High-Performance Inference of Random Graph Convolutional Networks: A Study of SIMD-based and Multithreading-based Hardware Optimizations

Amruth Niranjan

May 5, 2025

Contents

1 Introduction

Neural networks have become the primary tools of use in modern machine learning, with applications ranging from image classification to natural language processing. Among the many variants of neural architectures, Graph Neural Networks (GNNs) have emerged as powerful models capable of learning from graph-structured data, with specific applications in networking, medicine, and supply-chain management. These models extend the expressiveness of neural networks by leveraging the relational and structural information present in graph-based data structures.

Optimizing neural networks includes a process combining forward and backward passes, gradient-based updates, and careful management of computational and memory resources. In the case of GNNs, these challenges are further complicated by the non-Euclidean nature of the input data. Each node aggregates information from its neighbors in the graph, making the computation inherently sparse compared to traditional fully connected or convolutional layers.

GCNs, the mainly used subclass of GNNs, perform layer-wise propagation. Even though they are relatively simple conceptally, it tends to be computationally demanding to efficiently implement GCNs, especially for larger and sparser graphs. Memory bandwidth, parallelization, and data locality are all critical factors that affect performance.

This report explores the algorithms and implementation techniques used to optimize the forward pass of single-layer undirected GCNs using single and multithreaded approaches. In the next section, I will touch on the nuances of GNNs and GCNs!

1.1 Graph Neural Networks

GNNs are a class of neural architectures that generalize deep learning to graph-structured data [1]. Unlike traditional neural networks that operate on fixed-size grid-like structures (i.e., vectors), GNNs are designed to learn representations over arbitrary graphs, which allows them to capture both local and global topological information.

Formally, let $G = (V, E)$ be an undirected graph with n nodes, where V is the set of nodes and E the set of edges. Each node $v_i \in V$ is associated with a feature vector $x_i \in \mathbb{R}^F$, where F is the number of input features per node. The graph structure is typically encoded by an adjacency matrix $A \in \mathbb{R}^{n \times n}$, where $A_{ij} = 1$ if there is an edge between nodes i and j, and $A_{ij} = 0$ otherwise.

At the core of GNNs lies the principle of **message-passing** (also known as **neighborhood aggregation**), which is a recursive mechanism where each node aggregates information from its neighbors to update its own representation. This process can be expressed generically as:

$$h_i^{(k)} = \text{UPDATE}^{(k)}\left(h_i^{(k-1)}, \text{AGGREGATE}^{(k)}\left(\left\{h_j^{(k-1)} : j \in \mathcal{N}(i)\right\}\right)\right), \tag{1}$$

where $h_i^{(k)}$ is the embedding of node i at layer k, and $\mathcal{N}(i)$ is the set of neighbors of node i. The `UPDATE` function computes a summary of the neighborhood features, while the `AGGREGATE` function combines the aggregated message with the current state.

It is important to note for later that the summary computed by the Z **permutation-invariant**, which means the learned representation is independent of the ordering of neighbors. This will be significant in later sections regarding parallelization.

1.2 Graph Convolutional Networks

Graph Convolutional Networks (GCNs) are a widely adopted subclass of GNNs that define a specific layer-wise propagation rule rooted in spectral graph theory. GCNs, as introduced by Kipf and Welling (2017) [2], offer a scalable and efficient approximation that performs localized, first-order neighborhood aggregation.

The forward pass of a single GCN layer is generally given by:

$$H^{(l+1)} = \sigma\left(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}H^{(l)}W^{(l)}\right), \tag{2}$$

where $H^{(l)} \in \mathbb{R}^{n\times d_l}$ is the matrix of node representations at layer l (with $H^{(0)} = X$ being the input feature matrix), $W^{(l)} \in \mathbb{R}^{d_l \times d_{l+1}}$ is a trainable weight matrix for layer l, $\hat{A} = A + I$ is the adjacency matrix with added self-loops, $\hat{D}$ is the diagonal degree matrix of $\hat{A}$, where $\hat{D}_{ii} = \sum_j \hat{A}_{ij}$, and $\sigma(\cdot)$ is a non-linear activation function, typically ReLU.

The expression $\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}$ corresponds to a symmetric normalization of the graph structure, ensuring that feature aggregation does not disproportionately favor high-degree nodes. This operator performs a weighted average of neighboring features, including the node itself due to the added self-loops.

Each layer in the GCN model performs two operations. First, in **feature transformation**, a linear projection of the node features is performed via $H^{(l)}W^{(l)}$, analogous to weight multiplication in standard neural networks. Next, in **neighborhood aggregation**, a normalized propagation of transformed features ($\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}$) is performed using the structure of the graph.

These operations collectively implement the message-passing framework discussed in the previous section, with the aggregation step implicitly encoded via matrix-matrix multiplication (MMM).

1.2.1 Node-wise Expression

In GCNs, a commonly used [3] practical form of the message-passing expression is defined by the following node-wise rewriting of $\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}$, which allows for parallelizable computation:

$$\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2} = \sum_{v\in\mathcal{N}(u)} \frac{1}{\sqrt{d_u d_v}} H'_v, \tag{3}$$

where d_u and d_v are the degrees of nodes u and v respectively, and H'_v denotes the transformed feature vector of neighbor node v.

1.3 A Note on Model Depth

In practice, stacking multiple GCN layers allows nodes to incorporate information from increasingly distant neighbors. However, deeper GCNs often suffer from the problem of *over-smoothing* [4], where node representations become indistinguishable as more layers are added. As such, most practical GCN architectures use only 2 or 3 layers.

The success of GCNs is largely attributed to their ability to integrate structural context with feature information in a computationally efficient manner. In the following sections, I will describe more about how the core GCN layer can be optimized for high-performance inference using serial, vectorized, and parallelized implementations.

2 Preprocessing

Before implementing the core graph convolutional operations, I developed a preprocessing pipeline to generate, store, and efficiently access synthetic graph data for benchmarking. My objective was not only create reproducible yet randomized graph instances of controlled size and density, but to also represent these graphs in a space- and compute-efficient format that could be quickly accessed by the main benchmarking code. To this end, I chose to initialize graphs using the probabalistic Erdős–Rényi model and store them in a Compressed Sparse Row representation serialized as binary files (`*.bin`). In the next few sections, I'll go over what these are!

2.1 Graph Memory Efficiency

There are probably near-infinite different ways of representing a graph in a computer's memory. However, this number drops severely when constraining on efficiency, especially for larger, sparser graphs. With some research, I was able to find the CSR representation, which serves as a highly efficient method of storing large, sparse graphs in memory, especially when compared to adjacency matrices.

Storing a full $n \times n$ adjacency matrix is impractical for both larger and sparser graphs, which is why sparse formats like CSR are better for hardware optimizations.

2.1.1 Compressed Sparse Row Representation

The Compressed Sparse Row (CSR) representation normally formats a sparse matrix using three arrays. First, a `row_ptr` array of size $(n + 1)$ holds the start index in the `col_idx` array for each row. Next, the `col_idx` array of size equal to the number of nonzero elements (edges) stores the column indices of the nonzero elements. Lastly, a `data` array stores the actual values of nonzero entries.

In the implementation of this project, I chose to use unweighted, unvalued, undirected graphs. This actually allowed me to completely remove this array from consideration, which reduced storage considerably!

Consider the following example of a 4 x 4 graph being represented as an adjacency matrix:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This matrix would be encoded in CSR format as

$$\texttt{row\_ptr} = \begin{bmatrix} 0 & 1 & 3 & 5 & 6 \end{bmatrix}$$

$$\texttt{col\_idx} = \begin{bmatrix} 1 & 0 & 2 & 1 & 3 & 2 \end{bmatrix}$$

2.2 Random Graph Initialization

To simulate a wide variety of graphs with controllable sparsity and reproducibility, I decided to use the Erdős–Rényi $G(n, p)$ model. In this probabilistic model, a graph is constructed by connecting each possible pair of n nodes with independent probability p. Notice there exist $\binom{n}{2}$ possible node pairs in an undirected graph.

2.2.1 Erdős-Rényi Random Graphs

As a formal introduction to Erdős-Rényi Graphs, let $G \sim G(n, p)$ be an Erdős–Rényi graph, where n is the number of nodes and p is the probability of an edge existing between any two nodes.

Then, the expected number of edges in the graph is:

$$\mathbb{E}[|E|] = p \cdot \binom{n}{2}$$

The expected degree of a node can also be calculated as:

$$\mathbb{E}[\deg(v)] = p(n-1)$$

I primarily chose this model because it offered an extremely easy way to tune graph size and sparsity, simply by tuning n and p. For example, setting $p = 0.001$ with $n = 10,000$ gives an average degree of approximately 10, resulting in a graph with about 50,000 edges. If stored in CSR form, this is well within the range of practical memory constraints!

2.2.2 Graph Initialization Code

Here is a code sample of how I generated graphs using the `igraph` library in Python:

```
import igraph as ig
import numpy as np
from scipy.sparse import csr_matrix
import os
from scipy.sparse import eye
def generate_erdos_renyi_graph(num_nodes=100000, param=0.001, seed=999,
    output_prefix="graph", save_binary=False):
     np.random.seed(seed)
```

```
    G = ig.Graph.Erdos_Renyi(n=num_nodes, p=param, directed=False, loops=
        False) # undirected w/o selfloops

    edges = np.array(G.get_edgelist())
    row = edges[:, 0]
    col = edges[:, 1]
    data = np.ones(len(row))

    # undirected
    row_full = np.concatenate([row, col])
    col_full = np.concatenate([col, row])
    data_full = np.ones(len(row_full))

    A = csr_matrix((data_full, (row_full, col_full)), shape=(num_nodes,
        num_nodes))

    # Add selfloops manually
    A += eye(num_nodes, format='csr')

    # CSR
    row_ptr = A.indptr
    col_idx = A.indices

    os.makedirs(os.path.dirname(output_prefix), exist_ok=True)

    if save_binary:
        row_ptr.astype(np.int32).tofile(f"{output_prefix}_row_ptr.bin")
        col_idx.astype(np.int32).tofile(f"{output_prefix}_col_idx.bin")
    else:
        np.savetxt(f"{output_prefix}_row_ptr.txt", row_ptr, fmt="%d")
        np.savetxt(f"{output_prefix}_col_idx.txt", col_idx, fmt="%d")

    print(f"Graph saved: {num_nodes} nodes, {A.nnz} edges (incl self-loops
        ) [{output_prefix}]")

if __name__ == "__main__":
    for num_nodes in [1000, 10000]:
        os.makedirs(f"{num_nodes}_nodes", exist_ok=True)
        for i in range(10):
            generate_erdos_renyi_graph(
                num_nodes=num_nodes,
                param=0.001,  # sparsity
                seed=i,
                output_prefix=f"{num_nodes}_nodes/erdos_renyi_{num_nodes}_
                    {i}",
                save_binary=True,
            )
```

Listing 1: Graph Generation Script

2.3 A Note on Self-Loops

One might have noticed the GCN update equation, specifically in the message-passing layer, uses a modified $\hat{A} = A + I$. In the original graph A, the node's own feature vector $h_i^{(l)}$ is not included in the aggregation step. Only features from neighboring nodes $j \in \mathcal{N}(i)$ are considered. As a result, the transformation applied to node i depends entirely on its neighborhood and excludes any direct contribution from its current state.

Adding a self-loop ensures that each node is included in its own neighborhood. This has two primary effects. First, it allows the node to keep part of its identity across layers, which avoids over-reliance on neighboring information. Second, it also enables the propagation mechanism to blend a node's current embedding with aggregated features from its neighbors, which preserves local features. Ultimately, self-loops are considered in most practical implementations of undirected GCNs, which is why I also decided to consider them in this study.

3 Specifications

Now that the Erdős–Rényi model and CSR have been introduced, I will show the graphs that I used to benchmark the serial, vectorized, and parallelized versions.

Given the $G(n, p)$ model, all of my graphs used the same edgewise probability $p = 10^{-3}$. I generated 10 graphs with $n = 10^3$ and 10^4, and 5 graphs with $n = 10^5$ and 10^6. The larger graphs' `col_idx` arrays did end up taking considerably more space (scaling exponentially with edge count).

I also benchmarked this code using a CPU present on Boston University's Shared Computing Cluster, which was an Intel(R) Xeon(R) Gold 6242 CPU @ 2.80GHz.

To standardize, I used 64 dimensions as the standardized input and output dimensions of my GCN layer. I did this for several reasons, most notably because practical implementations of GCNs (i.e., neuroscience) tend to have far more dimensions than two (binary-valued) as inputs. Second, I wanted to make the vectorized version have at least a notable speedup, although one will see this did not present itself extraordinarily.

Therefore, the weight matrix used in the feature transformation layer was randomly generated (with a range from $[-0.5, 0.5]$), with dimension $\mathbb{R}^{64\times64}$.

4 Serial Version

The serial implementation of forward-pass inference is as naive as one might expect. The only "optimization" present was representing all matrices as lists in C, which improves cache locality. However, I argue this cannot be considered a notable optimization for serial benchmarking, as this is usually a stylistic choice that can be expected with performing large-scale serial computations in C.

4.1 Feature Transformation

As stated in section 3, the weight matrix was of a static size chosen during development. This implementation greatly reduced the amount of cases to consider, especially when only varying one variable (size of input graph's nodes), let alone other variables like sparsity.

4.1.1 Time Complexity of Matrix-Matrix Multiply for GCN Inference

Because of the static size of the weight matrix, one can calculate the expected time complexity of the feature transformation portion of the GCN forward-pass.

For $G(n, p)$ graphs with a weight matrix $W \in \mathbb{R}^{d \times d}$:

$$H^{(l)}W \in O(n \cdot d^2)$$

$$H^{(l)}W \in O(n \cdot 4096) \in O(n) \tag{4}$$

4.2 Message-Passing

Now, after the feature transformation layer in this project, $H^{(l)}W \in \mathbb{R}^{n \times 64}$.

4.2.1 Design Constraints on Message-Passing for GCNs

The message-passing step in GCNs is fundamentally sparse and local, which introduces both opportunities and challenges when optimizing for performance. Each node's update is independent of all others except its immediate neighbors due to permutation-invariance. This independence implies that, in theory, the aggregation across all nodes can be computed in parallel, which will be demonstrated later. This makes traditional SISD and SIMD techniques less effective, and requires careful design of memory access and scheduling to fully exploit parallel hardware.

4.2.2 Time Complexity of Message-Passing for GCN Inference

Now, using a similar analysis as I performed earlier, one can derive the expected time complexity of the message-passing layer as follows.

Given a $G(n, p)$ graphs with a weight matrix $W \in \mathbb{R}^{d \times d}$:

$$\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2} \in O\left(p \cdot \binom{n}{2} \cdot d\right)$$

$$\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2} \in O\left(p \cdot n^2 \cdot d\right)$$

$$\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2} \in O\left(n^2\right) \tag{5}$$

I have intentionally left the constants in the time complexities for feature transformation and message-passing, as these will become relevant in section 4.3.1.

4.3 Activation

I implemented an extremely simple ReLU function using the C expression `x>0?x:0`. In future iterations, especially for larger graphs, more layered GCNs, and backpropagation involved, I would love to see how varying this to something like GeLU or Swish impacts overall performance.

4.3.1 Preliminary Considerations

Given the Erdős-Rényi model, one can observe the crucial fact that, given a constant p, the number of edges in a graph scale roughly hundredfold as n increases tenfold.

This is foundational, and it is precisely why the message-passing stage scales with $O(n^2)$, while the feature transformation stage scales with $O(n)$.

However, it must be noticed that, while the constants were dropped in time complexity calculation, they are somewhat significant, and the fact only begins to hold true (and show itself) as graphs' node counts increase. This will also be shown in later results.

4.4 Code and Results

For the serial implementation, the code is straightforward:

```
#include "gcn.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

float relu(float x) {
    return x > 0 ? x : 0;
}

GcnLayer* create_gcn_layer(int input_dim, int output_dim) {
    if (input_dim <= 0 || output_dim <= 0) {
        return NULL;
    }
    GcnLayer *layer = (GcnLayer*)malloc(sizeof(GcnLayer));
    if (!layer) {
        return NULL;
    }
    layer->input_dim = input_dim;
    layer->output_dim = output_dim;
    layer->weights = (float*)malloc((size_t)input_dim * output_dim *
       sizeof(float));
    if (!layer->weights) {
        free(layer);
        return NULL;
    }
    return layer;
}

void free_gcn_layer(GcnLayer* layer) {
    if (layer) {
        free(layer->weights);
        free(layer);
    }
}

// Simple random weight initialization (Glorot/Xavier style often better)
void initialize_weights_random(GcnLayer* layer) {
    if (!layer || !layer->weights) return;
```

```
    size_t num_weights = (size_t)layer->input_dim * layer->output_dim;
    // Basic random initialization between -0.5 and 0.5
    for (size_t i = 0; i < num_weights; ++i) {
        layer->weights[i] = ((float)rand() / RAND_MAX) - 0.5f;
    }
}
// GCN forward pass
void gcn_forward(
    CsrGraph* graph,
    GcnLayer* layer,
    float* input_features,
    float* output_features
) {
    if (!graph || !layer || !input_features || !output_features) {
        return;
    }
    int num_nodes = graph->num_nodes;
    int in_dim = layer->input_dim;
    int out_dim = layer->output_dim;

    // H' = HW
    float* transformed_features = (float*)calloc((size_t)num_nodes *
       out_dim, sizeof(float));
    if (!transformed_features) {
        return;
    }

    for (int i = 0; i < num_nodes; ++i) { //nodewise
        for (int j = 0; j < out_dim; ++j) {
            float sum = 0.0f;
            for (int k = 0; k < in_dim; ++k) {
                sum += input_features[i * in_dim + k] * layer->weights[k *
                    out_dim + j];
            }
            transformed_features[i * out_dim + j] = sum;
        }
    }

    // Aggregation, init output features to zero
    memset(output_features, 0, (size_t)num_nodes * out_dim * sizeof(float)
       );

    for (int u = 0; u < num_nodes; ++u) {
        int start = graph->row_ptr[u];
        int end = graph->row_ptr[u + 1];
        for (int edge_idx = start; edge_idx < end; ++edge_idx) {
            int v = graph->col_idx[edge_idx]; // Neighbor node

            float norm_uv = 1.0f / sqrtf((float)graph->degrees[u] * graph
               ->degrees[v]);
            for (int k = 0; k < out_dim; ++k) {
                output_features[u * out_dim + k] += norm_uv *
                   transformed_features[v * out_dim + k];
            }
```

```
        }
    }
    // Activation (ReLU)
    for (int i = 0; i < num_nodes * out_dim; ++i) {
        output_features[i] = relu(output_features[i]);
    }

    free(transformed_features);
}
```

Listing 2: Serial Execution Benchmark

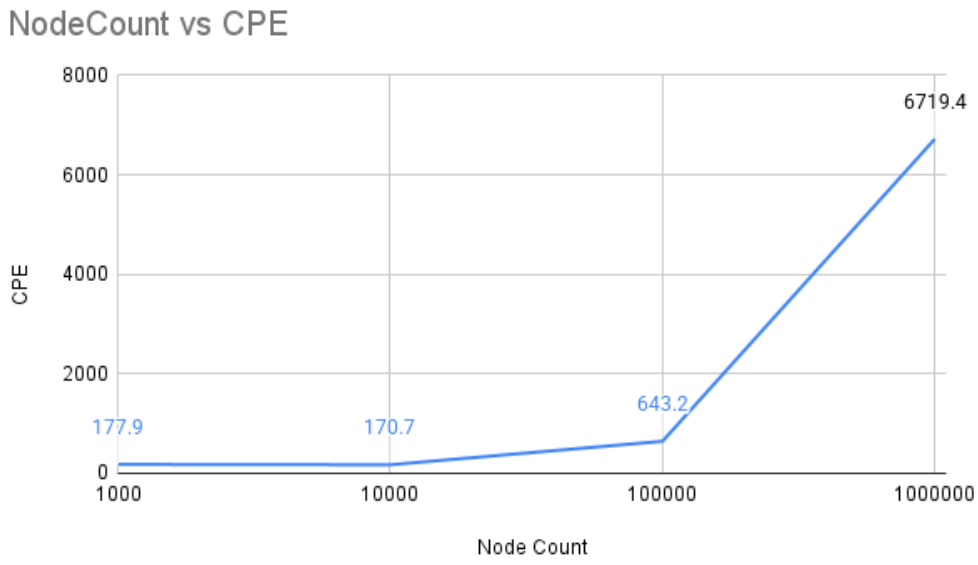

Figure 1: A plot of the cycles per iteration as a function of node count for serial execution.

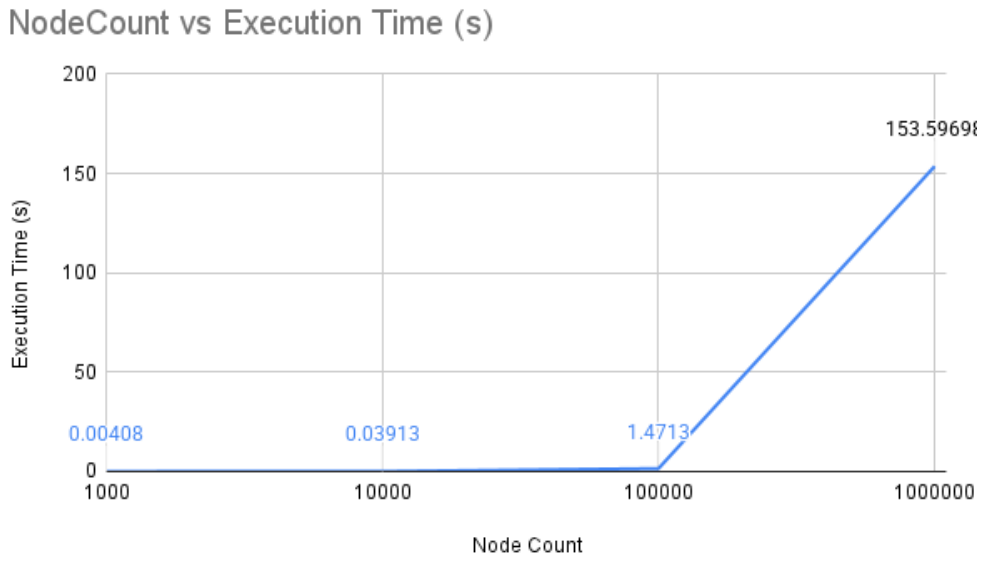

Figure 2: A plot of the execution time taken in seconds as a function of node count for serial execution.

4.4.1 Discussion of CPE

Notice that CPE as a function of node count in serial execution is relatively predictable. For graphs with 10^3 and 10^4, one can observe a relatively small CPE, which reduces for the latter. However, as the number of nodes increases to 10^5, one can see a substantial increase in CPE by around 3.8×. One can then see an over 10× increase in CPE.

4.4.2 Discussion of Time Taken

Execution time as a function of node count in serial execution is also relatively predictable, largely for the same reasons. One can observe the same slight drop in graphs with 10^4 nodes as before, along with a noticeable increase in execution time for graphs with 10^5 nodes, and an overwhelming 150× increase for the 10^6 case.

4.5 Overall Comments

The reduction in CPE between graphs with 10^3 and 10^4 nodes likely may be due to some more rigorous caching and instruction locality present. The additional overhead of ten times the number of nodes, along with retrievals from further caches, likely causes the noticeable increase in both CPE and execution time when observing 10^5 nodes. Lastly, due to the overwhelming size of the `col_idx` array for graphs with 10^6 nodes, caching becomes obsolete, which is why one sees 10× and 150× increases in CPE and execution time, respectively.

Overall, one can see significant slowdown as the node count becomes large. To verify the important considerations I noted in 4.3.1, the rest of the experiments will feature first vectorizing the feature transformation stage, and then parallelizing the message-passing stage.

5 Vectorized Version (AVX2)

After implementing a scalar baseline for GCN inference, I observed that the feature transformation step, was regular, memory-aligned. Additionally, because of the 64-dimensional input/output pairing, this made feature transformation a strong candidate for low-level vectorization using Advanced Vector Extensions 2 (AVX2) intrinsics.

5.1 Why Exclusivity?

Although both feature transformation and message passing are core components of GCN inference, only the former was selected for explicit vectorization. I decided on this design choice due to the fundamental difference in data access patterns and arithmetic structure between the two stages.

The feature transformation stage performs a series of predictable dot products between each node's feature vector and a weight matrix. This operation is compute-bound, with a high ratio of arithmetic operations to memory accesses, involving structurally regular memory access over contiguous arrays.

These properties make it well-suited to using AVX2 intrinsics, which can operate on multiple floating-point values in a single instruction cycle.

On the other hand, the message-passing stage requires each node to access and aggregate transformed feature vectors from its arbitrary set of neighbors. These neighbors are not contiguous in memory and vary in number per node, making SIMD vectorization extremely inefficient.

Due to these challenges, along with the computational complexity calculated and hypothesized earlier, I opted to keep the scalar implementation of message passing and focus optimization efforts on the vectorizable portion of the pipeline: the feature transformation.

5.2 Vectorizing Feature Transformation

To accelerate the matrix-matrix multiplication operation, I used AVX2 intrinsics to perform 256-bit vectorized dot products. Each AVX2 register can hold 8 single-precision FP values, enabling partial sums of dot products to be computed in parallel. The main loop was unrolled to align with register width, and memory accesses were padded to avoid misalignment penalties.

5.3 Code and Results

```
// other code same as serial...
static inline float hsum256(__m256 v) {
    __m128 vlow  = _mm256_castps256_ps128(v);
    __m128 vhigh = _mm256_extractf128_ps(v, 1);
    vlow  = _mm_add_ps(vlow, vhigh);
    __m128 shuf = _mm_movehdup_ps(vlow);
    vlow = _mm_add_ps(vlow, shuf);
    shuf = _mm_movehl_ps(shuf, vlow);
    vlow = _mm_add_ss(vlow, shuf);
    return _mm_cvtss_f32(vlow);
}
```

```
void gcn_forward_avx(CsrGraph* graph, GcnLayerAvx* layer, float*
   input_features, float* output_features) {
    int num_nodes = graph->num_nodes;
    int in_dim = layer->input_dim;
    int out_dim = layer->output_dim;
    int* row_ptr = graph->row_ptr;
    int* col_idx = graph->col_idx;
    int* degrees = graph->degrees;

    // temp buffer for transformed features
    float* transformed_features = (float*)malloc((size_t)num_nodes *
       out_dim * sizeof(float));
    if (!transformed_features) {
        return;
    }

    // Feature transformation: (input_features) x (weights)
    for (int i = 0; i < num_nodes; ++i) {
        for (int j = 0; j < out_dim; ++j) {
            __m256 acc = _mm256_setzero_ps();
            int k;
            for (k = 0; k <= in_dim - 8; k += 8) {
                __m256 v_in = _mm256_loadu_ps(&input_features[i * in_dim +
                    k]);
                __m256 v_w  = _mm256_loadu_ps(&layer->weights[k * out_dim
                   + j]);
                acc = _mm256_add_ps(acc, _mm256_mul_ps(v_in, v_w));
            }
            float sum = hsum256(acc);
            // handle leftovers if % 8 !=0
            for (; k < in_dim; ++k) {
                sum += input_features[i * in_dim + k] * layer->weights[k *
                    out_dim + j];
            }
            transformed_features[i * out_dim + j] = sum;
        }
    }
// other code same as serial...
```

Listing 3: AVX2-Vectorized Feature Transformation

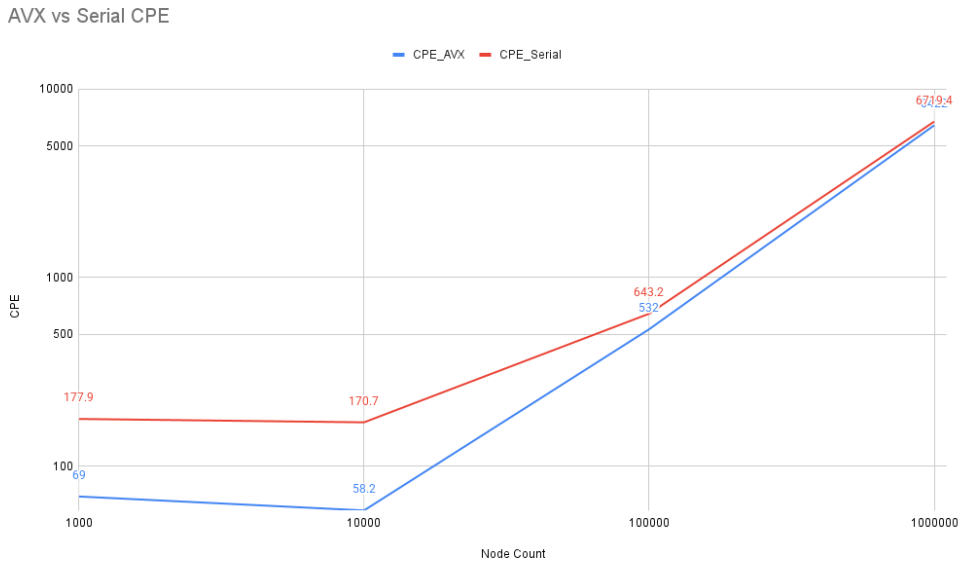

Figure 3: A plot of the cycles per iteration as a function of node count for vectorized execution.

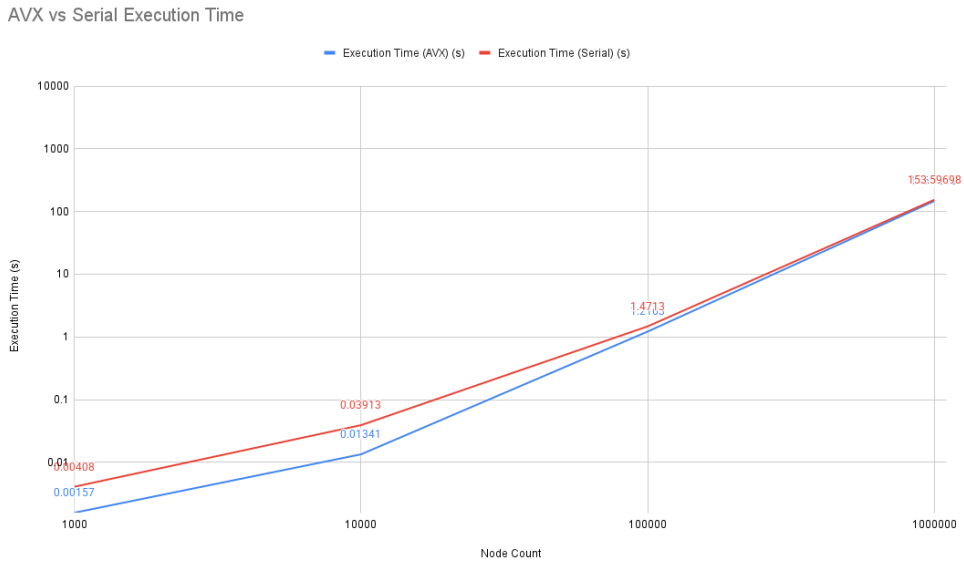

Figure 4: A plot of the execution time taken in seconds as a function of node count for vectorized execution.

5.3.1 Discussion of CPE

To better show these results, I opted to use a log-scale on the y-axis for CPE. Notice that the AVX2 implementation features similar noticeable traits for CPE. There is a slight drop from 10^3 to 10^4 nodes, and a noticeable increase $4\times$ and $10\times$ going forward. The AVX2 implementation's results are all below the results for serial execution, with varying gaps proportional to the number of nodes.

5.3.2 Discussion of Time Taken

Once again, to better show these results, I opted to use a log-scale on the y-axis for execution time. Similarly enough, one sees a gradual increase, with the AVX2 implementation's results consistently being under (better) than the serial execution benchmark, once again with varying gaps proportional to the number of nodes.

5.4 Overall Comments

The gaps between serial execution and vectorized execution were not as clear on a linear y-axis, which is why a log-scale was necessary. Ultimately, although the AVX2 implementation is considerably faster for smaller graph sizes as the feature transformation stage is vectorized, these returns begin to diminish as the 10^5-node graph size is approached. This aligns with all previous hypotheses and calculation, as it will soon be revealed that the message-passing stage is the true bottleneck for GCNs.

6 Parallelized Version (OpenMP)

While vectorization using AVX2 somewhat accelerated the feature transformation stage, the message-passing step was still a because of its inherently sparse and irregular structure. To improve performance in this stage, I implemented coarse-grained thread-level parallelism using OpenMP. This allowed multiple nodes to perform aggregation in parallel, taking advantage of modern multi-core CPUs without requiring SIMD-friendly memory layouts.

To do so, I implemented threading using 2, 4, 8, 16, 32, and 64 threads.

6.1 Parallelizing Message-Passing

In message-passing, each node's update is independent of all others, provided the output array is written in a thread-safe way. This is trivially parallel, as shown below. I therefore parallelized the outer loop over nodes with OpenMP.

Because each thread writes to a disjoint portion of the 'output_features' array, no locks or atomic operations were required.

The parallel implementation scales well for large graphs with balanced node degrees but suffers in two edge cases. When the graph is small, the overhead of spawning threads outweighs the performance gains. Also, when degree variance is high, threads may finish unevenly and cause stragglers.

However, within the scope of this project and for most realistic graph sizes and density levels, OpenMP parallelization provided substantial speedups over the scalar baseline.

6.2 Code and Results

```
// feature transformation same as vectorized
#pragma omp parallel for
for (int u = 0; u < num_nodes; ++u) {
    for (int j = row_ptr[u]; j < row_ptr[u+1]; ++j) {
        int v = col_idx[j];
        float norm = normalization[u][v];
        for (int k = 0; k < out_dim; ++k) {
            output_features[u * out_dim + k] +=
                norm * transformed_features[v * out_dim + k];
        }
    }
}
// other code same as serial
```

Listing 4: OpenMP-Parallelized Message Passing

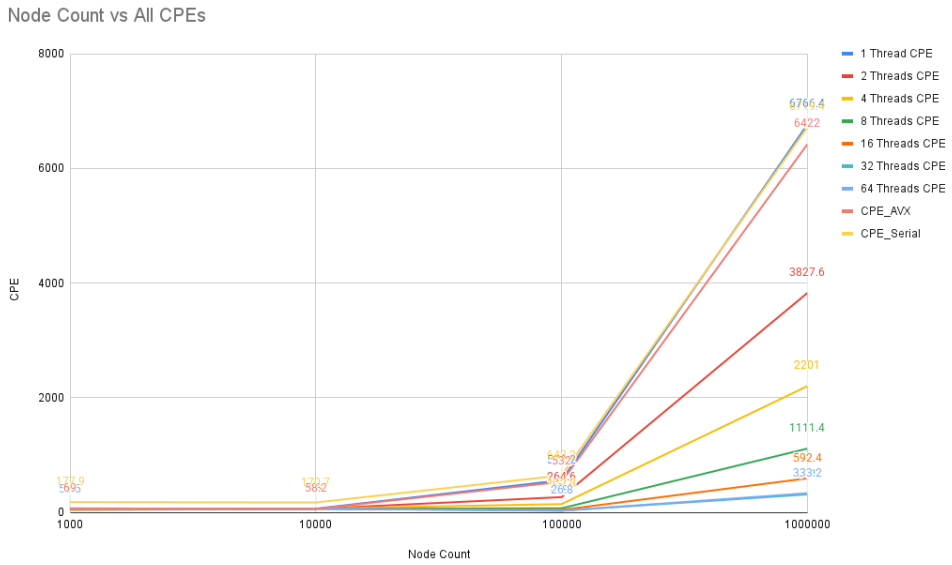

Figure 5: A plot of the cycles per iteration as a function of node count for multithreaded execution.

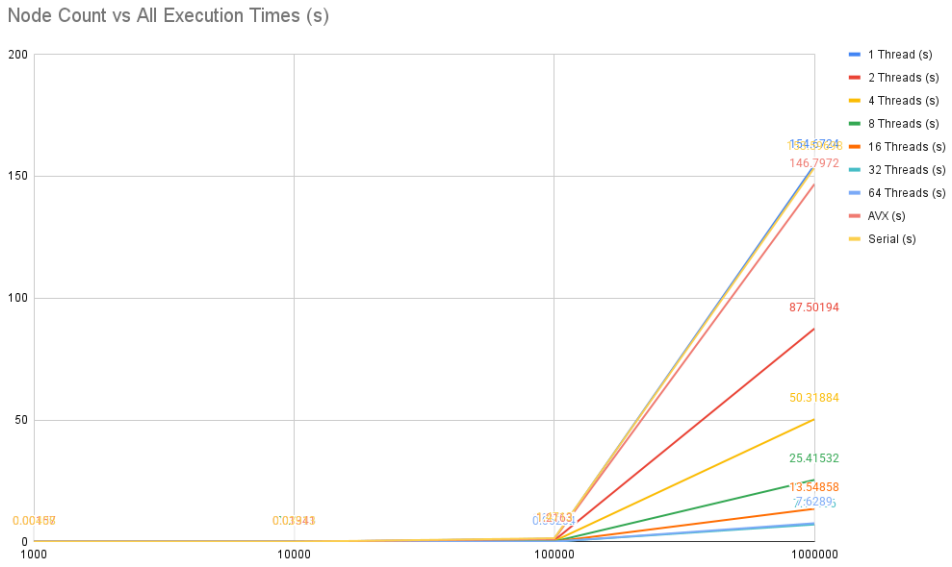

Figure 6: A plot of the execution time taken in seconds as a function of node count for multithreaded execution.

6.2.1 Discussion of CPE

At last, one can observe a steady but somewhat predictable drop in CPE by slightly less than half as the number of threads doubles up until 64, when the additional threading overhead starts to play a part and it becomes slightly less efficient than the 32-threaded implementation.

6.2.2 Discussion of Time Taken

Once again, one can see a slightly-less-than-half drop in the execution time as the number of threads doubles, with the same noticeable stall in efficiency when using 64 threads.

6.3 Overall Comments

Overall, this reinforces and confirms the initial hypothesis and calculations performed about the time complexities, **especially** for larger graphs. Notice how the performance gains are completely negligible in both CPE and execution time between multithreaded and vectorized approaches for graphs with 10^3 and 10^4 nodes, and noticeable performance increases only start to appear around the 10^5-node mark. This once again confirms the secondary hypothesis from earlier.

7 Closing Thoughts and Final Results

Table 1: Average CPE ($\pm$ Std. Dev.)

Nodes	1T	2T	4T	8T	16T	32T	64T	Serial	Vect.
1,000	50.8 ± 1.6	48.8 ± 1.5	47.6 ± 1.4	49.6 ± 1.6	51.7 ± 1.9	54.5 ± 2.9	50.6 ± 1.8	177.9 ± 80.2	69.0 ± 25.1
10,000	63.0 ± 1.6	61.4 ± 1.6	59.4 ± 1.5	60.6 ± 1.6	63.0 ± 1.6	61.7 ± 2.1	63.1 ± 1.7	170.7 ± 12.0	58.2 ± 3.6
100,000	559.2 ± 10.9	264.6 ± 5.4	132.5 ± 2.7	88.6 ± 1.8	41.8 ± 1.3	28.0 ± 0.4	26.8 ± 0.3	643.2 ± 1.3	542.6 ± 21.5
1,000,000	$6\,766 \pm 125$	$3\,828 \pm 67$	$2\,194 \pm 47$	$1\,108 \pm 23$	592 ± 14	311 ± 11	333 ± 9	$6\,719 \pm 41$	$6\,567 \pm 125.5$

Table 2: Average Execution Time (s) ($\pm$ Std. Dev.)

Nodes	1T	2T	4T	8T	16T	32T	64T	Serial	Vect.
1,000	0.0012 ± 0.0000	0.0011 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0010 ± 0.0000	0.0013 ± 0.0000	0.0012 ± 0.0000	0.0041 ± 0.0018	0.0016 ± 0.0006
10,000	0.0145 ± 0.0000	0.0142 ± 0.0000	0.0140 ± 0.0000	0.0140 ± 0.0000	0.0140 ± 0.0000	0.0142 ± 0.0000	0.0146 ± 0.0000	0.0391 ± 0.0028	0.0134 ± 0.0008
100,000	1.280 ± 0.0000	0.606 ± 0.0000	0.303 ± 0.0000	0.202 ± 0.0000	0.103 ± 0.0000	0.0649 ± 0.0000	0.0620 ± 0.0000	1.471 ± 0.0026	1.241 ± 0.0488
1,000,000	154.67 ± 0.0009	87.50 ± 0.0103	50.30 ± 0.0068	25.40 ± 0.0025	13.50 ± 0.0015	7.12 ± 0.0011	7.63 ± 0.0012	153.60 ± 0.9447	150.11 ± 2.8703

As one can see, the time difference between feature transformation and message-passing has been demonstrated as node counts increase. This reveals a lot of information, first (and most simply) that the message-passing stage is considerably more computationally expensive due to poor cache locality.

However, as one has seen, it is definitely possible to parallelize independent node-wise computations for the following layer. It is precisely because of GCNs' layer-like structure that this fact is true, and one can safely avoid any race conditions or concurrency problems.

Note that this is really only observable as graphs become really large. It must also be mentioned that this project used a single sparsity. Though nondeterministic, because of the large number of nodes, I would imagine little variance in the number of edges with $p = 10^{-3}$. I would love to spend more time in a future iteration of this project exploring different sparsities, especially considering $p \in O(\frac{1}{n}$ or $p \in O(\log(n))$.

Some further work would include varying sparsity as a baseline. I would also love to explore N-layer networks, as this is typically how GCNs are implemented in practice. Of course, N can't be too large due to over-smoothing, but I wonder how these optimizations stack up across multiple layers.

Following an N-layer implementation, the only reasonable subsequent implementation would be to extend this beyond the simple forward-pass "inference" optimization, and explore optimizing backpropagation for sparse graphs. Though this isn't trivial, it would be interesting to see how the concepts of sparsity, node count, and the message-passing stage all play a part.

8 Appendix: Compilation Instructions

The code provided in this paper includes functions taken from the implementation files which are used in benchmarking files that execute on pre-generated graphs. The actual generation of graphs and their subsequent benchmarking files can be found on my GitHub.

To compile the code, first clone my GitHub repository located below. Then, run the Makefile using `'make all'`.

`https://github.com/amruth-sn/high-perf-gnn/`

References

[1] Jie Zhou et al. "Graph neural networks: A review of methods and applications". In: *AI Open* 1 (2020), pp. 57–81.

[2] Thomas N Kipf and Max Welling. "Semi-supervised classification with graph convolutional networks". In: *ICLR* (2017).

[3] PyTorch Geometric. *torch_geometric.nn.conv.GCNConv*. Accessed: 2024-05-01. 2023. URL: `https://pytorch-geometric.readthedocs.io/en/2.4.0/generated/torch_geometric.nn.conv.GCNConv.html`.

[4] Kenta Oono and Taiji Suzuki. "Graph neural networks exponentially lose expressive power for node classification". In: *ICML* (2019).