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

Nicolás Meseguer-Iborra[1], Francisco Muñoz-Martínez[1], José L. Abellán[2], Manuel E. Acacio[1]

Resumen— **Debido al gran éxito de las Redes Neuronales Convolucionales (*Convolutional Neural Networks* o CNNs) en el ámbito del Aprendizaje Profundo (*Deep Learning* o DL), la operación convolucional se ha trasladado más allá del procesamiento de datos mapeados en el espacio Euclídeo (por ejemplo, las imágenes), a datos en el espacio no Euclídeo (por ejemplo, redes de citaciones estructuradas como grafos), dando lugar a las Redes Convolucionales de Grafos (*Graph Convolutional Networks* o GCNs). Dado el alto coste computacional de las GCNs, que involucran procesamiento de grafos en una fase denominada de agregación, y procesamiento de redes neuronales en una fase denominada combinación, implicando ambas fases multiplicaciones de grandes tensores (vectores y matrices), los procesos de entrenamiento de las GCNs se suelen llevar a cabo utilizando mayormente arquiteturas GPU en grandes centros de procesamiento de datos. Para acelerar aún más este procesamiento, PCGCN es una nueva técnica software implementada sobre arquitecturas GPUs que se basa en dividir el grafo de entrada en subgrafos y computarlos dependiendo de su nivel de dispersión (*sparsity* o presencia de operandos con valor cero) bien con un algoritmo de multiplicación de tensores *sparse* o tradicional (denso). Así, el procesamiento del grafo completo y por ende el procesamiento de GCN se hace mucho más rápido. Sin embargo, en entornos más cercanos al usuario y/o sensores, las plataformas de cómputo que ejecutan GCNs ya entrenadas o realizan entrenamiento *in-situ* de las mismas, ofrecen muchas menos prestaciones y pueden carecer de dispositivos GPU, imposibilitando la ejecución de PCGCN. Es por ello que en este trabajo presentamos CPU-PCGCN, una implementación de PCGCN para acelerar el cálculo de las GCNs aprovechando la CPU como unidad de cómputo. En comparación con la implementación base de GCN (PyGCN) usando conjuntos de datos reales y sintéticos, CPU-PCGCN consigue un aumento de velocidad de hasta 3.94 veces en el mejor de los casos.**

Palabras clave— **Redes Convolucionales de Grafos, PCGCN, Acelerador software para Deep Learning.**

I. Introducción

LAS Redes Neuronales Profundas o *Deep Neural Networks* (DNNs) han logrado muchos avances en muchas áreas, por ejemplo, el reconocimiento de imágenes o el procesamiento del lenguaje natural y, por tanto, muchas aplicaciones del mundo real se basan actualmente en ellas, como la conducción autónoma, motores de búsqueda o tareas de recomendación [1], [2], [3]. A día de hoy, dependiendo del ámbito de aplicación, además de las redes totalmente conectadas entre capas (las MLPs) que evolucionaron a partir del modelo primigenio *perceptrón*, existen otra gran variedad de modelos DNN, como las categorizadas como convolucionales (CNNs) o las recurrentes (RNNs). Todas estas DNNs operan sobre datos mapeados en el espacio Euclídeo (imágenes o series temporales).

Sin embargo, los objetos del mundo real a menudo se definen en términos de sus conexiones. Un conjunto de objetos (vértices o nodos) y las conexiones (aristas) entre ellos que se expresan naturalmente como grafos, estructuras de datos mapeadas en el espacio no Euclídeo. Es más, los análisis de grafos con *Machine Learning* (ML) están recibiendo cada vez más atención por parte de la comunidad científica, debido a la gran capacidad expresiva de los grafos y a su similitud con la representación de datos, por ejemplo, redes sociales, publicaciones científicas o redes de interacción entre proteínas [4], [5], [6].

Desafortunadamente, las DNNs tradicionales no son capaces de extraer adecuadamente la información de un grafo debido a la propia naturaleza de los datos. Por este motivo, las Redes Neuronales de Grafos o *Graph Neural Networks* (GNNs) surgen como un nuevo tipo de DNN especialmente adaptadas para procesar este tipo de estructuras de datos basadas en grafos.

Las GNNs han irrumpido en la escena del aprendizaje automático de los últimos años debido a su capacidad para modelar y aprender de datos estructurados en forma de grafo, centrándose en tareas como la clasificación o la predicción de vértices [7].

El procesamiento de cada una de las capas de una GNN involucra la ejecución de dos fases que, dependiendo del tipo de GNN, pueden intercambiar el orden de su ejecución. La primera fase, denominada **agregación**, consiste en recoger para cada vértice del grafo las características de sus vecinos mediante alguna función lineal como la media aritmética. En la segunda fase, llamada **combinación**, se actualizan las nuevas características de los vértices generalmente mediante el uso de una DNN. El entrenamiento de una GNN consistirá en ir procesando cada una de sus capas y actualizando los pesos siguiendo los mismos principios del entrenamiento de las DNN tradicionales (cálculo de la función de pérdida, algún tipo de *backpropagation* y actualización de pesos).

Entre los diferentes tipos de GNNs estudiados hasta ahora por la comunidad científica, las Redes Convolucionales de Grafos o *Graph Convolutional Networks* (GCNs) están recibiendo actualmente mucha atención. Las GCNs se inspiran en las CNNs y, más

[1]Dpto. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, e-mail: {n.mesegueriborra,francisco.munoz2,meacacio}@um.es.

[2]Dpto. de Grado en Ingeniería Informática, Universidad Católica de Murcia, e-mail: jlabellan@ucam.edu.

concretamente, aplican el concepto de capa convolucional. En una GCN, una sola operación de convolución transforma y agrega información de características de nodos vecinos al vértice; múltiples convoluciones de este tipo se apilan para propagar la información de los vértices a través del grafo, alcanzando así los límites del grafo. Cabe destacar que el paso de agregación utiliza una estructura de datos muy dispersa, la matriz de adyacencia.

Uno de los mayores retos a los que hay que enfrentarse es acelerar el procesamiento de los grafos, que cada vez está más limitado por los accesos a la memoria principal que carecen de un patrón predecible y regular. En particular, estos accesos presentan escasa localidad temporal, ya que la estructura irregular de los grafos (los grafos suelen tener muchos vértices pero relativamente pocas aristas, lo que resulta una enorme matriz de adyacencia con muy pocas conexiones) da lugar a accesos aparentemente aleatorios que son difíciles de predecir con antelación. Además, debido al alto grado de dispersión (nivel de *sparsity*) de la matriz de adyacencia, la localidad espacial es también muy escasa.

Partition-Centric Processing for Accelerating Graph Convolutional Network (PCGCN) [8], es un acelerador software para entrenamiento de GCNs sobre sistemas equipados con GPU, que propone particionar el grafo de entrada a las GCN en subgrafos para procesarlos durante la etapa de agregación dependiendo del nivel de sparsity. Así, PCGCN elegirá un modo de procesamiento *sparse*, si el nivel de sparsity del subgrafo supera cierto umbral, o un modo de procesamiento *denso* en caso contrario. Para el caso sparse, para evitar el almacenamiento de operandos con valor cero y operaciones de multiplicación sobre los mismos, se hace necesaria una representación más eficiente comprimida del subgrafo, como por ejemplo COO o CSR. Con este modo híbrido y selectivo de ejecución de sugrafos, PCGCN consigue acelerar notablemente la ejecución en comparación con los modelos GCN tradicionales.

En este trabajo, presentamos el *framework* CPU-PCGCN, una implementación de PCGCN especialmente concebida para ser utilizada en sistemas de bajas prestaciones, típicamente alimentados por baterías que pueden carecer de GPU (por ejemplo, dispositivos del Internet de las Cosas o IoT por sus siglas en inglés), donde el modelo GCN desplegado puede estar ya preparado para inferir un resultado o, por otro lado, requerir de entrenamiento *in-situ*. CPU-PCGCN se ha construido a partir de un modelo GCN escrito en PyTorch, PyGCN [9]. Además, hace uso de varias herramientas contemporáneas como METIS [10], que permite realizar distintas estrategias de particionado atendiendo a diversas propiedades de los grafos (agrupar nodos en cada partición en función de su conectividad), o generadores de grafos sintéticos, como Graphlaxy[1] o PaRMAT [11], para un estudio más amplio de la herramienta. Además, CPU-PCGCN emplea diferentes técnicas (paralelismo a nivel de tarea, propiedades de las matrices o incluso propiedades de representación de los grafos) para acelerar el cálculo y procesamiento del modelo. En comparación con la implementación base de PyGCN usando conjuntos de datos reales y sintéticos, CPU-PCGCN consigue un aumento de velocidad de hasta 3.94 veces en el mejor de los casos. [2]

El resto de este trabajo se organiza de la siguiente manera: la Sección II analiza diferentes técnicas para acelerar el cálculo de las GCNs y los modelos relacionados con PCGCN, así como técnicas de partición eficientes. Después, la Sección III presenta el groso de nuestro trabajo, CPU-PCGCN. En la Sección IV se evalúan los resultados obtenidos con CPU-PCGCN y se comparan con respecto a otros modelos GCN. Por último, la Sección V presenta las conclusiones y las líneas de trabajo futuro.

II. Antecedentes y Trabajo Relacionado

A. Dominio de grafos

Los objetos del mundo real se definen a menudo en términos de sus conexiones con otras cosas. Un conjunto de objetos (vértices) y las conexiones (aristas) entre ellos, se expresan naturalmente como un grafo. Además, podemos caracterizar los grafos asociando direccionalidad a las aristas (dirigidos, no dirigidos), donde para un grafo dirigido, un vértice e tiene un origen e_{src} y un destino e_{dst}; en este caso, la información fluye desde *src* hasta *dst*. También pueden ser no dirigidos, donde no existe la noción de origen ni destino y la información fluye en ambos sentidos. Aunque los grafos dirigidos son eficaces para modelar ciertas estructuras del mundo real, los grafos no dirigidos son bastante frecuentes en la práctica y se modelan mejor para muchas interacciones del mundo real, por ende, los grafos no dirigidos serán un concepto clave para el particionado eficiente del grafo.

Consideremos el ejemplo de un grafo de citaciones (p.ej. autores que citan el trabajo de otros autores en sus artículos), cada artículo es un vértice, y cada arista es una cita entre un artículo y otro. Además, podemos añadirle características al grafo, características únicas a cada artículo (p.ej. cuántas palabras de un diccionario con un tamaño estimado, cada artículo contiene), nos referiremos a esto como *features vector* para cada artículo, o *features matrix* para referirnos al grafo. Para finalizar, podemos etiquetar los artículos dentro de un tópico (p.ej. ciencia, medicina, arquitectura, etc.). Esto sustenta el funcionamiento de una GNN.

Asumamos la terminología *vértices* V y *aristas* E (*edges*). Con ellas podremos construir una estructura de datos que almacene el grafo, la matriz de adyacencia (utilizando un entero de valor 0/1 para representar las conexiones entre vértices). Esta estructura de datos resulta ser (en muchos casos) muy dispersa (mayor densidad de ceros). Para evitar almacenar o manejar esta gran cantidad de ceros, se suele aceptar una representación de la matriz utilizando diferentes

[1] `https://github.com/BNN-UPC/graphlaxy`

[2] CPU-PCGCN está distribuido abiertamente a la comunidad científica:`https://github.com/NicolasMeseguer/pcgcn`.

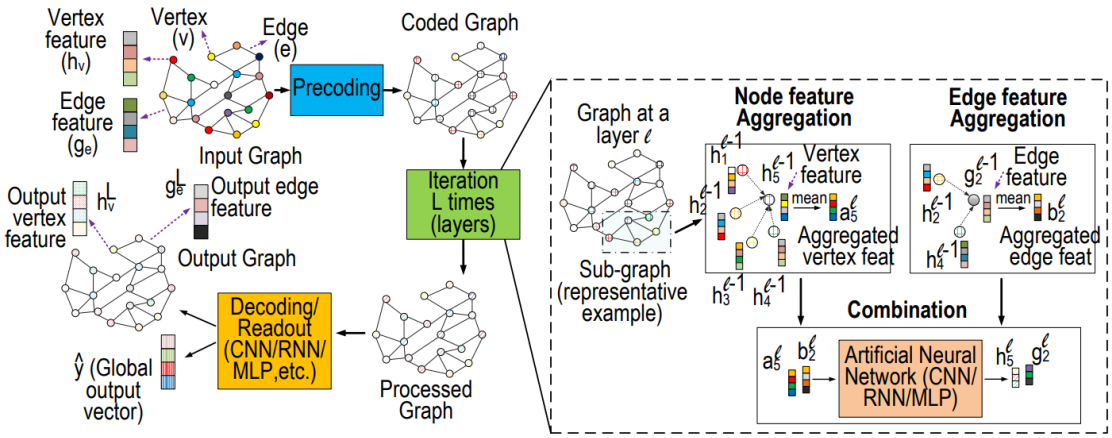

Fig. 1: Procesamiento de una GCN [7].

estructuras de datos comprimidos; por ejemplo, los formatos COO (*Coordinate List*) y CSR (*Compressed Sparse Row*) proporcionan un acceso más eficiente a los datos y facilitan las operaciones matriciales.

Finalizamos esta sección explicando conceptos y métricas relevantes que serán significativos para el resto del trabajo. Cuando nos referimos a grafos, un grafo G es un conjunto de puntos V, llamados vértices, que están conectados por un conjunto de enlaces E, llamados aristas. Así, $G = (V, E)$.

En la teoría de grafos, el concepto de *cluster* (o comunidad) se refiere a un grupo de vértices que están más conectados entre sí que con el resto de vértices fuera del cluster (llamado *small-world*), y es un concepto relevante que tiende a aparecer en la mayoría de los grafos del mundo real. Una métrica que define esta tendencia dentro de un grafo se llama *cluster coefficient*, y mide el grado en que los vértices de un grafo tienden a agruparse (localidad). Volviendo al ejemplo propuesto anteriormente de citaciones, esto ocasiona que aquellos vértices (artículos) de tópicos similares tiendan a estar relativamente cerca unos de otros.

B. Graph Neural Networks

La intuición de las GNNs es que los vértices se definen naturalmente por sus vecinos y conexiones; en esencia, el objetivo es (véase la Figura 1) aprender un estado que, para cada vértice, encapsula tanto el vector de características inicial del vértice (atributos o características) como la información de los vecinos (que representa la estructura del grafo local que los rodea). Este estado se utiliza para producir el resultado. A través de un proceso iterativo de paso de mensajes de información entre vértices, estos algoritmos capturan las complejas dependencias de los vértices y sus conexiones.

El primer paso de una GNN es, para cada nodo, recoger las características del vértice, los vértices de la vecindad y el grafo, y **agregarlas** en un solo conjunto. Hay que tener en cuenta que esta operación es muy costosa desde el punto de vista computacional debido al alto grado de dispersión de la matriz de adyacencia. Después, hay un proceso de **combinación** de todos los atributos; este puede realizarse mediante diferentes estrategias, dependiendo de la GNN concreta. Esta combinación da como resultado un nuevo vector de características que se utiliza para actualizar la información de los vértices o aristas. En la práctica, estos dos pasos no tienen por qué realizarse en el mismo orden; en ocasiones se realiza primero la combinación, seguida de la agregación.

C. Graph Convolutional Networks

Generalizar modelos DNN bien establecidos, como las RNNs o las CNNs, para que puedan trabajar con grafos es un problema difícil. Algunos artículos recientes introducen arquitecturas especializadas para problemas específicos, mientras que otros utilizan convoluciones de grafos basadas en la teoría de grafos espectrales [12] [13]. Sin embargo, GCN [14] parte de la base de teoría de grafos espectrales pero introduciendo simplificaciones que en muchos casos permiten tanto tiempos de entrenamiento significativamente más rápidos como una mayor precisión predictiva.

Las GCNs reciben el nombre en base a los filtros que se utilizan en el grafo (hablaremos de esto más adelante). Para estos modelos, el objetivo es aprender una función de señales/características en un grafo $\boldsymbol{G = (V,E)}$ que toma como entrada:

- Un vector de características $\boldsymbol{x_i}$ por cada vértice $\boldsymbol{i}$; resumido en una matriz $\boldsymbol{N \times D}$ de características $\boldsymbol{X}$ ($\boldsymbol{N}$: número de vértices, $\boldsymbol{D}$: número de características).
- Una descripción representativa de la estructura del grafo en forma de matriz; normalmente en forma de matriz de adyacencia comprimida $\boldsymbol{A}$.

Y produce una salida a nivel de nodo $\boldsymbol{Z}$ (una matriz $\boldsymbol{N \times F}$ de carcterísticas, donde $\boldsymbol{F}$ es el número de características de salida por nodo). Entonces, cada capa de la red neuronal puede escribirse como una función no lineal:

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

Con $H^0 = X$ y $H^L = Z$ (o Z para la salida del grafo), siendo $\boldsymbol{L}$ el número de capas. El modelo di-

fiere entonces sólo en cómo se elige y parametriza $f(,)$. Consideremos la siguiente fórmula muy simple de propagación por capas:

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

Donde W^l es la matriz de pesos para la ***l*-ésima** capa de la red neuronal y $\sigma(-)$ es una función de activación no lineal como ***ReLu***.

Pero primero, vamos a abordar dos limitaciones de este modelo simple: (1) la multiplicación con $\boldsymbol{A}$ significa que, para cada nodo, sumamos todos los vectores de características de todos los vértices vecinos **pero no el propio vértice**. Esto se resuelve fácilmente añadiendo la matriz de identidad a $\boldsymbol{A}$.

La segunda limitación (2) importante es que $\boldsymbol{A}$ no suele estar normalizada y, por tanto, la multiplicación cambiará completamente la escala de los vectores de características. Para solucionarlo **normalizamos** $\boldsymbol{A}$ de tal manera que **todas las filas suman igual a uno**. En la práctica, se vuelve más interesante cuando se utiliza una **normalización simétrica**, es decir, convertimos la matriz de adyacencia en simétrica y posteriormente la normalizamos (por esto es más práctico usar grafos no dirigidos). Combinando estos dos trucos, llegamos a la regla de propagación que utilizaremos más adelante en el modelo que proponemos (Sección III).

D. METIS

Aquellos algoritmos capaces de encontrar una buena partición en grafos altamente desestructurados, son fundamentales para el desarrollo de soluciones eficientes. Recientemente, varias investigaciones han estudiado una clase de algoritmos de partición de grafos que reducen el tamaño del grafo colapsando vértices y aristas. De estos estudios se observa que estas técnicas son muy prometedoras; sin embargo, no se sabe si se puede conseguir que produzcan particiones de alta calidad de forma consistente.

Serial Graph Partitioning and Fill-reducing Matrix Ordering (METIS) [10] presenta varias heurísticas novedosas para obtener mejores resultados y en un tiempo sustancialmente menor que otros algoritmos de partición, siendo capaz de fraccionar un grafo **no dirigido** en un número especifico de $\boldsymbol{K}$ subgrafos, teniendo en cuenta las propiedades de los grafos.

E. PCGCN

Partition Centric Graph Convolutional Network (PCGCN) [8] es un método de aceleración basado en software que tiene como objetivo mejorar la eficiencia del cálculo de las GCNs, tanto en memoria como en rendimiento. El aspecto en el que se centra PCGCN para alcanzar dicha mejora y, que el resto de modelos GCN no lo hacen, es en las propiedades del grafo, que pueden ser fundamentales para el cálculo de las redes neuronales.

PCGCN usa un método centrado en el particionado (se centra en computar las particiones, en vez del grafo completo), que aprovecha la característica de localidad de los grafos. Simultáneamente, el algoritmo se ajustará a la dispersión de cada partición resultante, acelerando aún más la etapa de propagación.

La propagación de las GCNs en PCGCN se modifica de forma que, en lugar de considerar únicamente el grafo, las operaciones a realizar dependen de cada subgrafo. Para cada capa GCN, primero se recolectan los vectores de características que pertenecen al subgrafo (fase de agregación) y para cada partición, se ejecuta el método de propagación. Una vez completado esto para todos los subgrafos, los resultados de cada uno se concatenan para generar la salida de la capa (fase de combinación). Gracias al enfoque de los subgrafos, el rango de vértices y aristas requeridos para los cálculos disminuye, lo que permite accesos más rápidos al cargar esta información desde la caché.

De esta manera, PCGCN consigue obtener una ventaja en la localidad. Sin embargo, el cómputo de las particiones llega a ser muy complejo dependiendo del grado de dispersión de cada una; por ello, PCGCN soporta dos modos de cómputo (cálculo dual), que varía dependiendo de las características del subgrafo a procesar: *Selective Mode* o modo selectivo (cuando el número de aristas entre el subgrafo que se procesa y una de sus particiones vecinas es escaso; por ende, se usa un formato comprimido, p.ej. CSR) y *Full Mode* o modo completo (cuando el número de aristas entre el subgrafo que se procesa y una de sus particiones es muy denso).

De este modo, PCGCN es una herramienta diseñada, evaluada y propuesta para ejecutarse en la GPU y ejecutar eficientemente tanto el entrenamiento como la inferencia de las GCNs.

III. CPU-PCGCN

A pesar de que PCGCN ha sido especialmente diseñado para aprovechar las características de las arquitecturas GPUs modernas, en este trabajo hemos elegido la CPU como plataforma de cómputo destino. Esto abre la aplicación de PCGCN al procesamiento eficiente de la inferencia en otros campos, como el *edge computing* o *IoT*, donde los dispositivos utilizados para el procesamiento tienen límites de potencia computacional y arquitecturas no GPU, o el entrenamiento *in-situ* de *datasets*.

Por tanto, en este trabajo presentamos nuestra propuesta llamada *CPU-PCGCN*, una extensión de PCGCN que está destinada a sistemas que constan solo de CPU. CPU-PCGCN ha sido programado mediante el popular framework PyTorch [15] lo que facilita su utilización y extensión, y que parte del framework para procesamiento de GCNs (tanto para entrenamiento como para inferencia) denominado PyGCN [9].

Como demostraremos en la Sección IV, donde evaluamos el rendimiento de CPU-PCGCN en una variedad de conjuntos de datos reales y sintéticos, nuestra implementación supera a la línea de referencia (PyGCN), por un factor de hasta 3,94 veces, en el

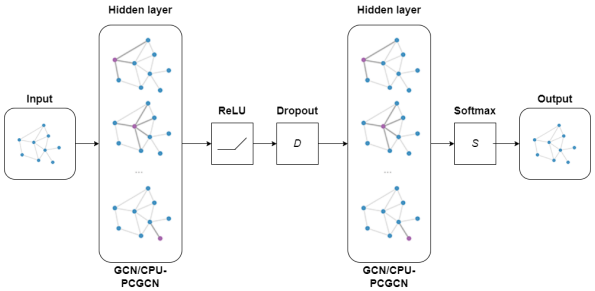

Fig. 2: Modelo CPU-PCGCN.

mejor de los casos.

Para obtener una mejor comprensión de CPU-PCGCN, la Figura 2 ofrece un esquema de alto nivel. Como puede observarse, el modelo contiene dos capas ocultas, denominadas GCN o CPU-PCGCN. De hecho, ambas capas son idénticas (arquitecturalmente hablando) y sólo se diferencian en la forma de calcular los parámetros de entrada (la primera siendo el grafo completo en un formato comprimido, y la segunda son las particiones generadas a partir del grafo, también en un formato comprimido). Nuestra propuesta sigue una serie de pasos bien establecidos: (A) procesamiento de los datasets; (B) particionado del grafo; (C) determinación de los *edge blocks* y su *sparsity*; y (D) ejecución del modelo.

A. Procesamiento de los datasets

CPU-PCGCN admite dos tipos de conjuntos de datos de grafos, los del mundo real, como Cora[3] o PubMed[4], y los sintéticos. En el caso de los del mundo real, estamos utilizando los conjuntos de datos en un formato preprocesado[5], por comodidad de almacenamiento.

En cuanto al post-procesado de los datasets, debemos destacar las dos limitaciones importantes ya mencionadas (ver Sección II-C); la primera limitación (1) puede resolverse fácilmente añadiendo la matriz de identidad I a la matriz de adyacencia A. En el caso de la segunda limitación (2), la solución es un poco más compleja: primero hacemos la matriz de adyacencia simétrica, $A = A + (A^T \times (A^T > A) - A \times (A^T > A))$[6] y posteriormente podemos normalizarla, teniendo en cuenta que la suma de las filas debe ser igual a 1. Junto con la matriz de adyacencia, también se normaliza la matriz de características.

Aplicando estas dos técnicas, resolvemos las cuestiones ya mencionadas anteriormente. Las representaciones de ambas matrices están en formato comprimido (ver Figura **??**), la primera en representación COO y la segunda en CSR. En el caso de las etiquetas, se almacenan en un vector del tipo *one-hot*.

B. Particionado del grafo

Para utilizar la técnica centrada en el particionado, CPU-PCGCN aplica una partición de grafos en 2D. Como se muestra en la Figura 3, se divide el grafo completo en K subgrafos y por lo tanto crea K bloques de vértice disjuntos y $K \times K$ *edge blocks* donde $E_{i,j}$ representa las *aristas* entre dos subgrafos i y j.

Hay multitud de algoritmos de particionado (particionado aleatorio, *min-cut partitionig*, etc.), sin embargo, nos hemos dado cuenta de que un método de particionado aleatorio perjudicará la localidad de los subgrafos ya que ignora la localidad y asigna aleatoriamente los vértices a las particiones. Es por ello que en este trabajo utilizamos la herramienta de particionado METIS, a la cual, además del grafo de entrada, el usuario debe proporcionar el número K de subgrafos en los que debe hacerse la división.

Trabajos anteriores han demostrado que METIS asegura una alta calidad y particiones uniformes (equilibra el número de vértices en todas las particiones), aunque debido a que el grafo es simétrico (y no dirigido), hay que tener en cuenta que en la mayoría de los casos, los subgrafos triangulares K pueden tener la misma cantidad de vértices y de *sparsity*, lo que abre una nueva ventana de mejora que discutiremos más adelante.

C. Determinación de los edge blocks y su sparsity

Una vez obtenidas las particiones, podemos empezar a formar los *edge blocks*. Gracias a la simetría de la matriz de adyacencia sabemos que la triangular superior de los *edge blocks* es igual a la traspuesta de la triangular inferior.

Asumimos que: $edge\_block[0][1] = (edge\_block[1][0])^T$, utilizando esta idea podemos calcular fácilmente la identidad, la triangular inferior y su grado de *sparsity*, para finalmente trasponer la triangular inferior y obtener la triangular superior. Se ha demostrado que este enfoque consigue un aumento de velocidad de hasta 9 veces en comparación con la implementación anterior, que calculaba los *edge blocks* y su *sparsity* secuencialmente, para $K \times K$ *edge blocks*.

El *sparsity* de cada uno de los subgrafos se representa con un número entero entre 0 y 100, lo que significa que 100 es una matriz totalmente dispersa (es decir, no hay ninguna arista dentro del *edge block*). Calculamos la dispersión de un subgrafo dado K como:

$$100 - \left(\frac{E_{i,j}}{K_i \times K_j} \times 100\right) \quad (1)$$

D. Ejecución del modelo

Una cuestión planteada es la equivalencia de los modelos CPU-PCGCN y GCN: CPU-PCGCN sólo cambia el orden de cálculo de vecinos para un determinado vértice comparado con el modelo GCN original. Las operaciones en la etapa de propagación del grafo de GCN son la multiplicación y la suma de elementos, que son conmutativas y asociativas respectivamente. Por lo tanto, CPU-PCGCN es igual a GCN y puede producir los mismos resultados numéricos, por lo tanto misma convergencia por época.

[3] https://relational.fit.cvut.cz/dataset/CORA
[4] https://pubmed.ncbi.nlm.nih.gov/download/
[5] https://github.com/kimiyoung/planetoid
[6] https://github.com/yao8839836/text_gcn/issues/17

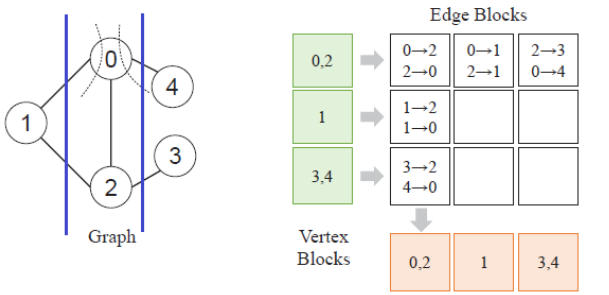

Fig. 3: Particionado y procesamiento de un grafo no dirigido.

En CPU-PCGCN, proponemos aprovechar la localidad de los grafos para acelerar el cálculo de las GCNs mediante la aceleración de la fase de propagación. En concreto, CPU-PCGCN introduce un esquema de particionado céntrico (**1**) en la etapa de propagación del grafo, lo que hace que PCGCN consiga esa localidad en el procesamiento de grafos. Además, se introduce un método de cálculo de subgrafos en modo dual (**2**) para acelerar aún más la propagación del grafo mediante el diseño de modos de cálculo acordes con la densidad (sparsity) de un subgrafo.

1. *Particionado céntrico de grafos*: CPU-PCGCN modifica la etapa de propagación de grafos de GCN, pasando de un esquema de grafo completo a uno centrado en subgrafos. Para cada capa GCN, el estado de la anterior (*hidden state*) es transformado por una MLP, que es la misma que la empleada en la GCN original (combinación). A continuación, para cada subgrafo, reúne y acumula estados de sí mismo y de los subgrafos vecinos para ejecutar la fase de propagación del grafo en la partición (agregación). Las salidas de cada subgrafo se combinan como la salida de esa capa.
   Dado que el rango de vértices de origen y destino se limita al subgrafo, esta técnica de procesamiento basado en particiones consigue sacar un mayor provecho de la jerarquía de memoria. Como resultado, procesar aristas con los mismos orígenes o destinos puede cargar datos de la caché, lo que es especialmente útil para los grafos con un alto grado de localidad.
2. *Modo de cálculo dual*: Para aprovechar la localidad del grafo, CPU-PCGCN procesa la GCN desde la perspectiva del subgrafo para cada capa. Los grafos del mundo real suelen tener distribuciones irregulares. La irregularidad de un grafo suele dar lugar a densidades variables en los subgrafos. Por ello, presentamos un enfoque de cálculo centrado en los subgrafos; utilizando un modo de cómputo dual basado en la densidad del subgrafo para acelerar significativamente el cálculo.
   a) *Modo Selectivo*: Cuando hay pocas aristas en el *edge block* $E_{k,i}$, CPU-PCGCN utiliza un modo selectivo para procesar este subgrafo. Las características del vértice de origen se multiplicarán con el valor escalar de la arista, y el resultado se añadirá a h_k^l usando la función dispersa (*spmm*).
   b) *Modo Completo*: Cuando hay muchas aristas en un subgrafo, el procedimiento de decodificación en el modo selectivo puede tener un gran impacto en la eficiencia del procesamiento, por ello presentamos el modo completo. Asume que los subgrafos S_k y S_i están completamente conectados; si no hay ninguna arista $e_{p\to q}$ entre el vértice p y q, se insertará esta arista con valor 0. Este modo usa la función densa (*mm*).
   c) *Modo Híbrido*: El modo *selectivo* es adecuado para *edge blocks* dispersos, mientras que el modo *completo* es mejor para *edge blocks* densos. Debido a la irregularidad, un grafo del mundo real puede contener *edge blocks* tanto dispersos como densos. La complejidad computacional de ambos modos está relacionada principalmente con el grado de dispersión de *edge block*. Por lo tanto, tomamos ese grado de dispersión SS_k para seleccionar el modo de procesamiento para un *edge block* K. En concreto, perfilamos el tiempo de ejecución de los modos completo y selectivo para un bloque de dispersión p, según un umbral establecido por el usuario, o por defecto (60 %) si no se especifica. Elegimos el modo *selectivo* si $SS_k > 60\,\%$, en caso contrario seleccionamos el modo *completo* para el procesamiento de un *edge block* determinado.

Algorithm 1 Cálculo a nivel de capa del modelo CPU-PCGCN

Símbolos: grafo de entrada: $G = (V, E)$, capas: $l = \{1, ..., L\}$, subgrafos: $\{S_k = (V_k, E_k, SS_k) | k = 1, ..., K\}$, vértices en el subgrafo k: V_k, aristas en el subgrafo k: E_k, dispersión del subgrafo k: SS_k, aristas entre el subgrafo i y j: $E_{i,j}$, características de la capa l: h^l (h^0 indica las características de entrada), peso de la capa l: w^l

for $l = 1, ..., L$ **do**
  $a^l = h^{l-1} \times w^l$
  Divide $a^l \to \{a_k^l | k = 1, ..., K\}$ de acuerdo a los subgrafos;
  // propagación de grafos
  for $k = 1, , K$ **do**
    // modo de cálculo dual
    if $SS_k > args.threshold$ **then**
      $h_k^l = \sum_{i=1}^{K} f_{spmm}(E_{k,i}, a_i^l)$
    else
      $h_k^l = \sum_{i=1}^{K} f_{mm}(E_{k,i}, a_i^l)$
    end if
    // combina los estados de k
    $h^l = concat(h_k^l)$
  end for
end for
return h^L;

Datasets	#vertices	#aristas	#características	#etiquetas	coeficiente de *clustering*
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

Tabla I: Datasets usados en la evaluación. (K: Miles, NM: No Medido)

IV. Evaluación

En este capítulo, demostramos la eficiencia de CPU-PCGCN evaluándolo en conjuntos de datos reales y sintéticos. La sección IV-A describe la metodología de los experimentos (configuración del entorno, generadores de conjuntos de datos sintéticos, propiedades de los conjuntos de datos, etc.). La sección IV-B muestra las comparaciones generales de tiempo de ejecución de CPU-PCGCN y la implementación base PyGCN.

A. Metodología

1. *Configuración del entorno*: Evaluamos CPU-PCGCN en una plataforma única equipada con un procesador dual Intel Xeon E5-2640v4 a 2.4 GHz (20 núcleos físicos en total) y 125 GB de memoria. El sistema operativo instalado es Ubuntu 16.04. Comparamos CPU-PCGCN con la implementación GCN de código abierto [9] en PyTorch v1.2.0 [15] (PyGCN), SciPy [16] v1.5.4 y Python NumPy [17] v1.19.5. Para que la comparación sea justa, todos los sistemas de referencia son las últimas versiones estables (en comparación con las versiones utilizadas en PyGCN), y utilizan la misma versión de Python, 3.6.15.
2. *Generadores de datasets sintéticos*: Para evaluar las mejoras que CPU-PCGCN puede proporcionar en grafos de diferentes niveles de *sparsity*, utilizamos las herramientas PaRMAT y Graphlaxy. Esto también nos sirve para demostrar el modo híbrido cuando se utilizan grafos de *sparsity* irregular. *PaRMAT* [11], es un generador de grafos multihilo ampliamente utilizado, para generar grafos sintéticos similares a los grafos del mundo real. *Graphlaxy*[7] es una herramienta desarrollada por el Barcelona Neural Networking Center (BNN) utilizada para crear datasets de grafos sintéticos con distribuciones uniformes.
3. *Datasets*: La Tabla I muestra los conjuntos de datos reales y sintéticos utilizados para la evaluación, incluyendo, la red de citaciones PubMed (pubmed) [18], el conjunto de datos científicos Cora (cora)[8] y la red de citas citeseer (citeseer)[9].
   En PaRMAT, establecemos un número variable de vértices y generamos diferentes números de aristas para conseguir densidades de 0.01 %, 0.06 %, 0.12 % y 1 % (por ejemplo, PaRMAT-1 indica un grafo con 1 % de densidad)[10]. Como algunos conjuntos de datos no tienen las características o etiquetas requeridas para el modelo GCN, utilizamos una distribución aleatoria para generarlas. La columna de características de la Tabla I representa el tamaño del vector de características, y la de etiquetas, el número de etiquetas por vector disponibles. Para los conjuntos de datos sintéticos no se ha medido el coeficiente de *clustering*, NM[11]

[7] `https://github.com/BNN-UPC/graphlaxy`

Tiempo (seg.)					
Fases	V1	V2	V3	V4	V5
Dividir a^l	0.098	0.0959	0.185	0.004	0.0065
Propagación del grafo	0.001	0.578	0.604	0.018	0.0196
Concatenación	0.642				
Conversión a Torch	NA	NA	NA	$\simeq 0$	$\simeq 0$
Total (seg.)	0.74	0.67	0.789	0.022	0.026

Tabla II: Desglose del tiempo de procesamiento para distintas versiones de CPU-PCGCN (en seg.) para 1 época de entrenamiento

B. Resultados

Consideramos, en primer lugar, el *paralelismo a nivel de capa*. Al principio, el tiempo de cálculo de una sola iteración en CPU-PCGCN era muy superior al de su homólogo GCN; ello motivó a estudiar distintas técnicas para acelerar el tiempo de cómputo. La forma de computar las capas ha pasado por una serie de etapas bien definidas: (V1), todo el cálculo que se realiza a nivel de capa se hace con la librería *Torch*; (V2), la fase de concatenación de la capa h_k^l se realiza inmediatamente después de su cálculo; (V3), igual que V1, excepto que el cálculo de a^l se paraleliza a nivel de tarea utilizando K hilos; (V4), el cálculo ahora se realiza con la librería *NumPy*, ello incluyendo las conversiones necesarias a *PyTorch* posteriormente; y, (V5), como V4, se ha intentado (aunque sin éxito) paralelizar el cálculo de a^l utilizando K hilos.

La Tabla II muestra la implementación base y la evolución por la que ha pasado. Estos resultados se han obtenido particionando el grafo en 4 subgrafos y con el dataset de *cora*.

[8] `https://relational.fit.cvut.cz/dataset/CORA`

[9] `https://linqs.soe.ucsc.edu/data`

[10] Estos conjuntos de datos están disponibles en el repositorio git, con el siguiente nombre (en orden con la densidad): *Magenta_Spoonbill*, *Red_Magpie*, *Lime_Hawk* y *Silver_Parrot*

[11] **N**no **M**medido: al tratarse de conjuntos de datos sintéticos, sólo consideramos su densidad, no el *clustering*.

Tiempo (seg.)							
Datasets	GCN (D)	GCN (S)	CPU-2	CPU-4	CPU-8	CPU-16	Aceleración
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**

Tabla III: El tiempo de ejecución global (s) de GCN y CPU-PCGCN, denotado como CPU-*particiones*, para 100 épocas de entrenamiento.

Tiempo (seg.)					
Datasets	*mejor-tiepo*	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

Tabla IV: Tiempo de ejecución global (seg.) variando el umbral (*threshold*), denotado como CPU-*particiones-umbral*

Como puede verse, el paralelismo a nivel de tarea para la aceleración del cálculo a nivel de capa parece añadir más sobrecarga que aceleración. Observe las dos celdas azules de V3 y V5, comparando ambas con la versión secuencial (V2 y V4 respectivamente) el tiempo consumido para crear un solo hilo es de hasta 0,046 para V3 y 0,001 para V4, lo que es exactamente 1/4 del tiempo de división de a^l.

En cuanto al *rendimiento*, la Tabla III compara el resultado de la mejor versión de CPU-PCGCN con la implementación del modelo base PyGCN en PyTorch. En general, CPU-PCGCN consigue un aumento de velocidad medio de 2,11 (hasta 3,94). Comparamos la implementación base PyGCN (GCN) utilizando el cálculo disperso (S) o denso (D) (no se particiona el grafo) frente a CPU-PCGCN con una cantidad específica de particiones, que denotamos como CPU-particiones.

Como se puede observar, en todos los casos CPU-PCGCN obtiene una mejora en el tiempo. Además, en el caso de los grafos con mayor densidad (**sólo para PaRMAT-1**), se puede observar cómo el número de particiones (4) beneficia el tiempo de cálculo. Destacar que, aumentar el número de particiones a 16, no mejora el tiempo ni la localidad en la caché (véase PaRMAT-1, donde a mayor partición, mayor densidad de subgrafos).

Para evaluar la contribución de los dos modos (selectivo y completo), ejecutamos CPU-PCGCN modificando el umbral (*threshold*) de *sparsity* para forzar la ejecución de uno de los modos duales. La tabla IV muestra los resultados comparados con el mejor tiempo obtenido anteriormente. Los nombres de las tablas se denotan como CPU-particiones-umbral.

Como se puede ver, en la mayoría de los casos, al variar el umbral necesario para ejecutar la operación en modo *sparse*, el tiempo de cálculo tiende a empeorar. Sin embargo, si nos fijamos en PaRMAT-1 (el grafo más denso) podemos ver que se obtiene una ligera mejora en el tiempo. A partir de esto podemos proyectar cómo una mayor densidad en los grafos (características de los grafos del mundo real) conduce a un tiempo de cálculo progresivamente mejor; el cómputo de particiones densas en modo completo resulta en un mejor rendimiento.

Por último, si observamos el dataset PubMed (que es muy disperso), podemos ver que al aumentar el número de particiones y reducir el umbral requerido (dos últimas columnas de la tabla IV), el tiempo aumenta; probablemente alguna partición densa está superando dicho umbral y se está ejecutando en modo disperso; como resultado, la representación en formato comprimido (COO/CSR) está afectando tanto al rendimiento como a la memoria. Para el resto de los experimentos, se observa como el tiempo de cálculo variando el umbral es muy similar.

V. Conclusiones

En este trabajo, presentamos CPU-PCGCN, una alternativa a PCGCN basada en la CPU para el cálculo rápido de GCNs. A diferencia de PCGCN, esta nueva implementación abre la aplicación de PCGCN a otros campos, como dispositivos del Internet de las Cosas (IoT), donde los dispositivos utilizados para el procesamiento tienen límites de potencia computacional y arquitecturas no GPU.

Las principales contribuciones de nuestro trabajo son: (1) Nueva implementación de PCGCN adaptada

específicamente al procesamiento eficiente, tanto del entrenamiento como de inferencia, de modelos GCN en plataformas con limitaciones arquitecturales (no GPU); y (2) Evaluación detallada de CPU-PCGCN, que incluye un análisis exhaustivo mostrando el tiempo empleado en cada fase y una evaluación completa utilizando grafos reales y sintéticos.

Finalmente, presentamos algunas vías potenciales para futuras extensiones de CPU-PCGCN: (A) Determinar un umbral de *sparsity* preciso, para ello se pueden realizar pruebas exhaustivas con distintos conjuntos de datos para perfilar aún mas el umbral óptimo; (B) CPU+GPU-PCGCN, que será una implementación heterogénea que aprovechará tanto la CPU como la GPU de manera colaborativa para acelerar aún mas el procesamiento de PCGCN; y (C) Optimizaciones al código usando TVM [19], compilador fuente-a-fuente para aplicaciones de ML, que permitirá obtener un código de CPU más optimizado (incluyendo diferentes tipos de paralelismo, aceleración de la capa computacional, etc.).

Agradecimientos

Trabajo financiado por RTI2018-098156-B-C53 (MCIU/AEI/FEDER,UE), NSF OAC 1909900 y US Department of Energy ARIAA co-design center. Además de por el proyecto PID2020-112827GB-I00 (MCIN/AEI/10.13039/501100011033). Francisco Muñoz-Martínez ha sido financiado mediante la beca 20749/FPI/18 de Fundación Séneca, Agencia de Ciencia y Tecnología de la Región de Murcia.

Referencias

[1] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," New York, NY, USA, 2018, ICSE '18, p. 303–314, Association for Computing Machinery.

[2] Alessio Schiavo, Filippo Minutella, Mattia Daole, and Marsha Gomez Gomez, "Sketches image analysis: Web image search engine usinglsh index and DNN inceptionv3," *CoRR*, vol. abs/2105.01147, 2021.

[3] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, and Misha Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," *CoRR*, vol. abs/1906.00091, 2019.

[4] 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*, vol. 34, pp. 1054–1061, 04 2020.

[5] Yingxue Zhang, Soumyasundar Pal, Mark Coates, and Deniz Üstebay, "Bayesian graph convolutional neural networks for semi-supervised classification," *arXiv e-prints*, p. arXiv:1811.11103, Nov. 2018.

[6] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur, "Protein interface prediction using graph convolutional networks," in *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. 2017, vol. 30, Curran Associates, Inc.

[7] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," *CoRR*, vol. abs/2010.00130, 2020.

[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)*, 2020, pp. 936–945.

[9] Thomas N. Kipf and Max Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016.

[10] George Karypis and Vipin Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM JOURNAL ON SCIENTIFIC COMPUTING*, vol. 20, no. 1, pp. 359–392, 1998.

[11] 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)*, 2015, pp. 39–50.

[12] 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 *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds. 2015, vol. 28, Curran Associates, Inc.

[13] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun, "Spectral networks and locally connected networks on graphs," 2013.

[14] Thomas N. Kipf and Max Welling, "Semi-supervised classification with graph convolutional networks," 2016.

[15] 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.

[16] 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*, vol. 17, pp. 261–272, 2020.

[17] 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*, vol. 585, no. 7825, pp. 357–362, Sept. 2020.

[18] Shobeir Fakhraei, James Foulds, Madhusudana Shashanka, and Lise Getoor, "Collective spammer detection in evolving multi-relational social networks," in *PubMed Collective Detection*, 08 2015.

[19] 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.