CUDA Lab Helpers

ECE 455: GPU Algorithm and System Design

This file contains helper functions for the CUDA Matrix Multiplication labs.

Helper Functions

Error Checking

```cpp
// Macro wrapper: converts the CUDA call 'val' into a string (#val)
// and passes it with file and line info to 'check'
#define checkCuda(val) check((val), #val, __FILE__, __LINE__)

// Checks the return code of CUDA API calls
void check(cudaError_t err, const char* const func,
           const char* const file, const int line)
{
    if (err != cudaSuccess) {                    // If CUDA call failed
        std::cerr << "CUDA Runtime Error at: "
                  << file << ":" << line << std::endl;
        std::cerr << cudaGetErrorString(err)     // Print readable error
                  << " " << func << std::endl;   // and the function that
                     failed
        std::exit(EXIT_FAILURE);                 // Abort program
    }
}
```

Random Initialization

```cpp
// Create a vector filled with random numbers in [-256, 256]
template <typename T>
std::vector<T> create_rand_vector(size_t n)
{
    std::random_device r;                        // Non-deterministic seed
    std::default_random_engine e(r());           // Random engine
    std::uniform_int_distribution<int> uniform_dist(-256, 256);

    std::vector<T> vec(n);
    for (size_t i = 0; i < n; ++i) {
        vec.at(i) = static_cast<T>(uniform_dist(e));  // Fill each element
    }
    return vec;
}
```

CPU Reference Implementation

```cpp
// Naive triple-loop matrix multiplication: C = A * B
template <typename T>
void mm(T const* mat_1, T const* mat_2, T* mat_3,
        size_t m, size_t n, size_t p)
{
    for (size_t i = 0; i < m; ++i) {              // Loop over rows of A
        for (size_t j = 0; j < p; ++j) {          // Loop over cols of B
            T acc_sum = 0;
            for (size_t k = 0; k < n; ++k) {      // Dot product
                acc_sum += mat_1[i * n + k] * mat_2[k * p + j];
            }
            mat_3[i * p + j] = acc_sum;           // Store result in C
        }
    }
}
```

Validation (allclose)

```cpp
// Compare two vectors elementwise within a tolerance
template <typename T>
bool allclose(std::vector<T> const& vec_1,
              std::vector<T> const& vec_2,
              T const& abs_tol)
{
    if (vec_1.size() != vec_2.size()) return false;   // Size mismatch
    for (size_t i = 0; i < vec_1.size(); ++i) {
        // Check absolute difference
        if (std::abs(vec_1.at(i) - vec_2.at(i)) > abs_tol) {
            std::cout << vec_1.at(i) << " " << vec_2.at(i) << std::endl;
            return false;                                 // First mismatch
               printed
        }
    }
    return true;                                          // All elements close
}
```

Random Test

```cpp
// Run one randomized test comparing CPU vs GPU results
template <typename T>
bool random_test_mm_cuda(size_t m, size_t n, size_t p)
{
    // --- Allocate and initialize host matrices ---
    std::vector<T> const mat_1_vec{create_rand_vector<T>(m * n)};
    std::vector<T> const mat_2_vec{create_rand_vector<T>(n * p)};
    std::vector<T> mat_3_vec(m * p);   // CPU result
    std::vector<T> mat_4_vec(m * p);   // GPU result

    // Raw pointers for convenience
    T const* mat_1 = mat_1_vec.data();
```

```
    T const* mat_2 = mat_2_vec.data();
    T* mat_3 = mat_3_vec.data();
    T* mat_4 = mat_4_vec.data();

    // --- Compute reference result on CPU ---
    mm(mat_1, mat_2, mat_3, m, n, p);

    // --- Allocate GPU memory ---
    T *d_mat_1, *d_mat_2, *d_mat_4;
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * mat_1_vec.size()));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * mat_2_vec.size()));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * mat_4_vec.size()));

    // --- Copy input matrices to device ---
    checkCuda(cudaMemcpy(d_mat_1, mat_1, sizeof(T) * mat_1_vec.size(),
                         cudaMemcpyHostToDevice));
    checkCuda(cudaMemcpy(d_mat_2, mat_2, sizeof(T) * mat_2_vec.size(),
                         cudaMemcpyHostToDevice));

    // --- Launch your CUDA kernel (user-defined function) ---
    mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
    cudaDeviceSynchronize();                          // Wait for kernel to
        finish

    // --- Copy result back to host ---
    checkCuda(cudaMemcpy(mat_4, d_mat_4, sizeof(T) * mat_4_vec.size(),
                         cudaMemcpyDeviceToHost));

    // --- Free device memory ---
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));

    // --- Compare CPU vs GPU results ---
    return allclose<T>(mat_3_vec, mat_4_vec, 1e-4);
}
```

Multiple Random Tests

```
// Run multiple random tests in a loop (for stress testing)
template <typename T>
bool random_multiple_test_mm_cuda(size_t num_tests)
{
    size_t m{MAT_DIM}, n{MAT_DIM}, p{MAT_DIM};
    for (size_t i = 0; i < num_tests; ++i) {
        if (!random_test_mm_cuda<T>(m, n, p)) {   // Stop if any test
            fails
            return false;
        }
    }
    return true;                                  // All tests passed
}
```

Runtime Measurement

```cpp
// Measure average runtime of mm_cuda using CUDA events
template <typename T>
float measure_latency_mm_cuda(size_t m, size_t n, size_t p,
                              size_t num_tests, size_t num_warmups)
{
    cudaEvent_t startEvent, stopEvent;
    float time = 0.0f;

    // --- Create CUDA events for timing ---
    checkCuda(cudaEventCreate(&startEvent));
    checkCuda(cudaEventCreate(&stopEvent));

    // --- Allocate device matrices once ---
    T *d_mat_1, *d_mat_2, *d_mat_4;
    checkCuda(cudaMalloc(&d_mat_1, sizeof(T) * m * n));
    checkCuda(cudaMalloc(&d_mat_2, sizeof(T) * n * p));
    checkCuda(cudaMalloc(&d_mat_4, sizeof(T) * m * p));

    // --- Warm-up runs (not timed) ---
    for (size_t i = 0; i < num_warmups; ++i) {
        mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
    }

    // --- Timed runs using CUDA events ---
    checkCuda(cudaEventRecord(startEvent, 0));
    for (size_t i = 0; i < num_tests; ++i) {
        mm_cuda(d_mat_1, d_mat_2, d_mat_4, m, n, p);
    }
    checkCuda(cudaEventRecord(stopEvent, 0));
    checkCuda(cudaEventSynchronize(stopEvent));           // Wait for GPU
    checkCuda(cudaEventElapsedTime(&time, startEvent, stopEvent)); // Time
        in ms

    // --- Free device memory ---
    checkCuda(cudaFree(d_mat_1));
    checkCuda(cudaFree(d_mat_2));
    checkCuda(cudaFree(d_mat_4));

    // Return average runtime per test (milliseconds)
    return time / num_tests;
}
```