CPU-PCGN: Procesamiento Eficiente de Redes Convolucionales de Grafos en Arquitecturas CPU

Máster Nuevas Tecnologías en Informática

Trabajo Fin de Máster

Autor:
Nicolás Meseguer Iborra
Tutor/es:
Manuel Eugenio Acacio Sánchez
José Luis Abellán Miguel
Francisco Muñoz Martínez

Facultad Informática Universidad Murcia

7 de Junio de 2022

CPU-PCGN: Procesamiento Eficiente de Redes Convolucionales de Grafos en Arquitecturas CPU

Implementación, Evaluación y Mejora

Autor
Nicolás Meseguer Iborra

Tutor/es
Manuel Eugenio Acacio Sánchez
Ingeniería y Tecnología de Computadores
José Luis Abellán Miguel
Ingeniería y Tecnología de Computadores
Francisco Muñoz Martínez
Ingeniería y Tecnología de Computadores

Facultad de Informática

Máster Nuevas Tecnologías en Informática

Departamento de Ingeniería y Tecnología de Computadores

UNIVERSIDAD DE MURCIA

Murcia, 7 de Junio de 2022

Declaración firmada sobre originalidad del trabajo

D. **Nicolás Meseguer Iborra**, con DNI [redacted], estudiante de la titulación de **Máster Nuevas Tecnologías en Informática** de la Universidad de Murcia y autor del TF titulado "**CPU-PCGN: Procesamiento Eficiente de Redes Convolucionales de Grafos en Arquitecturas CPU**".

De acuerdo con el Reglamento por el que se regulan los Trabajos Fin de Grado y de Fin de Máster en la Universidad de Murcia (aprobado C. de Gob. 30-04-2015, modificado 22-04-2016 y 28-09-2018), así como la normativa interna para la oferta, asignación, elaboración y defensa delos Trabajos Fin de Grado y Fin de Máster de las titulaciones impartidas en la Facultad de Informática de la Universidad de Murcia (aprobada en Junta de Facultad 27-11-2015).

DECLARO:

Que el Trabajo Fin de Máster presentado para su evaluación es original y de elaboración personal. Todas las fuentes utilizadas han sido debidamente citadas. Así mismo, declara que no incumple ningún contrato de confidencialidad, ni viola ningún derecho de propiedad intelectual e industrial.

Murcia, a 7 de Junio de 2022

[redacted]

Fdo.: Nicolás Meseguer Iborra
Autor del TFM

A mi madre Carmen, a mi pareja y a mi familia, sin los cuales no habría podido acabar esta tesis. Al Dr. D. Manuel E. Acacio Sánchez, Dr. D. José L. Abellán Miguel y D. Francisco Muñoz Martínez, por haber sido profesores, tutores y compañeros.

Resumen

Debido al gran éxito de las redes neuronales convolucionales (*Convolutional Neural Networks* o CNNs) en Deep Learning (DL), la operación convolucional se ha trasladado más allá del procesamiento de datos mapeados en el espacio Euclídeo (por ejemplo, las imágenes), a datos estructurados en forma de grafos (espacio no Euclídeo) como por ejemplo los grafos de la web, redes sociales, redes de citaciones a artículos, etc., dando lugar a las redes convolucionales de grafos (*Graph Convolutional Networks* o GCNs).

Las GCNs han ido ganando popularidad debido a su eficacia en aplicaciones del mundo real como la recomendación, el procesamiento del lenguaje natural, etc. Dado que las redes neuronales y la propagación de grafos tienen una alta complejidad de cálculo, las GPU se han introducido para el entrenamiento. Sin embargo, es muy difícil realizar un cálculo eficiente de GCN debido a la irregularidad de los grafos.

En este trabajo, presentamos CPU-PCGCN, un método novedoso y alternativo a PCGCN para acelerar el cálculo de GCN aprovechando la localidad de los grafos usando únicamente la CPU como unidad de cómputo. Demostramos que los grafos del mundo real suelen tener la propiedad de agrupación que puede utilizarse para mejorar la localidad de los datos en la computación de las GCNs.

A continuación, CPU-PCGCN propone dividir el grafo completo en trozos según la localidad y procesar los subgrafos con una estrategia de computación dual que incluye métodos de procesamiento selectivo y completo para subgrafos dependiendo de su dispersión.

En comparación con la implementación base de GCN (PyGCN) en conjuntos de datos reales y sintéticos, nuestra implementación únicamente sobre CPU consigue un aumento de velocidad de hasta 3.94 veces en el mejor de los casos.

CPU-PCGCN ha sido conectada con dos herramientas novedosas para la generación sintética de grafos, PaRMAT (versión paralela de la popular herramienta RMAT) y Graphlaxy (framework reciente para la generación masiva de grafos sintéticos). Esto permitirá ayudar al usuario a estudiar cargas de trabajo sintéticas en el entrenamiento de una GCN, así como evaluar optimizaciones de la propia herramienta CPU-PCGCN para acelerar todavía más el costoso proceso de entrenamiento de las GCNs.

Abstract

Due to the great success of *Convolutional Neural Networks* (CNNs) in Deep Learning (DL), the convolutional operation has moved beyond the processing of data mapped in Euclidean space (e.g. images), to data structured in the form of graphs (non-Euclidean space) such as web graphs, social networks, article citation networks, etc., giving rise to *Graph Convolutional Networks* (GCNs).

GCNs have been gaining popularity due to their huge success in real-world applications such as recommendation, natural language processing, etc. Since neural networks and graph propagation have high computational complexity, GPUs have been introduced for training. However, it is notoriously difficult to perform efficient computation of GCNs due to the irregularity of graphs.

In this thesis, we present CPU-PCGCN, a novel and alternative method of PCGCN to accelerate GCN computation by exploiting the locality of graphs using only the CPU as the computational unit. We show that real-world graphs often have the clustering property that can be used to improve data locality in GCN computation.

CPU-PCGCN then proposes to split the complete graph into chunks according to locality and process the subgraphs with a dual mode computation strategy that includes selective and complete processing methods for subgraphs depending on their sparsity.

Compared to the baseline implementation of GCN (PyGCN) on real-world and synthetic datasets, our CPU-only implementation on top of PyTorch achieves up to 3.94× speedup over the best case.

CPU-PCGCN has been developed along with two novel tools for synthetic graph dataset generation, PaRMAT (a parallel version of the popular RMAT tool) and Graphlaxy (a recent framework for massive synthetic graph generation). This will help the user to study synthetic workloads in the training of a GCN, as well as to evaluate optimisations of the CPU-PCGCN tool itself to further accelerate the costly process of training GCNs.

Contents

List of Figures

List of Tables

1 Introduction and Motivation

In recent years, data has played a very important role in society and, as a consequence, the need to preserve and organize it has arisen. The amount of data generated on a daily basis is overwhelming, therefore we need algorithms that are able to automatically process such an enormous amount of information.

Machine learning (ML) is a branch of artificial intelligence (AI) in computer science, which focuses on the use of data and algorithms to give the computers the ability to learn without being explicitly programmed, and that is gradually improving its accuracy. Because of its scalable capacity and wide range of applications, it has become a high-powered branch of engineering [9].

Since the beginning of ML, the goal has been to mimic the way the human brain works, hence computational learning approaches have been built with the human biological system in mind. These Brain-Inspired approaches are now commonly employed for Artificial Intelligence issue solving. Deep Learning (DL), in particular, is the outperforming approach.

DL is a machine learning technique that teaches computers to do what comes naturally to humans: learn by example. It is one of the most promising sub-fields and has experienced strong growth in the last few years. In deep learning, a computer model learns to perform classification tasks directly from images, text, sound, i.e., data.

The basis of DL models is the Neural Network (NN), which aims to imitate the functioning of the brain by grouping together a collection of small artificial neurons in a series of layers, with the outputs of one layer forming the inputs of the next one. The capacity of NNs to extract high-level features directly from input data is a significant advance over earlier strategies relying on expert-designed rules or hand-drawn features.

As a result, the term Deep applied to a NN (Deep Neural Network) derives from the prospect of having an usually big number of layers, each with a high number of neurons, to reduce prediction error.

DNN models typically serve two purposes, either training (teaching the network through a vast number of examples so that it can later formulate a proper conclusion from actual data) or inference (once the network has been trained, it is able to generate a result that is close to the real one for data never seen before).

DNNs have achieved a lot of advances in many areas, e.g., image and speech recognition, natural language processing, and many real-world applications are currently based on them, like self-driving, search engine, recommendation and so on. In fact, nowadays there are many DNN models, although depending on the task targeted, not all models perform well. For example, Convolutional Neural Networks (CNNs) are used mainly for image processing and classification, and have achieved a huge success in computer vision [10]; Recurrent Neural Networks (RNNs) have been typically used in Natural Language Processing (NLP) [11], etc.

Because of the vast amount of data generated today on the Internet or in sensor deployments from the Internet of Things (IoT) field, as well as the large computational capacities of today's hardware, deep and complex neural networks can be trained with minimal precision error, yielding results never seen before.

Graphs are all around us; real world objects are often defined in terms of their connections to other things, from which knowledge can be extracted. A set of objects (vertices), and the connections (edges) between them, are naturally expressed as a graph. Indeed, recent researches on analyzing graphs with machine learning have been receiving more and more attention from the community because of the great expressive power of graphs and similarity to real-word data representation, i.e. social networks [12], scientific publications [13], or protein-to-protein interaction networks [14].

Traditional DNNs (CNNs, MLPs or RNNs) are not able to correctly extract information from a graph due to the very nature of the data. For this reason, Graph Neural Networks (GNNs) emerge as a new type of DNN especially adapted to process this type of data structures.

GNNs have exploded onto the machine learning scene in recent years owing to their capability to model and learn from graph-structured data. Such an ability has strong implications in a wide variety of fields whose data is inherently relational, for which conventional neural networks do not perform well [6]. This way, GNNs can be defined as deep learning models that operate on a graph domain, focusing on tasks such as node classification or link prediction [15].

Among the different types of GNNs studied so far by the scientific community, Graph Convolutional Networks (GCNs) are currently receiving a lot of attention. In particular, GCNs get their inspiration from CNNs, and more precisely, apply the concept of convolutional layer to graph-structured data (e.g., social networks, knowledge graphs, etc.), and hence their name [16].

Most common target applications for GCNs include performing image differentiation problems like “Zero-Shot Learning” [17] (identifying an unknown labelled image and grouping it into known ones), or solving various problems related to research operations and combinatorial optimisation applications, even taking a certain length of molecular fingerprints as input and generate predicted molecular structures (MolGAN [18] is a GCN model which helps to create new molecular structures). Another significant application of GCNs is to solve community prediction problems, such as Karate Club of Zachary[1] (a problem based on the dispute between the administrator and the instructor of the club).

In a GCN, a single convolution operation on a graph transforms and aggregates feature information from a node’s one-hop graph neighborhood, and multiple such convolutions are stacked to propagate vertices’ information across far reaches of a graph. It is worth mentioning that the aggregation step uses a heavily sparse data structure, i.e., the adjacency matrix.

Even though this model seems to be the solution for “real-world” graphs in the deep-learning paradigm, its tremendous size is a conditioning factor that has an impact on the computation. This fact has recently led to the proposal of a large number of software acceleration models [19] [20] [21] [8].

One of the biggest challenges yet to face is to speed up the graph processing which is increasingly constrained by main memory accesses. In particular, they suffer from poor temporal locality, as the irregular structure of graphs (graphs usually have many vertices but relatively few edges, resulting in a huge matrix with very few connections) results in seemingly random accesses that are hard to predict ahead of time. Not only that, but, they suffer from poor spatial locality, due to the high degree of sparsity of the adjacency matrix [22].

Partition-Centric Processing for Accelerating Graph Convolutional Network (PCGCN), which is a recently software acceleration model for GCNs, [8] proposes to partition the input graph and to employ two alternative ways of computation, depending on the sparsity of each of the subgraphs. To this end, PCGCN proposes to leverage the locality of real-world graphs to accelerate GCN computing by accelerating the graph propagation.

Recent advances in GPU hardware technologies offer potentially new avenues to accelerate the inference and training of GCNs, PCGCN takes advantage of the Tensor Cores in the GPU (and proposes to move all computation to the GPU) to further accelerate graph propagation, despite this, the sparsity and irregularity in graphs make it notoriously difficult to perform efficient computing.

[1]`http://konect.cc/networks/ucidata-zachary/`

In this work, we present the CPU-PCGCN framework, an implementation of PCGCN especially conceived for being used on CPU-powered systems. CPU-PCGCN enhances cache-efficiency and memory performance to efficiently execute GCNs. This framework is built on top of a GCN model written in PyTorch, PyGCN [23]. It also makes use of several third-party contemporary tools, such as METIS [24] for efficient partitioning taking into account graph properties, or synthetic data generators, such as Graphlaxy or PaRMAT [25]. In addition to this, CPU-PCGCN employs different techniques (task-level parallelism, matrix properties, or even graph representation properties) to speed up computation and model processing.

When CPU-PCGCN is applied, a GCN model is executed in bulk synchronous stages of message exchange across vertex subsets known as partitions or subgraphs. This model employs a hybrid partition-centric processing method that can choose the best mode (sparse or dense computing) for any pair of graph partitions with distinct computation properties. CPU-PCGCN pulls excellent performance from the memory hierarchy in this manner compared to traditional GCNs models. CPU-PCGCN is openly distributed to the scientific community[2].

The rest of this work is organized as follows: Chapter 2 provides some context for the reader, related to the field of DL and graph domain. Chapter 3 discusses different techniques for accelerating the computation of GCNs and PCGCN-related models, as well as partitioning techniques. Then, Chapter 4 presents the bulk of the work, CPU-PCGCN. Finally, Chapter 5 evaluates the results obtained with CPU-PCGCN compared to various other GCNs models, and Chapter 6 wraps-up with the conclusions and future work.

[2]`https://github.com/NicolasMeseguer/pcgcn`

2 Background

2.1 Artificial Neuron

AI is now a growing and inventive discipline in a variety of study fields, with the goal of automating human work. To that purpose, algorithms must mimic the functioning of the human brain and the way we learn.

In Figure 2.1 (a), we can see the smallest unit of learning in the brain, the neuron. Then, in (c) we can see how our neurons work and how they communicate, receiving input signals from dendrites and producing output signals to other neurons via the axon, this communication is called synapse.

As mentioned before, these AI algorithms try to mimic the way the human brain works, thus, in (b) we can see the "artificial" neuron used by the models, and in (d) we can see how models work; the output of one neuron is connected to the input of multiple neurons.

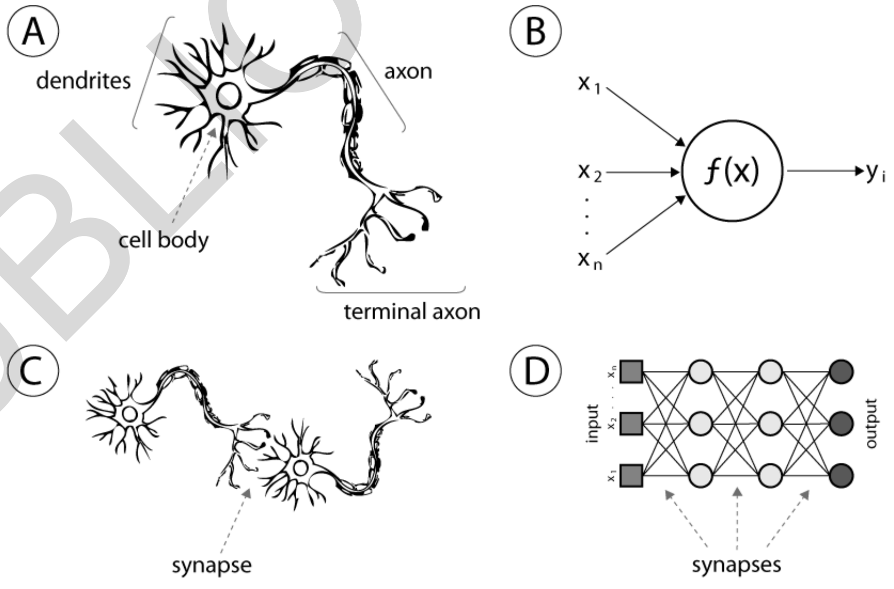

Figure 2.1: Biological and Artificial Neuron: (a) human neuron; (b) artificial neuron; (c) biological synapse; and (d) ANN synapses [1]

Learning in the human brain involves changing the value sent at this synapse, known as the weight. When the value of this weight is changed, the value conveyed between a dendrite and an axon terminal changes, and the neuron learns. This theory underpins the operation of a Neural Network (NN).

In Figure 2.2, each neuron computes by multiplying each input value from another neuron by a weight and adding it to the remainder of the input values. Then. the accumulated total is subjected to a non-linear activation function, such as ReLU, sigmoid or hyperbolic, and finally, the result (output y) is spread to the connected neurons.

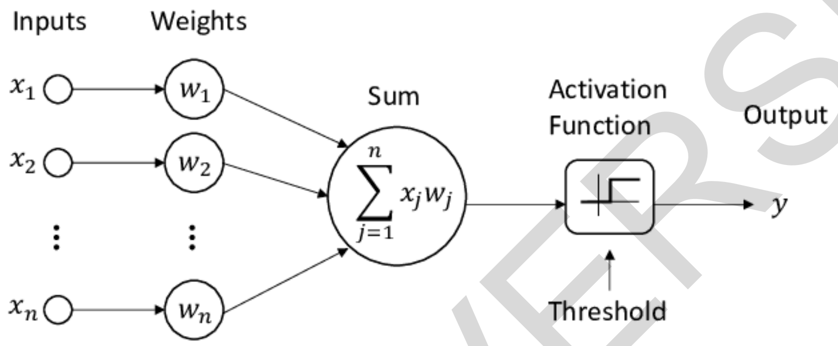

Figure 2.2: How an artificial neuron works [2]

In general, processing a DNN involves two fundamental phases: first, a training phase in which the neural network is trained by modifying the weights of the model using a large number of input examples, and then, an inference phase in which a prediction is made from previously unseen data using the (already trained) weights of the DNN.

2.2 Structure of a Neural Network

The fundamental operation of a NN is depicted in Figure 2.3. As we can see, in general, neurons (circles) are organized in layers, with connections forming from one layer to the next. When a NN has more than 3 layers, we are discussing a Deep (Learning) Neural Network (DNN). There are three sorts of layers in particular:

- *Input Layer*: It is the neural network's input layer (red color), and it holds the initial data with which to train or infer, such as pixels in an image or sensor information, etc. that make up the data to be analyzed. The output connections link to the network's next layer.

- *Hidden Layer*: The DNN's hidden layers are its intermediate layers (orange color). The amount of intermediate layers is unknown and determined by the network's architecture. It accepts the outputs of the preceding layer (another intermediate or input layer) as input data and transfers the results to the network's next layer (another intermediate or output layer).
- *Output Layer*: This is the neural network's last layer. It receives data from the previous layer (either the input layer or another intermediate hidden layer) and returns the final results, typically a vector of probabilities (a.k.a., scores).

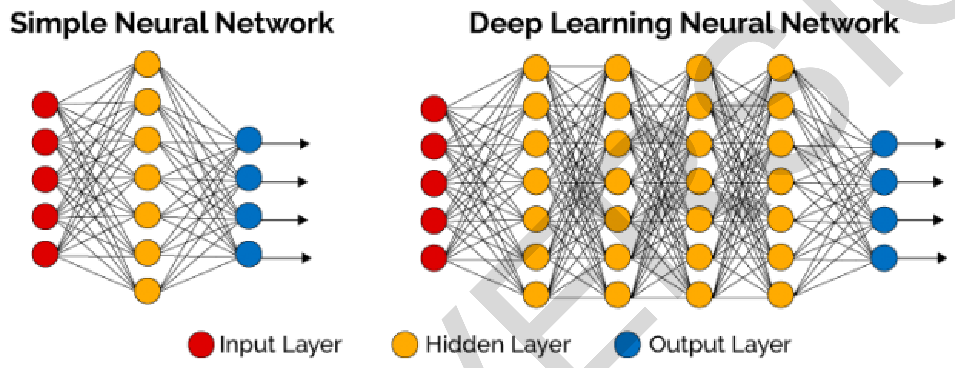

Figure 2.3: NNs and DNNs [3]

Furthermore, there are two unique types based on how the model processes the information:

- *Feed-forward networks*: These neurons have solely forward connections. That is, connections from a neuron in layer i towards layer $i+1$. In this category, we have several well-known models such as: MLPs, CNNs and GCNs, which will be discussed below.
- *Backward networks*: A neuron in layer i can be connected to neurons in its own layer (feedback), as well as, to neurons in the preceding layer $(i-1)$. Some examples belonging to this category are Long Short Term Memory Networks (LSTM) and Recurrent Networks (RNNs).

2.3 Training vs Inference

As previously stated, the deployment of a NN involves two different phases:

- *Training*: NNs are learning models that are built on the *inductive learning approach*, which entails teaching the network through a vast number of examples so that it can later infer a proper conclusion from actual data. This type of training can be done in 2 different ways: Offline Training (differentiate between the training phase and the inference phase) and Online Training (while producing real conclusions, the network is being trained).

- *Inference*: The inference phase begins once the network has been trained. Thus, for a data set never seen before, it is able to generate a result that is close to the real one.

The training phase is usually performed in datacenters, using powerful processors (typically GPUs) and specialized accelerators. Once the weights have been obtained, the model can be passed into production through the inference phase.

2.4 Graph Domain

Graphs are all around us; real world objects are often defined in terms of their connections to other things. A set of objects, and the connections between them, are naturally expressed as a graph [4]. Figure 2.4 introduces a graph representation, which represents the relations (edges) between a collection of entities (vertices).

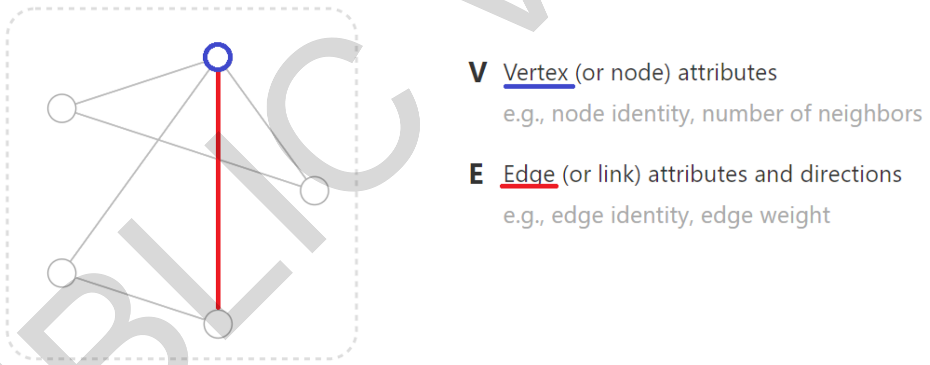
V Vertex (or node) attributes
e.g., node identity, number of neighbors
E Edge (or link) attributes and directions
e.g., edge identity, edge weight

Figure 2.4: Attributes of a graph [4]

We can additionally specialize graphs by associating directionality to edges (directed, undirected), see Figure 2.5. The edges can be directed, where an edge e has a source node, v_{src}, and a destination node v_{dst}. In this case, information flows from v_{src} to v_{dst}. They can also be undirected, where there is no notion of source or destination vertex, and information flows in both directions. This will be a key-concept for Chapter 4.

Graphs are very flexible data structures that allow multiple types of information to be represented. For example, Figure 2.6 shows how we can model text to be represented as a graph.

Undirected edge
Directed edge

Figure 2.5: Undirected vs Directed Edges [4]

In particular, we can digitize text by associating indices to each character, word, or token, and representing text as a sequence of these indices. This creates a simple directed graph, where each character or index is a node and is connected via an edge to the node that follows it.

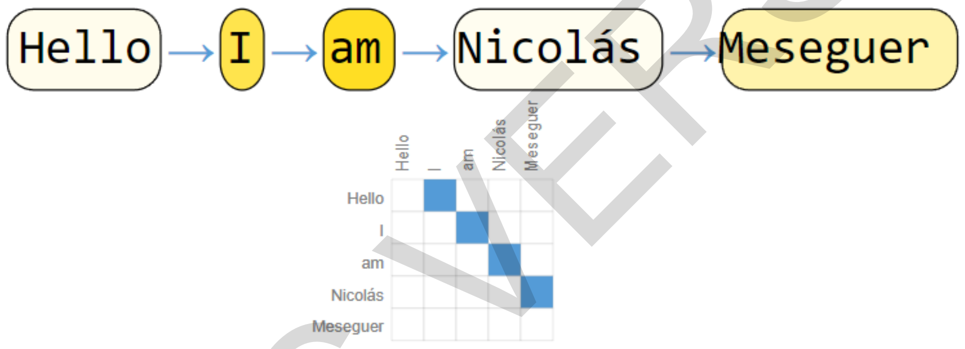

Figure 2.6: Adjacency matrix of a text graph [4]

As another example, scientists routinely cite other scientists' work when publishing papers. We can visualize these networks of citations as a graph, where each paper is a node, and each directed edge is a citation between one paper and another. This kind of graphs will be used in Chapter 5 for evaluating CPU-PCGCN.

Let's step forward and take a look at the pubmed dataset [26]. It consists of 19,700 biomedical publications (vertices) classified into one of three classes (labels). The citation network consists of 108,300 links (edges). Each publication in the dataset is described by a 0/1-value word vector indicating the absence/presence of the corresponding word from the dictionary (features). The dictionary consists of 500 unique words (max. features).

Graphs have a series of characteristics attached. Considering pubmed dataset, one of them is the label, which classifies the paper into a class, i.e. a paper is related to one topic or another. Another one is the number of words present in the paper out of the 500, this is called the features vector, and each node/vertex has it's own features vector. This underpins the operation of a GNN.

For the moment, let's stick with *vertices V* and *edges E*. With these we will be able to build a data structure that stores the graph, the adjacency matrix (using a 0/1-value integer to represent the connections between vertices). This data structure turns out to be (in many cases) very sparse (greater density of zeros). To avoid storing or handling this large amount of zeros, it is typically accepted a representation of the matrix using different compressed data structures (Figure 2.7).

This compressed structures provide a more efficient access to data and facilitates matrix operations. In this category, the most popular (and the ones that we will be using) are: Coordinate list (COO) and Compressed Sparse Row (CSR).

The COO format (Figure 2.7 bottom left) stores the matrix in the form of a list of tuples containing row, column and data. To enhance the time of the random access, the tuples in the list are sorted first by row and second by column. The memory consumption savings derived from the usage of this format start to be noticeable as the size of the matrix increases.

The CSR format (Figure 2.7 bottom right) is similar to the COO format, but it compresses the columns. It consists of a set of three arrays, containing the data, the extent of rows (indices) and the indices of columns (index pointers).

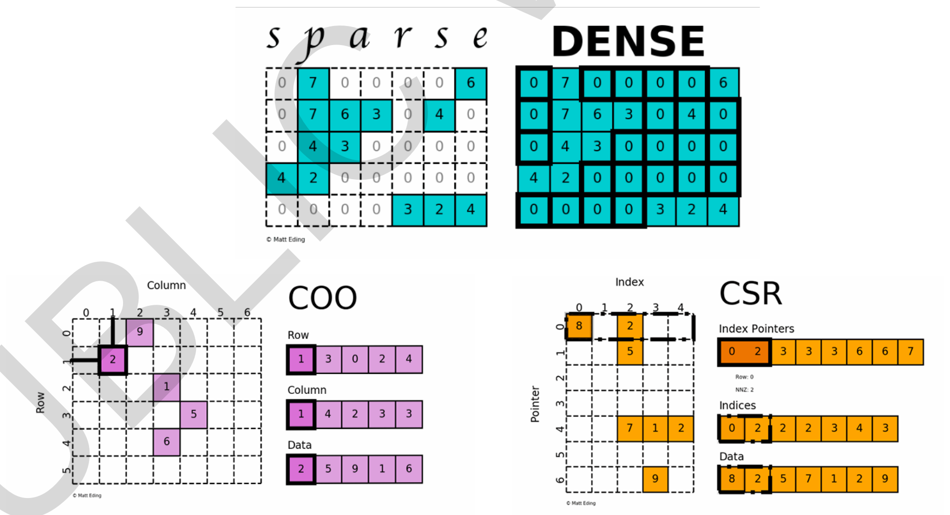

Figure 2.7: Sparse Matrices [5]

Let's conclude this section explaining relevant concepts and metrics that will be significant for the rest of the work. When referring to graphs, a graph G is a set of points V, called vertices, which are connected by a set of links E, called edges. Thus, $G = (V, E)$.

When discussing the symmetry and reciprocity of the set of edges, we distinguished between two types, *directed* and *undirected* graphs. While directed graphs are effective for modeling certain real-world structures and appear to preserve the hierarchical structure, undirected graphs are quite prevalent in practice and are better modeled for many real-world interactions [27].

In graph theory, the concept of cluster (or community) refers to a group of vertices that are more connected between them than to the other vertices outside the cluster ('small-world' property), and it is a relevant concept that tends to appear in most real-world graphs. One metric that defines this tendency inside a graph is called *cluster coefficient*, and it measures the degree to which vertices in a graph tend to cluster together (measures the locality). Cluster coefficient is expressed as a real number between zero (no clustering), and one (maximal clustering) [28] [29].

2.5 Graph Neural Networks

GNNs were initially proposed in 2009 [30] as an extension of neural networks that can process data in graph format, providing a convenient way for node level, edge level, or graph level prediction task. The intuition of GNN is that vertices are naturally defined by their neighbors and connections, which they are; in essence, the objective (see Figure 2.8) is to learn a state embedding that, for each vertex, encapsulates both the vertex' initial feature vector (attributes/features) and the information of the neighborhood (which represents the local graph structure that surrounds them). This embedding is used to produce the output [31]. Through an iterative process of information message-passing across vertices, these algorithms capture the complex dependencies of the network.

The first step of a GNN computation is, for each node, to collect the features from the edges, neighbourhood vertices and graph, and **aggregate** them into a single set. Note that, this operation is computationally very expensive due to the high degree of sparsity of the graph's adjacency matrix. Afterwards, there is a process for **combining** all the attributes, which can be done through different strategies, depending on the GNN. This combination results in a new embedding feature vector that is used to update the information of the vertices or edges, determined by the task prediction of the algorithm. In practice, these two steps may not necessarily have to be performed in the same order; occasionally the combination is performed first, followed by the aggregation. In Section 2.6, we will introduce GCNs with a basic example.

Finally, if the work is done at the graph level, the final output might be a node or edge information vector (label prediction) or a graph embedding that sums all the

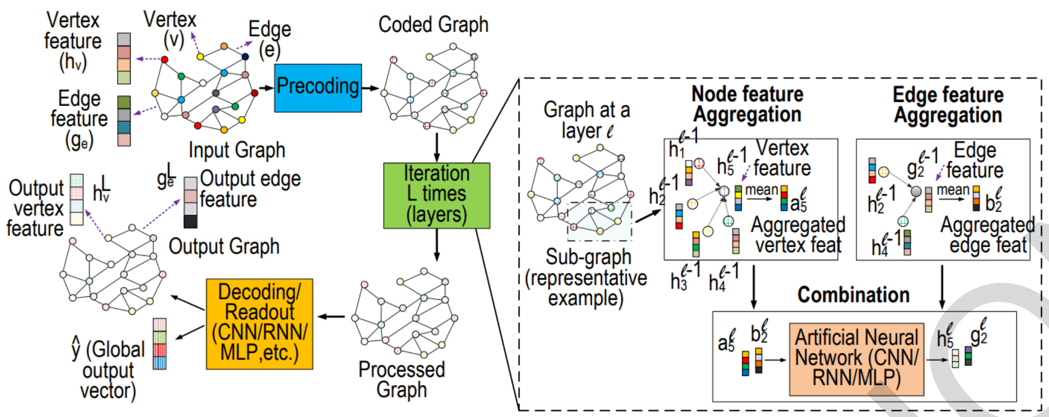

Figure 2.8: Computation process of a GNN [6]

information about the whole output graph. A GNN algorithm can be customized in many different ways, from the aggregation and combination strategies (swapping the order), to the number of layers to apply (like 2 or 4 GNN layers).

2.6 Graph Convolutional Networks

Generalizing well-established neural models like RNNs or CNNs to work on arbitrarily structured graphs is a challenging problem. Some recent papers introduce problem-specific specialized architectures [32], others make use of graph convolutions known from spectral graph[1] theory [33]. In [34], the authors take a somewhat similar approach and start from the framework of spectral graph convolutions, yet introduce simplifications that in many cases allow both for significantly faster training times and higher predictive accuracy, reaching state-of-the-art classification results on a number of benchmark graph datasets.

GCNs receive the name based on the filter parameters that are typically shared over all locations in the graph (we will talk about this in a moment). For these models, the goal is then to learn a function of signals/features on a graph $\boldsymbol{G = (V,E)}$ which takes as input:

- A feature description $\boldsymbol{x_i}$ for every vertex $\boldsymbol{i}$; summarized in a $\boldsymbol{N} \times \boldsymbol{D}$ feature matrix $\boldsymbol{X}$ ($\boldsymbol{N}$: number of vertices, $\boldsymbol{D}$: number of features).
- A representative description of the graph structure in matrix form; typically in the form of a compressed adjacency matrix $\boldsymbol{A}$.

[1] A spectral graph convolution is defined as the multiplication of a signal with a filter in the Fourier space of a graph.

And produces a node-level output $\boldsymbol{Z}$ (an $\boldsymbol{N} \times \boldsymbol{F}$ feature matrix, where $\boldsymbol{F}$ is the number of output features per node). Then, every neural network layer can then be written as a non-linear function:

$$H^{(l+1)} = f(H^l, A)$$

With $H^0 = X$ and $H^L = Z$ (or z for graph-level outputs), being $\boldsymbol{L}$ the number of layers. The specific models then differ only in how $f(\cdot, \cdot)$ is chosen and parameterized. Let's consider the following very simple form of a layer-wise propagation rule:

$$f(H^l, A) = \sigma(AH^lW^l)$$

Where W^l is a weight matrix for the ***l*-th** neural network layer and $\sigma(\cdot)$ is a non-linear activation function like the ***ReLu***.

But first, let us address two limitations of this simple model: multiplication with $\boldsymbol{A}$ means that, for every node, we sum up all the feature vectors of all neighboring vertices **but not the vertex itself**. This is easily solved by **adding the identity matrix to $\boldsymbol{A}$**.

The second major limitation is that $\boldsymbol{A}$ is typically not normalized and therefore the multiplication with $\boldsymbol{A}$ will completely change the scale of the feature vectors. **Normalizing $\boldsymbol{A}$** such that **all rows sum to one** (i.e. $D^{-1}A$, where D is the diagonal node degree matrix), gets rid of this problem [7]. $D^{-1}A$ now corresponds to taking the average of neighboring node features. In practice, it gets more interesting when using a **symmetric normalization**, like $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$.

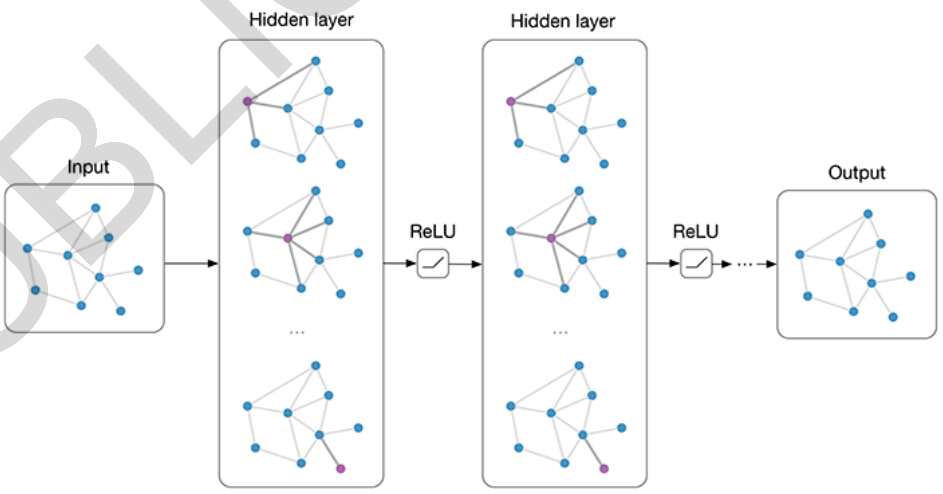

Figure 2.9: Multi-layer Graph Convolutional Network (GCN) [7]

Combining these two tricks, we essentially arrive at the propagation rule that we will be using later on in our proposed model (Chapter 4):

$$f(H^l, A) = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^lW^l)$$

Being $\hat{A} = A + I$, where I is the identity matrix and $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$.

With this, we have the definition of GCNs and how they work with a simple example. Finally, in Figure 2.9 we can see an actual model with two hidden layers (GCN layers) and activations (propagation rules) in-between (ReLU). This model is comparable to the one that will be used for CPU-PCGCN.

Let's conclude reminding that, what we do in each GCN layer are the two steps mentioned in Section 2.5. First the aggregation $Agg = H^l \times W^l$, and then, the combination $C = A \times Agg$, this output is then used for the ReLu propagation rule $\sigma(C)$.

3 Related Work

Several acceleration approaches and frameworks have been proposed for the costly Graph and GCN computation in recent years. These algorithms may be classified into many categories based on whether they are software-based (pre-processing techniques and software acceleration) or hardware-based (hardware acceleration). In the next sections, some state of the art examples of each category will be described, with a focus on software-based solutions, which are the main focus of this work.

3.1 Pre-processing acceleration techniques

These accelerators speedup GNN models without altering them. Instead, they design additional pre-processing strategies to boost training time by creating batches in a certain way that takes advantage of specific aspects of the input graph.

3.1.1 Cluster-GCN

Cluster-GCN [35] seeks to address the two primary problems of the GCNs training phase. The increased computation cost, which grows in an exponential ratio with the number of layers in the model, and the requirement for a large amount of memory in order to keep all of the information about the graph and node embeddings required for the calculation.

This technique aims to tackle the aforementioned concerns by making use of the graph's existing clustering structures (clustering coefficient). Despite its simplicity, this strategy is successful, delivering appropriate test accuracy while reducing memory utilization and boosting overall computing efficiency.

To reach this, the program uses a subgraph created by a graph clustering technique (partitioning algorithm) and obtains the block of vertices contained in the subgraph for each step. Therefore, given a graph, it is partitioned in K subgraphs. Each partition contains a block on vertices (which is unique for each partition) and a set of edges (only the ones which create the connections between the vertices of the cluster). As a consequence, the adjacency matrix, as well as the feature matrix and the training labels are partitioned accordingly.

3.1.2 GraphSAINT

To scale GCNs to large graphs, state-of-the-art methods use various layer sampling techniques to alleviate the "neighbor explosion" problem during training. Graph sampling based inductive learning method (GraphSAINT) [36] is a graph sampling-based technique that improves training efficiency and accuracy in a fundamentally different way.

GraphSAINT creates minibatches[1] by sampling the training graph rather than the vertices or edges across GCN layers. As a result, a fixed number of well-connected vertices in all layers in ensured.

3.1.3 METIS

Although this is not a technique directly applicable to GNNs or GCNs, it complements the computation process and is one of the key concepts applied in this work. Serial Graph Partitioning and Fill-reducing Matrix Ordering (METIS) [24] is a set of serial programs for partitioning graphs, partitioning finite element meshes, and so on.

Recently, a number of researchers have investigated a class of graph partitioning algorithms that reduce the size of the graph by collapsing vertices and edges. From the early work it was clear that this techniques held great promise; however, it was not known if they can be made to consistently produce high quality partitions.

METIS presents various novel heuristics to consistently perform better and in substantially smaller time than other partitioning algorithms, such as random cutting or min-edge cutting.

Algorithms that find a good partitioning of highly unstructured graphs are critical for developing efficient solutions. For example, large-scale numerical simulations on parallel computers, this partitioning must be done so that the number of elements assigned to each processor is the same (balance the computations among the processors), and the number of adjacent elements assigned to different processors is minimized (minimize the communication). This logic may be implemented to a single CPU, that processes partitions sequentially while taking advantage of their spatial locality (minimum communication between subgraphs).

As already mentioned, partitioning a graph offers many advantages to the computation stage as we will see briefly in Section 3.2.2. Thus, it is important to apply a locality-aware graph partitioning algorithm that takes into account graph properties and produces high-quality partitioning.

[1]**Minibatches** - refers to equally sized subsets of the dataset over which the weights are updated.

METIS can partition an undirected graph into a user-specified number $\boldsymbol{k}$ of partitions using either the multilevel recursive bisection or the multilevel k-way partitioning paradigms. Since multilevel k-way partitioning algorithm provides additional capabilities (e.g., minimize the resulting subdomain connectivity graph, enforce contiguous partitions, minimize alternative objectives, etc.) is the paradigm selected for partitioning datasets in the rest of this work.

3.2 Software acceleration

This sort of technique achieves acceleration by enhancing the efficiency of the model's computing process via software, allowing it to operate faster on ordinary CPUs/GPUs. Techniques such as node reordering, faster scheduling of matrix multiplications, and graph partitioning are a few examples.

3.2.1 HAG

Hierarchically Aggregated computation Graphs (or HAGs) [19] is a novel way of representing graph neural networks. The development of such technique arises out of the idea of reducing the highly common repeated computations during the aggregation phase of GNNs. Thanks to this representation, the efficiency and scalability of graph neural networks is enhanced.

The computation of a GNN consists of a set of trees (one for each node), indicating for a given node V which neighbor's previous-layer activations are going to be aggregated. However, those computation graphs do not take in consideration the overlapping computations, and hence all this redundancy generates inefficiency.

HAGs get rid of the overhead of the repeated computations by using intermediate results of the aggregations and creating a hierarchy scheme to manage them. It is important to note that in order to a HAG representation to be valid, it must be equivalent. Therefore, HAG retains the same performance in accuracy but speeds up the training and inference of the GNN.

3.2.2 PCGCN

Partition Centric Graph Convolutional Network (PCGCN) [8] is a software-based acceleration method which aims to improve the efficiency of the computation of Graph Convolutional Networks (GCN), both in memory and time performance. The critical aspect on which PCGCN focuses in order to achieve such enhancement in comparison to the available open source versions of GCN, is the fact that said implementations fail when it comes to taking into account graph properties that may be critical for neural network computation, because such versions are solely based on common sparse matrix calculations.

PCGCN is a partition-centric method that takes advantage of the locality characteristic of graphs. Simultaneously, the algorithm will adjust to the sparsity of each resultant partition, accelerating the propagation stage even more.

The GCN propagation technique in PCGCN is modified in such a manner that, rather than considering only the graph, the operations to be performed are dependent on each subgraph. The overall method for each layer of the neural network consists of collecting the states from the vertices that belong to the subgraph (aggregation phase) and for each partition, execute the partition-centric propagation. Once this is completed for all subgraphs, the results of each are merged to generate the layer's output (combination phase). This is done for each layer of the model.

The advantage of PCGCN is that, due to its subgraph-centric approach, the range of probable vertices and edges required for calculations is decreased, allowing for quicker accesses by loading this information from the cache. This is mainly how PCGCN exploits graph locality.

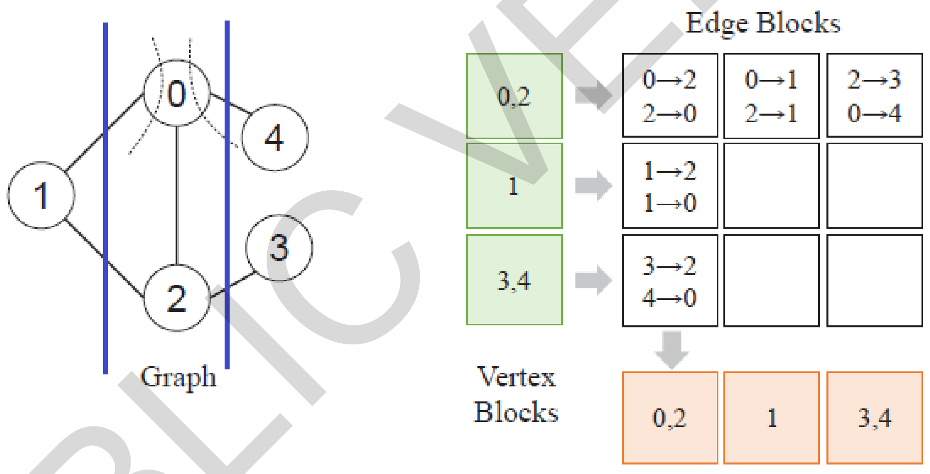

Figure 3.1: Graph partition and partion-centric processing for an undirected graph [8]

The first thing PCGCN does is the graph partitioning, it divides the complete graph into K subgraphs where every node is included only in one of the partitions, along with $\boldsymbol{K}\times\boldsymbol{K}$ blocks of edges (see Figure 3.1). In order to partition the graph in a way in which the locality is taken in consideration, the best option is to apply a locality-aware graph partitioning algorithm, METIS.

Once the grah partitioning is done, PCGCN moves to the partition-centric processing of each subgraph. This stage is repeated in every layer. First of all, the neural network transformation is calculated and divided according to the subgraphs. Hereafter, every partition is processed, taking in consideration not only the partition itself but also the ones which have at least an edge connected to the subgraph being processed. The results of the calculations are gathered in the hidden state h_k^l, where $\boldsymbol{l}$ is the layer and $\boldsymbol{k}$ the current subgraph. When the entirety of the partitions has been treated, all the hidden states are concatenated to set up the hidden state of the layer h^l (see Algorithm 1).

Algorithm 1 Forward computation of Partition-Centric Graph Convolutional Network [8]

```
Symbols: input graph: $G = (V, E)$, layers: $l = \{1, \cdot\cdot\cdot, L\}$, subgraphs: $\{S_k = (V_k, E_k)|k = 1, \cdot\cdot\cdot, K\}$, vertices in subgraph $k$: $V_k$, edges in subgraph $k$: $E_k$, edges between subgraph $i$ and $j$: $E_{i,j}$, features of layer $l$: $h^l$ ($h^0$ indicates the input features), weight of layer $l$: $w^l$
Ensure: Partition $G \rightarrow \{S_k|k = 1, \cdot\cdot\cdot, K\}$
  // calculate a L layer GCN model
  for $l = 1, \cdot\cdot\cdot, L$ do
      $a^l = h^{l-1} \times w^l$                                    ▷ Combination
      Split $a^l \rightarrow \{a_k^l|k = 1, \cdot\cdot\cdot, K\}$ according to subgraphs;
      // execute graph propagation for each subgraph
      for $k = 1, \cdot\cdot\cdot, K$ do
          // gather and accumulate states from neighbor subgraphs
          $h_k^l = \sum_{i=1}^{K} f(E_{k,i}, a_i^l)$                  ▷ Aggregation
      end for
      // combine hidden states of the subgraph k
      $h^l = concat(h_k^l, \forall k \in \{1, \cdot\cdot\cdot, K\})$;
  end for
  return $h^L$;
```

In this way, PCGCN is capable of taking advantage of the locality properties of the graph. However, the processing complexity of individual partitions may vary depending on their sparsity, which might be inefficient. PCGCN supports two graph propagation modes to counteract the usual uneven distribution of real-world graphs: *Selective Mode* and *Full Mode*.

PCGCN makes use of the *Selective Mode* when the number of edges between the subgraph being processed and one of its neighbor partitions is scarce, so, the compressed sparse row (CSR) is used to store the subgraph.

For dense subgraphs, *Full Mode* propagation is the preferred compute mode. When using this approach, PCGCN does not decode the vertices' indices but instead processes the edges sequentially. Full Mode assumes that the subgraph and the neighbor partition are both completely connected (every node in one is connected to every node in the other). Because it is an assumption, if a node $\boldsymbol{x}$ in one partition has no relationship

(edge) with a node ***y*** in the other, a new one with value 0 is established between them.
When working with the same graph, both modes can be used simultaneously. So, given a subgraph's edge block, PCGCN will pick which mode to use based on the sparsity. It should be noted that because PCGCN makes use of the graph's locality attribute, it is possible for this approach to perform better or worse depending on the input graph (and its properties).

PCGCN takes advantage of the Tensor Core acceleration offered by the latest versions of NVIDIA GPUs. To this end, they design various GPUs-specific optimizations for the partition-centric processing framework and the dual-mode subgraph computing strategy. PCGCN is a tool designed, evaluated and proposed to run on the GPU and efficiently execute GCNs. However, the work we present here is CPU-PCGCN a CPU alternative to PCGCN to efficiently execute GCN.

3.3 Hardware acceleration

By creating specialized hardware, hardware-based accelerators attempt to manage the computational needs of GCNs and increase the efficiency of their calculations. There are many hardware acceleration models, being the ones that have achieved the largest performance improvements AWB-GCN [37], EnGN [38] and HyGCN[39]; each implementing a specific dataflow which is heavily co-designed with the microarchitecture.

3.3.1 AWB-GCN

The Autotuning-Workload-Balancing GCN accelerator mainly advocates for an aggressive adaptation to the structural sparsity of the GNN. The authors motivate their design by analyzing the power-law distribution of most graphs, arguing that some parts of the computation will be dense and others extraordinarily sparse, creating unbalances.

3.3.2 EnGN

Among the first accelerators to appear, EnGN presents a unified architecture heavily inspired by CNN accelerators. The GNN is fundamentally treated as concatenated matrix multiplication of feature vectors, adjacency matrices, and weights, all scheduled in a single dataflow.

3.3.3 HyGCN

Hybrid architecture GCN Accelerator (HyGCN) arises from the observation that GNNs have two major alternating phases with opposing computing demands (memory and computation). Then, for the GCN model, HyGCN proposes a hybrid design with two independent engines, one for aggregation and the other one for combination.

4 CPU-PCGCN

In this work, we present CPU-PCGCN [40], a PCGCN implementation on top of PyGCN [23], solely based on CPU that enhances cache-efficiency and memory performance to efficiently execute GCNs, and implemented on top of the popular deep learning framework PyTorch [41]. This is particularly important for users to leverage GCNs in more data analysis applications.

As we demonstrate in Chapter 5, where we evaluate the performance of CPU-PCGCN by executing a typical GCN algorithm on a variety of real-world and synthetic datasets, our implementation of PCGCN outperforms the baseline implementation (PyGCN), by a factor of up to 3.94×.

Despite the fact that PCGCN discusses the usage of Tensors in modern GPUs architectures to further improve graph propagation in some stages, we have chosen a different CPU-running paradigm. This opens up the application of CPU-PCGCN to other fields, such as edge computing or IoT, where the devices used for processing have computational power limits and non-GPU architectures.

Among its main features, CPU-PCGCN is capable of running both models, the based PyGCN (GCN) implementation and the PCGCN (CPU-PCGCN) model.

Let's start with a high-level explanation of how our implementation works. Figure 4.1 depicts the model's flowchart, which clearly shows the many phases it goes through.

First and foremost, if the user has supplied any parameters, read them and load them, this is the *Read Parameters* stage. Examples of parameters can be the unique seed for the unpredictability of the libraries, the GCN specific parameters, like the *weight_decay* or *dropout ratio*, etc. Then, it follows the process of building a synthetic dataset if the user has requested so, or pre-processing an existing dataset (either real or synthetic); we will briefly explain both phases in Section 4.1. Although we will discuss synthetic dataset generators in Section 5.1 along with their properties.

After the dataset has been loaded and pre-processed, which entails having the adjacency matrix, features matrix, and labels in the right format, the model can be computed.

As previously mentioned, the model can be run either using CPU-PCGCN or the model based implementation, PyGCN; let's suppose the user decides to go for CPU-

PCGCN, then we partition the input graph into N subgraphs using the locality-aware partition algorithm, METIS (see Section 4.2), and then calculate the edge blocks (*vertices* and *edges* of each partition) –details in Section 4.3. Only after we can call the *training stage* of the model.

In fact, the training stage is divided into four distinct phases: *forwarding* (the only phase where, depending on the implementation, GCN computation or CPU-PCGCN is used), *error calculation* (examines the difference in error induced by the model), *backpropagation* using the gradient (i.e. linear regression), and *model weight update* (weights are updated so the next time the output matches).

Reached this point, let's step back and take a look at Figure 4.2 which also provides a high-level visualization of the model we propose. Let's use this figure to further explain what comes next, the forwarding of the model.

As it can be seen, the model contains two different hidden layers, named GCN or CPU-PCGCN. Indeed both layers are identical (architecturally speaking) and only differ in the way of computing the input parameters (the former being the complete graph in a compressed format, and the latter being the subsequent partitions of the graph, also in a compressed format). We will enter in detail and go through the algorithms of each of these layers in Section 4.4. By now, in *GCN vs CPU-PGCN* we will highlight the most important features.

In-between we have several different activation functions, like *ReLU*, an applied *Dropout* [42] to the previous output (during training, uses samples from a Bernoulli distribution to randomly zero some of the components of the input tensor with probability p), and finally a *Softmax* [43] activation.

After the model has been trained, we execute the inference step, which involves re-running only the forwarding of the model with a new set of data.

All of this work is available on a public git repository[1], where the interested reader can find instructions on library requirements, how to install or run the model etc., along with a detailed description of all the parameters accepted and clear examples.

4.1 Datasets

CPU-PCGCN admits two kinds of graph datasets, real-world ones, like Cora[2] or PubMed[3] [26] and synthetic ones. In the case of synthetic ones, CPU-PCGCN is integrated along with two different tools that are able to generate this kind of graphs,

[1]`https://github.com/NicolasMeseguer/pcgcn`
[2]`https://relational.fit.cvut.cz/dataset/CORA`
[3]`https://pubmed.ncbi.nlm.nih.gov/download/`

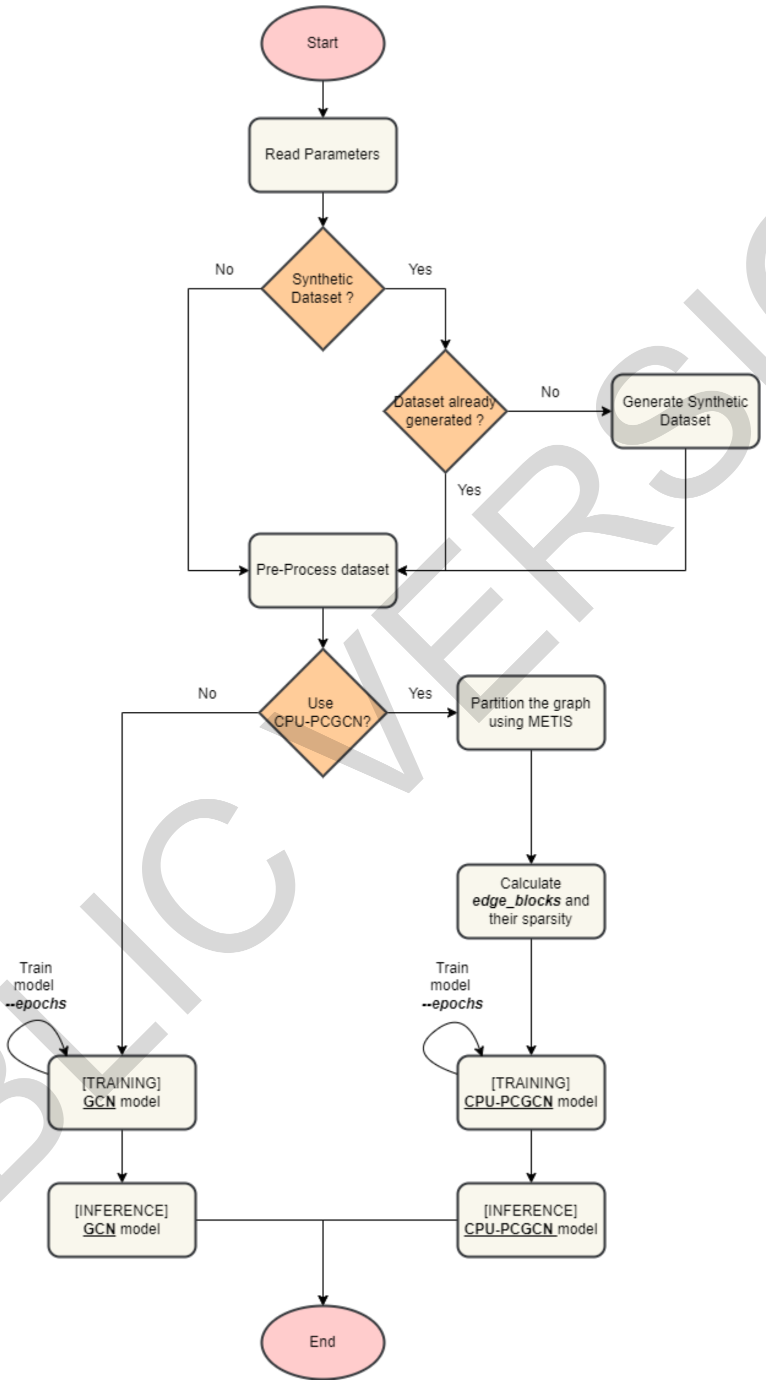
Start
Read Parameters
Synthetic Dataset ?
No
Yes
Dataset already generated ?
No
Generate Synthetic Dataset
Yes
Pre-Process dataset
Use CPU-PCGCN?
No
Yes
Partition the graph using METIS
Calculate *edge_blocks* and their sparsity
Train model --*epochs*
[TRAINING] GCN model
Train model --*epochs*
[TRAINING] CPU-PCGCN model
[INFERENCE] GCN model
[INFERENCE] CPU-PCGCN model
End

Figure 4.1: Flowchart of CPU-PCGCN

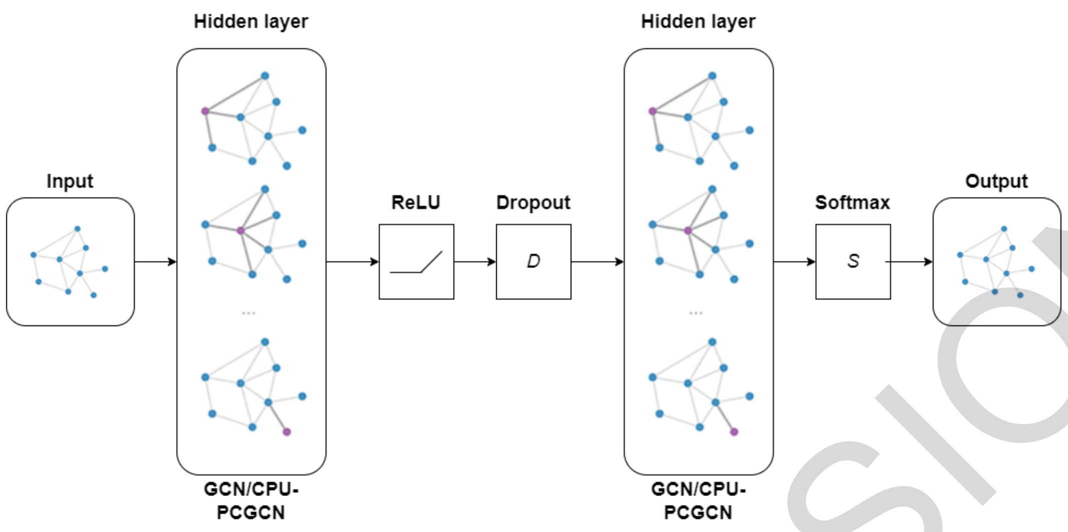

Figure 4.2: CPU-PCGCN Model

PaRMAT [25] and Graphlaxy[4], thes two tools will be later explained in Chapter 5.

In the case of real-world ones, we are using the datasets in a preprocess format from the planetoid repository[5], for storage convenience and with some preprocess already done, as the name suggest.

Regarding the pre-processing step of the datasets, we must highlight two major limitations we already addressed at Section 2.6; (1) given a node V, sum all feature vectors of neighbor vertices but itself and (2) not normalizing matrix A and thus, the multiplication will completely outscale the proportion of the feature vectors.

The first issue can be easily solved by adding the identity matrix I to the adjacency matrix A. The second issue, however, is a bit more tricky: we first make the adjacency matrix symmetric, $A = A + (A^T \times (A^T > A) - A \times (A^T > A))$, doing this we enforce the matrix to be symmetric. Further details about this specific technique can be found in: `https://github.com/yao8839836/text_gcn/issues/17`

With these two tricks already done, we resolve the issue already mentioned above. We can then normalize the adjacency matrix A, taking into account that the sum of the rows must be equal to 1, and the features matrix. The representations of both of these matrices are in a compressed format (see Figure 2.7), the former being in COO representation and latter in CSR. In the case of labels, they are stored in a one-hot encoding vector.

Once the datasets are ready to be used (loaded and processed), the user could opt to run the default GCN model, which would jump right into the training stage, or they

[4]`https://github.com/BNN-UPC/graphlaxy`
[5]`https://github.com/kimiyoung/planetoid`

could opt for running the CPU-PCGCN model.

4.2 Partitioning the graph

For a given graph, in order to process it in the CPU partition-centric computing scheme, the input graph should be firstly partitioned into a set of subgraphs. Therefore, CPU-PCGCN applies a 2D graph partitioning. As shown in Figure 3.1, it partitions the whole graph into K subgraphs and thus creates K disjoint vertex blocks and $K \times K$ edge blocks where $E_{i,j}$ represents edges between two vertex blocks V_i and V_j.

Recently, there have been proposed plenty of graph partitioning algorithms, e.g., random partitioning, min-cut partitioning, etc. We notice that a random partitioning will hurt the locality of graphs because it ignores the locality and randomly assigns vertices to partitions. Thus, we choose the locality-aware algorithm as the graph partition method to enhance the locality.

When utilizing CPU-PCGCN, the user must provide the number K of subgraphs into which the input graph should be partitioned using METIS. The conversion of the adjacency matrix to METIS input format is optimised to be performed with the sparse COO format so that the conversion time is relative to zero ($\simeq$ 0) regardless of the matrix size.

When processing a synthetic graph, various particular instances are considered, such as when the number of vertices is odd even though the graph is symmetrical; to fix this, we delete the initial edge of the graph, [0][0], and thus, we make it even so that METIS can partition it with no problems.

In terms of METIS, it ensures high-quality and even partitions (balances the number of vertices in all partitions), though because the graph is symmetric, it should be noted that in the majority of cases, the triangular subgraphs K may have the same amount of vertices and sparsity, which opens up a new window of speedup that we will discuss shortly.

4.3 Calculating *edge blocks* and their sparsity

Continuing what we have mentioned above, once we have obtained the partitioning of METIS, we can start to form the edge blocks. Thanks to the symmetry of the adjacency matrix we know that the upper triangular of the edge blocks equals to the transpose of the lower triangular (Figure 3.1), i.e. lets suppose we have a variable called *edge_blocks* which is a matrix of matrices, then we assume that: $edge\_block[0][1] = (edge\_block[1][0])^T$, using this approach we can easily compute only the identity and lower triangular, form the edge blocks, calculate its corresponding sparsity and then, transpose the results to obtain the upper triangular.

This approach has been demonstrated to achieve an up to 9× speedup compared to the previous implementation where all the edge blocks were computed sequentially not taking into account matrix properties, and normalize characteristics.

The sparsity of each of the subgraphs is represented by an integer between 0 and 100, meaning 100 is a fully sparse matrix (i.e. there is no edge within the edge block). We calculate the sparsity of a given subgraph K as:

$$100 - (\frac{E_{i,j}}{V_i \times V_j} \times 100) \tag{4.1}$$

Being $E_{i,j}$ the number of edges between subgraphs i and j, V_i vertices of subgraph i and V_j vertices of subgraph j.

4.4 GCN vs CPU-PCGCN

As mentioned, CPU-PCGCN is able to run both models, Algorithm 2 shows the feed-forward computation steps of a 2 layer GCN model. It is important to note that this layers can only perform the aggregation phase in one of two ways: either dense matrix multiplication or sparse matrix multiplication. It cannot do both, that's the special ingredient of PCGCN, which will discuss later on.

Algorithm 2 Forward computation of GCN with 2 layers

```
Symbols: input graph: $G = (V, E)$, layers: $l = \{1, \cdots, L\}$, features of layer $l$: $h^l$
  ($h^0$ indicates the input features), weight of layer $l$: $w^l$
  // calculate a 2 layer GCN model
  for $l = 1, \cdots, 2$ do
      $a^l = h^{l-1} \times w^l$                                ▷ Combination
      ----------------------------------------------------------------
      $h^l = spmm(adj_{matrix} \times a^l)$   ▷ Aggregation using Sparse Matrix-Multiplication
      OR
      $h^l = mm(adj_{matrix} \times a^l)$     ▷ Aggregation using Dense Matrix-Multiplication
  end for
  return $h^2$;
```

A question that is often asked is the equivalence of CPU-PCGCN and GCN models: CPU-PCGCN only changes the order of calculating neighbors of a given vertex compared to the original GCN. The operations in the graph propagation stage of GCN are element-wise multiplication and add, which are commutative and associative. Thus, the changed order does not affect the results because of the property of commutative and associative operations [44]. Therefore, the CPU-PCGCN is equal to the original GCN and can produce the same numerical results, hence the same per-epoch convergence.

Having understood the two phases of a GCN layer (aggregation and combination), let's explain how our CPU-PCGCN works.

In CPU-PCGCN, we propose to leverage the locality of real-world graphs to accelerate GCN computing by accelerating the graph propagation. Specifically, CPU-PCGCN introduces a partition-centric processing scheme (*a*) in the graph propagation stage, which makes PCGCN achieve locality-friendly in processing graphs. Moreover, a dual mode subgraph computing (*b*) method is introduced to further accelerate the graph propagation by design different computing mode according to the density (sparsity) of a subgraph.

Algorithm 3 Forward computation of CPU-PCGCN with 2 layers

```
Symbols: input graph: $G = (V, E)$, layers: $l = \{1, \cdot\cdot\cdot, L\}$, subgraphs: $\{S_k = (V_k, E_k, SS_k) | k = 1, \cdot\cdot\cdot, K\}$, vertices in subgraph $k$: $V_k$, edges in subgraph $k$: $E_k$, sparsity of the subgraph $k$: $SS_k$, edges between subgraph $i$ and $j$: $E_{i,j}$, features of layer $l$: $h^l$ ($h^0$ indicates the input features), weight of layer $l$: $w^l$
Ensure: Partition $G \rightarrow \{S_k | k = 1, \cdot\cdot\cdot, K\}$
  // calculate a 2 layer CPU-PCGCN model
  for $l = 1, ..., 2$ do
      $a^l = h^{l-1} \times w^l$                                  ▷ Combination
      Split $a^l \rightarrow \{a^l_k | k = 1, \cdot\cdot\cdot, K\}$ according to subgraphs;
      // execute graph propagation for each subgraph
      for $k = 1, \cdot\cdot\cdot, K$ do
          // dual-mode computing
          if $SS_k > args.threshold$ then
              $h^l_k = \sum_{i=1}^{K} f_{spmm}(E_{k,i}, a^l_i)$              ▷ Sparse Aggregation
          else
              $h^l_k = \sum_{i=1}^{K} f_{mm}(E_{k,i}, a^l_i)$                ▷ Dense Aggregation
          end if
          // combine hidden states of the subgraph k
          $h^l = concat(h^l_k)$
      end for
  end for
  return $h^2$;
```

a. *Partition-Centric Graph Processing:*

CPU-PCGCN modifies the graph propagation stage of GCN from a whole graph computing scheme to a subgraph-centric one. Algorithm 3 shows the feed-forward computation steps of a 2 layer model. For each GCN layer, the hidden state from the previous is transformed by a fully connected neural network, which is the same as the original GCN (combination). Then, for each subgraph, it gathers and accumulates states from itself and neighbor subgraphs to execute the partition-centric graph propagation (aggregation). The outputs of each subgraph are combined as the output of this layer.

After partitioning the graph into a set of subgraphs, CPU-PCGCN can apply the partition centric processing. For layer l of the model, CPU-PCGCN calculates the neural network transformation a^l and then partitions it to corresponding subgraphs. Then, CPU-PCGCN processes these subgraphs one by one. For subgraph S_k, CPU-PCGCN traverses all the subgraphs that have edges $E_{k,i}$ connected to S_k, and processes the edge block. For each edge in $E_{k,i}$, it calculates the multiplication of data in the source vertex and the edge, and accumulates it in h^l_k, this can be done in two different ways, thus, it is a dual-mode computing. After all of the neighbor subgraphs being processed, it will generate the hidden state h^l_k for vertices in S_k. Finally, the concatenated $h^l_k(\forall k \in \{1, \cdot\cdot\cdot, K\})$ is the output hidden state h^l of layer l.

Because the range of source and destination vertices is confined to the subgraph, the partition-centric processing technique is memory hierarchy friendly. As a result, processing edges with the same sources or destinations can load data from the cache, which is especially useful for graphs with a high degree of locality.

b. *Dual Mode Subgraph Computing Strategy*

To take advantage of graph locality, CPU-PCGCN processes the GCN from the subgraph perspective for each layer. Real-world graphs, according to Section 2.4, typically have irregular distributions. The irregularity of a graph frequently results in variable densities in subgraphs. We offer the dual-mode subgraph computing approach based on density to significantly speed subgraph calculation.

a) *Selective Mode*

When there are a few edges in the edge block $E_{k,i}$ of the subgraph S_k and S_i in the layer l of the model, CPU-PCGCN uses a selective mode to process this edge block. Then, according to GCN, the features of the source vertex will be multiplied with the scalar value on the edge on the fly, and the result will be added to the hidden state h^l_k using the function $f_{spmm}(E_{k,i}, a^l_i)$. This subgraph layout is in a compressed matrix format (COO).

b) *Full Mode*

When there are lots of edges in a subgraph, the decoding procedure in the selective mode may lead to a huge impact on the processing efficiency. We introduce the *full* mode, it assumes the subgraph S_k and S_i is fully connected, i.e., each vertex in S_k is connected to all the vertices in S_i. If there is no edge $e_{p\to q}$ between vertex p and q, it will insert this edge with value 0 on the edge. This mode uses the function $f_{mm}(E_{k,i}, a_i^l)$. Different with the selective mode, the *full* mode processes all the edges in the fully connected chunk sequentially instead of decoding the indexes of edges and fetching the corresponding vertices.

c) *Hybrid Mode*

The above *selective* mode is suitable for sparse edge blocks, while the *full* mode is better for dense edge blocks. Due to the irregularity, a real-world graph may contain both sparse and dense edge blocks. The computing complexity of both modes mostly relates to the sparsity of an edge block. So, we take the sparsity SS_k to select the processing mode for a given edge block K.

Specifically, we profile the runtime of full and selective modes for a block of sparsity p, according to a threshold set by the user, or by default (60%) if not specified. We choose *selective* mode if $SS_k > 60\%$, otherwise we select *full* mode for the processing of a given edge block.

Several methods for parallelizing part of the calculation at layer level have been proposed to further speedup the model on the CPU without achieving any improvement in time. In the next chapter, we will examine this proposals along with a profiling to further understand where the time is being consumed.

5 Evaluation

In this chapter, we demonstrate the efficiency of CPU-PCGCN by evaluating it on real-world and synthetic datasets. Section 5.1 describes the methodology of the experiments (environment setup, synthetic dataset generators, dataset properties, etc.). Section 5.2 shows the overall runtime comparisons of CPU-PCGCN and the baseline PyGCN.

5.1 Mehodology

a. *Experiment setup*

   We evaluate CPU-PCGCN on a single platform equipped with dual 2.4 GHz Intel Xeon E5-2640v4 processors (20 physical cores in total), 125 GB memory, and NVIDIA GeForce GTX 780 GPU. The installed operating system is Ubuntu 16.04.

   We compare CPU-PCGCN to the open-source GCN implementation [23] on PyTorch v1.2.0 [41] (PyGCN), SciPy [45] v1.5.4 and Python NumPy [46] v1.19.5. For fair comparison, all of the baseline systems are the latest stable versions (compared with the versions used in PyGCN), and using the same Python version, 3.6.15.

   We focus on metrics for system performance, for example, time to train one epoch of data over 5 different executions to discard variability. We have proved the equivalence of GCN and CPU-PCGCN in Section 4.4. Thus, CPU-PCGCN produces the same numerical results compared with GCN. We report the average results over 100 epochs if no specified.

b. *Synthetic Dataset Generators*

   As noted in Section 4.1, to evaluate the superiority of CPU-PCGCN in graphs of different sparsity, we use the PaRMAT and Graphlaxy. Also to demonstrate the hybrid computing mode when using graphs of irregular sparsity.

   a. *PaRMAT* [25], is a multi-threaded (parallel) RMAT [47] graph generator. A widely used graph generator, to generate synthetic graphs characterized by the skewed distribution and fractal community structure which are similar to real-world graphs.

   b. *Graphlaxy* is a tool developed by Barcelona Neural Networking Center (BNN) who has been a partner in this project. Graphlaxy is a tool used to create synthetic graph datasets with an even distribution over a set of metrics (or projection) using 'Nash Bargain Scheme' optimization. It is available at `https://github.com/BNN-UPC/graphlaxy`.

Both of these tools are explained, along with examples on the public repository that has been derived as part of this work [40].

With the implementation of these two dataset generating tools, the window is open for further dataset generators that exploit other types of graph properties and, as a result, determine if CPU-PCGCN may be exploited in the same way.

c. *Datasets*

Table 5.1 lists the real-world and synthetic datasets used for evaluation including, PubMed citation network (pubmed) [26], Cora scientific dataset (cora) [48] and citeseer citation network (citeseer) [49].

In PaRMAT, we set a variable number of vertices and generate different number of edges to achieve the graph density of $\{0.01\%, 0.06\%, 0.12\%, 1\%\}$ (marked as PaRMAT-density, e.g. PaRMAT-1 indicates a PaRMAT with 1% of density)[1]. As some datasets do not have features or labels that are required by the GCN, we use a random distribution to generate the vertex features and the vertex labels. The column feature in Table 5.1 represents the size of vertex feature vector, and the label column means the number of label classes, for the synthetic datasets the clustering coefficient has not been measured, NM[2].

Datasets	#vertex	#edge	#feature	#label	clustering coefficient
cora	2.7K	5.4K	1.4K	7	0.24
citeseer	3.3K	4.7K	3.7K	6	0.14
pubmed	19.7K	108.3K	500	3	0.06
PaRMAT-0.01	996	8K	259	30	NM
PaRMAT-0.06	1K	30K	731	61	NM
PaRMAT-0.12	100	600	48	6	NM
PaRMAT-1	996	800K	6.8K	405	NM

Table 5.1: Datasets used in evaluation. (K: Thousands)

[1] These datasets are available in the git repository, with the following name (in order with the density): *Magenta_Spoonbill*, *Red_Magpie*, *Lime_Hawk* and *Silver_Parrot*

[2] **N**ot **M**easured: since these are synthetic datasets, we just consider their density, not the clustering coefficient.

Due to some server limitations we have not been able to generate synthetic datasets with a higher density of edges.

5.2 Experimental Results

a. *Layer-level-parallelism*

At first the computation time of a single iteration in the CPU-PCGCN technique was severely superior to that of its GCN counterpart. This encouraged the study of the different operations done at the layer level in order to profile the CPU-PCGCN forward function for a single iteration.

a) *V1, Torch implementation*, all the computation that is done at the layer level is done with the Torch library.

b) *V2, Concat included*, the concatenation phase of the layer h^l_k is performed immediately after its computation.

c) *V3, Torch parallel*, same as V1 except that the computation of a^la is parallelised with task-level parallelism using K threads.

d) *V4, NumPy implementation*, the computation of the layer has been done with NumPy, including the necessary conversions to PyTorch.

e) *V5, NumPy parallel*, like V4, an unsuccessful attempt has been made to parallelise the computation of a^l using K threads.

As a result, and already mentioned in Section 4.4, the computation of the layer for CPU-PCGCN has gone through several phases. The computation performed at the layer level is strongly linked to the libraries with which it has been developed, so data conversions are necessary. Table 5.2 shows the base implementation and the evolution it has gone through. These results have been obtained by partitioning the graph into 4 subgraphs and with the cora dataset. Figure 5.1 shows the time in the form of a graph.

Time (s)					
Phases	V1	V2	V3	V4	V5
Splitting a^l	0.098	0.0959	0.185	0.004	0.0065
Graph propagation	0.001	0.578	0.604	0.018	0.0196
Concat	0.642				
Torch conversion	NA	NA	NA	$\simeq 0$	$\simeq 0$
Total (s)	0.74	0.67	0.789	0.022	0.026

Table 5.2: Profiling (s) of CPU-PCGCN layer computation

As it can be seen, task-level parallelism for the acceleration of the layer computation seems to add more overhead than speedup (this is due to the cost of keeping threads in a queue + initialization). Take a look at the two blue cells from V3 and V5, comparing both of them to the sequential version (V2 and V4 respectively) the time consumed to create a single thread is up to 0.046 for V3 and 0.001 for V4, which is exactly $1/4th$ of the time of splitting a^l (remember we were using the same number of threads as partitions, 4). This shows the slightly poor performance the task-level parallelism offers, almost *twice*.

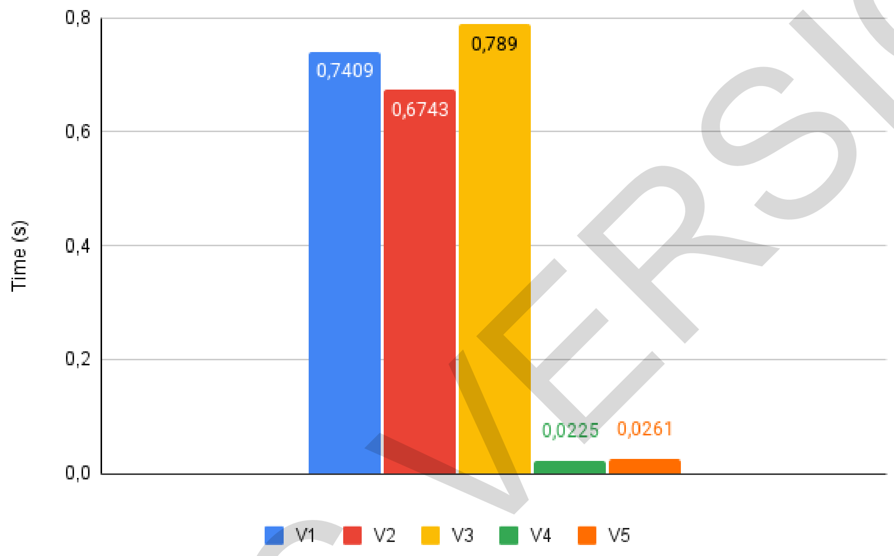

Figure 5.1: Profiling of the different phases of the computation at layer level

As a consequence of this profiling, it is clear how substantially the time has been reduced by modifying the implementation. NumPy employs low-level libraries like MKL to accelerate computing, allowing for quicker execution of operations such as matrix transposition or data accesses. This provides the possibility to further improvements with future versions of these libraries.

b. *Overall Performance*

We evaluate the overall model performance by comparing it with state-of-the-art base model implementation PyGCN on PyTorch.

Table 5.3 shows the end-to-end performance. Overall, CPU-PCGCN achieves an average 2.11× speedup (up to 3.94×) over the best one of baselines. We compare the base implementation PyGCN (GCN) using either sparse (S) or dense (D) computation vs CPU-PCGCN with an specific amount of partitions denoted as CPU-partitions. A more general view can be seen in Figure 5.2.

Time (s)							
Datasets	GCN (*D*)	GCN (*S*)	CPU-2	CPU-4	CPU-8	CPU-16	speedup
cora	88.74	81.53	46.74	48.58	56.96	72.53	**1.74x**
citeseer	82.61	69.42	46.11	49.58	56.90	74.97	**1.50x**
pubmed	1,259.93	795.184	248.56	303.22	347.44	446.47	**3.19x**
PaRMAT-0.01	30.70	28.86	19.78	21.71	25.36	33.12	**1.45x**
PaRMAT-0.06	43.01	37.81	21.67	23.70	26.31	34.06	**1.74x**
PaRMAT-0.12	3.76	2.41	1.9	2.04	2.63	4.54	**1.26x**
PaRMAT-1	1,171.2	1,050.3	266.48	229.98	262.54	387.72	**3.94x**

Table 5.3: The overall runtime (s) of GCN and CPU-PCGCN, denoted as CPU-*partitions* in powers of two

As it can be seen, in all cases CPU-PCGCN obtains an improvement in time. Moreover, in the case of higher density graphs (**only for PaRMAT-1**), it can be seen how the number of partitions (4) benefits the computation time. Finally, it should be noted that increasing the number of partitions to 16, even though the resulting subgraphs may be very dense, does not improve the time neither the cache locality. The computational cost of processing 16 partitions is higher than computing 4, for example see PaRMAT-1.

To evaluate the contribution of selective and full execution modes, we also run CPU-PCGCN modifying the sparsity threshold to force one of the dual-modes to be executed. Table 5.4 shows the results compared to the best time obtained previously. The table names are denoted as CPU-partitions-threshold.

Time (s)					
Datasets	*best-time*	CPU-4-20	CPU-4-40	CPU-16-20	CPU-16-40
cora	46.74	53.17	50.65	74.81	71.27
citeseer	46.11	48.84	48.88	74.18	74.47
pubmed	248.56	287.19	288.13	445.91	446.56
PaRMAT-0.01	19.78	21.47	21.27	32.57	32.78
PaRMAT-0.06	21.67	23.28	23.55	33.78	34.38
PaRMAT-0.12	1.9	2.04	2.06	4.72	4.77
PaRMAT-1	229.98	224.13	220.90	398.34	403.47

Table 5.4: The overall runtime (s) of varying sparsity, denoted as CPU-*partitions-threshold*

As it can be seen, in most cases, changing the sparsity required to run the operation in sparse mode tends to result in a worse computation time. However, if we look at PaRMAT-1 (the densest graph) we can see that a slight improvement in time is gained. From this we can project how increasing the density in the graphs (characteristics of real-world graphs) leads to progressively better computation time.

On the other hand, if we look at PaRMAT-0.01 (very dense graph) when lowering the required sparsity level we still get the same time, this is because the graph is already processed in a dense way automatically.

Finally, if we look at the PubMed dataset (which is very sparse) we can see that by increasing the number of partitions and reducing the required sparsity, the time increases; probably some partition is running in dense mode and therefore all the zero values are being penalised.

In the rest of the cases, it can be observed that the computation time varying the sparsity is very similar.

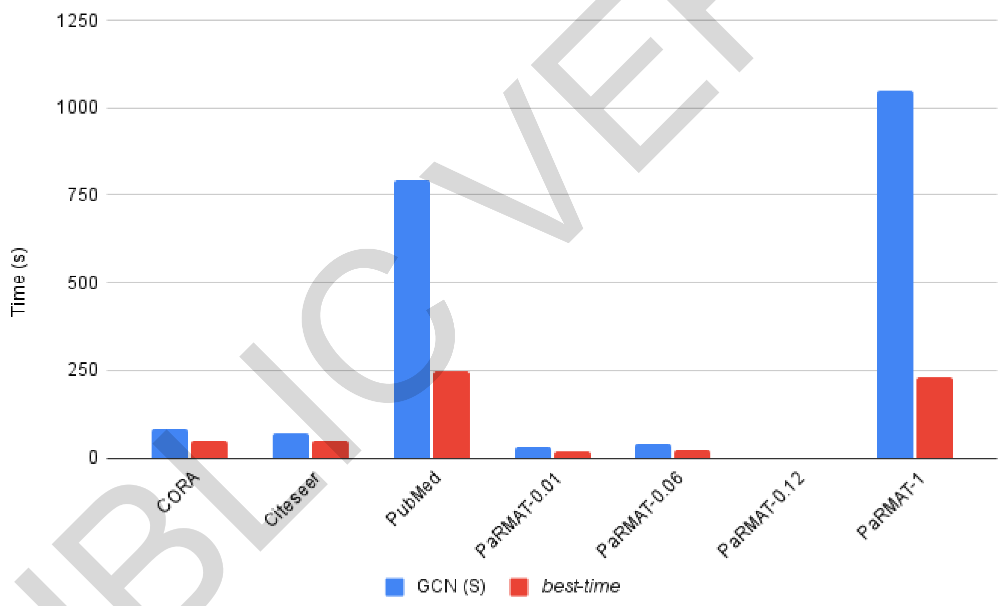

Figure 5.2: Runtime (s) of the different datasets

6 Conclusions

In this work, we present CPU-PCGCN, a CPU-only alternative to PCGCN for fast GCN computation. Unlike PCGCN, this new implementation opens up the application of PCGCN to other fields, such as edge computing or IoT, where the devices used for processing have computational power limits and non-GPU architectures. In short, CPU-PCGCN perceives a graph as a set of links between vertices and partitions instead of a sparse adjacent matrix, and depending on their sparsity, it selects a computation mode (*full* or *selective*). Despite using only the CPU, our model already offers severe advantages over native GCN processing.

We presented and developed system-level optimizations to achieve high performance by leveraging graph properties. PCGCN is a stepping stone in building systems for GNNs.

Two fundamental contributions have been made in this work:

1. *CPU-PCGCN*: a new implementation of PCGCN specifically tailored to efficient processing of both training and inference of GCN models in CPU-constrained computing platforms. CPU-PCGCN has been designed to facilitate integration with popular third-party tools such as METIS, aimed to explore different partitioning techniques, and PaRMAT and Graphlaxy for generation of synthetic graphs under different user-defined features.

2. *Detailed evaluation of CPU-PCGCN*: a detailed evaluation of CPU-PCGCN, including a thorough profiling showing the time spent in each phase and a complete evaluation using both real and synthetic graphs.

Having developed this work on top of high-performance libraries, it opens up the possibility of further research in the field of GCNs using the work proposed here.

Because the work we have conducted is constrained by a number of hours and a deadline, below are some potential avenues for future extensions of CPU-PCGCN. These are as follows:

a. *Accurate sparsity.* As of now, the sparsity threshold is predefined on a 60% value, which means that, if the sparsity of a given subgraph is bigger than 60%, it will be computed using the sparse matrix multiplication, otherwise, it will do with dense matrix multiplication. This threshold is defined based on a previous analysis of the datasets and a heuristic determination. As future work, we would like to study/find an accurate threshold that determines the computation mode.

b. *CPU+GPU acceleration.* CPU-PCGCN is based solely on CPU computation, whereas PCGCN is a GPU-standalone implementation. As contemporary computing platforms are heterogeneous, i.e., they integrate both CPU and GPU architectures, we will explore a new *CPU+GPU*-PCGCN version where both CPU and GPU collaborate for processing the GCN. More specifically, where either the CPU or the GPU is used to process a subgraph depending on the properties (subgraph density, size, clustering coefficient, etc.) that best suite for each kind of architecture –CPUs are typically more efficient to exploit irregular task-level parallelism (sparse computation), while GPUs excel at more regular data-level parallelism (dense computation).

c. *Comprehensive datasets.* Conduct a more thorough study of the datasets, varying only one of their properties at a time, like the sparsity, the density, increasing only the edges, etc. In addition, the hybrid mode could be varied so that the graph is computed only in dense or sparse, and then, using the hybrid mode. In this way we can do a study for each graph and see what time the hybrid mode gets, if it only uses one of the two modes (same time), or if it alternates between them for different subgraphs (less time).

d. *Optimizations to the CPU code.* Finally, we would like to employ TVM [50], an automated end-to-end optimizing compiler for DL. Using this tool, we could obtain considerably more optimized CPU code (including different types of parallelism, computational-layer acceleration, etc.). As an alternative, if we want to employ GPU Tensor core acceleration, we may use this tool to optimize the CUDA code in charge of doing the calculation.

Bibliography

[1] Zhenzhu Meng, Yating Hu, and Christophe Ancey. Using a data driven approach to predict waves generated by gravity driven mass flows. *Water*, 12, 02 2020. doi: 10.3390/w12020600.

[2] Hamza Jaffali. Illustration of an artificial neuron., 2022. URL https://www.researchgate.net/figure/Illustration-of-an-artificial-neuron_fig1_335442226.

[3] Iron Hack. What is machine learning?, 2022. URL https://www.ironhack.com/en/data-analytics/what-is-machine-learning.

[4] Benjamin Sanchez. A gentle introduction to graph neural networks, 2022. URL https://distill.pub/2021/gnn-intro/.

[5] Matt Eding. Sparse matrices, 2022. URL https://matteding.github.io/2019/04/25/sparse-matrices/.

[6] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *arXiv*, 2020. doi: 10.48550/ARXIV.2010.00130. URL https://arxiv.org/abs/2010.00130.

[7] Thomas Kipf. Graph convolutional networks, 2022. URL http://tkipf.github.io/graph-convolutional-networks/.

[8] Chao Tian, Lingxiao Ma, Zhi Yang, and Yafei Dai. Pcgcn: Partition-centric processing for accelerating graph convolutional network. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 936–945, 2020. doi: 10.1109/IPDPS47924.2020.00100.

[9] IBM. What is machine learning?, 2022. URL https://www.ibm.com/cloud/learn/machine-learning.

[10] Christopher Tomas. Common uses for cnns, 2022. URL https://towardsdatascience.com/an-introduction-to-convolutional-neural-networks-eb0b60b58fd7.

[11] Davis David. What is an rnn in dl?, 2022. URL https://hackernoon.com/what-is-an-rnn-recurrent-neural-network-in-deep-learning.

[12] Yongji Wu, Defu Lian, Yiheng Xu, Le Wu, and Enhong Chen. Graph convolutional networks with markov random field reasoning for social spammer detection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34:1054–1061, 04 2020. doi: 10.1609/aaai.v34i01.5455.

[13] Yingxue Zhang, Soumyasundar Pal, Mark Coates, and Deniz Üstebay. Bayesian graph convolutional neural networks for semi-supervised classification, 2018. URL `https://arxiv.org/abs/1811.11103`.

[14] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. Protein interface prediction using graph convolutional networks. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL `https://proceedings.neurips.cc/paper/2017/file/f507783927f2ec2737ba40afbd17efb5-Paper.pdf`.

[15] Jie Zhou, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018. URL `http://arxiv.org/abs/1812.08434`.

[16] Inneke Mayachita. Understanding graph convolutional networks, 2022. URL `https://towardsdatascience.com/understanding-graph-convolutional-networks-for-node-classification`.

[17] Yongqin Xian, Christoph H. Lampert, Bernt Schiele, and Zeynep Akata. Zero-shot learning – a comprehensive evaluation of the good, the bad and the ugly, 2017. URL `https://arxiv.org/abs/1707.00600`.

[18] Nicola De Cao and Thomas Kipf. Molgan: An implicit generative model for small molecular graphs, 2018. URL `https://arxiv.org/abs/1805.11973`.

[19] Zhihao Jia, Sina Lin, Rex Ying, Jiaxuan You, Jure Leskovec, and Alex Aiken. Redundancy-free computation graphs for graph neural networks, 2019. URL `https://arxiv.org/abs/1906.03707`.

[20] Dalong Zhang, Xin Huang, Ziqi Liu, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Zhiqiang Zhang, Lin Wang, Jun Zhou, Yang Shuang, and Yuan Qi. Agl: a scalable system for industrial-purpose graph machine learning, 2020. URL `https://arxiv.org/abs/2003.02454`.

[21] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 187–198, 2020. URL `https://proceedings.mlsys.org/paper/2020/file/fe9fc289c3ff0af142b6d3bead98a923-Paper.pdf`.

[22] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–14, 2018. doi: 10.1109/MICRO.2018.00010.

[23] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[24] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM JOURNAL ON SCIENTIFIC COMPUTING*, 20(1):359–392, 1998.

[25] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 39–50, 2015. doi: 10.1109/PACT.2015.15.

[26] Shobeir Fakhraei, James Foulds, Madhusudana Shashanka, and Lise Getoor. Collective spammer detection in evolving multi-relational social networks. In *PubMed Collective Detection*, 08 2015. doi: 10.1145/2783258.2788606.

[27] Bael Dung. What is the difference between a directed and an undirected graph, 2022. URL `https://www.baeldung.com/cs/graphs-directed-vs-undirected-graph`.

[28] Surin Der Dawra. Clustering coefficient in graph theory, 2022. URL `https://www.geeksforgeeks.org/clustering-coefficient-graph-theory/`.

[29] Jerome Junegis. Defining the clustering coefficient, 2022. URL `https://networkscience.wordpress.com/2013/09/08/defining-the-clustering-coefficient/`.

[30] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.

[31] Derrick Mwiti. The essential guide to gnn, 2022. URL `https://cnvrg.io/graph-neural-networks/`.

[32] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL `https://proceedings.neurips.cc/paper/2015/file/f9be311e65d81a9ad8150a60844bb94c-Paper.pdf`.

[33] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs, 2013. URL `https://arxiv.org/abs/1312.6203`.

[34] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2016. URL `https://arxiv.org/abs/1609.02907`.

[35] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-GCN. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining*. ACM, jul 2019. doi: 10.1145/3292500.3330925. URL `https://doi.org/10.1145%2F3292500.3330925`.

[36] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method, 2019. URL `https://arxiv.org/abs/1907.04931`.

[37] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin Herbordt. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing, 2019. URL `https://arxiv.org/abs/1908.10834`.

[38] Shengwen Liang, Ying Wang, Member, IEEE, Cheng Liu, Lei He, Huawei Li, Senior Member, IEEE, And, Xiaowei Li, Senior Member, and IEEE. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks, 2019. URL `https://arxiv.org/abs/1909.00155`.

[39] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Hygcn: A gcn accelerator with hybrid architecture, 2020. URL `https://arxiv.org/abs/2001.02514`.

[40] Nicolas Meseguer. Cpu-pcgcn, 2022. URL `https://github.com/NicolasMeseguer/pcgcn`.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL `https://arxiv.org/abs/1912.01703`.

[42] PyTorch. Dropout, 2022. URL `https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html`.

[43] PyTorch. Softmax function, 2022. URL `https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html`.

[44] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 17–30, USA, 2012. USENIX Association. ISBN 9781931971966.

[45] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, and C J Carey. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[46] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, and Matti Picus. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

[47] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SIAM Proceedings Series*, volume 6, 04 2004. doi: 10.1137/1.9781611972740.43.

[48] Arnaud Barragao. Cora dataset, 2022. URL https://relational.fit.cvut.cz/dataset/CORA.

[49] LINQS. Citeseer dataset, 2022. URL https://linqs.soe.ucsc.edu/data.

[50] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning, 2018. URL https://arxiv.org/abs/1802.04799.