HPC Tutorial - 13
CUDA - Parallelizing Multi-Head Self Attention Transformers

Rohan G
CS22B1093

CUDA Version of project code -

Device Code -

```
// CUDA kernel for matrix addition
__global__ void addMatrixKernel(double* A, double* B, double* C, int rows, int cols) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < rows && col < cols) {
        int idx = row * cols + col;
        C[idx] = A[idx] + B[idx];
    }
}

// CUDA kernel for applying bias
__global__ void addBiasKernel(double* matrix, double* bias, int rows, int cols) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < rows && col < cols) {
        int idx = row * cols + col;
        matrix[idx] += bias[col];
    }
}
```

```
// CUDA kernel for ReLU activation with NaN handling
__global__ void reluKernel(double* matrix, int rows, int cols) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < rows && col < cols) {
        int idx = row * cols + col;
        double val = matrix[idx];

        // Check for NaN or Inf and handle it
        if (!isfinite(val)) {
            matrix[idx] = 0.0; // Replace NaN/Inf with zero for ReLU
        } else {
            matrix[idx] = max(0.0, val);
        }
    }
}

// CUDA kernel for scaling tensor values
__global__ void scaleKernel(double* matrix, double scale, int rows, int cols) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < rows && col < cols) {
        int idx = row * cols + col;
        // Prevent division by zero or very small numbers
        if (fabs(scale) > 1e-10) {
            matrix[idx] /= scale;
        }
    }
}
```

```cuda
// CUDA kernel for softmax operation
__global__ void softmaxKernel(double* matrix, int rows, int cols) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < rows) {
        // Find max value in the row for numerical stability
        double max_val = -INFINITY;
        for (int j = 0; j < cols; j++) {
            max_val = max(max_val, matrix[row * cols + j]);
        }

        // Calculate exp and sum
        double sum = 0.0;
        for (int j = 0; j < cols; j++) {
            int idx = row * cols + j;
            matrix[idx] = exp(matrix[idx] - max_val);
            sum += matrix[idx];
        }

        // Normalize
        for (int j = 0; j < cols; j++) {
            int idx = row * cols + j;
            matrix[idx] /= sum;
        }
    }
}
```

```cuda
// CUDA kernel for split heads
__global__ void splitHeadsKernel(double* input, double* output, int seq_len, int num_heads, int d_head) {
    int h = blockIdx.z;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    if (h < num_heads && i < seq_len && j < d_head) {
        output[(h * seq_len + i) * d_head + j] = input[i * (num_heads * d_head) + h * d_head + j];
    }
}

// CUDA kernel for concatenate heads
__global__ void concatenateHeadsKernel(double* input, double* output, int seq_len, int num_heads, int d_value) {
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int k = blockIdx.x * blockDim.x + threadIdx.x;

    if (j < seq_len && k < num_heads * d_value) {
        int h = k / d_value;
        int d = k % d_value;
        output[j * (num_heads * d_value) + k] = input[(h * seq_len + j) * d_value + d];
    }
}
```

```cuda
// CUDA kernel for layer normalization
__global__ void layerNormKernel(double* input, double* gamma, double* beta, double* output, int rows, int cols, float epsilon) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < rows) {
        // Calculate mean
        double mean = 0.0;
        for (int i = 0; i < cols; i++) {
            mean += input[row * cols + i];
        }
        mean /= cols;

        // Calculate variance
        double var = 0.0;
        for (int i = 0; i < cols; i++) {
            double diff = input[row * cols + i] - mean;
            var += diff * diff;
        }
        var /= cols;

        // Normalize, scale, and shift
        double stddev_inv = 1.0 / sqrt(var + epsilon);
        for (int i = 0; i < cols; i++) {
            int idx = row * cols + i;
            double norm = (input[idx] - mean) * stddev_inv;
            output[idx] = gamma[i] * norm + beta[i];
        }
    }
}
```

```cpp
// Utility functions for host-device memory transfers
void copyMatrixToDevice(double* d_matrix, const vector<vector<double>>& h_matrix, int rows, int cols) {
    double* h_temp = new double[rows * cols];
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            h_temp[i * cols + j] = h_matrix[i][j];
        }
    }
    CHECK_CUDA_ERROR(cudaMemcpy(d_matrix, h_temp, rows * cols * sizeof(double), cudaMemcpyHostToDevice));
    delete[] h_temp;
}

void copyMatrixToHost(vector<vector<double>>& h_matrix, double* d_matrix, int rows, int cols) {
    double* h_temp = new double[rows * cols];
    CHECK_CUDA_ERROR(cudaMemcpy(h_temp, d_matrix, rows * cols * sizeof(double), cudaMemcpyDeviceToHost));
    for (int i = 0; i < rows; i++) {
        h_matrix[i].resize(cols);
        for (int j = 0; j < cols; j++) {
            h_matrix[i][j] = h_temp[i * cols + j];
        }
    }
    delete[] h_temp;
}

void copyVectorToDevice(double* d_vector, const vector<double>& h_vector, int size) {
    CHECK_CUDA_ERROR(cudaMemcpy(d_vector, h_vector.data(), size * sizeof(double), cudaMemcpyHostToDevice));
}
```

```cpp
// Helper function for matrix multiplication using cuBLAS with enhanced stability
void matrixMultiply(cublasHandle_t handle, double* A, double* B, double* C, int m, int n, int k) {
    const double alpha = 1.0f;
    const double beta = 0.0f;

    // Perform matrix multiplication: C = A * B
    // Note: cuBLAS uses column-major order, while C/C++ uses row-major order
    // C(m,n) = A(m,k) * B(k,n)
    CHECK_CUBLAS_ERROR(cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, n, m, k, &alpha, B, n, A, k, &beta, C, n));
}

// Utility function to generate a random matrix
vector<vector<double>> genRandomMatrix(size_t rows, size_t cols, double scale = 0.01) {
    vector<vector<double>> matrix(rows, vector<double>(cols, 0.0f));
    for (size_t i = 0; i < rows; i++) {
        for (size_t j = 0; j < cols; j++) {
            // Xavier/Glorot initialization approach - scale by sqrt(1/n)
            double glorot_scale = sqrt(2.0 / (rows + cols));
            // Use a smaller scale for better numerical stability
            matrix[i][j] = ((double)rand() / RAND_MAX * 2.0 - 1.0) * glorot_scale * scale;
        }
    }
    return matrix;
}
```

```cpp
// Utility function to add vectors
vector<vector<double>> add_vectors(vector<vector<double>>& a, vector<vector<double>>& b) {
    size_t rows = a.size();
    size_t cols = a[0].size();
    vector<vector<double>> result(rows, vector<double>(cols, 0.0f));

    // Allocate device memory
    double *d_a, *d_b, *d_result;
    CHECK_CUDA_ERROR(cudaMalloc((void**)&d_a, rows * cols * sizeof(double)));
    CHECK_CUDA_ERROR(cudaMalloc((void**)&d_b, rows * cols * sizeof(double)));
    CHECK_CUDA_ERROR(cudaMalloc((void**)&d_result, rows * cols * sizeof(double)));

    // Copy data to device
    copyMatrixToDevice(d_a, a, rows, cols);
    copyMatrixToDevice(d_b, b, rows, cols);

    // Set up grid and block dimensions
    dim3 blockDim(16, 16);
    dim3 gridDim((cols + blockDim.x - 1) / blockDim.x, (rows + blockDim.y - 1) / blockDim.y);

    // Launch kernel
    addMatrixKernel<<<gridDim, blockDim>>>(d_a, d_b, d_result, rows, cols);

    // Copy result back to host
    copyMatrixToHost(result, d_result, rows, cols);

    // Free device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_result);

    return result;
}
```

```cpp
// Concatenate heads function with CUDA acceleration
vector<vector<double>> concatenate_heads(vector<vector<double>>& x, size_t num_heads, size_t seq_len, size_t d_value) {
    vector<vector<double>> X(seq_len, vector<double>(num_heads * d_value, 0.0f));

    // Allocate device memory
    double *d_x, *d_X;
    CHECK_CUDA_ERROR(cudaMalloc((void**)&d_x, num_heads * seq_len * d_value * sizeof(double)));
    CHECK_CUDA_ERROR(cudaMalloc((void**)&d_X, seq_len * num_heads * d_value * sizeof(double)));

    // Copy data to device
    copyMatrixToDevice(d_x, x, num_heads * seq_len, d_value);

    // Set up grid and block dimensions
    dim3 blockDim(16, 16);
    dim3 gridDim((num_heads * d_value + blockDim.x - 1) / blockDim.x,
                 (seq_len + blockDim.y - 1) / blockDim.y);

    // Launch kernel
    concatenateHeadsKernel<<<gridDim, blockDim>>>(d_x, d_X, seq_len, num_heads, d_value);

    // Copy result back to host
    copyMatrixToHost(X, d_X, seq_len, num_heads * d_value);

    // Free device memory
    cudaFree(d_x);
    cudaFree(d_X);

    return X;
}
```

```cpp
// Matrix multiplication wrapper for the GPU
vector<vector<double>> matmul(vector<vector<double>>& a, vector<vector<double>>& b) {
    size_t m = a.size();
    size_t k = a[0].size();
    size_t n = b[0].size();
    vector<vector<double>> c(m, vector<double>(n, 0.0f));

    // Create cuBLAS handle
    cublasHandle_t handle;
    CHECK_CUBLAS_ERROR(cublasCreate(&handle));

    // Allocate device memory
    double *d_a, *d_b, *d_c;
    CHECK_CUDA_ERROR(cudaMalloc((void**)&d_a, m * k * sizeof(double)));
    CHECK_CUDA_ERROR(cudaMalloc((void**)&d_b, k * n * sizeof(double)));
    CHECK_CUDA_ERROR(cudaMalloc((void**)&d_c, m * n * sizeof(double)));

    // Copy data to device
    copyMatrixToDevice(d_a, a, m, k);
    copyMatrixToDevice(d_b, b, k, n);

    // Perform matrix multiplication
    matrixMultiply(handle, d_a, d_b, d_c, m, n, k);

    // Copy result back to host
    copyMatrixToHost(c, d_c, m, n);

    // Free device memory and destroy cuBLAS handle
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);
    cublasDestroy(handle);

    return c;
```

Host Code -

```
// Add positional encoding
vector<vector<double>> pos_encoding = get_positional_encoding(sequence_length, d_model);

// Check positional encoding for NaN values
if (check_for_nan(pos_encoding, "Positional encoding")) {
    cout << "Warning: Positional encoding contains NaN values. Sanitizing..." << endl;
    sanitize_matrix(pos_encoding);
}

vector<vector<double>> embedded_input = add_vectors(input, pos_encoding);

// Check embedded input for NaN values
if (check_for_nan(embedded_input, "Embedded input")) {
    cout << "Warning: Embedded input contains NaN values. Sanitizing..." << endl;
    sanitize_matrix(embedded_input);
}

// Initialize CUDA device
int device = 0;
cudaDeviceProp deviceProp;
CHECK_CUDA_ERROR(cudaGetDeviceProperties(&deviceProp, device));
CHECK_CUDA_ERROR(cudaSetDevice(device));

cout << "Using CUDA device: " << deviceProp.name << endl;

// Create encoder layers with smaller initialization values
vector<EncoderLayer> layers;
for (size_t i = 0; i < num_layers; i++) {
    layers.push_back(EncoderLayer(d_model, num_heads, ff_dim));
}
```

```
// Forward pass through encoder layers
vector<vector<double>> encoder_output = embedded_input;

LIKWID_MARKER_START("transformer");
auto start_time = chrono::high_resolution_clock::now();

// Process each layer with checks for NaNs between layers
for (size_t i = 0; i < num_layers; i++) {
    // Check before processing layer
    if (check_for_nan(encoder_output, "Before layer " + to_string(i))) {
        cout << "Warning: NaN values detected before layer " << i << ". Sanitizing..." << endl;
        sanitize_matrix(encoder_output);
    }

    // Process through the layer
    encoder_output = layers[i].forward(encoder_output);

    // Check after processing layer
    if (check_for_nan(encoder_output, "After layer " + to_string(i))) {
        cout << "Warning: Layer " << i << " produced NaN values. Using fallback..." << endl;
        // If a layer produces NaNs, fall back to a sanitized version of input
        if (i > 0) {
            // Try to use output from the previous layer
            encoder_output = layers[i-1].forward(embedded_input);
            sanitize_matrix(encoder_output);
        } else {
            // For the first layer, fall back to the embedded input
            encoder_output = embedded_input;
        }
    }
}

auto end_time = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(end_time - start_time);
```

Serial Code Output -

```
rzeta@rzeta:~/sem6/hpc/tutorials/tutorial_7/openmp$ ./transformer 1
Loading the data from the file...
Data read successfully with sequence length 1000
Output after input data is passed through the transformer :
Threads: 1 | Time: 1.86799 seconds.
```

Parallelized CUDA Output -

```
rzeta@rzeta:~/sem6/hpc/tutorials/tutorial_13$ ./transformer_cuda
Attempting to read input data from 'dataset_vectors.txt'...
Successfully loaded data with sequence length: 512 and embedding dimension: 512
Sample input values (first 3 rows, first 5 columns):
0.299295 0.955671 0.638526 0.556211 0.594222
0.620756 0.329521 0.067501 0.67424 0.352096
0.004244 0.087782 0.718957 0.202941 0.426891
Using CUDA device: NVIDIA GeForce RTX 3060 Laptop GPU
Time taken for forward pass: 950 ms
Sample output values:
-1.21767 1.6035 -0.639876 0.923191 -0.715311
0.796888 -0.280892 -0.245296 0.399886 0.236334
-0.205035 -2.51297 1.17368 -2.17769 0.671306
-0.546175 -3.56844 -0.0566555 -2.31128 -0.466916
-2.04639 -1.56591 -2.32427 -2.51147 -2.28466
```

Speedup -

S = T(1) / T(P) = 1.87 / 0.95 = 1.968

Speedup Achieved ~ 2x

CUDA Profiling -

```
Generating '/tmp/nsys-report-a686.qdstrm'
[1/8] [========================100%] report1.nsys-rep
[2/8] [========================100%] report1.sqlite
[3/8] Executing 'nvtx_sum' stats report
SKIPPED: /home/rzeta/sem6/hpc/tutorials/tutorial_13/report1.sqlite does not contain NV Tools Extension (NVTX) data.
[4/8] Executing 'osrt_sum' stats report

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)       Med (ns)      Min (ns)     Max (ns)    StdDev (ns)           Name

 --------  ---------------  ---------  ------------  ------------  -----------  -----------  ------------  ------------------
----
     53.2    1,466,792,127         23  63,773,570.7  100,107,361.0        1,433  264,454,267  63,828,509.0  poll

     36.3    1,000,137,828          2 500,068,914.0  500,068,914.0  500,065,893  500,071,935       4,272.3  pthread cond timed
```

```
[5/8] Executing 'cuda_api_sum' stats report

 Time (%)  Total Time (ns)  Num Calls   Avg (ns)     Med (ns)    Min (ns)    Max (ns)    StdDev (ns)              Name

 --------  ---------------  ---------  -----------  -----------  ---------  -----------  -----------  ------------------------
--------
     67.7      591,373,815        771    767,021.8    279,858.0      5,881   11,152,944  1,464,238.9  cudaMemcpy

     19.6      171,240,666      1,053    162,621.7     67,287.0      1,353  114,425,390  3,524,752.6  cudaMalloc

      8.9       78,047,907      1,185     65,863.2     69,361.0        310   10,240,655    300,611.7  cudaFree

      2.0       17,315,508        582     29,751.7      1,433.0        561      336,214     88,416.2  cudaDeviceSynchronize
```

```
[6/8] Executing 'cuda_gpu_kern_sum' stats report

 Time (%)  Total Time (ns)  Instances   Avg (ns)     Med (ns)    Min (ns)   Max (ns)    StdDev (ns)
           Name
 --------  ---------------  ---------  -----------  -----------  ---------  ---------  -----------  -------------------------
---------------------------------------------------------
     56.7      197,447,781         48  4,113,495.4  3,761,261.5  3,744,190  5,218,262    622,168.4  softmaxKernel(double *, int
, int)
     41.3      143,846,420        132  1,089,745.6    313,637.0    199,747  8,576,301  1,762,225.8  void cutlass::Kernel2<cutla
ss_80_tensorop_d884gemm_64x32_16x4_nn_align1>(T1::Params)
      1.3        4,366,470         12    363,872.5    342,309.0    340,965    473,767     50,557.0  layerNormKernel(double *, d
ouble *, double *, double *, int, int, float)
      0.4        1,264,986         48     26,353.9     24,001.0     23,968     33,825      4,046.1  scaleKernel(double *, doubl
e, int, int)
```

```
[7/8] Executing 'cuda_gpu_mem_time_sum' stats report

 Time (%)  Total Time (ns)  Count  Avg (ns)   Med (ns)   Min (ns)  Max (ns)   StdDev (ns)           Operation
 --------  ---------------  -----  ---------  ---------  --------  ---------  -----------  ----------------------------
     56.9      128,801,183    476  270,590.7  317,877.5       480  1,312,405    292,669.9  [CUDA memcpy Host-to-Device]
     43.1       97,594,684    295  330,829.4  312,774.0    39,744  2,580,682    282,967.5  [CUDA memcpy Device-to-Host]

[8/8] Executing 'cuda_gpu_mem_size_sum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)           Operation
 ----------  -----  --------  --------  --------  --------  -----------  ----------------------------
    847.471    476     1.780     2.097     0.004     8.389        1.937  [CUDA memcpy Host-to-Device]
    643.826    295     2.182     2.097     0.262     8.389        1.722  [CUDA memcpy Device-to-Host]
```

Observations from CUDA Nsight Profiling -

- Batch kernel launches to reduce CPU-GPU synchronization overhead.
- Optimize data transfers with pinned memory and asynchronous streaming.
- Increase block sizes to maximize SM occupancy and warp efficiency.