms)
Attention Comp.	~4.9x faster (12.549ms → 2.559ms)
Total Execution	~3.3x faster (14.297ms → 4.293ms)

Implementation Details

Flash Attention Configuration

Parameter	Value
Tile Size	32
Threads per block	256
Number of blocks	16
Shared memory	24576 bytes

Key Optimizations

1. Memory Access Optimization

- Tiled processing reduces global memory access
- Efficient use of shared memory cache

2. Computational Efficiency

- Fused operations in attention kernel

- Optimized thread utilization
- Efficient tile-based processing