A CPU-GPU HYBRID ALGORITHM FOR EMBEDDING LARGE GRAPHS

by
AMRO ALABSI ALJUNDI

Submitted to the Graduate School of Social Sciences
in partial fulfilment of
the requirements for the degree of Master of Arts

Sabancı University
September 2020

A CPU-GPU HYBRID ALGORITHM FOR EMBEDDING LARGE GRAPHS

Approved by:

Asst. Prof. Kamer Kaya .....
(Thesis Supervisor)

Assoc. Prof. Hüsnü Yenigün .....

Assoc. Prof. Hasan Sözer .....

Date of Approval: September 4, 2020

ABSTRACT

A CPU-GPU HYBRID ALGORITHM FOR EMBEDDING LARGE GRAPHS

AMRO ALABSI ALJUNDI

Computer Science and Engineering M.A. Thesis, 2020

Thesis Supervisor: Asst. Prof. Kamer Kaya

keywords: Graph embedding, HPC, parallel algorithms, machine learning,

Graphs have become ubiquitous in this day and age, and their sizes are only becoming larger and harder to deal with. Graph embedding is the process of transforming graphs into a d-dimensional vector space to carry out machine learning tasks on them. However, time- and memory-wise, it is a very expensive task. Many approaches have been proposed to optimize the process of graph embedding using distributed systems and GPUs, however, state-of-the-art GPU implementations fail to embed graphs unless the total memory of the available GPUs satisfies the cost of embedding. We propose a hybrid CPU-GPU graph embedding algorithm that enables arbitrarily large graphs to be embedded using a single GPU even when the GPU's memory capabilities fall short. The embedding is partitioned into smaller embeddings and the GPU carries out embedding updates on embedding portions that fit the GPU's memory. The system generates samples on the CPU and sends them to the GPU as they become needed without any global synchronization across the system. The system adopts a generalizable DAG execution model to minimize the dependencies between its sub-tasks. We embed a graph with 60 million vertices and 1.8 billion edges in 17 minutes and report a link prediction AUC ROC score of 97.84% making us 67× faster than the state-of-the-art GPU implementation.

ÖZET

BÜYÜK ÇAPLI ÇİZGELERDE ÇİZGE GÖMME İŞLEME İÇİN BİR CPU-GPU HİBRİT ALGORİTMA

AMRO ALABSI ALJUNDI

Bilgisayar Bilimi, Yüksek Lisans Tezi, 2020

Tez Danışmanı: Asst. Prof. Kamer Kaya

Anahtar Kelimeler: çizge gömme, yüksek performanslı bilgi işlem, parallel algoritmalar, makine öğrenmesi

Günümüzde çizgeler birçok alanda karşımıza çıkmaktadır, ve çizgelerin boyutu her geçen gün büyümektedir. Çizge gömme, çizgeler üzerinde makine öğrenmesi işlemleri gerçekleştirmek için çizgeleri çok boyutlu bir vektör uzayında temsil etme işlemidir. Fakat bu işlem zaman ve bellek açısından pahalıdır. Birçok çalışma, dağıtılmış sistemler ve ekran kartı kullanarak, çizge gömme işlemini optimize etmek üzerine algoritmalar öne sürmüştür fakat son teknoloji ürünü algoritmalar ekran kartının belleği gömme maliyetini karşılayamadığı takdirde işlemi gerçekleştirememektedir. Bu çalışmada büyük ölçekli çizgeleri, ekran kartının belleği yeterli olmasa da, sadece bir ekran kartı ile işleyebilen bir hibrit CPU-GPU çizge gömme algoritması önermekteyiz. Bu algoritmada gömme matrisi GPU belleğine sığacak parçalara ayrılarak sıralı bir şekilde işlenmektedir. Sistem, global bir senkronizasyon gerekmeden, örnekleri CPU'da yaratarak, GPU'ya gerektikçe göndermektedir. Ek olarak sistem genelleştirilebilir yönlü ve döngüsüz bir çizge modeli kullanarak yan işlerin bir birine bağımlılığını en aza indirgemektedir. Önerilen algoritma 60 milyon nokta ve 1.8 milyar kenar bulunduran bir çizgeyi 17 dakikada işlerken literatürdeki en hızlı algoritmadan 67 kat hızlı olmakta, ve bağlantı tahmini problemi için %97.84 AUCROC skoru elde etmektedir.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Kamer Kaya for his endless support and guidance. Dr. Kamer taught me how to be a better researcher and a better programmer and I am always going to be grateful for his continuous patience and valuable advice.

My parents who were always there to support me despite being physically far away. My sister for getting me through the harder times, and my siblings for being my inspiration to always keep moving forward.

My friends and loved one s for always being there when the pressure was overwhelming. The students I had the pleasure of working with throughout my teaching assistanship at Sabancı University.

Finally, I would like to thank Sabanci University for supporting the effort of this research and providing ample resources and support when needed.

For my family & loved ones,
my pride and joy

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Today, data is the most valuable asset of any technology. Many technologies we use in our everyday life are driven by the data they collect and the data they produce. Medical applications are driven by the medical literature data, as well as the historic data of patients and medication. Social networks are nothing but a massive collection of user data which is given in a simple interface for people to use. These aggregates of data can be utilized to gain insights into many different fields of research and industry. The study of data that is focused on the relationships within the data and the information such relationships entails utilizes the mathematical concept of a graph. Graphs are a special type of data representation that is used to represent collections of data with a focus on how the elements making up the data are connected. They are extremely useful for modeling data that can be reduced to a set of connected entities. They allow researchers to extract valuable information about the data from its structure. Graphs are used heavily in scientific research and industrial applications. Protein-protein interaction networks, social networks, hyperlink graphs, and co-authorship graphs are all such examples.

Graphs carry unique information, and understanding this information and extracting it from graphs is not an easy process, especially with the scale in which graph sizes are growing. With graphs that model hundreds of millions of vertices and billions of edges, ordinary methods of data extraction are becoming computationally infeasible. That is why the scientific community looked for ways to use machine learning (ML) to study graphs as machine learning approaches are proven to be an invaluable tool for analyzing data aggregates. For graphs, machine learning is used in many tasks, e.g node classification, in which vertices in a graph are classified into labels using a labeled set of nodes, and link prediction, where possible connections between nodes are predicted. These ML approaches face a very important problem; the structures of graphs are highly irregular. Unlike other data formats like text, audio, and images, which have standard structures, representations, memory layouts etc., a graph usually has a structure that does not lend itself to be trivially used with currently existing machine learning models. The procedure of *graph embedding* is

an unsupervised machine learning task that takes arbitrary graphs and produces standard d-dimensional vectors for entities of the graph (vertices, edges, or even full graphs). These vectors capture the connectivity information of the graph and encapsulate it into a vector space that is easily usable by machine learning models, which in turn abstracts away the irregularity inherent in the structure of graphs and expands the arsenal of machine learning models that are usable with graphs. Many successful models have been introduced in recent years with much success (Grover & Leskovec, 2016; Perozzi, Al-Rfou & Skiena, 2014; Tang, Qu, Wang, Zhang, Yan & Mei, 2015; Tsitsulin, Mottin, Karras & Müller, 2018; Zhu, Xu, Tang & Qu, 2019).

As successful as graph embedding procedures are, they are heavily compute-intensive and require hours and sometimes days to embed a single graph. This problem is even more critical when considering the size of contemporary graphs. Graphs with millions or even billions of vertices and edges are known to be used in fields like social networks and e-commerce (Zang, Cui & Faloutsos, 2016). The Facebook graph which captures the interactions in the social network has two billion nodes and more than a trillion edges between these nodes. This computational complexity has lead to research going in the direction of optimizing the process of graph embedding to reduce its runtime overhead. Many such attempts have been made with different approaches to solving the problem. Coarsening methods have been used to compress the graph into smaller, more manageable sizes to produce embeddings faster Liang, Gurukar & Parthasarathy (2018). Distributed systems-based approaches, like (Lerer, Wu, Shen, Lacroix, Wehrstedt, Bose & Peysakhovich, 2019), utilize multiple nodes to lighten the load of embedding. Also, accelerators like GPU were exploited to produce embeddings much faster (Zhu et al., 2019). However, accelerators, despite their incredible computation ability, suffer from a lack of memory capability when it comes to generating embeddings. This means that the hardware requirement increases as the graph under embedding becomes larger.

In this work, we introduce an embedding algorithm with an efficient and specialized scheduling schema that allows arbitrarily large graphs to be embedded using a single GPU - even when the GPU's memory capabilities are not sufficient to embed the complete graph. Our approach partitions the graph into smaller sub-graphs and carries out embedding updates on these sub-graphs. Besides, we utilize the CPU to generate the samples used during the embedding in parallel with GPU computation. Our approach does not have any global synchronization points, leading to non-stop computation on the GPU.

The algorithm proposed in this work is part of GOSH (Akyildiz, Aljundi & Kaya, 2020), a tool that utilizes graph coarsening to produce high-quality graph embed-

dings quickly and using a single GPU only. We briefly discuss GOSH in the thesis. We summarize the contributions of this thesis as follows:

- We propose a highly flexible embedding algorithm that utilizes the capabilities of GPUs while bypassing their harsh memory restrictions.
- Utilize the CPU's computation and memory capabilities to accelerate the process of embedding by generating the positive samples required for embedding locally and sending the samples to the GPU without needing any global synchronization mechanism.
- Propose a generalizable Directed Acyclic Graph (DAG) based execution model for the embedding procedure that enables seamless communication between the GPU and the CPU.
- Produce highly accurate embeddings at a fraction of the time needed by state-of-the-art graph embedding algorithms. For instance, the state-of-the-art GPU-based embedding algorithm, GRAPHVITE, takes 5.36 hours to embed a graph with around 40 million vertices and 600 million edges, and scores 94.3% AUC ROC score on the task of link prediction while using 4 Tesla P100 GPUs. Our approach, on the other hand, after embedding for 7 minutes only, scores 98.44% on the same task with a single TITAN X GPU.

The remaining chapters are organized as follows: Chapter 2 provides some preliminary information and presents an introduction to graph embedding. Chapter 3 discusses the GPU accelerated embedding procedure and introduces the partitioning schema used in the embedding, and Chapter 4 describes the DAG model which carries out the embedding procedure using a single GPU. Chapter 5 includes an analysis of the proposed algorithm and demonstrates the efficacy of GOSH in machine learning tasks. Finally, Chapter 6 summarizes the work and outlines future work to be conducted.

2. BACKGROUND

2.1 Preliminaries and notation

A graph $G = (V, E)$ is a data structure composed of the sets V and E. We define $V = \{v_0, v_1, \cdots, v_{|V|}\}$ as the set of vertices within the graph, and $E = \{(v_i, v_j), (v_l, v_m), \cdots, (v_a, v_b)\}$ as the set of edges in the graph, where $(u, v) \in E$ indicates that there is an edge between vertex u and vertex v. Graphs can be weighted, in which case, the edges have numerical weights. In addition, graphs can be directed or undirected. Edges in directed graphs are oriented, i.e $(u, v) \neq (v, u)$ and $(u, v) \in E \not\Rightarrow (v, u) \in E$. Edges in undirected graphs, however, do not have any orientation. In this work, we will assume all graphs are unweighted and undirected.

An embedding matrix $\mathbf{M}$ of a graph G is a d-dimensional matrix with d columns and $|V|$ rows, where vectors in the matrix correspond to embeddings of vertices in G, i.e $\mathbf{M}[u]$ is the embedding vector of vertex $u \in V$. The notation used in this work is shown in Table 2.1.

2.2 Graph Embedding

A graph is an essential data representation that is ubiquitous in contemporary research and industrial applications. Graphs capture the structure of data elegantly and provide insights that are hard to grasp otherwise. However, their highly irregular structure prevents their richness of representation from being exploited by contemporary ML models; it is highly desirable to open up the application of any machine

Table 2.1 Notation used throughout the thesis.

Symbol	Definition
$G = (V, E)$	A graph with vertex set V and edge set E.
$\mathcal{V}$	The set of sub-graphs of a partitioned graph.
V_i	The ith sub-graph of of the partitioned graph.
K	# of parts in $\mathcal{V}$.
d	# features per vertex, i.e., dimension of the embedding.
s	# negative samples per vertex.
σ	Sigmoid function.
$\mathbf{U} \odot \mathbf{V}$	The dot product operation between the vectors $\mathbf{U}$ and $\mathbf{V}$.
sim_m	Similarity metric modeled as a distribution.
e	Total number of epochs that will be performed
lr	Learning rate.
$\mathbf{M}$	The embedding matrix of the entire graph.
P_{GPU}	# embedding parts to be placed on the GPU.
$\mathcal{M}$	The set of embedding sub-matrices of the partitioned graph.
$\mathbf{M}_i$	Embedding sub-matrix of sub-graph V_i.
$\mathbf{M}_i^d$	Sub-matrix bin i on the GPU.
B	# Positive samples per vertex in a single sample pool.
$\mathbf{K}_{i,j}$	Embedding kernel of the the sub-graphs i, j.
S_{GPU}	# Sample pools to be placed on the GPU.
z	# of sample pool sets on the CPU.
$\mathbf{S}_{i,j,k}^h$	The sample pool in sample pool set k on the CPU containing positive samples of the sub-graph pair i and j.
$\mathbf{S}_i^d$	Sample pool bin i on the GPU.
$\mathcal{D}_i$	The directed acyclic graph (DAG) i consisting of task nodes.
$\mathcal{Q}_i$	The execution queue of $\mathcal{D}_i$.
τi	# of threads executing tasks from $\mathcal{Q}_i$.
$ST_{i,j,k}$	A sampling task node that samples into sample pool $\mathbf{S}_{i,j,k}^h$.
$MST_{i,j}^k$	A sub-matrix swap task that switches out $\mathbf{M}_i$ and switches in $\mathbf{M}_j$ from $\mathbf{M}_k^d$.
$SCT_{i,j,k}$	A sample pool copy task that copies $\mathbf{S}_{i,j,k}^h$ to the GPU.
$KT_{i,j}$	A task that executes $\mathbf{K}_{i,j}$ on the GPU.
$\mathcal{X}$	The execution order set.
$S_{i,j,k}^h$	The shared variable of $\mathbf{S}_{i,j,k}^h$.
S_i^d	the shared variable of $\mathbf{S}_i^d$.
$K_{i,j}$	The shared variable of $\mathbf{K}_{i,j}$.
C	# of concurrent samplers on the GPU.
T_s	# of sampling threads.

learning model to graphs. Graph embedding techniques transform the connectivity information of a graph into a d-dimensional vector space that is easily usable with many ML models. It provides a data format that is extremely efficient in many ML tasks including link prediction (Liben-Nowell & Kleinberg, 2003), node classification (Perozzi et al., 2014), and anomaly detection (Hu, Aggarwal, Ma & Huai, 2016). There have been many different approaches to graph embedding, and different taxonomies classify them into a variety of sets of classes (Cai, Zheng & Chang, 2018; Goyal & Ferrara, 2017; Wang, Mao, Wang & Guo, 2017). Different approaches target different elements of graphs. The most basic graph embedding flavor is an embedding of the vertices of the graph, but embeddings of other elements of the graph can be learned as well. This includes edges (Gao, Fu, Ouyang, Tsutsui, Liu & Ding, 2018), sub-graphs (Adhikari, Zhang, Ramakrishnan & Prakash, 2018), and even entire graphs (Narayanan, Chandramohan, Venkatesan, Chen, Liu & Jaiswal, 2017). Besides, embedding different types of graphs has been an important field of research. This is especially true for knowledge graphs due to their flexibility and richness with information (Lerer et al., 2019; Xiao, Huang, Hao & Zhu, 2015).

The earliest attempts at graph embedding are matrix factorization methods, which take a relationship matrix (such as an adjacency matrix) and factorize it to produce a d-dimensional matrix. Local Linear Embedding (Roweis & Saul, 2000), Laplacian Eigenmaps Belkin & Niyogi (2002), and the more recent HOPE (Ou, Cui, Pei, Zhang & Zhu, 2016) are all such methods. Matrix factorization methods, despite their impressive results, suffer from a lack of scalability. The matrices these approaches factorize scale with the square of the number of vertices in the graph. And for graphs with millions or billions of vertices, the storage and time requirements become astronomical.

More recently, deep learning-based graph embedding methods have been receiving a great deal of attention due to the non-linearity of their models and the recent advancements in the field of deep learning. Structural Deep Network Embedding (SDNE) and Variational Graph Auto-Encoders both use auto-encoders to generate high-quality embeddings (Kipf & Welling, 2016; Wang, Cui & Zhu, 2016). Graph Convolutional Networks (GCN) (Kipf & Welling, 2017), a class of neural networks that interface a graph directly, have also been successful at producing high-quality embeddings.

Sampling-based graph embedding approaches are a class of deep learning-based algorithms which uses sampling to optimize an objective function. The approach proposed in this work belongs to this class of algorithms. The following sections will review some of the most prominent sampling-based embedding algorithms.

2.2.1 DeepWalk

DeepWalk is the first of a series of embedding algorithms to adopt random walks in the process of graph embedding. It builds upon advancements in the area of using neural networks for building latent representations of natural languages (Collobert & Weston, 2008). DeepWalk builds in parallel with natural language modeling techniques and proves that many of the existing natural language models can be used to model community networks.

This algorithm carries out embeddings by generating random walks, then updating the embeddings of the vertices in a walk such that vertices within a certain distance from one another will have a bigger co-occurrence probability. The idea is to treat vertices as words in a language model, and random walks as sentences.

DeepWalk's embedding procedure begins with choosing a random root vertex r uniformly from the graph and carrying out a random walk starting at r for a certain walk length. Once the walk is generated and given a certain window size w, DeepWalk will iterate through every node in the walk and carry out updates to the embeddings of these vertices that will maximize the co-occurrence probability between every vertex and the w vertices before it in the walk, as well as the w vertices after it in the walk. Maximizing the co-occurrence probability corresponds to updates to the embedding matrix, which are done through SkipGram (Mikolov, Chen, Corrado & Dean, 2013), a model designed originally to maximize the co-occurrence of words appearing in a sentence. SkipGram is optimized using Hierarchical Softmax (Mnih & Hinton, 2008) to reduce the complexity of updating the embeddings. This process is repeated many times until the embedding is complete.

DeepWalk is evaluated on the task of node classification against several baseline approaches and it produces very high classification F1-scores, especially when the amount of labeled data is scarce (less than 60%). This goes to prove that the embeddings DeepWalk generates capture information about the graph independent of any labels on the graph vertices.

2.2.2 LINE

LINE is a graph embedding algorithm designed to scale up to very large graphs that previous approaches had struggled with. It uses a novel embedding approach that optimizes the embeddings not only based on the proximity between nearby nodes

but also the overlap between vertices' neighborhoods. This method proves to be very effective and delivers highly accurate results.

The embedding of LINE runs two sets of embeddings for a single graph. The first one preserves the first-order proximity between vertices, which LINE defines as the pairwise proximity (weight of the edge between vertices). The second embedding preserves what LINE defines as the second-order proximity between vertices. The second-order proximity between two vertices u and v is described as the similarity of their neighborhood structure, which is determined by the shared neighbors of u and v.

LINE trains its embeddings using the negative sampling technique (Mikolov, Sutskever, Chen, Corrado & Dean, 2013). In each sampling iteration, one (existing) edge is sampled from the graph and another n_s (probably non-existing) edges are chosen from some noise distribution as negative samples. These edges are used to optimize an objective function that distinguishes the positive and negative samples. The sampling process is further optimized for weighted graphs by creating a sampling table in which edges are unrolled based on their weight, i.e., taking an edge (u, v) with weight w and adding to the sample pool w instances of that edge. Its optimization procedure is carried out using *asynchronous gradient descent* (ASGD) by batching sampled edges and updating the embeddings accordingly. The embedding is carried out by training embeddings that preserve first-order proximity, another set that preserves second-order proximity, and concatenating the two sets to produce the final embedding matrix.

The model provides two different parameter sets. The first is the embedding matrix $\mathbf{M}$, and the second is a context embedding matrix $\mathbf{C}$. The context embedding matrix has the same dimensions as $\mathbf{M}$ but it is used for the optimization of second-order proximity. During the second stage of embedding, when embeddings are optimized to reflect the second-order proximity, sampling the edge (u, v) would result in updating $\mathbf{M}[u]$ and $\mathbf{C}[v]$. This way, two vertices who share a neighbor v will update their embeddings with $\mathbf{C}[v]$.

2.2.3 node2vec

node2vec takes the concept of random walks for generating embeddings of vertices and generalizes it to produce embeddings that capture neighborhood information as well as the structural equivalence between nodes. node2vec observes that classic

random walks, like those used in DeepWalk, are very linear and cannot capture the richness of information in a network, which is why it proposes a randomized search strategy that explores different neighborhoods of the same root vertex.

node2vec claims that different random walk strategies can capture different aspects of a network. Carrying out a random walk that moves away from a root in DFS fashion will create embeddings that reflects the neighborhood of the root node at the macro level. On the other hand, a BFS-like walk strategy captures the structural information of a node as it entails exploring multiple nearby neighborhoods.

node2vec guides the random walks in such a way that the degree to which the nearby neighborhood of a vertex is explored, and the distance traveled away from the vertex, can be controlled with the two tuning parameters p and q. p controls the likelihood of hopping backward in a random walk which biases it to stay close to its community, while q controls the likelihood of moving away from the community of the node, which makes it lean more toward a DFS strategy.

The hyper-parameters p and q are chosen at the beginning of the embedding, and the embedding proceeds by aggregating random walks that are biased with these parameters. Similar to DeepWalk, the vertices within a certain window (or context) in a walk are updated through the process of negative sampling and using stochastic gradient ascent.

The model was evaluated with the tasks of node classification and link prediction its random walk strategy was shown to be superior over both the classical walk strategy of DeepWalk, as well as the first-order and second-order proximity optimization method of LINE. However, node2vec suffers an important drawback. To make the most out of node2vec's random walk strategy, one must find the best p and q values for the downstream machine learning task at hand, which needs to be done through searching the space of p and q independently for each graph. This adds a quadratic element to the time complexity of embedding. Even though it is shown that understanding the graph being embedded and its structure can guide the choice of these tuning parameters, searching the input space can still be necessary.

2.2.4 VERSE

VERSE is a highly versatile graph embedding approach, meant to generalize the process of graph embedding and make it faster and more scalable. VERSE follows a very straightforward approach to embedding; model the embedding matrix

as a similarity distribution between vertices in the graph, model any other empirical similarity measure as another distribution, and minimize the Kullback-Leibler divergence between the two distributions through the process of Noise Contrastive Estimation (NCE), a variant on negative sampling (Mikolov et al., 2013). We discuss the similarity distribution approximation process in more detail in Section 3 as we use it for our embedding algorithm.

In their paper, Tsitsulin et al. (2018) instantiate VERSE with three similarity measures, namely Personalized Page-Rank (Page, Brin, Motwani & Winograd, 1999), SimRank (Jeh & Widom, 2002), and adjacency similarity, which is similar to the first-order proximity that LINE uses. Each one of these similarity measures was reduced to distributions that can be used in training, and the effectiveness of these approaches was analyzed with a variety of tasks. Similarity distributions are not calculated beforehand, as that would incur an $O(|V^2|)$ space complexity overhead to the embedding. Instead, samples were generated in-time during the embedding procedure. A version that calculates the distributions before the embedding starts was implemented and it was shown that it produces slightly better results.

VERSE was evaluated against various embedding algorithms based on tasks like link prediction and node classification and was shown to perform very well against state-of-the-art methods.

2.2.5 Pytorch Big Graphs

Pytorch Big Graphs is an embedding approach for knowledge graphs, but it can be generalized for other graphs as well. It provides a partitioning schema that allows distributing the work of embedding over multiple machines even when these machines' memory capabilities do not allow for a full embedding procedure. Also, work can be spread across multiple machines in parallel to gain speedups from the parallelism.

A partitioning schema is introduced in PBG in which vertices in the graph are partitioned into P parts, and the edges of the graph are partitioned into P^2 bins such that every part pair has a corresponding edge bin. The embedding matrix is sharded across the machines and a single, centralized lock server controls access to the embedding matrix. The lock server controls which machines get access to which partitions and moves the embedding procedure forward.

PBG optimizes the embeddings using a margin-based ranking objective between

edges in the graph (positive edges) and edges that have been corrupted (negative edges). It uses mini-batch SGD to lighten the load on processing units when working with larger graphs. Negative samples are generated partially uniformly from a random distribution, and partially by following the data distribution.

PBG produces highly accurate embeddings in tasks such as link prediction and node classification. Besides, its parallelization schema proved to provide a performance speedup as the number of contributing machines increased.

2.2.6 Graphvite

Graphvite is a multi-GPU graph embedding approach that distributes the workload of embedding larger graphs onto one or more GPUs while utilizing the CPU to generate samples and provide them to the GPU on the fly as they become needed.

Graphvite uses the embedding procedure of LINE's second-order proximity; it has a context embedding matrix in addition to the original embedding matrix with the same dimensions. It bypasses the immense memory overhead of these two matrices by partitioning the graph into multiple subgraphs and partitioning the embedding and context matrices to match these of the sub-graphs. The embedding takes place by switching embedding and context embedding sub-matrices in and out of the GPU along with CPU-generated positive samples and letting the GPU carry out the embedding updates.

Sampling in Graphvite is done on the CPU. Every pair of parts in the partitioned graph has a sample pool in the host, which the samplers will populate with samples. These samples are generated by conducting DeepWalk or node2vec random walks, then iterating through the generated walks and placing samples from these walks into the appropriate sample pools based on the matrix partitioning. Two sample pool sets are allocated on the CPU such that when the GPU is currently fetching from a pool, the CPU can continue sampling on the other. Also, samples are copied to the GPU in small batches such that sample pools do not take up extra space on the GPU and that the GPU doesn't wait idly for samples. Negative samples are generated on the GPU itself; when updating the vertices of a part A with samples from part B, negative samples are uniformly drawn from part B. This way, no data transfers between the CPU and the GPU need to happen to fetch negative samples.

The embedding procedure is split into episodes. At the beginning of an episode, embedding and context sub-matrices are sent to different GPUs, such that no sub-

matrix is in two GPUs at the same time. The CPU proceeds to send batches of positive samples to the GPUs and parallel embedding starts. Once an episode is over, the CPU will synchronize with the GPUs, fetch back the updated embedding and context sub-matrices, and start a new episode.

Graphvite has been evaluated against many state-of-the-art embedding algorithms in both link prediction and node classification and its shown substantial speedups over multi-core CPU implementations, as well as highly accurate embeddings. However, it suffers from an important hardware constraint - if using a single GPU, it cannot embed graphs which cannot physically fit inside the GPU (Zhu et al., 2019). This means that multiple GPUs are required to embed large-scale graphs.

2.3 General Purpose GPU Computing with CUDA

Graphics Processing Units (GPU) are high-performance hardware chips designed to accelerate the rendering of graphical elements. GPUs have a high number of specialized computation cores that provide them with extremely high throughput. For these reasons, scientists and engineers saw the potential of these devices, especially in terms of the parallelism they allow. General Purpose GPU (GPGPU) computing is a term used to describe using GPUs for scientific research or industrial applications that are beyond the specialty of GPUs, i.e, rendering graphics.

Many GPGPU programming interfaces are currently being used to develop GPGPU programs. The `CUDA` language is Nvidia's proprietary GPU programming interface that's designed to run on Nvidia's GPUs. It is a programming language with `C` like syntax supported on all the major operating systems. The programming paradigm of `CUDA` is different from CPU based programming languages. With `CUDA`, the programmer is working with thousands of threads running in parallel which provides substantial speedups when used appropriately.

In this work, we will use the `CUDA` jargon to describe the inner workings of the algorithm. In the remainder of this section, we give a few pointers about the GPU operations:

- The CPU that runs the `CUDA` program is called the **host**. For the remainder of this work, we will use the terms host and CPU interchangeably.
- A `CUDA` **block** is a collection of **threads** which share fast on-chip memory

called the **shared memory**. Every thread has its registers and local private memory.

- Threads in a `CUDA` program are grouped into groups of 32 lock-stepped **warps**. The threads in the same warp are controlled by the same controller hence, they always run the same instruction.

- **Global memory** is the memory region on the GPU that the host can write to and read from. All threads running on the GPU have access to data on global memory.

- **Kernels** are the computation execution units that the GPU carries out. The programmer writes these kernels as functions and dispatches them to the GPU in the host code. Also, the programmer must specify the number of blocks the kernel should use and the number of threads in each block. However, how these blocks and threads are distributed on the physical cores is decided by the GPU.

- Physical cores on the GPU are grouped into **streaming multiprocessors**, with each SM having a limited amount of shared memory and register space. An SM is bound by its memory capability; if its shared memory or register space is full while some of its threads are idle, and there are queued blocks to execute, it will not be able to execute them and these threads will remain idle. The rate of active warps on an SM to the maximum number of possible active warps supported by the SM is called the **occupancy.**

- All `CUDA` dispatches, including reads and writes to global memory, as well as kernels, are sent to a specific `cudaStream`. Each stream is a queue of jobs that the GPU will execute. Multiple streams can be active on the GPU concurrently. The order of dispatched jobs on a single stream is deterministic, but the relative order of jobs on different streams is not.

- `cudaEvents` are special constructs used to synchronize different `CUDA` jobs dispatched on different streams, as well as synchronize GPU work with the host's code.

- `cudaCallbacks` are special functions that can be enqueued into `cudaStreams`. However, these functions run on the *host* and they can be used for synchronization purposes.

3. EMBEDDING LARGE GRAPHS ON A SINGLE GPU

Graph embedding is an extremely costly procedure, both in terms of time and resources. That is why we designed a novel framework and algorithm to exploit the acceleration capabilities of GPUs. In this chapter, we first present the formal definition of the embedding optimization we follow and its sequential implementation. Afterward, we introduce our GPU parallelized embedding algorithm. Finally, we introduce the partitioning schema used to perform our global-synchronization free large-scale graph embedding. As discussed in Chapter 2, graph embedding can take many different forms, but it is essentially the process of optimizing a certain objective function which uses the embeddings as its parameters. For our algorithm, we chose the method presented in VERSE (Tsitsulin et al., 2018) as we recognize its high utility and, as the name suggests, versatility. The way VERSE approaches the process of embedding is by optimizing the embeddings in such a way that they resemble one of many graph similarity measures from the literature (like Personalized PageRank (Page et al., 1999) or SimRank (Jeh & Widom, 2002), for example).

VERSE defines two random distributions for every vertex v in the graph, namely sim_Q^v and sim_E^v, both of which will give a value for the similarity between v and any other vertex u in the graph. sim_Q^v is calculated from the structural information of the graph according to the definition of an empirical similarity measure Q_v, such that a higher probability of picking some vertex u over another vertex k in sim_Q^v implies that v is more similar to u than k. In other words, $sim_Q^v(u) > sim_Q^v(k) \iff Q_v(u) > Q_v(k)$.

On the other hand, sim_E^v is a distribution that is calculated from the values of the embedding matrix $\mathbf{M}$ itself. To elaborate, to calculate the value of $sim_E^v(u)$, the dot product of the vectors of v and u is calculated and softmax normalized with the dot products of v and every other vertex in the graph such that they sum up to 1. That is

$$sim_E^v(u) = \frac{exp(\mathbf{M}[v] \odot \mathbf{M}[u])}{\sum_{i=0}^{|V|-1} exp(\mathbf{M}[v] \odot \mathbf{M}[i])} \tag{3.1}$$

and

$$\sum_{\forall u \in V} sim_E^v(u) = 1 \tag{3.2}$$

where $\odot$ is the dot-product of two embedding vectors.

The objective function then becomes to simply minimize the Kullback-Leibler (KL) divergence between the two similarities sim_Q and sim_E by minimizing the following cross entropy loss:

$$\mathcal{L} = -\sum_{\forall u \in V} \left(\sum_{\forall v \in V} sim_Q^u(v) \right) \cdot log \left(\sum_{\forall v \in V} sim_E^u(v) \right) \tag{3.3}$$

Minimizing the loss defined by (3.3) is carried out using Noise Contrastive Estimation (NCE), which is a variant of negative sampling (Mikolov et al., 2013). NCE optimizes an objective function by training a binary classifier to distinguish between the true observations and other observations from a *noise* distribution. In our case, we train the classifier to distinguish the (edge) samples coming from the empirical similarity sim_Q and the samples coming from a random distribution N, with $\mathbf{M}$ being the parameter space of this binary classifier. The training is done through asynchroneous stochastic gradient descent (ASGD).

3.1 A Sequential Embedding Algorithm

Effectively, the optimization is carried out through repetitively choosing a vertex v from the original graph, sampling a positive vertex and one or more negative vertices for v, and updating the embeddings of v and the sampled vertices. Algorithm 1 shows the optimization process in more details. Given a graph G, an initial embedding matrix $\mathbf{M}$ (possibly initiated with random values), a negative sample count s, a learning rate lr, and an epoch count e, the algorithm will return the final, trained embedding matrix $\mathbf{M}$. For e rotations (i.e., epochs), $\forall v \in V$, a single vertex u is chosen from sim_Q^v as a positive sample (Line 4) and s vertices $(u_1, u_2, \cdots, u_s)$ are chosen from a random (uniform) distribution N as negative samples (Line 8). The embeddings of v and u are updated such that the distance in the embedding space between $\mathbf{M}[v]$ and $\mathbf{M}[u]$ is shortened (Line 4), while the embeddings of v and $u_1, u_2, \cdots, u_s$ are updated such that the distances between $\mathbf{M}[v]$ and $\mathbf{M}[u_1], \mathbf{M}[u_2], \cdots, \mathbf{M}[u_s]$ are

made longer (Line 9). Algorithm 2 shows the process of making a single embedding update. $\mathbf{M}$ is the embedding matrix, b is a binary variable indicating whether this sample is positive or negative, v is the source vertex and *sample* can either be a positive sample (sampled from sim_Q^v) in which case $b = 1$, or it can be a negative sample (sampled from N), in which case $b = 0$. In addition, σ is the sigmoid function, and $\odot$ is the dot-product of two embedding vectors.

In this work, we use the adjacency matrix as the empirical similarity measure sim_Q, i.e the call to GetPositiveSample(v, G) will return one of v's neighboring vertices. Adjacency similarity can be switched for another similarity measure without changing the algorithm (Tsitsulin et al., 2018).

Algorithm 1: FullGraphEmbedding

Input: G: input graph
$\mathbf{M}$: embedding matrix
s: number of negative samples
lr: learning rate
e: training epochs
Output: $\mathbf{M}$

```
1 for j = 1 to e do
2     lr' ← lr × max(1 − j/e, 10^−4);
3     for ∀v ∈ V do
4         u ← GetPositiveSample(v, G);
5         if u ≠ −1 then
6             SingleVertexUpdate(M[v], M[u], 1, lr' );
7             for k = 1 to s do
8                 u_k ← GetNegativeSample(G);
9                 SingleVertexUpdate(M[v], M[u_k], 0, lr');
```

Algorithm 2: SingleVertexUpdate

Input: $\mathbf{M}$: embedding matrix
b: sample is positive (1/0)
v: source vertex ID
sample: sample vertex ID
lr: learning rate
Output: $\mathbf{M}[v]$, $\mathbf{M}[sample]$

```
1 score ← b − σ(M[v] ⊙ M[sample]) × lr ;
2 M[v] ← M[v] + M[sample] · score;
3 M[sample] ← M[sample] + M[v] · score;
```

3.2 Parallelizing Graph Embedding with GPUs

The proposed algorithm parallelizes the graph embedding procedure described in Section 3.1 using GPUs. This task is not at all trivial as one must pay special attention to several GPU specific design restrictions, namely the Single Instruction Multiple Threads (SIMT) paradigm that GPUs follow, the limited size of shared memory on a GPU's Streaming Multiprocessor, and the global memory data access pattern.

Single instruction multiple threads (SIMT): A single GPU contains upwards of tens of thousands of microprocessors, all running concurrently. However, the (SIMT) execution model on a GPU is very different from that on the CPU. In this model, threads are organized into groups of 32 called "warps". All threads within a warp are thread-locked, meaning that they all execute the same instruction at all times. However, each thread has separate registers and address space. In this paradigm, every thread can store its independent data, but it must abide by the instruction flow of the other threads in the warp. To maximize the utilization of threads in this paradigm, we must make sure that all threads in a warp follow the same control flow (without divergence at any control statements).

Global memory access pattern: All threads on the GPU can access a memory region called the *global/device memory*. As a form of optimization, GPUs are designed to minimize accesses from SMs to global memory by grouping global memory accesses of threads in a warp into transactions such that a single transaction can service many threads simultaneously. This grouping, or coalescing, can only be utilized when threads access consecutive memory locations. If threads on a warp try to access uncoalesced memory locations then the number of memory transactions will increase, and hence the average latency.

Shared memory usage: The thread warps in the GPU are executed on *Streaming Multiprocessors* (SM), the physical devices the make up the GPU. Every SM provides its warps with a portion of fast-access memory called the shared memory. This portion of memory provides very fast access for the threads within a warp, but its capacity is much smaller than that of the global memory.

3.2.1 Implementation Details

To achieve the best utilization of the GPU, we parallelize Line 3 of Algorithm 1 on the warp level as shown in Figure 3.1; each warp in the GPU will carry out the positive update and negative updates of a source vertex v, with its 32 threads

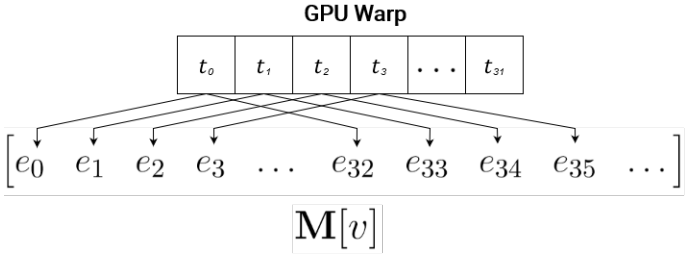

Figure 3.1 A GPU warp containing the threads $t_0, t_1, \cdots, t_{31}$ split the workload of processing a single vertex through having every thread t_i read and update specific elements within the vector $\mathbf{M}[v]$. More precisely, thread t_i handles embedding elements $e_i, e_{i+32}, e_{i+64}, \cdots$ for all $0 \geq i > 32$.

splitting the workload. With this approach, all 32 threads in a warp will operate on the same embedding vectors at all times. In Algorithm 2, instead of a single thread operating on every value of embedding vectors $\mathbf{M}[v]$ and $\mathbf{M}[sample]$, thread t_i in the warp will operate on the values $\mathbf{M}[.][i + c \times 32]$ where c is a positive integer such that $i + c \times 32$ is smaller than the length of vectors. This approach carries three very important benefits:

- Since all threads in a warp are operating on the same vertex and processing the same samples, it means that there will never be a case in which some threads are idle due to different control paths. In other words, there cannot be a case in which a thread will evaluate Line 5 of Algorithm 1 differently from the other threads in its warp.

- This method coalesces access to global memory to retrieve embedding values - accesses happen in such a way that two adjacent threads in the same warp access consecutive elements. This allows the GPU to make fewer memory transactions to global memory. In fact, using single-precision floats, a warp will make a single global memory transaction per instruction.

- Processing a single update using 32 threads effectively means that there are $32\times$ less concurrent updates which would greatly reduce the race conditions created by GPU parallelization.

To reduce the overall communication between SMs and global memory, we first carry the embeddings of source vertices to shared memory. This way, instead of threads having to read and write the embeddings of the source vertex v once in Line 6 and s times in Line 9, the embeddings of v will only be read from global memory once before the positive update, and written once after the negative updates. All intermediate reads and writes will happen on the shared memory. However, the reads and writes of the embeddings of positive and negative samples are done directly on global memory.

3.3 Large-Graph Embedding on a Single GPU

Although it is working extremely well on medium-scale graphs, the algorithm described in the previous section cannot embed large-scale graphs especially into spaces with large dimensions since it is limited by the memory of the GPU being used. In this work, we propose an embedding algorithm that can bypass the memory limitations of GPUs and embeds arbitrarily large graphs using a single GPU without the need for global synchronization. The algorithm consists of two concurrent processes on the host and the device:

- the host continuously samples edges from the input graph and sends them to the GPU as they are needed, and,
- the device performs embedding updates on sub-parts of the graph using the samples brought over from the host.

Here we explore the challenges our algorithm attempts to solve and the solutions.

3.3.1 Memory bottlenecks

The algorithm described in Sections 3.1 and 3.2 incur a heavy memory cost. Without any modifications to Algorithm 1, this memory cost puts a hard limit on the size of the graphs that can it can embed. There are two main memory costs on the GPU for embedding graphs: storing the embedding matrix $\mathbf{M}$, and storing the graph data itself. For the former, the algorithm requires that the entirety of the embedding matrix $\mathbf{M}$ be present on the GPU during the embedding process. This is because the accesses to the embeddings are completely random, specifically, the access to the embeddings of the positively and negatively sampled vertices. The exact cost of storing the embedding matrix, given that the embedding values are stored as single-precision floats:

$$d \times |V| \times 4 \tag{3.4}$$

For a graph with 100 million vertices, and an embedding dimensionality of 128, the embedding matrix would be 51.2 GB in size. That is twice the maximum available memory size of contemporary scientific GPUs.

There are many different standard methods for storing graph data structures, and in our algorithm, we use the Compressed Sparse Row (CSR) format for storing graphs. In CSR, an array, adj holds the neighbors of every vertex in the graph consecutively. It is a list of all the neighbors of vertex 0, followed by all the neighbors of vertex 1, and so on. Another array, $xadj$, holds the starting indices of each vertex's neighbors in adj, with the last index being the number of edges in the graph. In other words, the neighbors of vertex i are stored in the array adj from $adj[xadj[i]]$ until $adj[xadj[i+1]]$. This representation is very compact and assuming the vertex IDs are stored as 4-byte integers, the total cost of storing the graph can be calculated as:

$$(|V| + |E|) \times 4 \tag{3.5}$$

Storing the graph on the GPU is highly desirable for our algorithm. That is because it would allow us to generate the samples on the GPU itself without needing to communicate with the host. However, as graphs become larger, storing the graph on the GPU becomes a more significant cost. For example, given the same graph above with 100 million vertices, and assuming that its average degree is 3, then the size of the CSR is 1.6 GB.

3.3.2 Large-graph embedding

The memory costs shown in Section 3.3.1 make the process of accelerating graph embedding hardware dependent. Even though there have been successful attempts at using GPUs to accelerate embedding large graphs (Zhu et al., 2019), these solutions required that the number of GPUs increase with the graph size. This requirement puts a serious barrier between researchers who wish to explore the field of graph embedding while not having sufficient resources. That is why this algorithm is designed in such a way that graphs whose memory cost exceeds a single GPU's capability can be embedded using said GPU. To do so, we partition the embedding matrix into sub-matrices that fit the GPU and rotate them in and out of the GPU. Besides, we generate samples on the host and send them to the GPU. The GPU utilizes the embedding sub-matrices and the samples in its memory and executes the necessary embedding operations.

The embedding process can be reduced to a series of updates to vectors within the embedding matrix $\mathbf{M}$, in which a single update will read and write from not

more than *two* embedding vectors. As such, for the GPU to execute the embedding consisting of $U = \{(u_0, v_0), (u_1, v_1), \cdots, (u_i, v_i)\}$ updates where $u_j, v_j \in V$, it must have access to the embedding vectors of every pair of vertices involved in these updates. Since we cannot store the entire embedding matrix on the GPU, we need to either (**a**) store parts of the embedding matrix on the GPU; enough to carry out a subset of the updates, or (**b**) store the embeddings on the host, and access them on the GPU using the Unified Virtual Memory (UVM) interface, in which the CUDA runtime decides when to fetch embeddings to the GPU. Using UVM hides the process of moving embeddings from and to the GPU, but it takes away from our ability to control the movement of embeddings. This would lead to a large number of unnecessary data movements - especially given the random access pattern of the embedding vectors during the embedding process. On the other hand, we can resort to the former option by partitioning the embedding matrix into K sub-matrices that are small enough to be stored on the GPU. We would only require two sub-matrices to be stored on the GPU at the same time since, for a single update, we would need to access *at most* two sub-matrices. This way, an update involving any two vertices u and v where $u, v \in V$ can be executed on the GPU as long as the sub-matrices which include u and v reside on the GPU.

More formally, we partition V into K disjoint subsets of vertices $\mathcal{V} = \{V_0, V_1, \ldots, V_{K-1}\}$. Let $\mathcal{M} = \{\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_{K-1}\}$ be the sub-matrices of $\mathbf{M}$ corresponding to the vertex sets in $\mathcal{V}$ with $2 \times \texttt{sizeof}(\mathbf{M}_i) <$ GPU memory. With this partitioning, embedding G becomes the process of moving the sub-matrices in $\mathcal{M}$ to the GPU, carrying out the updates which involve vertices within these sub-matrices, and switching them out for the next sub-matrices, and so on.

3.3.3 Embedding rounds

Following the partitioning idea, the embedding procedure is executed in rounds. Each round consists of:

- a series of sub-matrix copy operations from/to the device, and
- executions of embedding kernels on the GPU that carry out the updates on the sub-matrices residing on the GPU.

The pattern in which the embedding sub-matrices are switched in and out of the GPU must maintain the condition that within a single round, for every possible pair of embedding sub-matrices, there must exist a time-point at which the corresponding

sub-matrices concurrently reside on the GPU. In other words, during an embedding round, there will be a time instance when the embedding sub-matrices $(\mathbf{M}_i, \mathbf{M}_j)$ will be on the GPU at the same time $\forall i, j : 0 \leq j \leq i < K$. To process sub-matrices on the GPU, we move the sub-matrices from the host to $P_{GPU} < K$ pre-allocated sub-matrix bins on the GPU $\mathbf{M}^d_0, \mathbf{M}^d_1, \cdots, \mathbf{M}^d_{P_{GPU}-1}$.

During a round, an embedding kernel $\mathbf{K}_{i,j}$ is executed for every embedding sub-matrix pair $(\mathbf{M}_i, \mathbf{M}_j)$ where $0 \leq j \leq i < K$. An embedding kernel $\mathbf{K}_{i,j}$ executes Algorithm 3 and is explored in more detail at the end of this section. Every embedding kernel will carry out up to B positive samples (and s negative samples for every positive sample) for every vertex in $\mathbf{M}_i$. For each one of these updates, the (positively and negatively) sampled vertices will be from $\mathbf{M}_j$. The same will happen in the opposite direction. B is the batch size of a single round. This schema of execution means that, in each rotation, we run a total of *at most* $B \times K$ positive and $B \times K \times s$ negative samples for every vertex. It should be noted that for a vertex $v \in V_i$, *no* updates will be executed for v during zero or more embedding kernels $\mathbf{K}_{i,j}$ or $\mathbf{K}_{j,i}$. This happens when there are no vertices $u \in V_j$ that are positive samples for v. We add an atomic global counter for the samples to ensure that no additional sampling is performed beyond $|V| \times e$. Otherwise, we would execute p samples where $|V| \times e \leq p \leq |V| \times e \times K$.

Execution order: During the execution of the aforementioned embedding kernels, we follow an order resembling the *inside-out order* proposed in (Lerer et al., 2019) as it showed the best results in terms of embedding quality. Formally, we define the execution order to be the order of part pair in the set $\mathcal{X} = \{(V_{a_0}, V_{b_0}), (V_{a_1}, V_{b_1}), \cdots, (V_{a_\ell}, V_{b_\ell})\}$ where $\ell = \frac{K(K+1)}{2}$ and

$$(V_{a_j}, V_{b_j}) = \begin{cases} (V_0, V_0) & j = 0 \\ (V_{a_{j-1}}, V_{b_{j-1}+1}) & j > 0 \text{ and } a_{j-1} > b_{j-1} \\ (V_{a_{j-1}+1}, V_0) & a_{j-1} = b_{j-1} \end{cases}$$

Samples: As mentioned at the beginning of Section 3.3.2, we avoid the memory cost of storing the graph on the GPU by generating the positive samples needed for embedding on the host, and sending these samples to the GPU as they become needed. We generate the samples needed for embedding on the host and store them in the $z \geq 1$ sample pool sets $\mathcal{S}$ where every embedding kernel $\mathbf{K}_{i,j}$ has z designated sample pools $\{\mathbf{S}^h_{i,j,0}, \mathbf{S}^h_{i,j,1}, \cdots, \mathbf{S}^h_{i,j,z-1}\}$ which contain positive samples from V_i to V_j, and, if $i \neq j$, samples from V_j to V_i. On the GPU, we allocate S_{GPU} sample pool bins $\mathbf{S}^d_0, \mathbf{S}^d_1, \cdots, \mathbf{S}^d_{S_{GPU}-1}$ and move sample pools from the host to these bins on the

GPU according to the kernels about to be executed. As for negative samples, we generate them on the GPU itself. For every positive sample executed for a vertex $v \in V_i$, s vertices are chosen randomly from V_j and used as negative samples. The same happens for vertices $u \in V_j$.

Algorithm 3: SubGraphEmbedding

Input: $\mathbf{S}^d$: pool of positive samples
$\mathbf{M}_m^d$: sub-matrix bin a which contains the sub-matrix of sub-graph i
$\mathbf{M}_n^d$: sub-matrix bin b which contains the sub-matrix of sub-graph j
s: number of negative samples
lr: learning rate
Output: $\mathbf{M}_m^d$, $\mathbf{M}_m^d$

```
1  num_i ← S^d[0];
2  num_j ← S^d[1];
3  index ← 1;
4  for j = 1 to num_i do
5      src ← S^d[index × 2];
6      sample ← S^d[index × 2 + 1];
7      SingleVertexUpdate(M_m^d[src], M_n^d[sample], 1, lr );
8      for k = 1 to s do
9          u ← GetNegativeSampleEmbedding(M_n^d);
10         SingleVertexUpdate(M_m^d[src], u, 0, lr);
11     index ← index + 1;
12 for j = 1 to num_j do
13     src ← S^d[index × 2];
14     sample ← S^d[index × 2 + 1];
15     SingleVertexUpdate(M_n^d[src], M_m^d[sample], 1, lr );
16     for k = 1 to s do
17         u ← GetNegativeSampleEmbedding(M_m^d);
18         SingleVertexUpdate(M_n^d[src], u, 0, lr);
19     index ← index + 1;
```

Kernel execution: An embedding kernel $\mathbf{K}_{i,j}$ carries out Algorithm 3. It uses positive samples from the sample pool bin $\mathbf{S}^d$ to update the two sub-matrices $\mathbf{M}_i$ and $\mathbf{M}_j$ while they are on the GPU in bins $\mathbf{M}_m^d$ and $\mathbf{M}_n^d$, respectively. The number of positive samples in the pool is fetched from the sample pool itself (Lines 1–2) and the samples are read and used to perform updates on $\mathbf{M}_m^d$ (loop on Line 4) and $\mathbf{M}_n^d$ (loop on Line 12). For every positive sample update to a vertex in $\mathbf{M}_m^d$ and $\mathbf{M}_n^d$, s negative samples from the opposing sub-matrix are fetched and updated, with calls to GetNegativeSampleEmbedding($\mathbf{M}_k$) returning a random embedding vertex from the sub-matrix $\mathbf{M}_k$ (Lines 9 and 17).

3.3.4 Impact of the parameters on the performance

Recall that P_{GPU} is the number of sub-matrices that can be stored on the GPU at a time. Since we require every sub-matrix pair to exist on the GPU together during a single rotation, the smallest acceptable value is 2. However, $P_{GPU} = 2$ means that there will be time-points where all the kernels processing the current sub-matrices finish, and a new kernel cannot start until a new sub-matrix is copied to the GPU. This leaves the GPU idle during the copy operation. On the other hand, using $P_{GPU} > 2$ will reduce the size of a single sub-matrix (and the number of samples executed on it per kernel) but allows an overlap of data transfers with kernel executions. For instance, assume $\mathbf{M}_1, \mathbf{M}_2$ and $\mathbf{M}_4$ are on GPU and the three upcoming kernels are $\mathbf{K}_{4,1}, \mathbf{K}_{4,2}$ and $\mathbf{K}_{4,3}$. The first two kernels are dispatched and after the first finishes, while the second is running, $\mathbf{M}_1$ is replaced with $\mathbf{M}_3$, thus hiding the latency.

A large P_{GPU} increases the amount of overlap. However, it also consumes more space on the GPU and increases K, i.e., the number of sets in $\mathcal{V}$. This leads to a rotation containing more kernels, i.e., pairs to be processed. We explore the relationship between the P_{GPU} and the speed of embedding in Section 5.1.2 .

Since we do not keep the large graphs on GPU memory and draw positive samples on the CPU, these samples must also be transferred to the GPU for the kernels to execute. Let S_{GPU} be the number of sample pools stored on the GPU concurrently. Smaller values of S_{GPU} would lead to the same issue stated above; once a sample pool is used up, updates must halt and the GPU will be idle until a new sample pool is fetched from the host. Larger values for S_{GPU} would bypass the idling issue. However, it should be noted that increasing the number of sample pools on the GPU increases the memory space they occupy, leaving less space for the embedding parts and potentially increasing K, which, as stated previously, can slow down the embedding. We explore the performance of different values of S_{GPU} further in Section 5.1.3.

Another equally important hyperparameter is the batch size B. Larger values of B increase the size of a single sample pool which could potentially increase K, however, it allows for more updates to be carried out per rotation, reducing the total number of rotations required to run e epochs. We explore the effect of B on the embedding quality and speed in Section 5.1.4.

The next chapter describes the proposed directed acyclic graph execution model we used to maximize the GPU utilization with efficient synchronization mechanisms.

4. DIRECTED ACYCLIC GRAPH EXECUTION MODEL

To achieve the highest possible utilization of the GPU, we must make sure that it is never idle and that it is always carrying out embedding updates. For that, we need to reduce the number of synchronization points in the algorithm. We do so by adopting a Directed Acyclic Graph (DAG) model of execution. As described in the previous chapter, the proposed approach involves the coordination of four main tasks:

- generating samples on the host,
- copying the sample pools to the device,
- copying the embedding sub-matrices to and from the device, and
- executing the embedding kernels.

In this model, instantiations of the tasks above are represented as nodes, and when a task $\mathcal{T}_2$ is dependent on task $\mathcal{T}_1$, an edge will go from $\mathcal{T}_1$ to $\mathcal{T}_2$. With this model, a task $\mathcal{T}_i$ will only synchronize with its incoming neighbors, thus eliminating the need for algorithm-wide synchronization points while maintaining correctness. Formally, we define the tasks that comprise our system as follows:

1.1 **Sampling Task** ($ST_{i,j,k}$): utilize a team of sampling threads on the host to generate positive samples into a sample pool $\mathbf{S}_{i,j,k}$ for some $k < z$, where z is the number of sample pool sets.

1.2 **Sub-Matrix Swap Task** ($MST^k_{i,j}$): dispatch two GPU memory copies, one to copy sub-matrix $\mathbf{M}_i$ out of the kth sub-matrix bin $\mathbf{M}^d_k$ on the GPU to the host, and one to copy another sub-matrix $\mathbf{M}_j$ from the host to the same sub-matrix bin on the GPU. If $i = -1$ then no copy out of the GPU is carried out, and if $j = -1$ then no copy to the GPU is carried out.

1.3 **Sample Pool Copy Task** ($SCT_{i,j,k}$): dispatch a GPU memory copy to copy the sample pool $\mathbf{S}_{i,j,k}$ to the GPU (the exact location on the GPU is resolved at execution time).

1.4 **Kernel Execution Task** ($KT_{i,j}$): dispatch an embedding kernel $\mathbf{K}_{i,j}$ on the GPU.

We establish a node for each task instance mentioned above and connect the nodes with edges to define their dependency structure, forming an execution DAG $\mathcal{D}$. We also define an execution queue $\mathcal{Q}$ associated with every execution DAG for nodes in the DAG which are ready to be executed. Given a DAG $\mathcal{D}_i$ and a task node $n \in \mathcal{D}_i$, whenever node n's incoming neighbors finish executing, it is added to the execution queue $\mathcal{Q}_i$. A node $n \in \mathcal{Q}_i$ is executed using an independent thread such that multiple nodes from the execution queue can be executed in parallel.

More specifically, we define two DAGs:

- the **Embedding DAG** $\mathcal{D}_e$ consisting of the Sub-Matrix Swap Tasks, Sample Pool Copy Tasks, and Kernel Execution Tasks.
- The **Sampling DAG** $\mathcal{D}_s$: consisting of Sampling Tasks.

The construction of both DAGs, as well as the definition of each of their edges, are elaborated on in the following in Sections 4.1 and 4.2.

4.1 The Embedding DAG ($\mathcal{D}_e$)

The Embedding DAG consists of the tasks which interact directly with the GPU by dispatching work to it. More precisely, it comprises of the tasks that dispatch memory copies of sample pools and sub-matrices and the tasks that dispatch kernels. These tasks are executed on the host and only asynchronously dispatch jobs to the GPU. A task finishes executing once it dispatches its jobs to the GPU; it does not wait for the job to finish executing on the GPU - or even start executing. That is why we need a method to guarantee that the dependency structure of $\mathcal{D}_e$ is transferred to the GPU. We discuss this topic further in Section 4.1.1.

4.1.1 GPU Dependency Using `cudaEvents`

One method to enforce dependency between different jobs on the GPU is by dispatching all the GPU jobs on a single `cudaStream` since all jobs on a stream are sequentially executed based on the order they were dispatched in. However, this will greatly limit the parallelism and eliminate any overlap between memory copies and computation. That is why we use `cudaEvent`s to achieve this dependency structure without sacrificing parallelism. `cudaEvent`s are the canonical method for creating a dependency between different GPU jobs running on different streams. If we wish to create a dependency between the GPU job A dispatched on `cudaStream` S_i and the GPU job B dispatched on the `cudaStream` S_j, we would dispatch A, record `cudaEvent` E on the stream S_i, instruct the stream S_j to wait for event E, then dispatch B to stream S_j. This guarantees that B (and all subsequent jobs on S_j) do not start until job A is complete.

We utilize `cudaEvent`s in $\mathcal{D}_e$ by making it so that every node $n \in \mathcal{D}_e$ will have a single `cudaEvent` E_n which it will record after it dispatches its GPU work, and will use to communicate with its neighbors. Given a node $n_i \in \mathcal{D}_e$ with incoming neighbors $(n_0^I, n_1^I, \cdots, n_a^I)$, outgoing neighbors $n_0^O, n_1^O, \ldots, n_b^O$ and `cudaEvent` E_{n_i}. Before n_i dispatches its job on its stream S_i, it instructs S_i to wait for the events of all of its incoming neighbors $(E_{n_0^I}, E_{n_1^I}, \cdots, E_{n_a^I})$, then it dispatches its GPU job and records the event E_{n_i} on S_i. With this method, the dispatched jobs of any node n will always wait for its incoming nodes' jobs to finish executing, and n's outgoing neighbor nodes' jobs are guaranteed to not execute until n's job is finshed executing, effectively maintaining the correctness of the dependency structure of $\mathcal{D}_e$.

4.1.2 Structure of $\mathcal{D}_e$

The three tasks comprising $\mathcal{D}_e$ can be thought of as running in three parallel lanes. The first lane consists of all the sub-matrix swap tasks, the second of the kernel execution tasks, and the third of the sample pool copy tasks. Given a graph that's partitioned into K sub-graphs, that we are using P_{GPU} sub-matrix bins and S_{GPU} sample pool bins on the GPU, and z sample pool sets on the host, that we are running an embedding of e epochs, and given the execution order $\mathcal{X}$, the DAG is constructed as shown in Algorithm 4.

At the beginning, we calculate the number of embedding rounds r from the number of epochs e and the batch size B (Line 1 of Alg. 4). Then, we create an empty DAG and a shell *Beginning Node* to serve as an execution starting point (Lines 2–4). We create two arrays, $matrixPositions$ and $matrixGPUNodes$ to track the state of the

sub-matrix bins on GPU. Observing $matrixPositions[i] = k$ indicates that $\mathbf{M}_i$ is currently on the GPU in sub-matrix bin $\mathbf{M}_k^d$, and observing $matrixGPUNodes[k] = MST$ indicates that the last node to move a sub-matrix to $\mathbf{M}_k^d$ is MST (Lines 5–6). We also set up two node containers to store the last created sample copy and kernel nodes (Lines 7–8). Then, we create MST nodes that will move the first P_{GPU} sub-matrices to the GPU (Line 9).

The loop at Line 16 is responsible for the bulk of the algorithm. We repeat this loop r times to generate a graph that executes r rounds of embedding. This loop proceeds by dequeuing a part pair (V_i, V_j) from $\mathcal{X}$, and creating $SCN_{i,j}$ and $KN_{i,j}$ to dispatch a copy of the sample pool and dispatch an embedding kernel for said part pair, respectively (Lines 17–18). We make $KN_{i,j}$ dependent on $SCN_{i,j}$ to guarantee that a kernel does not start until its samples are on the GPU (Line 19). Besides, we make this iteration's sample copy node dependent on the last iteration's sample copy node (Line 19). As for consecutive kernel nodes, we connect them with a *weak edge*. This edge does *not* lead to the event propagation process mentioned in Section 4.1.1; when a kernel node is finished executing, it does not send its recorded event to the next kernel node. This is so that kernel nodes on the GPU do not run sequentially, and to allow for kernels to run out-of-order (in case a kernel is ready to execute before its preceding kernel in the execution order is done executing). Afterwards, we make it so the $KN_{i,j}$ is dependent on the nodes which copied sub-matrices $\mathbf{M}_i$ and $\mathbf{M}_j$ to the GPU (Lines 24–25). This guarantees that a kernel does not start until the sub-matrices it needs for it to run are on the GPU.

After a kernel node is executed, one (or both) of its sub-matrices are switched out. The function MATRICESTOSWITCH(X, $matrixPositions$) in Line 28 determines whether, after kernel $K_{i,j}$ is finished executing, any sub-matrices should be switched out, and which sub-matrices are to be switched in their stead (Line 27). If it is determined that sub-matrix a, residing in bin k, is to be switched out and replaced with sub-matrix b, we create the sub-matrix switch node $MST_{a,b}^k$ (Line 29). The execution of the copy job must not begin until all the kernels which depend on $\mathbf{M}_a$ are finished executing. We project this dependency on the graph structure in the loop in Line 30; $MST_{a,b}^k$ is made to depend on all the kernel nodes which depend on the MST node that brought $\mathbf{M}_a$ into the GPU. The $matrixGPUNodes$ and $matrixPositions$ arrays are updated to reflect the change in the sub-matrices' states (Lines 32–33).

Finally, after all the kernel nodes have been added to the graph, a shell *Terminal Node* is created and made to depend on the final kernel node as well as the part switch nodes which will take out whichever embeddings sub-matrices are on the

GPU (Lines 34–36) as well as the. A demonstration of this process is shown in Figure 4.1. In the preceding section, it can be seen that we determine at the graph generation stage the exact locations of sub-matrices on the GPU, while we do not do the same for sample pools. This is because the processing time of a sample pool is highly unpredictable; sample pools can have a variable number of positive samples and, consequently, would require varying amounts of time to process. We leave the location of sample pools on the GPU to be decided at the node execution time and we coordinate it using shared variables. We discuss this further in Section 4.3.

4.1.3 Task Queue of $\mathcal{D}_e$

The task queue of the embedding DAG is populated by the nodes whose incoming neighbors have been executed to completion. Nodes enqueued into the task queue are executed by a team of τ_e threads. The tasks the threads execute for nodes in $\mathcal{D}_e$ are not computationally significant, and hence do not incur any costs on the running host. In our experiments, we use $\tau_e = 5$ as we found it to be sufficient, and experimentally, using higher values does not yield any improvement in execution time.

4.2 Sampling DAG ($\mathcal{D}_s$)

Sample pools must be generated in the same order of the kernel executions mentioned in Section 3.3.3 since that is the order in which the sample pools will become needed by the embedding kernels. However, generating sample pools sequentially could lead to idleness on the GPU. That is because the GPU is capable of processing multiple sample pools in parallel and the sample pools vary greatly in the number of samples they contain. That is why up to C sample pools are sampled in parallel, and the order in which the sampling happens is in such a way that the sample pools needed sooner are sampled into first. An important distinction between the nodes in $\mathcal{D}_s$ and $\mathcal{D}_e$ is that the nodes in the former do not dispatch any GPU jobs, and therefore do not record `cudaEvents`. In other words, *all* edges in $\mathcal{D}_s$ are weak edges according to the definition in Section 4.1.2

Algorithm 4: constructDe

Input: S_{GPU}: number of sample pool bins on the GPU
P_{GPU}: number of sub-matrix bins on the GPU
e: number of epochs
z: number of sample pool sets
K: number of sub-graphs in the partition of the graph
$\mathcal{X}$: execution order set

Output: D_e: embedding DAG

```
1  r ← e/B;
2  D_e ← createEmptyDAG();
3  BN ← createBeginningNode();
4  setRoot(D_e, BN);
5  matrixGPUNodes ← array[P_GPU] = null;
6  matrixPositions ← array[K] = −1;
7  lastCopyNode ← BN;
8  lastKernelNode ← BN ;
9  for i = 0 to P_GPU do
10     MN ← createMatrixSwapTask(−1, i, i);
11     matrixGPUNodes[i] ← MN;
12     matrixPositions[i] ← i;
13     lastMatrixNode.addEdge(MN);
14     lastMatrixNode ← MN;
15 for r rounds do
16     for (i, j) ∈ X do
17         SCN ← createSampleCopyTask(i, j, r mod z);
18         KN ← createKernelTask(i, j);
19         SCN.addEdge(KN);
20         lastSampleCopyNode.addEdge(SCN);
21         lastKernelNode.addWeakEdge(KN);
22         lastSampleCopyNode ← SCN;
23         lastKernelNode ← KN ;
24         matrixGPUNodes[matrixPositions[i]].addEdge(KN);
25         matrixGPUNodes[matrixPositions[j]].addEdge(KN);
26         outMatrices, inMatrices ← matricesToSwitch(X,
            matrixPositions);
27         for (in, out) ∈ (outMatrices, inMatrices) do
28             outPosition ← matrixPositions[out];
29             MN ← createMatrixSwapTask(out, in, outPosition);
30             for KN ∈ matrixGPUNodes[outPosition].getOutEdges() do
31                 KN.addEdge(MN);
32             matrixGPUNodes[in] ← MN;
33             matrixPositions[in] ← outPosition;
34 TN ← createTerminalNode();
35 lastKernelNode.addEdge(TN);
36 for i = 0 to P_GPU do
37     matrixGPUNodes[i].addEdge(TN);
```

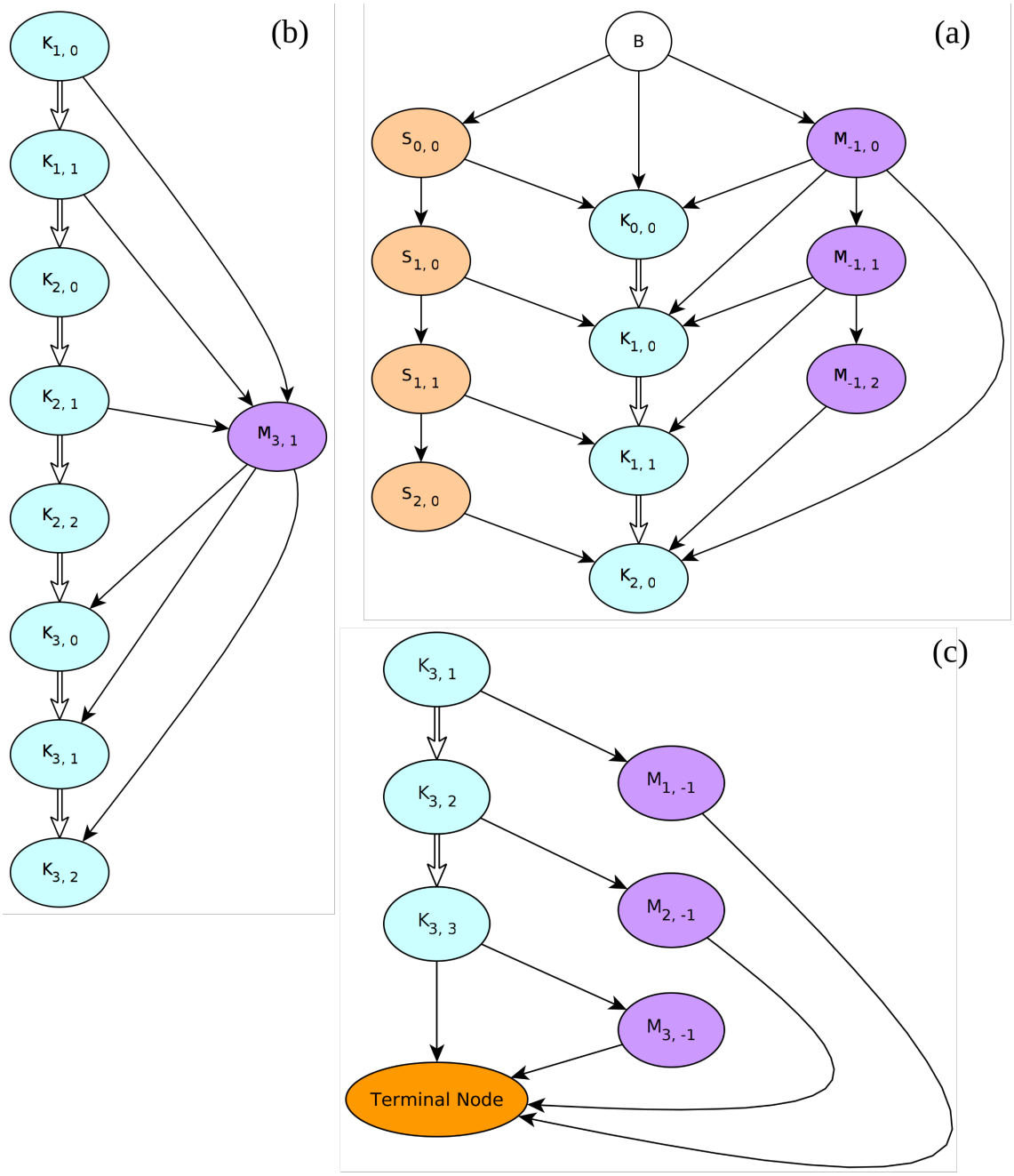

Figure 4.1 A demonstration of the construction of $\mathcal{D}_e$ given $K = 4$ and $P_{GPU} = 3$. (a) The graph starts with a beginning node that connects to the first sample task, kernel task, and part swap task. Consecutive kernels are connected with weak edges (white arrows) while other elements in the graph are connected with normal edges. (b) The sub-matrix swap node that will copy out $\mathbf{M}_1$ and copy in $\mathbf{M}_3$ will depend on kernel nodes that are using $\mathbf{M}_1$, and its dependents are the nodes that will use $\mathbf{M}_3$. (c) At the end of the embedding, the last kernel node, plus the last P_{GPU} sub-matrix switch nodes will have an edge to the terminal node.

4.2.1 Structure of $\mathcal{D}_s$

Nodes in the sampling DAG are set up linearly such that they are created in the same order as the execution order $\mathcal{X}$. The process of creating $\mathcal{D}_s$ is shown in Algorithm 5. The algorithm takes as inputs the execution order set $\mathcal{X}$, the number of sample pool sets z, the total number of rounds r, and the number of sub-graphs that the graph under embedding has been partitioned to K. It returns a fully constructed $\mathcal{D}_s$. There are two types of edges added to this graph. The first type contains the

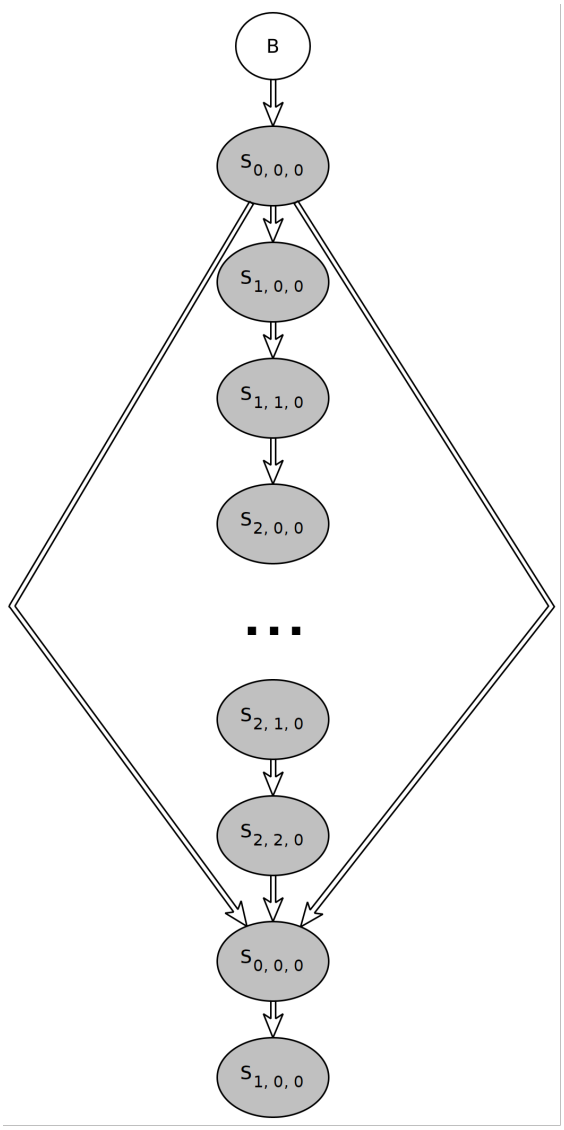

Figure 4.2 The construction of $\mathcal{D}_s$ is given for $K = 3$. There exist two types of edges; one between consecutive sample pools, and one between sample tasks that will be sampled into the same pool.

edges between consecutively created nodes (Line 13 of Alg. 5). These are added to guarantee that sample pools are made ready for kernels in time. The second type contains the edges between nodes which will sample into the same pool (Line 11).

Hence, no two thread teams attempt to sample into the same pool. A snippet from a D_s construction is shown in Figure 4.2.

Algorithm 5: CONSTRUCTDS

Input: $\mathcal{X}$: execution order set
z: number of sample pool sets
r: number of embedding rounds
K: number of sub-graphs in the partition of the graph

Output: D_s: sampling DAG

```
1  D_s ← createEmptyDAG();
2  BN ← createBeginningNode();
3  setRoot(D_e, BN);
4  k ← 0;
5  lastNode ← BN;
6  lastRoundsNodes ← array[z][K][K] = {null};
7  for r rounds do
8      for (i, j) ∈ X do
9          TS ← createSampleTask(i,j,k);
10         if lastRoundsNodes[k][i][j] ≠ null then
11             lastRoundsNodes[k][i][j].addWeakEdge(TS);
12         lastRoundsNodes[k][i][j] = TS;
13         lastNode.addWeakEdge(TS);
14         lastNode ← TS;
15     k ← (k+1) mod z;
```

4.2.2 Task Queue of D_s

Sampling tasks are heavily compute-intensive on the host. We control the number of concurrent sampling tasks by limiting τ_s, the size of the team of threads executing tasks from $\mathcal{Q}_s$, the task queue of $\mathcal{D}_s$. In other words, we set $\tau_s = C$, i.e to the number of concurrent samplers.

4.3 Host Dependency Using Shared Variables

We establish communication between different tasks using the inherent structure of the DAGs for both $\mathcal{D}_e$ and $\mathcal{D}_s$, and use `cudaEvents` to transfer our dependency

structure to the GPU in the case of $\mathcal{D}_e$ as mentioned in Section 4.1.1. However, these two methods do not cover the entire array of communication required for the correctness of the algorithm. Two additional types of communication arise between sample tasks and sample copy tasks, and between sample copy tasks and kernel tasks. Sample tasks and sample copy tasks are not part of the same graph, and yet they require to communicate to establish the states of sample pools on the host. A sample copy task needs to be informed of whether the pool it is assigned to copy has been sampled into or not. Similarly, a sample task must not sample into a pool that is being copied from by a sample pool.

Additionally, there is an element in the algorithm which is not coordinated during the graph generation phase, nor is it coordinated using `cudaEvents`: the locations to which sample pools are copied to on the GPU. To elaborate, while generating $\mathcal{D}_e$, we specify the *order* in which sample pools are copied, but we do not dictate the location on the GPU to which they should be copied, i.e in which sample pool bin $\mathbf{S}^d$ to place them. This is not the case with sub-matrices; we define the location to which sub-matrices are to be copied during graph generation time. The reason for this difference between sample pools and sub-matrices is that sample pool usage times are highly unpredictable - much more than sub-matrices. Sample pools vary greatly in the number of positive samples they may contain. This is because the bigger that a graph is, and the more partitions it has, the higher the odds for vertices in some sub-matrix $\mathbf{M}_i$ to not have any positive samples in another sub-matrix $\mathbf{M}_j$. This leads to a drastic contrast in sample pool processing times, and consequently, to many instances when a sample pool is processed to completion before its predecessor sample pools. For this reason, we leave the decision of choosing sample pool locations on the GPU to be made at sample copy time. Just as the location to which sample pools are copied is relevant to sample copy nodes, it is also important for kernel nodes since these nodes, when dispatching embedding kernels, must inform the embedding kernel of the location of its samples.

We establish communication between kernel nodes and sample copy nodes as well as sample copy nodes and sample task nodes using shared variables, mutexes, and condition variables. Shared variables store the states of the different resources on the host and the device, mutexes prevent race conditions and ensure correctness, and condition variables provide an interface by which threads can send messages amongst themselves. There are three system-wide control variables by which this communication is achieved:

- $S^h_{i,j,k}$ where $0 \leq j \leq i < K$ and $0 < k < z$: a variable which relays the status of sample pools on the host. $S^h_{i,j,k} = -1$ indicates that sample pool $\mathbf{S}^h_{i,j,k}$ is

empty and ready to be sampled into, $S^h_{i,j,k} = 0$ means that the sample pool is full and ready to be copied, and $S^h_{i,j,k} = 1$ means that the sample pool is currently being copied to the GPU.

- S^d_i where $0 \leq i < S_{GPU}$: holds the status of sample pools on the device. $S^d_i = -1$ indicates that GPU pool $\mathbf{S}^d_i$ is empty, $S^d_i = 0$ indicates that said pool is being copied into, but not ready for usage by a kernel yet, and $S^d_i = 1$ means that the pool is full and ready to be used for embedding.

- $K_{i,j}$ where $0 \leq j \leq i < K$: every kernel of execution is given a variable which indicates whether or not it has been assigned a sample pool on the device to fetch its samples from. $K_{i,j} = -1$ means that kernel $\mathbf{K}_{i,j}$ has not been assigned a sample pool yet, while $K_{i,j} = g$ where $0 \leq g < S_{GPU}$ indicates that the kernel $\mathbf{K}_{i,j}$ has been assigned the sample pool bin $\mathbf{S}^d_g$ on the GPU; it will use it as a source for its positive samples.

These control variables are read and written to by a variety of threads in the system and their usage will be elaborated upon in Section 4.4.

4.4 Implementation of Tasks

4.4.1 Sampling task

On the host, where all the sampling takes place, $z \times \frac{K \times (K+1)}{2}$ sample pools are generated, i.e., z pools are generated for every $\mathbf{K}_{i,j}$. These sample pools are continuously sampled into by a team of T_s sampling threads. Do note that not all the threads will sample into the same pool; sampling threads sample into up to C sample pools concurrently where $C \geq 1$. Using $C > 1$ means that multiple sample pools are prepared concurrently. We explore the effect of concurrent samplers on embedding runtime further in Section 5.1.5.

Sampling into a sample pool is shown in Algorithm 6. The sampling thread waits for the sample pool on the host to be empty through its shared variable $S^h_{i,j,k}$, locks the variable to prevent race conditions, and sets it to 0 to indicate that it is full (Lines 1–2). After sampling is over, the variable is unlocked (Line 32). The execution of a

Algorithm 6: SAMPLETOPOOL

Input: $\mathbf{S}^h_{i,j,k}$: sample pool k for sub-graph part pair V_i and V_j
V_i: sub-graph i
V_j: sub-graph j
B: embedding batch size
vertices_per_part: number of vertices in a single sub-graph
T_s: number of sampling threads
C: number of concurrent samplers
$S^h_{i,j,k}$: shared variable with the status of the sample pool $\mathbf{S}^h_{i,j,k}$

Output: $\mathbf{S}^h_{i,j,k}$

```
1  waitUntilAndLock(S^h_{i,j,k}, -1);
2  S^h_{i,j,k} ← 0;
3  T_l ← T_s / C;
4  vertices_per_thread ← vertices_per_part / T_l;
5  thread_counter ← array[T_l] = {0};
6  samples_per_thread ← array[T_l] = {null};
7  starting_src ← i× vertices_per_part;
8  for every thread t in T_l in parallel do
9      counter ← 0;
10     pool ← array[B × vertices_per_thread × 2] = 0;
11     for 0 to B do
12         src ← starting_src + vertices_per_thread ×t;
13         for k = 0 to vertices_per_thread do
14             ps ← SampleFromSubpart(src+k, V_j);
15             if ps ≠ −1 then
16                 pool[t][counter × 2] ← src+k;
17                 pool[t][counter × 2+1] ← ps;
18                 counter+ =;
19     if i ≠ j then
20         starting_src ← j× vertices_per_part;
21         src ← starting_src+vertices_per_thread ×t;
22         for 0 to B do
23             for k = 0 to vertices_per_thread do
24                 ps ← SampleFromSubpart(src+k, V_i);
25                 if ps ≠ −1 then
26                     pool[t][counter × 2] ← src+k;
27                     pool[t][counter × 2+1] ← ps;
28                     counter+ = 1;
29     thread_counter[t] ← counter;
30     samples_per_thread[t] ← pool;
31 S^h_{i,j,k} ← mergeThreadSamples(thread_counter, samples_per_thread);
32 unlockVariable(S^h_{i,j,k});
```

single sampling job uses T_l local threads, and each thread will generate samples for an equal portion of vertices (Line 4). Every thread allocates a thread-private sample pool to store its generated samples and a thread-private counter to write the number of samples it generates (Lines 9–10). After a thread is done sampling, it will write its pool and its counter to the shared memory space so that they can be merged and placed into $\mathbf{S}^h_{i,j,k}$ (Lines 29–30). These private data guarantee that sampling is completely parallelized and protected against false-sharing. However, this can lead to load-imbalance if regions in a sample pool are less dense with samples than others. We experimentally show in Section 5.1.5 that generally, this is not the case.

The function SAMPLEFROMSUBPART(v, V_k) chooses a sample $u \in V_k$ for vertex v, and if no samples are found returns -1. After a thread is finished generating samples for vertices in V_i from V_j (loop at Line 11), it will generate samples for vertices in V_j from V_i given that $i \neq j$ (loop at Line 22). After all the threads are done sampling, MERGETHREADSAMPLES(*thread_counters*, *samples_per_thread*) will use the thread counters to sequentially merge the per-thread sample pools and store the results in $\mathbf{S}^h_{i,j,k}$.

4.4.2 Sample pool copies

At the beginning of the algorithm, S_{GPU} sample pool bins are allocated on the GPU. These bins are what the embedding kernels use to carry out the embedding. Whenever kernel $\mathbf{K}_{i,j}$ is scheduled to run, it will wait until a sample pool $\mathbf{S}^h_{i,j,k}$ has been moved to one of the sample pools bins on the GPU. The task of dispatching the copies of sample pools to the GPU is carried out by sample pool copy tasks. In addition to dispatching the copy jobs to the GPU, sample copy tasks inform sample tasks when the latter should resample into a pool, and communicate with kernel tasks to inform them from which sample pool bin on the GPU the kernels must fetch their positive samples.

Let $SC_{i,j,k}$ be the task of copying a single sample pool $S_{i,j,k}$ to the GPU. The execution of $SC_{i,j,k}$ is shown in Algorithm 7. As shown, before the copying is initiated, the thread will wait until the control variable of the sample pool $S^h_{i,j,k}$ indicates that the sample pool on the host has been filled, and sets it to 1 afterward to indicate that it is being copied from (Line 1). Then, the task attempts to reserve a sample pool bin on the GPU with the call to RESERVEGPUSAMPLEPOOLBINANDLOCK(). This call will block until one of the S_{GPU} bins on the GPU is free. Once it acquires one, say, the bin at position *bin*, it will reserve it by locking its shared variable S^d_{bin},

Algorithm 7: SAMPLEPOOLCOPY

Input: $S_{i,j,k}$: shared variable containing the status of pool $\mathbf{S}^h_{i,j,k}$
$\mathbf{S}^h_{i,j,k}$: sample pool k of the sub-graph pair V_i and V_j
$K_{i,j}$: shared variable of the status of kernel $\mathbf{K}_{i,j}$

1 WAITUNTILANDLOCK($S^h_{i,j,k}$, 0);
2 $S^h_{i,j,k} \leftarrow 1$;
3 $bin \leftarrow$ RESERVEGPUSAMPLEPOOLBINANDLOCK();
4 WAITUNTILANDLOCK($K_{i,j}$, -1);
5 $K_{i,j} \leftarrow bin$;
6 $\mathbf{S}^d_{bin} \leftarrow$ GETSAMPLEPOOLBIN(bin);
7 DISPATCHCOPYFROMTO($S_{i,j,k}$, $\mathbf{S}^d_{bin}$);
8 ADDGPUCALLBACK(SAMPLECOPYCALLBACK, $S^h_{i,j,k}$, S^d_{bin});
9 UNLOCKVARIABLE($S^h_{i,j,k}$);
10 UNLOCKVARIABLE(S^d_{bin});
11 UNLOCKVARIABLE($K_{i,j}$);

and setting its value to $S^d_{bin} = 0$ to indicate that S^d_{bin} is being copied to and is not vacant (Line 3). Afterward, it also waits for the control variable of the kernel to become $K_{i,j} = -1$ to indicate that the kernel is not currently being executed. It then locks the variable and sets its value to *bin*, communicating that $\mathbf{S}^d_{bin}$ is the location from which the kernel is to fetch its samples upon its execution (Lines 3–5). Afterward, the GPU copy of the sample pool is dispatched (Line 7). After dispatching the copy, a `cudaCallback` is dispatched to the GPU (Line 8). This callback will be executed on the host once the copy job has been executed to completion and will make the required changes to the shared variables to indicate that the sample copy is complete, the callback's workings are shown in Algorithm 8.

The callback is tasked with signaling that the sample pool $\mathbf{S}^h_{i,j,k}$ is free to be resampled into (Line 3 in Alg. 8), and signaling that the sample pool bin $\mathbf{S}^d_{bin}$ on the GPU has now been filled to completion with samples (Line 4).

Algorithm 8: SAMPLECOPYCALLBACK

Input: $S^h_{i,j,k}$: shared variable containing the state of sample pool $\mathbf{S}^h_{i,j,k}$
S^d_{bin}: shared variable containing the state of sample pool bin $\mathbf{S}^d_{bin}$

1 LOCKVARIABLE($S^h_{i,j,k}$);
2 LOCKVARIABLE(S^d_{bin});
3 $S^h_{i,j,k} \leftarrow -1$;
4 $S^d_{bin} \leftarrow 1$;
5 UNLOCKVARIABLE($S^h_{i,j,k}$);
6 UNLOCKVARIABLE(S^d_{bin});

4.4.3 Matrix Swaps

Carrying out the embedding procedure happens on parts of the embedding matrix $\mathbf{M}$ which we partition suitably for the GPU memory capabilities. The GPU stores two or more embedding sub-matrices in sub-matrix bins allocated on it before the execution of the embedding and embedding is carried out on whichever sub-matrices are currently in these bins. The task of dispatching copies of embedding sub-matrices to and back from the GPU is carried out by matrix swap tasks. These tasks will take out an updated embedding sub-matrix from one of the sub-matrix bins on the GPU back to its location on the host, then they will copy a different embedding sub-matrix from the host and place it in the same sub-matrix bin on the GPU. This process is shown in Algorithm 9. It starts by acquiring a pointer to the sub-matrix bin on the GPU (Line 1). Afterwards, it will copy whichever sub-matrix is currently on the GPU back to the host (Lines 2–3), and will copy a sub-matrix from the host to the GPU (Lines 4–5). Do note that a swap job is not always in both directions. If the sub-matrix bin is empty, we do not wish to copy its contents to the host. Similarly, if we've reached the end of the embedding and a sub-matrix bin is not going to be used anymore, we do not wish to spend time and resources copying to it.

Algorithm 9: createMatrixSwapTask

Input: $\mathbf{M}_{out}$: sub-matrix of sub-graph *out* on the host
$\mathbf{M}_{in}$: sub-matrix of sub-graph *in* on the host
bin: index of the sub-matrix bin on the GPU the copy is occuring on

```
1 M^d_bin ← getSubMatrixBin(bin);
2 if M_out ≠ null then
3     dispatchCopyFromTo(M^d_bin, M_out);
4 if M_in ≠ null then
5     dispatchCopyFromTo(M_in,M^d_bin);
```

4.4.4 Kernel execution tasks

The embedding procedure operates by sending sub-matrices from the embedding matrix to the GPU and carrying out embedding updates on whichever sub-matrices are currently residing on the GPU. Kernel execution tasks dispatch the embedding kernels tasked with these updates. They coordinate with sample pool copy tasks to

inform the GPU of the locations of the samples required for a particular kernel execution. They do not need to directly coordinate with sub-matrix copy tasks since the locations of the sub-matrices that they will use for embedding are predetermined at DAG generation time, and the dependency relationship is decided with `cudaEvents`. The operation of a kernel execution task is shown in Algorithm 10. The execution requires the indices of the sub-graphs on which embedding is going to take place, as well as the sub-matrix bins that contain the sub-matrices of the aforementioned sub-graphs. Execution starts by waiting for the kernel to be assigned a sample-pool bin on the GPU by a sample pool copy task (Line 1). After the index of the sample pool bin on the GPU is acquired from the shared variable of the kernel (Line 2), the task will wait until the sample pool has been copied to completion by a sample copy task (Line 3). Once copying is finished successfully, the task will dispatch the embedding kernel to the GPU (Line 5). Afterward, a `cudaCallback` is dispatched on the GPU and scheduled to run right after the kernel is finished executing. This callback is tasked with signaling that the kernel is complete and that the sample pool bin is free for other sample pools to be copied into it. The callback is shown in Algorithm 11. The callback sets the shared variable of the kernel that was executed to -1 to indicate that the kernel is finished executing and is ready for another round of embedding (Line 3). Also, it sets the shared variable of the sample pool bin on the GPU to -1 to indicate that the sample pool bin is now free for other sample pools to be copied into it (Line 4).

Algorithm 10: kernelExecutionTask

Input: i: index of the first sub-graph the embedding kernel is updating
j: index of the second sub-graph the embedding kernel is updating
$\mathbf{M}_m^d$: the sub-matrix bin on which $\mathbf{M}_i$ is located
$\mathbf{M}_n^d$: the sub-matrix bin on which $\mathbf{M}_j$ is located

1 waitUntilAndLock($K_{i,j}$, > -1);
2 $samplePoolBin \leftarrow K_{i,j}$;
3 waitUntilAndLock($S^d_{samplePoolbin}$, 1);
4 $\mathbf{S}^d_{samplePoolBin} \leftarrow$ getSamplePoolBin(samplePoolID);
5 dispatchEmbeddingKernel($\mathbf{M}^d_m$, $\mathbf{M}^d_n$, $\mathbf{S}^d_{samplePoolBin}$);
6 addGPUCallback(kernelCallback, $K_{i,j}$, $S^d_{samplePoolID}$);
7 unlockVariable($K_{i,j}$);
8 unlockVariable($S^D_{samplePoolID}$);

Algorithm 11: kernelCallback

Input: $K_{i,j}$: shared variable containing the status of kernel $\mathbf{K}_{i,j}$
S^d_{bin}: shared variable containing the status of sample pool bin $\mathbf{S}^d_{bin}$

```
1 lockVariable(K_{i,j});
2 lockVariable(S^d_{bin});
3 K_{j,k} ← −1;
4 S^d_{bin} ← −1;
5 unlockVariable(K_{j,k});
6 unlockVariable(S^d_{bin});
```

5. EVALUATION

The embedding procedure proposed in this work accelerates graph embedding using GPUs while mitigating their memory limitations which prevent larger scale graphs from being embedded by a single GPU. As important as it is to accelerate the process of embedding, we must not sacrifice the quality of the embeddings it produces. In this chapter, we explore the effectiveness of the schema proposed in this work, both in terms of embedding runtime and prediction accuracy. The approach proposed in this work is a part of a tool named GOSH (Akyildiz et al., 2020) designed for accelerating graph embedding and reducing the procedure's memory overhead. GOSH utilizes a graph coarsening approach that compresses the graph under embedding to minimize the amount of work required to carry out an embedding. More precisely, given a graph G_0, GOSH compresses G_0 into a smaller graph by combining groups of one or more vertices in G_0 into super vertices to produce a new, smaller graph G_1. The graph G_1 is coarsened into a graph G_2, and so on. Once a certain stopping criterion has been met and D coarsened graphs have been generated, the final level generated, G_{D-1}, is embedded using the embedding algorithm proposed in this work. Then, its embedding is projected on a graph G_{D-1} by assigning every vertex in G_{D-2} the embedding of its super vertex in G_{D-1}. Then G_{D-2} is embedded with the projected embeddings as its initial starting point, and its embeddings are projected to G_{D-3}. The process continuous until G_0 is embedded. The algorithm presented in this work is mainly concerned with embedding the larger levels that do not fit inside the GPU memory.

In the following sections, we will introduce the graphs we use in our experiments, as well as the hardware and software environments used during our experiments. Afterward, we analyze the large graph embedding procedure. Finally, we provide experimental results that compare GOSH to other state-of-the-art graph embedding implementations both in terms of speed and embedding quality.

Datasets: For our experiments, we use four large-scale graphs to demonstrate the efficiency of the proposed embedding method. Also, we use eight medium-scale graphs to assess the machine learning quality of GOSH as a whole. The graphs

Table 5.1 Medium- and large-scale graphs used in the evaluation experiments.

Graph	**|V|**	**|E|**	**Density**
`com-dblp` (Leskovec & Krevl, 2014)	317,080	1,049,866	3.31
`com-amazon` (Leskovec & Krevl, 2014)	334,863	925,872	2.76
`youtube` (Mislove, Marcon, Gummadi, Druschel & Bhattacharjee, 2007)	1,138,499	4,945,382	4.34
`soc-pokec` (Leskovec & Krevl, 2014)	1,632,803	30,622,564	18.75
`wiki-topcats` (Leskovec & Krevl, 2014)	1,791,489	28,511,807	15.92
`com-orkut` (Leskovec & Krevl, 2014)	3,072,441	117,185,083	38.14
`com-lj` (Leskovec & Krevl, 2014)	3,997,962	34,681,189	8.67
`soc-LiveJournal` (Leskovec & Krevl, 2014)	4,847,571	68,993,773	14.23
`hyperlink2012` (Meusel, 2015)	39,497,204	623,056,313	15.77
`soc-sinaweibo` (Rossi & Ahmed, 2015)	58,655,849	261,321,071	4.46
`twitter_rv` (Rossi & Ahmed, 2015)	41,652,230	1,468,365,182	35.25
`com-friendster` (Leskovec & Krevl, 2014)	65,608,366	1,806,067,135	27.53

range in their vertices amount as well as densities. A summary of the graphs is shown in Table 5.1.

Experimental Environment: We compile and run our algorithm and the other systems used for evaluation on two different servers. The first machine, which we will refer to as `Gandalf`, has 4 NUMA sockets, each with 15 Intel E7-4870 v2 cores running at 2.30GHz and a single thread per core for a total of 60 logical cores. The server is equipped with 515GBs of RAM. It uses a single Tesla K40c GPU with compute capability 3.5, having 11.4GBs of global memory, and connected through PCIe 3. The server has `CentOS 6.5` as its operating system.

The second system, which we will refer to as `Nebula`, has 2 NUMA sockets with 8 Intel E5-2620 v4 cores running at 2.10GHz and two hyper-threads per core, adding up to 32 logical cores. The machine is equipped with 192GBs of RAM, and a TITAN X GPU with compute capability 6.1 and 12GB of global memory. The GPU is connected to the server via `PCIe 3.0 x 16`. The server's operating system is `Ubuntu 4.4.0-159`.

The code was compiled using `nvcc 10.1` on both servers with optimization flag `-O3`. For CPU parallelization, `OpenMP` was used with `C++11` multithreading data structures.

Experimental setup: We evaluate the accuracy of the embeddings generated with link prediction, which is one of the most commonly used machine learning tasks used for evaluating graph embeddings (Grover & Leskovec, 2016; Lerer et al., 2019; Tsitsulin et al., 2018; Zhu et al., 2019).

Evaluating a graph G starts by splitting the graph into a train sub-graph $G_{train} = (V_{train}, E_{train})$ and a test sub-graph $G_{test} = (V_{test}, E_{test})$. The split is made such that G_{train} contains 80% of the edges of G and G_{test} contains the remaining 20%. Afterwards, we remove all isolated vertices from G_{train}, i.e we remove all vertices $v \in V_{train}$ iff $(v, u) \notin E_{train} \forall u$. In addition, we remove any edge in E_{test} if one of the vertices making up the edge are not in V_{train}. More formally, we remove all $(u, v) \in E_{test} \iff u, v \notin G_{train}$. This is to prevent G_{test} from including vertices which will not have an embedding at the end of training phase. In other words, it guarantees that $V_{test} \subseteq V_{train}$. Next, we execute the embedding model under evaluation with G_{train} as input. Finally, we use the resultant embedding matrix to predict the existence of edges in E_{test} by employing a Logistic Regression model. We use the `SGDClassifier` module from the `scikit-learn` library with a logistic regression classifier (Pedregosa, Varoquaux, Gramfort, Michel, Thirion, Grisel, Blondel, Prettenhofer, Weiss, Dubourg, Vanderplas, Passos, Cournapeau, Brucher, Perrot & Duchesnay, 2011).

The logistic regression model is trained and evaluated using two matrices, $\mathbf{R}_{train}$ and $\mathbf{R}_{test}$. Each vector i in these matrices $\mathbf{R}[i]$ corresponds to a logistic regression sample. Positive samples in $\mathbf{R}_k$ correspond to edges that exist in the graph $\mathbf{G}_k$, while negative samples correspond to edges that do not. We generate a single sample corresponding to the edge (u, v) by element-wise multiplying the embeddings of u and v. In addition, we concatenate to each vector a binary value that indicates whether the sample is a positive or a negative one. $\mathbf{R}_{train}$ includes vectors for all the edges $\in E_{train}$ as positive samples, as well as $|E_{train}|$ negative samples. The same procedure is carried out for $\mathbf{R}_{test}$ by using G_{test} instead of G_{train}.

After training the logistic regression model with $\mathbf{R}_{train}$, we evaluate its prediction performance using $\mathbf{R}_{test}$ and report its acquired $AUCROC$ score (Fawcett, 2006).

5.1 Large Graph Embedding Analysis

The algorithm proposed in this work uses a highly flexible scheduling schema in terms of the parameters provided for the users to tune. These parameters affect many different elements of the algorithm. In this section, we explore the performance of the algorithm and gauge the sensitivity of some of the tunable hyper-parameters present.

5.1.1 GPU parallelization performance

We examine the efficiency of the GPU parallelization introduced in Section 3.2.1 as well as our partitioning approach for large-scale graphs that do not fit the GPU by comparing it to a multi-core CPU version that we implemented. Table 5.1 shows some of the speedups acquired from running our algorithm versus the multi-core CPU implementation running with 16 threads. The three leftmost graphs are medium-scale and do not require the large graphs scheduling schema. The other four are large-scale graphs that do not fit inside a single GPU and require partitioning. We find that using the GPU parallelization is at least 2× faster than a multi-core CPU implementation, and at most 7.3× faster. However, the partitioning schema entails a heavy penalty on the performance. The average speedup for all eight of the medium-scale graphs in Table 5.1 is 5.47×, while the average for large-scale graphs is 2.80×.

5.1.2 Number of embedding sub-matrix bins P_{GPU}

Large graphs embedded with our model are partitioned into smaller sub-graphs whose corresponding embedding sub-matrices can fit inside the GPU. The number of sub-matrices placed on the GPU is controlled with the parameter P_{GPU}. Table 5.2 presents the runtimes of embedding the four large-scale graphs in Table 5.1 with different values of P_{GPU}. These results are plotted in Figure 5.2. The smallest value used for P_{GPU} is $P_{GPU} = 2$ since it is the smallest value that maintains the correctness of the algorithm as explained in Section 3.3.4. We observe that going from two sub-matrix bins to a three sub-matrices drastically improves the runtime - we see an average improvement of 20% in runtime, with `hyperlink2012` seeing the biggest improvement with a 25% speedup. This improvement is due to the overlapping of kernel computation and memory copies of sub-matrices from and to

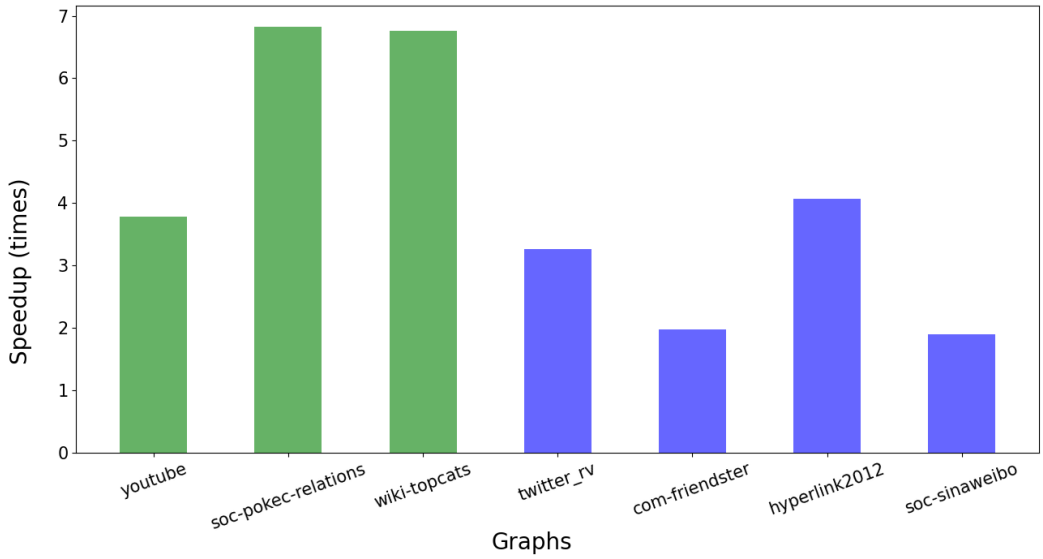

Figure 5.1 The speedup GPU parallelization achieves over a multicore parallel version. Experiments were run with 4 medium-scale graphs (green) and 4 larege-scale graphs (blue). The multi-core CPU implementation was run with 16 threads. We used $B = 5$, $P_{GPU} = 3$, $S_{GPU} = 4$, $C = 4$ and $T_s = 16$ as the hyper-parameters for our approach.

the GPU that an additional sub-matrix can introduce. We demonstrate this effect in Figure 5.3 which shows a trace of the embedding of `hyperlink2012` obtained through `nvvp`, the Nvidia visual profiling tool. As shown in the figure, when $P_{GPU} = 2$, the GPU is idle after every kernel execution (except $K_{i,j} : j = i - 1$). This happens because while the GPU is running some kernel $\mathbf{K}_{i,j}$ where $j < i - 1$, the sub-matrices on the GPU cannot be switched out until the kernel is complete, and the next kernel must wait for the new kernel to be swapped into the GPU. On the other hand, with $P_{GPU} = 3$, copies of the sub-matrices are hidden by embedding kernels. It should be noted that the time difference between the two figures is not all due to the aforementioned latency hiding. It is also the result of 2 partitioning the graph into fewer parts than $P_{GPU} = 3$; parts that have more vertices, and consequentially, kernels that carry out more work, and memory copies of bigger chunks of data. The difference in the number of sub-graphs generated for different values of P_{GPU} is shown in Table 5.2.

Increasing P_{GPU} past three, however, slows down the runtime consistently for all graphs under evaluation. This is because increasing P_{GPU} means that sub-graphs must become smaller, and the number of sub-graphs the graph is partitioned into increases, as well. This is translated into more sub-matrix copies done per round of embedding, and more kernel calls.

Table 5.2 The effect of the number of sub-matrix bins P_{GPU} on the the number of graph sub-parts K generated for a graph, as well as the runtime of embedding. The experiments were carried out on `Gandalf` using the four large-scale graphs in Table 5.1. Experiments were run with $e = 100$, $S_{GPU} = 4$, $C = 4$, $B = 5$, and $T_s = 16$.

Graphs	P_{GPU}	K	**Time (s)**	**Graphs**	P_{GPU}	K	**Time (s)**
	2	7	1431.07		2	5	744.29
	3	9	1173.85		3	6	556.36
`soc-sinaweibo`	4	12	1253.81	`hyperlink2012`	4	8	457.97
	5	14	1301.68		5	10	496.62
	6	17	1514.45		6	11	528.00
	2	5	740.29		2	7	1541.32
	3	7	607.93		3	10	1230.56
`twitter_rv`	4	8	510.88	`com-friendster`	4	13	1196.00
	5	10	531.15		5	16	1369.68
	6	12	615.93		6	19	1594.44

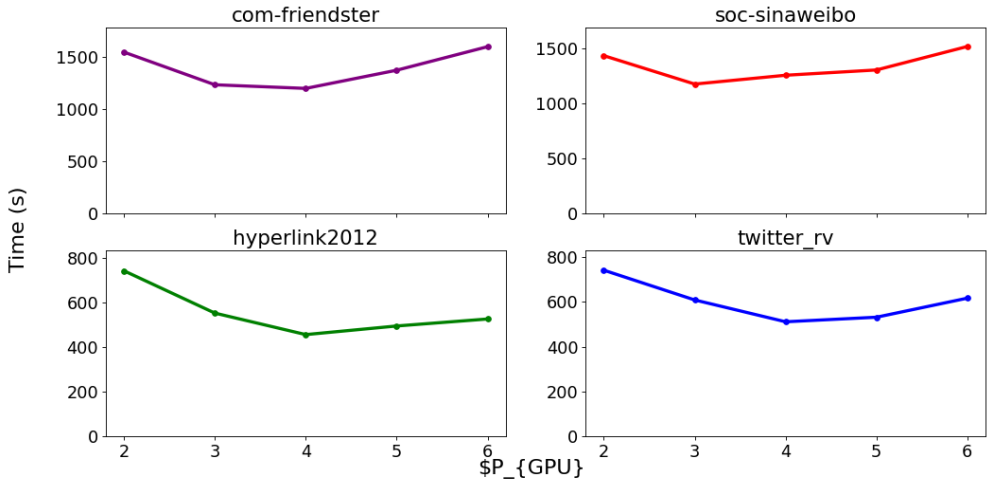

Figure 5.2 Embedding runtimes acheived from embedding the graphs in Table 5.1 with a range of values for P_{GPU}. This plot projects the results in Table 5.2.

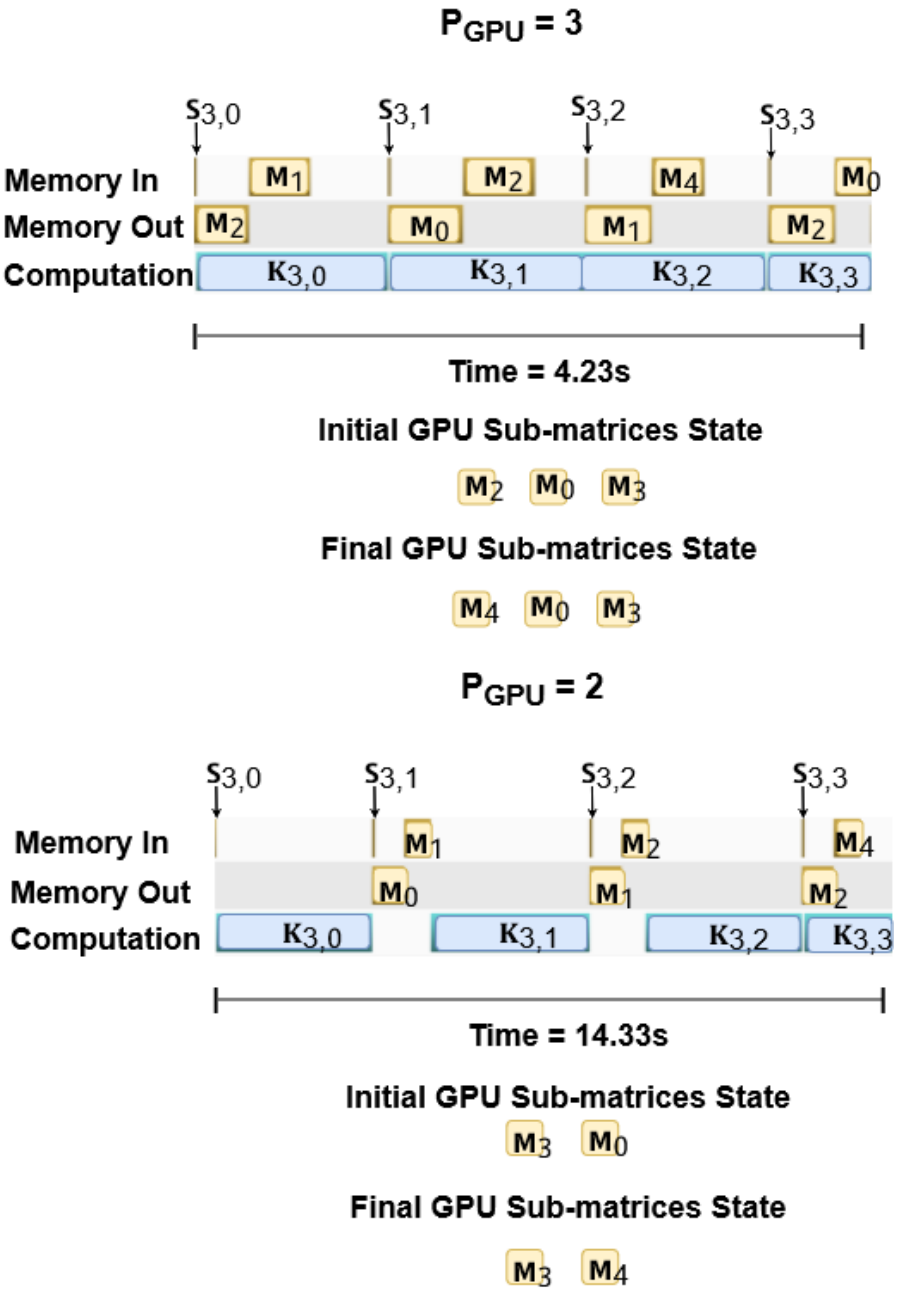

Figure 5.3 The figure shows an extract from the `nvvp` visual profiling tool profile of two embedding executions of the graph `hyperlink2012` on `Gandalf` with $S_{GPU} = 1$, $B = 5$, $C = 4$, and $T_s = 16$. However, the top execution is with $P_{GPU} = 3$ and the bottom execution is with $P_{GPU} = 2$. Figure 5.4 explains the notation used in this figure. The sub-matrices that are on the GPU at the beginning and at the end of each execution are shown below the time series. As shown, using $P_{GPU} = 3$ hides some of the latency of memory copies with computation.

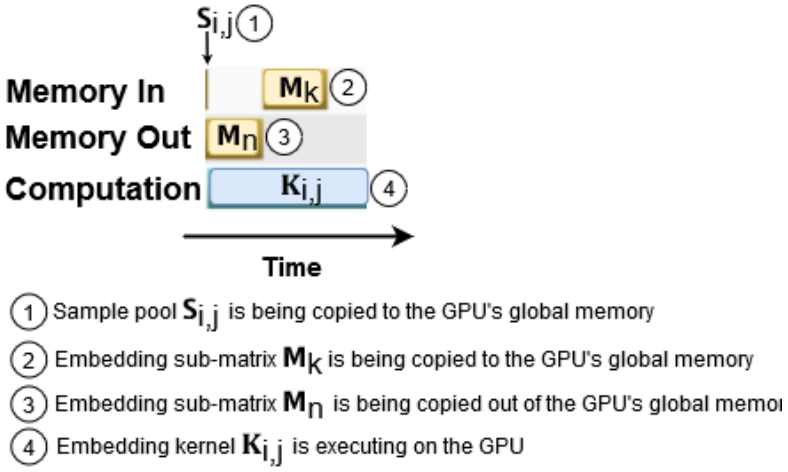

Figure 5.4 A time series figure is a snippet from the `nvvp` visual profiling tool used for profiling `CUDA` applications. From top to bottom, the rows in the time series show the memory copy jobs of data from the host to the GPU, the memory copy jobs from the GPU to the host, and the computation kernel jobs on the GPU.

5.1.3 Number of sample pool bins of S_{GPU}

Carrying out embeddings on the GPU requires positive samples to be prepared on the host and sent to the GPU. We allocate S_{GPU} sample pool bins on the GPU for embedding kernels to use during embedding, and we move sample pools to the GPU once they become needed as determined by the scheduling algorithm discussed in Chapter 2. We experiment with different values of S_{GPU} to gauge the effects of increasing the number of sample pools residing concurrently on the GPU. Table 5.3 shows the runtimes of the large-scale graphs in Table 5.1 with a range of values for S_{GPU}. The results show that the number of sample pools on the GPU does not affect the embedding speed directly. Instead, its influence on K, the number of sub-graphs in the graph partition, is what affects the embedding runtime. As S_{GPU} increases, more space on the GPU is required to store the sample pools, this leaves less space for embedding sub-matrices, and consequently, to splitting the graph into more sub-parts. The direct effect of increasing the number of parts in the partition is an increase in the number of kernels executed, and sub-matrices and sample pools copied.

An unexpected outcome from this experiment is that the change from $S_{GPU} = 1$ to $S_{GPU} = 2$ does not lead to any drastic speedup. A single sample pool bin on the GPU means that only a single embedding kernel can be executed on the GPU at any single time instance - removing kernel-kernel parallelization. That is why we anticipated

that increasing the number of sample pools from a single pool to multiple pools would improve runtime. Upon further experimentation, we found that having more than one sample pool bin resides on the GPU does allow for overlapping kernels; however, the bottleneck of the embedding becomes the sub-matrix swaps to and back from the GPU. Figure 5.5 illustrates these effects. We can make two main observations from the figure.

Kernel overlap: Even though there is a slight overlap between kernels, the kernels do not run in complete parallelism. For example, kernel $\mathbf{K}_{2,0}$ does not overlap $\mathbf{K}_{1,1}$ completely even though both the sub-matrices it requires and the sample pool are on the GPU. We believe that this because every single embedding kernel we dispatch completely saturates the GPU's *SM*s. To elaborate, every kernel we send is made up of 512 threads and 512 blocks. According to `nvvp`, the kernel's theoretical occupancy is 75%. When running on `Gandalf`, only three blocks can be active concurrently on a single *SM*. Since `Gandalf` is equipped with 15 multiprocessors, this means that the machine can only have $3 \times 15 = 45$ active blocks concurrently - less than 10% of the blocks of a single kernel. That is why we observe that only at the very end of a kernel, presumably when it has < 45 blocks left to process, does the next kernel begin processing and its blocks become active.

Compute idleness: Looking at Figure 5.5, we can see that there is a substantial period of idleness on the GPU after kernel $\mathbf{K}_{2,2}$ is finished executing. The idleness is not due to missing sample pools as it is clear that the samples of the next two kernels are already loaded into the GPU by the time $\mathbf{K}_{2,2}$ is finished executing. However, we can see that the sub-matrix M_3 is not on the GPU by the time $K_{2,2}$ finishes executing. This figure shows that even though increasing the number of sample pools does result in kernel-kernel overlapping, this benefit is lost due to the bottleneck of submatrix copies.

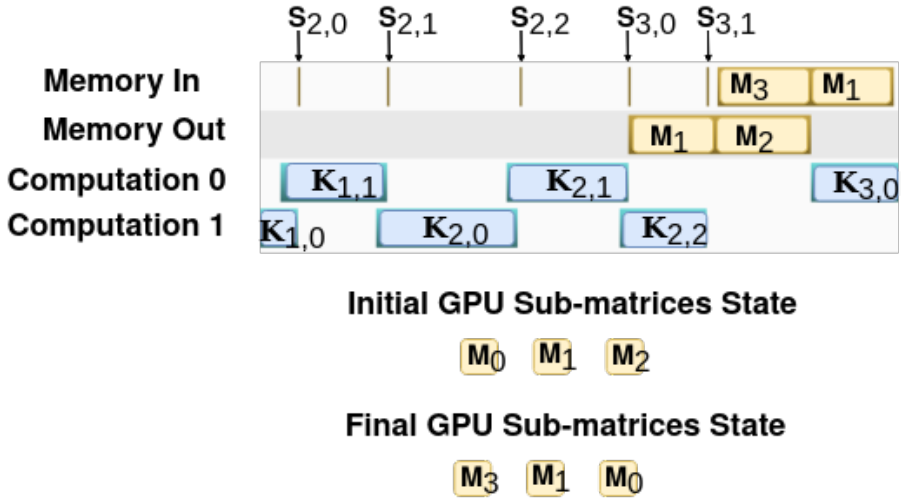

Figure 5.5 A time series of an embedding of the graph `twitter_rv` produced by the `nvvp` visual profiling tool. This execution was carried out on `Gandalf` and, $S_{GPU} = 2$, $P_{GPU} = 3$, $B = 5$, $C = 4$, and $T_s = 16$ were used as hyperparameters. The notation of this time series is explained in Figure 5.4. However, this figure contains two parallel computation streams, shown as "Computation 0" and "Computation 1". The sub-matrices which are on the GPU before the beginning of the time series are shown at the bottom of the figure, as well as the sub-matrices on the GPU after the time series is complete. A computation overlap between embedding kernels can be seen. In addition, a period in which the GPU is not doing any computation occurs between kernels $\mathbf{K}_{2,2}$ and $\mathbf{K}_{3,0}$.

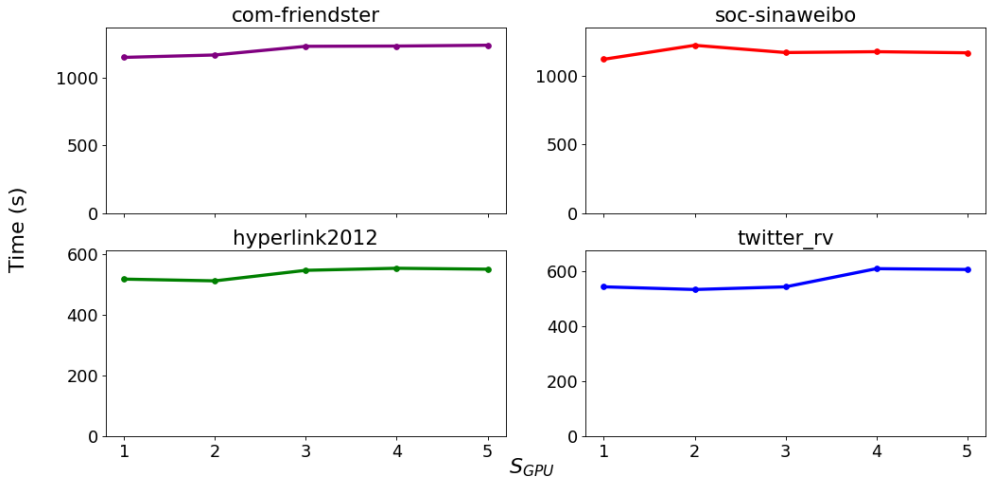

Figure 5.6 Visual representations of the effect of increasing S_{GPU} runtime of graph embedding.

5.1.4 Batch size of a single round B

Table 5.3 The effect of S_{GPU} on the the number of graph parts K and the embedding runtime for the large-scale graphs in Table 5.1. Experiments were run with $e = 100$, $P_{GPU} = 3$, $B = 5$, $C = 4$, and $T_s = 16$ and were done on `Gandalf`.

Graphs	S_{GPU}	K	**Time (s)**	**Graphs**	S_{GPU}	K	**Time (s)**
	1	8	1119.12		1	8	518.64
	2	9	1220.18		2	9	512.92
`soc-sinaweibo`	3	9	1167.75	`hyperlink2012`	3	9	547.58
	4	9	1170.69		4	9	554.36
	5	9	1165.90		5	9	551.29
	1	6	542.21		1	9	1147.37
	2	6	532.49		2	10	1164.96
`twitter_rv`	3	6	542.19	`com-friendster`	3	10	1228.60
	4	7	607.93		4	10	1230.56
	5	7	605.02		5	10	1236.65

Table 5.4 The effect of B on the the number of graph parts K, the runtime of embedding, as well as link prediction accuracy on the four large-scale graphs in Table 5.1. Experiments were run with $e = 100$, $P_{GPU} = 3$, $S_{GPU} = 4$, $C = 4$, and $T_s = 16$ and were carried out on `Gandalf`. Please note that these experiments do not incorporate coarsening; embedding is applied to the original level only.

Graphs	B	K	**Time (s)**	**AUCROC (%)**
	1	8	2107.46	98.02
`soc-sinaweibo`	5	9	842.73	99.78
	9	10	720.12	99.85
	1	8	1031.10	97.42
`hyperlink2012`	2	9	455.32	97.36
	3	10	413.71	97.09
	1	6	1103.09	93.97
`twitter_rv`	5	7	508.50	94.03
	9	6	436.28	90.00
	1	9	2476.13	93.18
`com-friendster`	2	10	874.43	91.96
	3	12	1026.57	90.36

The embedding procedure that we propose in this work carries out the embedding in multiple rounds, each round composed of large chunks of data copied into and out of the GPU. The batch size B controls the number of training epochs that are executed in a single round. A higher value of B means that there will be less embedding rounds, and therefore, fewer memory copies between the host and the GPU will be dispatched. The amount of computation carried out is not affected by B since the total number of samples generated by the algorithm is controlled using a global atomic counter; embedding ends once $e \times |V|$ positive samples are generated

and used for embedding. Table 5.4 shows the effect of tuning B on the runtime of the embedding, as well as the link prediction AUCROC scores. Figures 5.7 and 5.8 demonstrate the effect of increasing B on the link prediction scores and the runtime of embedding, respectively. We can see a consistent trend of the batch size being inversely proportional to the runtime of embedding, while either maintaining $AUCROC$ scores or degrading them slightly. On average, moving from $B = 1$ to $B = 5$ improves runtime by 58%, while only degrading the $AUCROC$ score by less than 1%. The improvement in runtime is natural since there will be a 5-fold decrease in rounds, which equals to a 5-fold decrease in the number of memory copy operations and kernel calls (despite the amount of embedding work not changing). These copy operations are of larger chunks of memory since sample pools become larger as B increases, but the effect is drastic nonetheless. The degradation in $AUCROC$, despite being almost negligible for some graphs, is expected. Increasing B means that more embedding updates are happening *in isolation* from the rest of the graph. To elaborate, when updates are happening in kernel $\mathbf{K}_{i,j}$, the vertices in this kernel will only be affected by updates to the small subset of the graph composed of V_i and V_j. Moving from $B = 5$ to $B = 9$ improves the runtime by a mere 2% while degrading the $AUCROC$ score by around 2%, and in the case of `twitter`, by almost 5%. Unlike the aforementioned jump from $B = 1$ to $B = 5$, going from $B = 5$ to $B = 9$ does not decrease the number of rounds by a large amount - when running 100 epochs, this amounts to reducing the number of rounds from 20 rounds to 11 rounds.

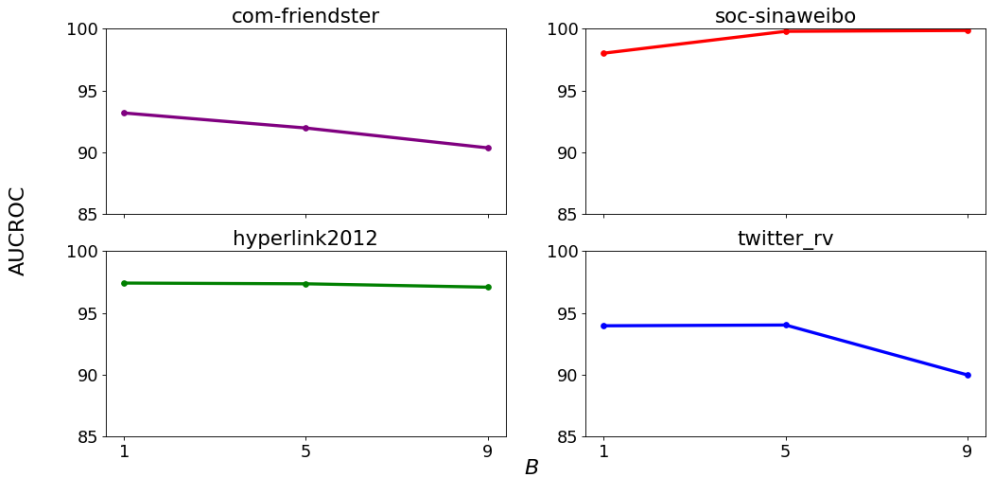

Figure 5.7 Effect of B on link prediction AUCROC scores for the large-scale graphs in Table 5.1. The data plotted in these line charts are from Table 5.4.

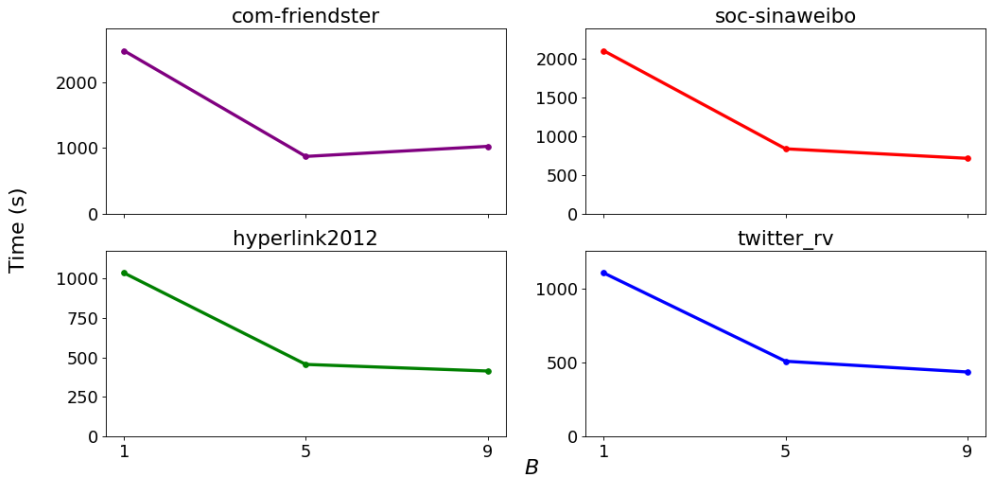

Figure 5.8 Effect of B on the embedding runtime of the large-scale graphs in Table 5.1. These plots use the data in Table 5.4.

5.1.5 Sampling time analysis

Sampling is a key part of the embedding procedure we propose in this work. We use teams of threads to carry out the embedding and ensure that their work is parallelized. We have two different levels of parallelism in the sampling work. **a)** we sample into multiple sample pools concurrently, and **b)** we utilize multiple threads to carry out the sampling procedure into a single sample pool.

Figure 5.9 shows, for all the large-scale graphs in Table 5.1, the speedup acquired from increasing the number of sampling threads after averaging the sampling time of more than 15,000 sample pools. It can be seen that the parallelism of sampling produces major speedups for all four graphs. However, `soc-sinaweibo` does not benefit from a greater number of threads as much as the other three; at 16 threads, `soc-sinaweibo` gains a speedup of 6.61×, while the other three gain $> 8\times$. We believe that this due to the different structure of the graph itself. To elaborate, we plot in Figure 5.10 histograms of the number of samples that are generated in a single pool for all four graphs. Unlike the other three graphs, `soc-sinaweibo` has a very high amount of sample pools with very few samples. In fact, 55% of sample pools have < 1000 samples in them. We think that since the sampling time of *most* pools is very small, the overhead of using additional threads becomes a bigger part of the sampling time.

Figure 5.11 compares the performance gain achieved by increasing the number of

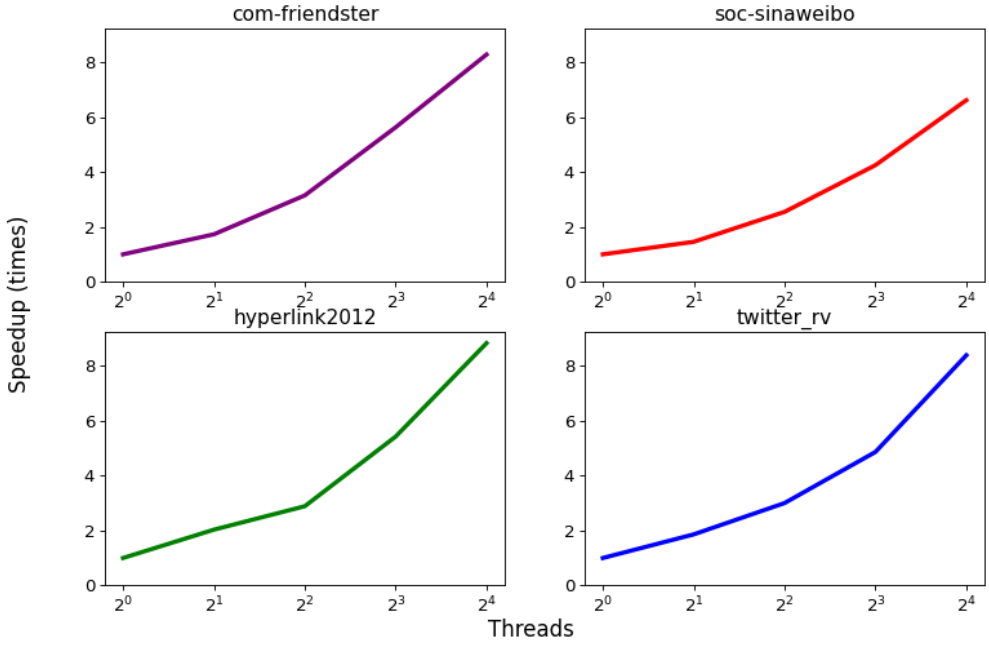

Figure 5.9 Sampling time as the number of sampling threads increases. Left: speedup acquired from increasing the number of sampling threads that are sampling into a single pool. Baseline is the time taken by a single thread. Right: the average time of sampling into sampling pools. Experiments were run with $C = 1$ and $B = 5$.

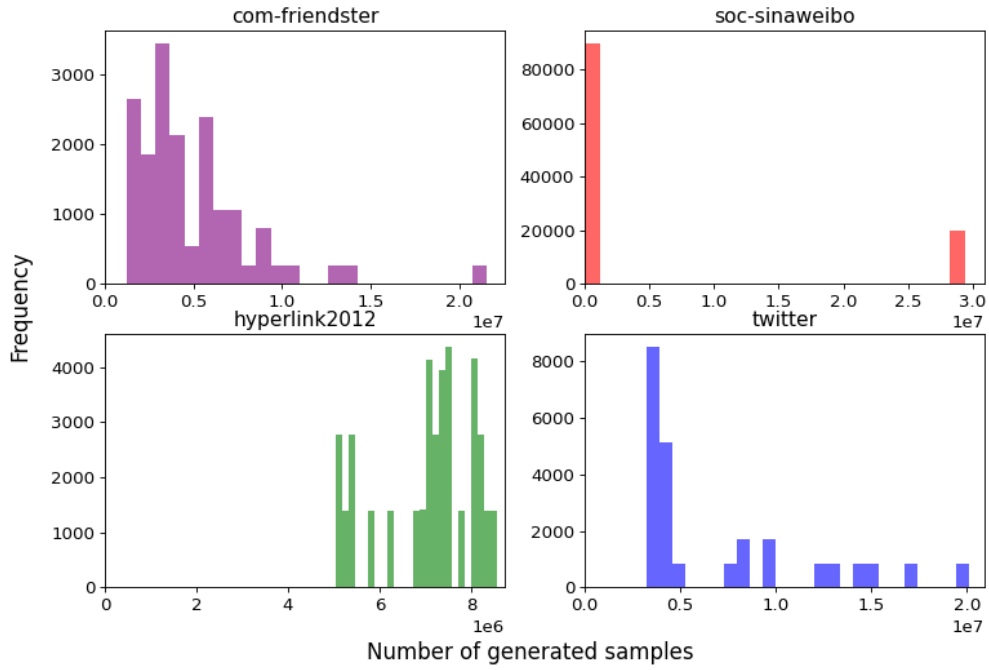

Figure 5.10 The frequency of different numbers of samples that are generated per sample pools modeled as histograms for the large-scale graphs in Table 5.1

sampling threads for a single concurrent sampler, and the one gained by increasing the number of concurrent samplers while giving each a single thread. The leftmost column shows the result of running 1 to 8 concurrent samplers with a matching number of threads, such that every concurrent sampler uses a single thread for sampling. The second column shows the result of running a single concurrent sampler, but increasing its team of threads linearly from 1 to 8. The third column is a superimposition of the two figures. For the graphs `com-friendster` and `hyperlink2012`, we can see that the effect of increasing the number of threads is almost identical; whether more threads were dispatched to sample a single pool, or threads were spread out to sample into multiple sample pools, a steady speedup in embedding is seen until a certain cut-off. At that point, the bottleneck of embedding is no longer the sampling. This trend can be seen in `twitter` as well, but the values are less stable and the trend is weaker. However, `soc-sinaweibo` shows a very peculiar behavior. It seems that using more threads in a single concurrent sampler is more important than spreading out the work into more samplers. This result can be explained by examining the histogram in Figure 5.12 which depicts the frequencies of sampling times of sample pools of the four large-scale graphs in Table 5.1 using $B = 5$. We notice that `soc-sinaweibo`'s sample times are very dispersed, with more than 90% of the pools being sampled into in less than 0.2 seconds, while a small portion takes more than twice that amount. We believe that this disparity in sampling times results in the difference between the effectiveness of increasing concurrent samplers versus increasing sampling threads per concurrent sampler that we see in Figure 5.11. To elaborate, this phenomenon happens because the highly-dense sample pools, which are only sampled using a single thread, are not being sampled fast enough and are holding back the embedding from progressing forward. For example, let's assume that sample pool $\mathbf{S}^h_{4,0,0}$ is highly dense, and requires a long amount of time to be sampled. While it is being sampled by a single thread, the following three sample pools, $\mathbf{S}^h_{4,1,0}$ $\mathbf{S}^h_{4,2,0}$, and $\mathbf{S}^h_{4,3,0}$ are smaller and have already finished sampling. When the sample copy task node of $\mathbf{S}^h_{4,0,0}$ is running, it will not terminate until $\mathbf{S}^h_{4,0,0}$ has been populated. Since successive sample pool tasks are dependent, this will prevent the following two sample pools from being copied to the GPU despite being full and ready to execute. This way, the sample pool $\mathbf{S}_{4,0,0}$ hindered the progress of the entire embedding. That is why we recommend users with scarce thread resources to use more sampling threads and less concurrent samplers, so as to avoid this effect.

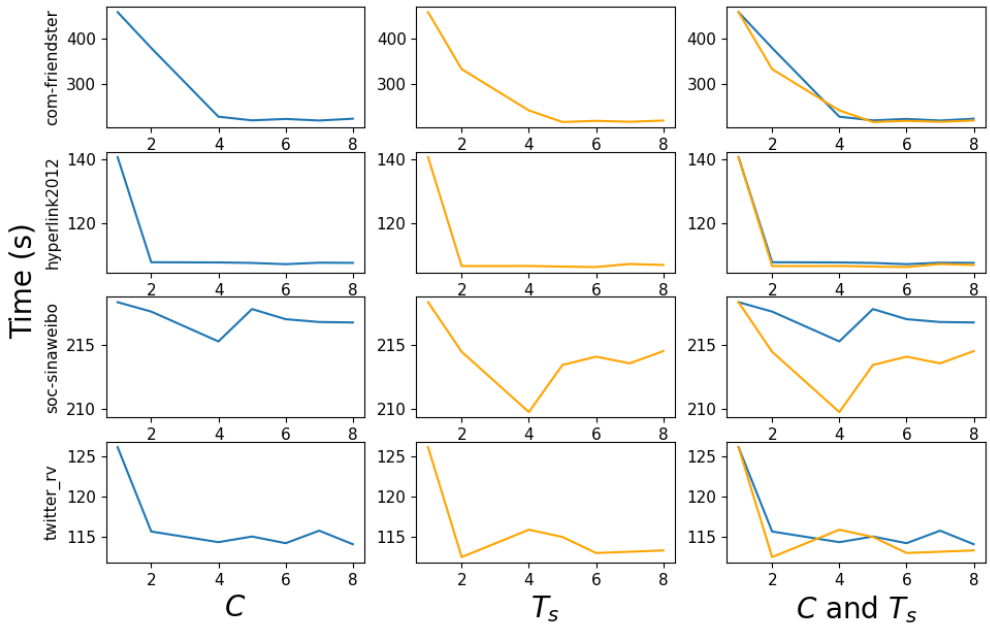

Figure 5.11 The figure depicts the distribution of sampling threads over multiple concurrent pool sampling tasks versus using more threads to sample into a single pool. All experiments were run using $e = 20$, $P_{GPU} = 3$, $S_{GPU} = 4$, and $B = 5$. The leftmost column depicts increasing the number of concurrent samplers one-to-one with the number threads so that every concurrent sampler receives a single thread. The middle column shows experiments that were run with a single concurrent sampler, but with an increasing number of threads, which means that all the threads in the pool will work on a single sample pool. The rightmost column is a superimposition of the graphs in the other two columns.

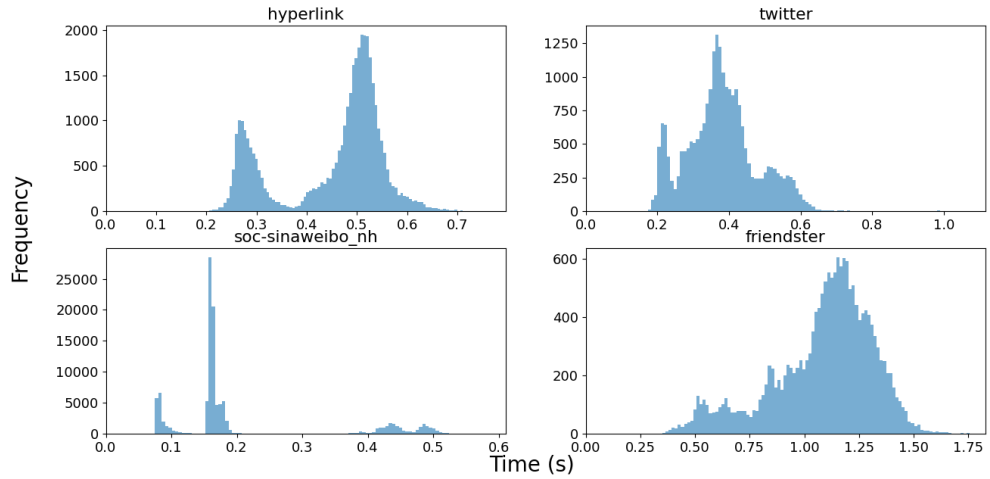

Figure 5.12 The frequencies of the time taken to sample pools using a single thread and $B = 5$.

5.2 Embedding Quality

This section examines GOSH in terms of its ability to produce high-quality embeddings in terms of their link prediction quality as well as its speed of embedding.

Baseline algorithms and tools: We evaluate the performance of the proposed embedding algorithm by comparing its runtime and embedding results to these state-of-the-art embedding algorithms.

- VERSE: a multi-core graph embedding algorithm (Tsitsulin et al., 2018) which employs the similarity measure-based sampling procedure we use in our tool. In our experiments, we use the PPR (Page et al., 1999) similarity measure for sampling as recommended by the authors for the best results. We run VERSE with three different epoch values, $e = 600, 1000$ and $e = 1400$m and use the run with the best AUCROC for each experiment. In addition, we set the learning rate to $lr = 0.0025$.

- GRAPHVITE: the state-of-the-art multi-GPU graph embedding algorithm at the time this work was published. Graphvite can use a variety of underlying sampling schemas including LINE (Tang et al., 2015), DeepWalk (Perozzi et al., 2014), and Node2Vec (Grover & Leskovec, 2016). Graphvite, however, suffers from a memory limitation that prevents it from embedding graphs whose memory requirement surpasses that of the GPU. That is why we were not able to use it to embed the large-scale graphs in Table 5.1. In the experiments in this section, we use two different configurations for GRAPHVITE. A slow configuration with 600 epochs and a fast configuration with 1000 epochs. We found that GRAPHVITE's AUCROC scores saturate at 1000 epochs and running more epochs does not benefit the AUCROC scores while reducing the runtime. We use LINE as the base embedding method and leave the remaining hyper-parameters to their default values.

- MILE: a coarsening based approach to embedding that iteratively coarsens graphs, embeds the final level, and projects the embedding to the original graph using a deep-learning-based approach. We set the base embedding method to DEEPWALL and the refinement method to MD-GCN. We also set the number of levels of coarsening levels to 8 and the learning rate to its default of $lr = 0.001$. It should be noted that MILE does not allow for the number of embedding epochs to be configured.

For GOSH, we use three different embedding configurations which we call *fast*, *nor-*

mal, and *slow*. These configurations are shown in Table 5.5. The parameter p in the table describes the smoothing ration in GOSH, which controls how the number of embedding epochs is distributed across coarsening levels, with smaller smoothing ratios resulting in more work assigned to coarser levels compared to higher values of p, resulting in faster but less fine-tuned embedding. Along with the variation in p across configurations, we vary the number of epochs and the learning rate. It should be pointed out that we use different epoch counts for medium- and large-scale graphs. We reduce the number of epochs for larger graphs as we found it experimentally sufficient to acquire satisfactory AUCROC scores. We use a fourth configuration option for GOSH: GOSH-*no-Coarse*. This version does not do any coarsening on the graph and embeds the original graph directly.

In the following experiments, we modify the definition of a single epoch of work during the embedding of the graph $G = (V, E)$ to become the execution of $|E|$ positive samples. This is to match the definition of GRAPHVITE for fairness in our experiments.

Table 5.5 Configurations of GOSH used in the experiments in Section 5.2. The three configurations vary in the amount of work they do during embedding and demonstrate the flexibility of GOSH.

Configuration	p	lr	e_{normal}	e_{large}
Fast	0.1	0.050	600	100
Normal	0.3	0.035	1000	200
Slow	0.5	0.025	1400	300
No coarsening	-	0.045	1000	200

5.2.1 Experiments on Embedding Quality

Table 5.6 shows the results of embedding the large-scale graphs in Table 5.1, and Tables 5.7 and 5.8 show the results of embedding the medium-scale graphs. In the aforementioned tables, we show the speedup achieved by each model as compared to VERSE. All large-scale graphs use the large-graph embedding schema proposed in this work to embed one to three of the coarsening levels that GOSH produces. For example, the first three coarsening levels for the graph `com-friendster` do not fit in the GPU (require more than 12GB of memory to store the embedding matrix and the graph information). It is important to note that the GOSH-NoCoarse configuration in Table 5.6 is different from that of the medium-scale graphs tables (Tables 5.7 and 5.8). We only used $e = 10$ and a learning rate of 0.04. This is because we

found that very few epochs are needed when training such large-scale graphs since the number of samples executed pre epoch is $|E| \times e$. The graph with the least amount of edges, `soc-sinaweibo`, would execute more than 2.6 billion samples in ten epochs.

None of the four large-scale graphs were embedded with GRAPHVITE since neither of our servers was equipped with GPUs that had sufficient memory capabilities to embed these large scale graphs. However, Zhu et al. (2019) report that GRAPHVITE embeds `hyperlink2012` in 5.36 hours using four Tesla P100 GPUs and achieves 94.3% link prediction AUCROC. GOSH-NoCoarse, meanwhile, can achieve an AUCROC of 98.44% after only 5 minutes of embedding, producing a speedup of 62.91×. In addition, they report that GRAPHVITE embeds the graph `com-friendster` in 20.3 hours on the same setup mentioned earlier. GOSH-NoCoarse, on the other hand, embeds the same graph in 17 minutes, which amounts to a speedup of 67.51× that of GRAPHVITE. The graph `com-friendster` was not assessed in terms of AUCROC in Zhu et al. (2019), and so we could not compare the embedding qualities for this graph.

MILE could not finish any embedding jobs due to timing out during the execution. VERSE timed out for three out of the four large-scale graphs. It finished execution successfully on `soc-sinaweibo` and it was able to surpass the AUCROC score of GOSH-NoCoarse by 0.03%. However, GOSH-NoCoarse is 127.33× faster than VERSE.

5.2.1.1 Medium-scale graphs

Table 5.7 and 5.8 show the results of embedding the medium-scale graphs in Table 5.1. These tables show the effectiveness of the embedding algorithm proposed in Section 3.2, specifically, the results of the configuration GOSH-NoCoarse demonstrate that even without any coarsening, the parallelism optimizations we introduce in Section 3.2.1 produce noticeable speedups while maintaining the accuracy of the embedding. On average, GOSH-NoCoarse is 15.64× faster than the multi-core CPU implementation of VERSE. These results demonstrate the power of coarsening and its potential as an accelerator for graph embedding, even compared to GRAPHVITE, the state-of-the-art GPU graph embedding algorithm. Looking at tables 5.7 and 5.8, we make the following observations:

- GOSH can carry out extremely fast embeddings with high accuracy using the GOSH-fast configuration. The speedup it can achieve over VERSE is up to three orders of magnitude, and its resultant accuracies do not show a loss of more than 2% (1.64% on average). In addition, it is on average 23.44× faster than GRAPHVITE, the state-of-the-art GPU embedding algorithm. Also, it surpasses MILE in both accuracy and runtime.

- The flexibility that GOSH possesses in terms of the trade-off between speed and accuracy is clear with its *normal* and *slow* configurations. The gap between VERSE and GOSH in terms of the AUCROC score goes from 1.64% to 0.76% when using GOSH-normal, and from 0.76% to 0.24% when using GOSH-slow. All the while maintaining its edge in embedding speed over VERSE, and MILE.

- We find that, compared to GOSH-slow, GRAPHVITE is able to produce embeddings at comparable speeds. However, when comparing the best results of GOSH and GRAPHVITE on each graph, we find that GOSH surpasses GRAPHVITE in 4/8 graphs, and is, on average, 5.2× faster than GRAPHVITE.

Table 5.6 Link prediction results on large graphs. Every data point is the average of 6 experiments. GRAPHVITE and MILE fail to embed any of the graphs due to excessive memory usage or an execution time larger than 12 hours. $\tau = 16$ threads used for both VERSE and MILE. These experiments were executed on the `Nebula` server. In this table, the GOSH-NoCoarse row was run with fewer epochs than the other runs and with the learning rate $lr = 0.04$ as it converges to very high AUCROC scores very early on.

Graph	**Algorithm**	**Time (s)**	**Speedup**	**AUC ROC** (%)
`hyperlink2012`	VERSE	Timeout	-	-
	GOSH-fast	201.02	-	87.60
	GOSH-normal	724.09	-	97.20
	GOSH-slow	1676.93	-	98.00
	GOSH-NoCoarse ($e = 10$)	306.71	-	98.44
`soc-sinaweibo`	VERSE	20397.79	1.00×	99.89
	GOSH-fast	48.88	417.30×	70.27
	GOSH-normal	352.86	57.81×	97.00
	GOSH-slow	759.85	26.84×	99.37
	GOSH-NoCoarse ($e = 10$)	160.28	127.26×	99.86
`twitter_rv`	VERSE	Timeout	-	-
	GOSH-fast	261.08	-	91.78
	GOSH-normal	994.46	-	97.36
	GOSH-slow	2128.70	-	98.50
	GOSH-NoCoarse ($e = 10$)	740.22	-	98.27
`com-friendster`	VERSE	Timeout	-	-
	GOSH-fast	680.33	-	85.17
	GOSH-normal	2720.82	-	93.40
	GOSH-slow	5000.96	-	94.98
	GOSH-NoCoarse ($e = 10$)	1068.16	-	97.84

Table 5.7 Link prediction results on medium-scale graphs. Every data-point is the average of 15 results. VERSE and GOSH uses $\tau = 16$ threads. MILE is a sequential tool. Both GRAPHVITE and GOSH use the same GPU. The speedup values are computed based on the execution time of VERSE. These experiments were performed on `Nebula`.

Graph	**Algorithm**	**Time (s)**	**Speedup**	**AUCROC**(%)
`com-dblp`	VERSE	247.99	1.00×	**97.82**
	MILE	136.65	1.81×	97.65
	GRAPHVITE-fast	13.97	17.70×	97.80
	GRAPHVITE-slow	19.93	12.40×	**98.08**
	GOSH-fast	0.72	344.43×	96.45
	GOSH-normal	2.08	119.23×	97.38
	GOSH-slow	3.84	64.58×	97.63
	GOSH-NoCoarse	29.97	8.27×	93.31
`com-lj`	VERSE	12502.72	1.00×	**98.86**
	MILE	3948.62	3.17×	80.19
	GRAPHVITE-fast	373.58	33.47×	98.04
	GRAPHVITE-slow	644.43	19.40×	98.33
	GOSH-fast	16.27	768.45×	96.82
	GOSH-normal	55.01	227.28×	98.33
	GOSH-slow	153.72	81.33×	**98.46**
	GOSH-NoCoarse	675.25	18.52×	98.32
`wiki-topcats`	VERSE	8709.48	1.00×	**99.31**
	MILE	4953.68	1.76×	86.04
	GRAPHVITE-fast	310.47	28.05×	96.42
	GRAPHVITE-slow	544.06	16.01×	96.28
	GOSH-fast	11.34	768.03×	98.13
	GOSH-normal	40.76	213.68×	98.33
	GOSH-slow	93.86	92.79×	**98.50**
	GOSH-NoCoarse	549.65	15.85×	98.51
`soc-pokec`	VERSE	9182.53	1.00×	**98.32**
	MILE	2848.78	3.22×	85.75
	GRAPHVITE-fast	370.73	24.77×	**97.42**
	GRAPHVITE-slow	607.07	15.13×	97.37
	GOSH-fast	16.34	561.97×	96.34
	GOSH-normal	54.66	167.99×	96.49
	GOSH-slow	131.06	70.06×	96.67
	GOSH-NoCoarse	598.95	15.33×	97.28

Table 5.8 A continuation of Table 5.7.

com-amazon	VERSE	216.18	1.00×	97.71
	MILE	146.29	1.48×	**98.14**
	GRAPHVITE-fast	12.45	17.36×	97.40
	GRAPHVITE-slow	16.84	12.83×	97.82
	GOSH-fast	0.69	313.30×	97.20
	GOSH-normal	1.88	114.99×	98.29
	GOSH-slow	3.59	60.22×	**98.43**
	GOSH-NoCoarse	24.60	8.79×	90.13
com-orkut	VERSE	45994.93	1.00×	**98.65**
	MILE	11904.31	3.86×	90.38
	GRAPHVITE-fast	1246.38	36.90×	98.02
	GRAPHVITE-slow	2199.25	20.91×	**98.05**
	GOSH-fast	43.30	1062.24×	97.35
	GOSH-normal	185.12	248.46×	97.63
	GOSH-slow	487.33	94.38×	97.69
	GOSH-NoCoarse	2301.89	19.98×	97.64
youtube	VERSE	1365.36	1.00×	**98.04**
	MILE	1328.62	1.03×	94.17
	GRAPHVITE-fast	63.90	21.37×	97.07
	GRAPHVITE-slow	104.76	13.03×	97.45
	GOSH-fast	2.76	494.70×	96.16
	GOSH-normal	7.15	190.96×	97.78
	GOSH-slow	15.32	89.12×	**97.93**
	GOSH-NoCoarse	158.60	8.61×	97.16
soc-LiveJournal	VERSE	14965.76	1.00×	**97.61**
	MILE	6210.58	2.41×	80.84
	GRAPHVITE-fast	745.33	20.08×	99.23
	GRAPHVITE-slow	1209.95	12.37×	**99.31**
	GOSH-fast	29.74	503.22×	98.58
	GOSH-normal	112.72	132.77×	98.87
	GOSH-slow	183.64	81.50×	98.76
	GOSH-NoCoarse	1348.74	11.10×	98.88

6. CONCLUSION

In this thesis, we presented a graph embedding algorithm that utilizes a single GPU to embed very large graphs while bypassing the GPUs memory limitation. The algorithm uses a sampling-based approach of embedding adopted from (Tsitsulin et al., 2018) that allows the use of any similarity measure, providing additional generalizability to the presented algorithm. The GPU embedding kernel used utilizes the GPU's architecture to provide fast and accurate embeddings.

Overcoming the memory limitations of the GPU is achieved by partitioning the embedding matrix into smaller sub-matrices and moving them to and back from the GPU to carry out embedding updates. This allows utilizing the accelerating capabilities of the scarce resource that is GPUs.

Sampling is carried out on the CPU and samples are sent to the GPU as they become needed without needing any global synchronization; sampling is done on the CPU in parallel with the GPU embedding, minimizing the idle time on the GPU that could result from waiting for samples.

We presented the Directed Acyclic Graph (DAG) execution model used to carry out the embedding. This model abstracts the embedding into smaller tasks, and is carefully designed with an execution graph in which a task does not synchronize with any work on the CPU or the GPU unless it is directly dependant on it.

We provided analytics of the algorithm and the sensitivity to the parameters demonstrated by the algorithm with regards to some of its key elements. We also present the embedding accuracy results of Gosh, which is a tool built with the algorithm presented in this work. We find our algorithm can achieve high AUCROC scores of large-scale graphs with millions of vertice and billions of edges in a fraction of the time. Using our algorithm the `com-friendster` which has 60 million vertices and over one billion edges is embedded in 17 minutes and an AUCROC score of 97.84% was achieved.

6.1 Future work

We plan to add multi-GPU capabilities to the algorithm and exploit the DAG structure to provide global-synchronization free embedding for the GPUs. In addition, we find from our experiments in Chapter 5 that the bottleneck of execution is the copy of sub-matrices. That is why we are considering different methods to reduce the number of sub-matrix copies, such as keeping the most updated vertices' embeddings on the GPU.

BIBLIOGRAPHY

Adhikari, B., Zhang, Y., Ramakrishnan, N., & Prakash, B. A. (2018). Sub2vec: Feature learning for subgraphs. In Phung, D., Tseng, V. S., Webb, G. I., Ho, B., Ganji, M., & Rashidi, L. (Eds.), *Advances in Knowledge Discovery and Data Mining*, (pp. 170–182)., Cham. Springer International Publishing.

Akyildiz, T. A., Aljundi, A. A., & Kaya, K. (2020). Gosh: Embedding big graphs on small hardware. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA. Association for Computing Machinery.

Belkin, M. & Niyogi, P. (2002). Laplacian eigenmaps and spectral techniques for embedding and clustering. In T. G. Dietterich, S. Becker, & Z. Ghahramani (Eds.), *Advances in Neural Information Processing Systems 14* (pp. 585–591). MIT Press.

Cai, H., Zheng, V. W., & Chang, K. C. (2018). A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, *30*(9), 1616–1637.

Collobert, R. & Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, (pp. 160–167)., New York, NY, USA. Association for Computing Machinery.

Fawcett, T. (2006). An introduction to roc analysis. *Pattern Recogn. Lett.*, *27*(8), 861–874.

Gao, Z., Fu, G., Ouyang, C., Tsutsui, S., Liu, X., & Ding, Y. (2018). edge2vec: Learning node representation using edge semantics. *CoRR*, *abs/1809.02269*.

Goyal, P. & Ferrara, E. (2017). Graph embedding techniques, applications, and performance: A survey. *CoRR*, *abs/1705.02801*.

Grover, A. & Leskovec, J. (2016). node2vec: Scalable feature learning for networks.

Hu, R., Aggarwal, C. C., Ma, S., & Huai, J. (2016). An embedding approach to anomaly detection. In *2016 IEEE 32nd Int. Conf. on Data Eng. (ICDE)*, (pp. 385–396).

Jeh, G. & Widom, J. (2002). Simrank: A measure of structural-context similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, (pp. 538–543)., New York, NY, USA. Association for Computing Machinery.

Kipf, T. & Welling, M. (2016). Variational graph auto-encoders. *ArXiv*, *abs/1611.07308*.

Kipf, T. & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. *ArXiv*, *abs/1609.02907*.

Lerer, A., Wu, L., Shen, J., Lacroix, T., Wehrstedt, L., Bose, A., & Peysakhovich, A. (2019). Pytorch-biggraph: A large-scale graph embedding system.

Leskovec, J. & Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

Liang, J., Gurukar, S., & Parthasarathy, S. (2018). Mile: A multi-level framework for scalable graph embedding.

Liben-Nowell, D. & Kleinberg, J. (2003). The link prediction problem for social networks. In *Proc. 12th Int. Conf. on Information and Knowledge Management*,

CIKM '03, (pp. 556–559)., NY, USA. ACM.

Meusel, R. (2015). The graph structure in the web – analyzed on different aggregation levels. *Journal of Web Science*, *1*, 33–47.

Mikolov, T., Chen, K., Corrado, G. S., & Dean, J. (2013). Efficient estimation of word representations in vector space. *CoRR*, *abs/1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *ArXiv*, *abs/1310.4546*.

Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., & Bhattacharjee, B. (2007). Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, (pp. 29–42)., New York, NY, USA. ACM.

Mnih, A. & Hinton, G. (2008). A scalable hierarchical distributed language model. In *Proceedings of the 21st International Conference on Neural Information Processing Systems*, NIPS'08, (pp. 1081–1088)., Red Hook, NY, USA. Curran Associates Inc.

Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., & Jaiswal, S. (2017). graph2vec: Learning distributed representations of graphs. *CoRR*, *abs/1707.05005*.

Ou, M., Cui, P., Pei, J., Zhang, Z., & Zhu, W. (2016). Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (pp. 1105–1114)., New York, NY, USA. Association for Computing Machinery.

Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab. Previous number = SIDL-WP-1999-0120.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Perozzi, B., Al-Rfou, R., & Skiena, S. (2014). Deepwalk: Online learning of social representations. In *Proc. 20th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, KDD '14, (pp. 701–710)., NY, USA. ACM.

Rossi, R. A. & Ahmed, N. K. (2015). The network data repository with interactive graph analytics and visualization. In *AAAI*.

Roweis, S. T. & Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, *290*(5500), 2323–2326.

Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., & Mei, Q. (2015). Line: Large-scale information network embedding. In *Proc. 24th Int. Conf. on World Wide Web*, (pp. 1067–1077). IW3C2.

Tsitsulin, A., Mottin, D., Karras, P., & Müller, E. (2018). Verse: Versatile graph embeddings from similarity measures. In *Proc. World Wide Web Conference*, WWW '18, (pp. 539–548)., Republic and Canton of Geneva, CHE. IW3C2.

Wang, D., Cui, P., & Zhu, W. (2016). Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (pp. 1225–1234)., New York, NY, USA. Association for Computing Machinery.

Wang, Q., Mao, Z., Wang, B., & Guo, L. (2017). Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, *29*(12), 2724–2743.

Xiao, H., Huang, M., Hao, Y., & Zhu, X. (2015). Transg : A generative mixture model for knowledge graph embedding. *ArXiv*, *abs/1509.05488*.

Zang, C., Cui, P., & Faloutsos, C. (2016). Beyond sigmoids: The nettide model for social network growth, and its applications. (pp. 2015–2024).

Zhu, Z., Xu, S., Tang, J., & Qu, M. (2019). Graphvite: A high-performance cpu-gpu hybrid system for node embedding. In *The World Wide Web Conference*, WWW '19, (pp. 2494–2504)., NY, USA. ACM.