GPU-Accelerated Approaches for Graph Data Processing

by

Yuanhang Yu

A THESIS SUBMITTED IN FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Australian Artificial Intelligence Institute (AAII)
Faculty of Engineering and Information Technology (FEIT)
University of Technology Sydney (UTS)

August, 2023

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, Yuanhang Yu, declare that this thesis is submitted in fulfilment of the requirements for the award of Doctor of Philosophy, in the Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Signature: Production Note:
Signature removed prior to publication.

Date: 10/11/2023

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my principal supervisor, Prof. Ying Zhang, for his unwavering support, guidance, and mentorship throughout this journey. His wisdom, patience, and commitment to my academic and personal growth have been the bedrock upon which this thesis stands. In addition to the academic and professional guidance, I am deeply grateful for the personal care and concern he has shown towards my well-being. Our bond transcends that of a typical student-supervisor relationship. I am proud to call him not just my supervisor but also a true friend.

In the same breath, I owe a profound debt of thanks to my co-supervisors, Prof. Lu Qin and Dr. Dong Wen. Their invaluable perspectives, constructive critiques, and unwavering support have guided my research trajectory and honed my ideas, ensuring the robustness and pertinence of my endeavors. Their expertise has been a beacon, illuminating the intricacies of my research area. At the same time, I express my deepest respect to Prof. Xuemin Lin and Prof. Wenjie Zhang for their support. They are outstanding scholars. Their dedication to research is truly inspirational.

Thirdly, I owe profound gratitude to Dr. Longbin Lai, Dr. Zhengping Qian, and Prof. Zengfeng Huang. Working alongside them has been a gratifying experience. They have consistently guided and supported me, offering invaluable insights that greatly benefited my research. I would also like to extend my heartfelt gratitude to Mr. Peilun Yang, Dr. Hanchen Wang and Mr. Zhibin Wang. We have forged a deep and enduring bond during my Ph.D. candidate. I am immensely grateful for their care, support, and the countless moments of camaraderie we shared. Their kindness and encouragement make this journey all the more fulfilling and memorable.

Special thanks to Prof. Bingsheng He and another anonymous examiner for their insightful comments and suggestions. Their expertise and constructive feedback have immensely enriched this thesis, rendering it more polished and coherent.

Finally, my heartfelt appreciation goes to my family. To my father Jianguo Yu and my mother Guixia Zheng, who instilled in me the value of perseverance and the love for learning, their unwavering faith in my abilities pushed me to reach greater heights. And to my wife, Liping Niu, whose love, understanding, and sacrifices have been my anchor during the highs and lows of this endeavor. You have been my strength, and this achievement is as much yours as it is mine.

ABSTRACT

Graphs become increasingly prevalent in a variety of domains such as bioinformatics, social networks, etc., the size of graph is expanding significantly at the same time. The need to process graph data in an acceptable time has gained prominence. As a device equipped with massive computational resources, there is a growing interest in the GPU for general-purpose computation. In this thesis, we focus on three common problems in graph data processing, which share the following two characteristics: (1) Given the vast data and inherent complexity, CPU performance is unsatisfactory, prompting the need for high-parallelism hardware acceleration; (2) Irregular data complicates optimal use of GPU memory and threads. With these concerns in mind, we propose GPU-based approaches for these problems.

The first problem is matrix factorization, widely used in in graph embedding and recommendation system. Matrix factorization, due to its data-intensive nature from large matrices, requires the massive parallel processing capabilities of both CPU and GPU in heterogeneous multiprocessors. The challenge lies in balancing the workload across these working units. We design a novel strategy to divide the input matrix into non-uniform blocks for optimal GPU resource utilization and workload balance. This is achieved by a tailored cost model that can gauge the performance of working units on blocks. Coupled with dynamic scheduling, our approach demonstrates enhanced performance with high training quality on real-world datasets through extensive experiments.

The second problem is approximate k nearest neighbors search, widely applied in community detection, anomaly detection, etc. Graph-based methods with outstanding performance are hindered by distance computations among high-dimensional data points, highlighting the potential for GPU acceleration. Existing work on the GPU decomposes the search algorithm and accelerates distance computation by leveraging multiple threads. We notice that it still suffers from the high expenses of data structure operations performed by single thread. Driven by this motivation, we design a novel GPU-friendly search approach, which can fully harness multiple threads at all critical steps in search. In addition, we propose a GPU-accelerated algorithm for building high-quality graphs with efficient parallelism. Extensive experiments on benchmark high-dimensional datasets demonstrate the outstanding performance of our algorithms in both ANN search and graph construction.

The third problem is subgraph matching, fundamental in a wide range of research fields such as protein interaction analysis and social network analysis. As an NP-hard problem, subgraph matching is inherently challenging. This offers a chance for GPU acceleration. Existing work employs a lightweight filtering method to make a trade-off between high parallelism and efficient pruning performance. We notice that its pruning performance falls short of expectations when the labels on graphs are not diverse. Inspired by this, we propose a novel GPU-friendly filtering approach with strong pruning performance. Additionally, we propose a 1^*-phase output scheme to reduce space consumption and optimize write transactions during enumeration. Together with a pipeline method, our approach outperforms existing work on real-world datasets through extensive experiments.

PUBLICATIONS

- **Yu, Y.**, Wen, D., Zhang, Y., Wang, X., Zhang, W., Lin, X. (2021, April). "Efficient matrix factorization on heterogeneous CPU-GPU systems." *ICDE 2021* (**Chapter 3**)
- **Yu, Y.**, Wen, D., Zhang, Y., Qin, L., Zhang, W., Lin, X. (2022, May). "GPU-accelerated Proximity Graph Approximate Nearest Neighbor Search and Construction." *ICDE 2022* (**Chapter 4**)
- **Yu, Y.**, Wen, D., Lai, L., Qian, Z., Qin, L., Zhang, Y. "A GPU-accelerated Subgraph Matching algorithm." *In Submission* (**Chapter 5**)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we briefly introduce the background of our research problems and explain the motivations. Following this, contributions of our approaches are summarized and a structured overview is provided.

1.1 Background

1.1.1 Matrix factorization

As a common technique in machine learning and data mining, matrix factorization (MF) has been widely applied in many areas, such as graph embedding [23], recommendation system [70, 43], social network analysis [68, 7], web mining [86] and word embedding [82].

Given a sparse matrix $R \in \mathbb{R}^{m \times n}$, MF aims to decompose R into two dense matrices $P \in \mathbb{R}^{m \times k}$ and $Q \in \mathbb{R}^{k \times n}$ such that the following condition is satisfied:

$$R \approx P \times Q$$

Here, the latent factor k is far less than m and n such that we can represent the sparse matrix with low-dimensional dense data effectively.

The existing methods for solving MF can be roughly classified into three broad categories, namely, alternating least squares (ALS) [58], coordinate descent (CD) [113] and stochastic gradient descent (SGD) [30]. Among these methods, SGD-based algorithms have received the most attention due to their algorithmic simplicity and effectiveness [109]. In order to achieve convergence during training, SGD-based algorithms perform multiple iterations. They terminate when either the training results have converged or the user-specified number of iterations have been completed. During each iteration, the gradient of every element in the sparse matrix is computed and the corresponding vectors in the result matrices are updated. For the purpose of accelerating this process, several parallel SGD-based algorithms have been proposed on different system settings such as GPUs (e.g., CuMF_SGD [109]), multi-core CPUs (e.g., FPSGD [121]) and distributed systems (e.g., NOMAD [115]). The main idea behind these methods is to partition the sparse matrix into a set of uniform blocks. During each iteration, multiple working units (e.g., GPU, node or thread) are assigned a group of mutually independent blocks to update. Here, two blocks are regarded as independent if they have no shared row or column. This resolves any writing conflicts arising from updates.

1.1.2 *k* nearest neighbors search

As a fundamental problem, k nearest neighbors search has been studied in many areas such as databases, computer vision and recommendation systems.

Given a distance metric δ and a query point q, k nearest neighbors search aims to find k points in the point set P that are closer to q than all other points under δ.

The exact search can be costly due to *the curse of dimensionality* [45]. Therefore, many researchers have diverted their attention to approximate algorithms for the trade-off between accuracy and efficiency. A wide range of approximate nearest neighbor (ANN) search algorithms have been proposed. Existing techniques can be roughly classified into three main categories: locality-sensitive hashing [34], product quantization [110] and proximity graph [95]. As stated in a recent ANN search benchmark paper [69], proximity graph-based methods have outstanding search performance and can achieve high recall by only retrieving a minimal portion of points.

A proximity graph-based method builds a proximity graph on the point set P. On the graph, a vertex denotes a point in P and chooses certain vertices as its neighbors based on a specified rule. Figure 1.1 shows an example of proximity graph. We can traverse the proximity graph from an entry vertex and quickly identify the promising nearest neighbors of the query point by greedy heuristics. Due to the high dimensionality of data points, the massive distance computation is the dominant cost while searching on proximity graph.

It is well-known that GPU is a promising device for solving simple data-intensive computing tasks (e.g., bulk distance computation) due to having such an enormous number of lightweight cores and high memory bandwidth. Considering the GPU-CPU interaction is not expensive in the context of ANN search, GPU-based solutions have been developed by industry such as Proxima from Alibaba Group [6] and Faiss from Facebook [53] for a variety of applications such as information retrieval, recommendation and advertisement. Therefore, it is rather natural to accelerate the proximity graph ANN search with GPU-based solutions. SONG [119] is the state-of-the-art algorithm that follows this line of research. Through breaking down the search algorithm into 3 stages, SONG can comfortably parallelize the performance-crucial distance computation on the GPU in the distance computation stage. This significantly speeds up the search

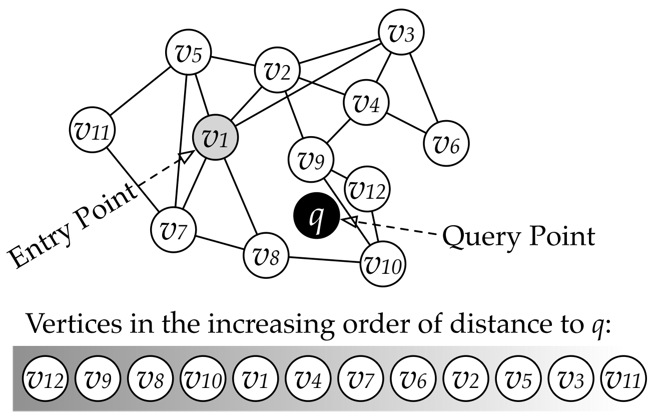

v_5
v_2
v_3
v_4
v_6
v_{11}
v_1
v_9
v_{12}
q
Entry Point
Query Point
v_7
v_8
v_{10}
Vertices in the increasing order of distance to q:
v_{12} v_9 v_8 v_{10} v_1 v_4 v_7 v_6 v_2 v_5 v_3 v_{11}

Figure 1.1: An example of proximity graph

performance, compared to CPU-based solutions.

1.1.3 Subgraph matching

Subgraph matching has been extensively studied for decades. As a fundamental problem, it has significant impact on a wide range of research fields such as protein interaction analysis [84], social network analysis [91], and chemical compound search [111].

Given a query graph q and a data graph G, subgraph matching is to extract all subgraph isomorphic matches of q in G.

Due to the NP-hardness of subgraph matching [52], the whole search space of subgraph matches is prohibitively large. Despite the intrinsic limitation that cannot be surpassed, substantial research efforts have been made to reduce computational complexity in the average case. By their methodologies, these algorithms can be classfied into two classes: join-based algorithms and backtracking search-based algorithms. The join-based algorithms [59, 60] regard the query graph q as a multi-way join and execute the join based on a join plan derived based on cardinality estimation. Following a matching order, the backtracking search-based algorithms iteratively find vertices in the data graph to match query vertices.

We focus on backtracking search-based algorithms. Similar to the critical influence of the join plan on join-based algorithms, the matching order has a substantial effect on the performance of backtracking search-based algorithms [96]. Identifying the optimal matching order is time-consuming due to the inherent complexity of the problem. Existing works [15, 38] resort to conduct pre-processing that serves two purposes: (1) it can guide a decent matching order when integrated with heuristics. (2) search space can be reduced by pruning branches of search tree. During both the pre-processing and enumeration processes, a substantial amount of computation involving set intersection is conducted. This makes it attractive for GPU acceleration. GSI [116] exhibits outstanding performance within the context of prior research. GSI employs a lightweight filtering method to make a trade-off between high parallelism and efficient pruning performance. In addition, GSI proposes the Prealloc-Combine strategy to avoid joining-twice.

1.1.4 GPU Architecture

We introduce GPU architecture from thread hierarchy and memory hierarchy.

In terms of thread hierarchy there are two perspectives: hardware and programming model, in charge of execution and organization respectively.

Hardware. The GPU consists of several streaming multiprocessors (SM), including some streaming processors and other resources such as dispatch units.

Programming model. A program executed by a GPU is called a *kernel*, the execution of a kernel is supported by thousands of threads. These threads are organized into several *thread blocks*. A thread block contains hundreds of threads. After a kernel is launched, thread blocks will be assigned to SMs, and cores inside each SM execute these threads in a single instruction multiple threads (SIMT) fashion, at the granularity of a single *warp* which is the smallest execution unit from the view of hardware and contains 32 consecutive threads.

In the memory hierarchy of a GPU, there are various types of memory, including global memory, shared memory, caches and registers. These differ in capacity, latency and visibility.

Global memory. This has the largest capacity, generally tens of GB. But, it is off-chip that leads to the highest latency. Data in global memory is visible to all threads. Therefore, data structures of large size usually reside in global memory.

Shared memory. This is on-chip and can achieve high throughput. However, it is a relatively scarce resource, which can only reserve a few KB per SM. It can be only accessed by threads from the same streaming multiprocessors. Therefore, shared memory is generally used to store frequently accessed data within thread blocks.

Caches. *L1 cache* and *L2 cache* are included. They can reduce the latency of access to global memory. While the L1 cache is exclusive to every SM, the L2 cache is shared among all SMs. Generally, we cannot access and manage them explicitly.

Registers. This is the fastest memory and larger than shared memory, which is roughly hundreds of KB per SM. A register can be only accessed directly by the thread to which it belongs. Generally, it is used to store local variables of each thread.

If the size of local variables exceeds the capacity of registers, the excess spills over into *local memory* which is actually part of global memory but is only visible in the thread where it is declared. In addition, *warp-level primitives* allow for the exchange of local variables between threads in the same warp. This supports communication within the warp. For a more detailed exploration of GPU architecture, refer to [48].

1.2 Motivations

1.2.1 Matrix factorization

With GPUs now commonly used for general-purpose data-parallel applications, heterogeneous systems with multi-core CPUs and GPUs are becoming increasingly attractive for many general tasks. In comparison to methods on CPU-only or GPU-only systems, approaches developed for heterogeneous systems not only hastens task processing but also makes efficient use of the computing resources accessible in heterogeneous systems. In order to meet performance requirements, CPU-only or GPU-only systems are often over-provisioned, resulting in low average utilization. For instance, when a task is allocated to a GPU (i.e., starting the kernel), the CPU remains idle. Similarly, in applications where GPU memory bandwidth is a significant bottleneck, the vast computational resources of GPUs are underutilized. The development of techniques for heterogeneous systems facilitates efficient utilization of resources that were originally inactive, thereby reducing running time for the program. Driven by this motivation, a significant amount of researches on heterogeneous computing techniques [72, 81, 93, 89] have been proposed. The hybrid algorithm in [89] demonstrates a two orders of magnitude performance improvement over CPU-only and GPU-only algorithms, underscoring the promising potential of algorithms designed for heterogeneous systems. To our best knowledge, no prior studies have tackled the MF task in heterogeneous CPU-GPU systems. The pursuit of an efficient MF algorithm suited for this system setting is desirable. In order to implement an SGD-based MF solution on heterogeneous CPU-GPU systems, our primary focus is on task partitioning and scheduling strategies. This makes our techniques not heavily reliant on any particular algorithm tailored for CPUs or GPUs.

Due to the intricacy of the GPU thread hierarchy, it is preferable to treat a single GPU as an entire working unit on the heterogeneous system, similar to a thread on the CPU. In this case, a straightforward method called HSGD can be achieved by naturally employing the state-of-the-art shared memory algorithm FPSGD for our system setting. Despite the success of HSGD, there are still opportunities for further improvement. First, we observe that the block size obtained through uniform division in HSGD is insufficient to fully utilize the processing capabilities of GPUs. This may limit the overall efficiency. Second, a weak training quality can result from the different computing power between working units, i.e., GPU and CPU thread. More specifically, there is a considerable difference in the number of computations between blocks assigned to a GPU and blocks assigned to a CPU thread. This inspires us to develop a novel approach to alleviate these issues.

1.2.2 *k* nearest neighbors search

Compared to CPU-based solutions, SONG demonstrates superior performance. We notice that the execution dependency in data structure operations results in a bottleneck for SONG, posing challenges for GPU efficiency. A time breakdown analysis of SONG reveals that a significant portion, approximately $50 - 90\%$, is dedicated to data structure operations such as lookup and maintenance while searching on navigable small world (NSW) graphs.

The primary cause for the low efficiency of data structure operations is that SONG continues to adhere to the search paradigm employed by CPU-based solutions. More specifically, the relevant data structure operations, such as searching and updating in the priority queue and the hash table, can be implemented efficiently on the CPU. However, a GPU encounters difficulties when it comes to dynamically maintain data structures that exhibit high irregular dependency. While a range of ANN-specific optimization techniques have been developed to eliminate dynamic GPU memory allocations and trade computations for lower GPU memory consumption, SONG relies on a single thread for data structure operations to maintain good overall performance in a warp. This naturally causes under-utilization of the GPU computing bandwidth.

This motivates us to design a novel graph-based search algorithm on the GPU such that all key steps can fully exploit the massive parallelism of GPU. Furthermore, the construction time of proximity graphs is typically more costly than that of other types of ANN search algorithms, as noted in a recent survey [69]. This motivates us to develop GPU-based algorithms for accelerating the construction of representative proximity graphs. Similar to the search algorithm, the primary challenge lies in harnessing the immense parallelism offered by the GPU. We discover that both straightforward methods suffer from significant weakness. More specifically, the sequential graph construction algorithm in CPU-based solutions is cost-prohibitive on the GPU, and the naive parallel implementation leads to poor-quality graphs. We propose a divide-and-conquer approach to address these problems.

1.2.3 Subgraph matching

In comparison to previous GPU-based subgraph matching efforts, GSI outperforms them by eliminating the redundant calculations from joining-twice and employing optimizations that hasten set intersection operations during computation. We notice that the pruning performance of GSI is sensitive to the number of distinct labels present in the data graph. GSI can only achieve a decent pruning performance on graphs with a large number of distinct labels. When labels on graphs is not pretty diverse, its pruning performance falls short of expectations. In many real-life applications, the number of distinct labels is not very large. The limited pruning performance of GSI can result in a deficient matching order and substantial search space, leading to a drop in search performance in these applications.

The foremost reason for limited pruning performance is that GSI employs a lightweight filtering method. Specifically, the neighborhood structure of each vertex in query graph and data graph is encoded with a bit-vector signature. The bit-wise operation on signatures enables the exclusion of some infeasible data vertices from the candidate set associated with a query vertex. However, the signature struggles to accurately represent neighborhood structure. This occurs as the degree is approximately represented by a very limited number of bits and the label is rehashed in the signature to save memory consumption. Consequently, the pruning performance of GSI leaves much to be desired, particularly when the graph has a limited number of labels. Specifically, after the pruning of GSI, the number of candidate vertices is approximately 10^4 on the *up* dataset with $|\mathcal{L}| = 20$, while [96] indicates that effective pruning can reduce the number of candidate vertices to the order of 10^3.

This prompts us to develop a GPU-friendly filtering approach with strong pruning performance. Furthermore, we notice that the Prealloc-Combine strategy demands more space than necessary to eliminates the redundant calculations from joining-twice, leading to an unignorable portion of memory consumption and uncoalesced memory transactions during enumeration. In order to alleviate this issue, we propose a 1*-phase output scheme that can get a tighter upper bound and more coalesced memory transactions during enumeration. To enhance scalability, we design a pipeline method that can switch from BFS to DFS when data structures during enumeration exceed the capacity of the global memory.

1.3 Contributions

1.3.1 Matrix factorization

The main contributions of our approach are summarized as follows.

- To our best knowledge, this is the first work for parallel SGD-based matrix factorization on heterogeneous CPU-GPU systems.
- We propose a non-uniform division strategy to improve both the efficiency and training quality.
- We design a cost model tailored to our problem to balance workload, which takes into account both data transfer and kernel execution in the GPU part.
- Extensive experiments on four benchmark datasets demonstrate that our approach can outperform competing methods and achieve effective utilization of the resources on the heterogeneous CPU-GPU systems.

1.3.2 k nearest neighbors search

The principal contributions of our approach are summarized as follows.

- We develop a novel graph-based ANN search algorithm on the GPU. To better leverage the massive parallelism of GPUs in all search steps, we propose a GPU-friendly search paradigm.
- We propose the first GPU-based approach for NSW graph construction, which allows for efficient utilization of the immense parallelism of the GPU

without sacrificing graph quality. Our technique can be readily extended to other representative proximity graphs.

- Comprehensive experiments on representative high-dimensional datasets demonstrate the superior performance of our approaches. Our search algorithm is up to 5 times faster than the state-of-the-art SONG with the same accuracy. Our graph construction approach delivers a 40-50x speedup on most datasets as the single-thread CPU-based NSW graph construction algorithm, while maintaining the same graph quality.

1.3.3 Subgraph matching

The principal contributions of our approach are summarized as follows.

- We develop a novel GPU-friendly filtering method that has strong pruning ability.
- We develop a 1^*-phase output scheme to better utilize the GPU by reducing memory consumption and increasing write throughput during enumeration.
- We propose a pipeline method that can adaptively switch between BFS and DFS to enhance the scalability.
- Experiments on real-life labeled graphs demonstrate that our approach outperforms the state-of-the-art method GSI, with up to 4 times speedup.

1.4 Roadmap

The subsequent content of this thesis is organized as follows. Chapter 2 delves into the related work pertaining to the three problems of our concern. Chapter 3 presents our approach that addresses matrix factorization on heterogeneous systems. Chapter 4 and Chapter 5 details our methods for kNN search and subgraph matching on GPUs, respectively. Finally, Chapter 6 concludes the thesis and discusses potential future work.

Chapter 2

Literature Survey

In this chapter, we conduct a systematic review of the current literature concerning the three key problems we explore.

2.1 Matrix factorization

In this section, we introduce a range of methods for solving matrix factorization in different system settings, primarily focusing on SGD-based approaches for multicore shared memory system and GPU, along with other related works. We highlight FPSGD [121] and CuMF_SGD [109], as they are closely related to our work.

2.1.1 Multicore SGD-based algorithms

There exist several algorithms that built upon the core ideas of SGD in multicore shared memory systems. Hogwild [87] adopts a lock-free update strategy, allowing for the fully parallel updating of dense matrices. With a matrix-blocking partition, DSGD [32] averts conflicts that may arise while updating. Following the strategy of DSGD, various approaches [121, 80, 20] have been proposed.

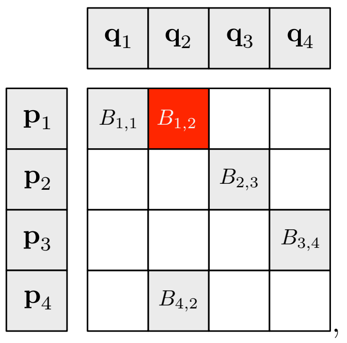


Figure 2.1: An example of independent and conflicting blocks in 4×4 grid matrix

Among these works, FPSGD [121, 20] is a representative method.

FPSGD uniformly divides the sparse matrix R into a set of sub-matrices, referred to as blocks. Two blocks are considered independent if they do not share any common columns or rows of the matrix R; otherwise, they are deemed to conflict. FPSGD employs the update strategy of SGD on a set of independent blocks in a batch and continues until the number of computed blocks reaches a predefined threshold. The reason for selecting independent blocks is to prevent overwriting caused by conflicts. It can be demonstrated through Example 1.

Example 1. *Considering the matrix R divided into 4×4 blocks as shown in Figure 2.1, the computation of $B_{1,1}$ updates vectors $\boldsymbol{p}_1$ and $\boldsymbol{q}_1$, while the computation of $B_{2,3}$ updates vectors $\boldsymbol{p}_2$ and $\boldsymbol{q}_3$. As they update vectors in distinct regions, the computations on $B_{1,1}$ and $B_{2,3}$ can be executed simultaneously. It is evident that $B_{1,1}, B_{2,3}, B_{3,4}$, and $B_{4,2}$ are mutually independent. However, $B_{1,1}$ and $B_{1,2}$ are in conflict with each other, as they both update vectors in $\boldsymbol{p}_1$.*

2.1.2 GPU SGD-based algorithms

Due to its data-intensive nature, matrix factorization can be significantly accelerated by leveraging the vast computational resources of GPUs. Several

SGD-based algorithms on the GPU have been developed for this purpose. GPUSGD [50] follows the matrix-blocking partition and computes independent blocks simultaneously using multiple thread blocks on the GPU. BIDMach [17] supports the use of GPU to accelerate SGD-based matrix factorization. Following the typical framework of SGD-based methods on the GPU, CuMF_SGD [109] stands as the state-of-the-art through high-performance GPU kernels and the strategy for effective utilization of CPU-GPU bandwidth.

CuMF_SGD implements high-performance GPU kernels by a range of optimizations fully exploiting GPU hardware characteristics, which includes warp shuffle, memory coalescing, register usage and half-precision. CuMF_SGD simultaneously performs the memory transfer and the computation to enhance overall efficiency. This is supported by CUDA streams. Specifically, each stream consists of a series of GPU commands executed sequentially, while commands in separate streams can run concurrently if hardware resources allow. CuMF_SGD employs three streams to manage CPU-GPU memory transfer, GPU kernel computation and GPU-CPU memory transfer, respectively. In addition, CuMF_SGD effectively utilizes the CPU-GPU memory transfer bandwidth during block scheduling by selecting multiple consecutive blocks at once instead of only one independent block for the GPU. This reduces the overhead of CPU-GPU memory transfer.

2.1.3 Other related work

Matrix factorization has been extensively investigated in the literature due to its broad range of applications. Various SGD-based algorithms for this problem in other system settings have been proposed such as distributed systems [67] [115] and parameter server [120].

Besides SGD-based algorithms, alternative methods such as Coordinate De-

Table 2.1: A summary of matrix factorization approaches

Methodology	**System**	**Approach**
SGD	Multicore	Hogwild [87], DSGD [32], MLGF-MF [80], FPSGD [121, 20]
	GPU	GPUSGD [50], BIDMach [17], CuMF_SGD [109]
	Distributed	Sparkler [67], NOMAD [115], DFM [120]
CD	Multicore/Distributed	CCD++ [113]
ALS	Distributed	softImpute-ALS [41]

scent (CD) [113] and Alternating Least Squares (ALS) [41] have been also proposed to solve matrix factorization. These methods employ different update rules to compute dense matrices. Specifically, ALS [58] updates only one dense matrix once while fixing the other and subsequently updates the second dense matrix similarly. An iteration is deemed complete once both dense matrices have been updated. CD [113] updates a single element in a dense matrix while maintaining all other elements in both matrices constant. An iteration is finished when all elements in both dense matrices have been updated following the same rule.

2.1.4 Summary

All the approaches are summarized in Table 2.1. We aim to develop an SGD-based matrix factorization method for heterogeneous systems. This indicates our dedication to improving performance compared to existing multicore or GPU-based methods. Specifically, we focus on FPSGD and CuMF_SGD due to their representative performance on multicore and GPU systems, respectively. We do not include distributed system methods in our comparison because they rely on multiple nodes, while our research is centered on single-system architectures.

2.2 *k* nearest neighbors search

In this section, we provide an overview of methods for *k* nearest neighbors search, implemented on the CPU and GPU.

2.2.1 CPU-based ANN methods

To strike the balance between accuracy and efficiency, various works have been proposed from multiple perspectives. Existing techniques can be broadly categorized into three groups: *LSH-based* methods, *product quantization-based* methods and *graph-based* methods.

LSH-based methods (e.g., [34, 44, 98, 71, 51]) use a set of hash functions, which have the property that similar points in the original high-dimensional space are more likely to collide than dissimilar points. By hashing the data points with these functions, an index structure is built, which can be used to quickly retrieve the approximate nearest neighbors.

Product quantization-based methods (e.g., [110, 75, 103, 11, 54, 31, 46]) partition each high-dimensional data point into lower-dimensional sub-vectors. Each sub-vector is then independently quantized using its own codebook. The final quantized representation of a data point is the concatenation of the indices of the code vectors for each sub-vector.

Graph-based methods (e.g., k-DR [8], DPG [69], PANNG [106], EFANNA [28], FANNG [40], NSG [29], HNSW [74] and Vamana [95]) have garnered significant attention due to their exceptional search performance in handling high-dimensional data. Researchers have dedicated considerable efforts to construct a range of proximity graphs such as kNN graphs, small world graphs, relative graphs and their various adaptations. These proximity graphs aim to capture the underlying structure and relationships within the dataset, enabling efficient traversal during search.

Furthermore, there has been a growing interest in providing benchmark and survey studies (e.g., [69, 104]) for researchers and practitioners. These studies serve as comprehensive resources that offer valuable insights into the state-of-the-art techniques and best practices.

2.2.2 GPU-based ANN methods

Given the remarkable performance demonstrated by GPUs in general-purpose data-parallel applications, a growing number of researchers have started to explore the potential of leveraging GPU technology to accelerate the approximate nearest neighbor problem.

Quantization-based methods have received considerable attention in the context of GPU-based implementation, due to their intrinsically high level of parallelism. Examples of such methods include Faiss [53], Robustiq [18] and PQT [108], which have demonstrated significant performance improvements in comparison to their CPU-based counterparts. Notably, Faiss [53] builds upon the idea of IVFADC by incorporating a fast k-selection implementation on the GPU, enabling billion-scale similarity search. SONG [119] proposes an efficient search algorithm on proximity graphs, outperforming Faiss in terms of search time and accuracy.

2.2.3 Summary

Table 2.2 provides a comprehensive overview of the approaches. We aim to address the k nearest neighbor search problem on the GPU. Multicore algorithms are beyond the scope of our consideration as GPU-based algorithms have been proven to offer superior performance compared to their multicore counterparts [53, 119]. Given that SONG has demonstrated superior performance

Table 2.2: A summary of k nearest neighbors search approaches

Methodology	**System**	**Approach**
Locality-Sensitive Hashing	Multicore	DSH [51], DGH [71], SRS [98], QALSH [44], iDEC [34]
Product Quantization	Multicore	IVFADC [46], OPQ [31], LOPQ [54], AQ [11], OCKM [103], LSQ [75], Online PQ [110]
	GPU	PQT [108], Robustiq [18], Faiss [53]
Proximity Graph	Multicore	PANNG [106], k-DR [8], NSG [29], FANNG [40], EFANNA [28], HNSW [74], DPG [69], Vamana [95]
	GPU	SONG [119]

over the PQ-based method Faiss, our performance evaluation is conducted in comparison with SONG.

2.3 Subgraph matching

In this section, we provide a comprehensive review of subgraph matching techniques across various system settings such as CPU, GPU and distributed system, as well as other related work.

2.3.1 CPU-based subgraph matching methods

As one of the pioneers in the development of subgraph matching algorithms, Ullmann [101] uses a candidate set and a partial mapping to keep track of the search progress and employs a backtracking mechanism when a dead-end is reached. This lays the foundation for many later algorithms. Through a set of feasibility rules, VF2 [22] reduces the number of possible mappings to be considered during search, thereby improving the efficiency. Later works [90, 42, 117, 118, 39, 88, 15, 38, 14] have been dedicated to deriving an efficient matching order. QuickSI [90] computes the matching order by estimating the presence of query vertices and edges in the data graph. In order to leverage the pruning ability of non-tree edges early, CFL [15] employ a core-forest-leaf structure for matching query vertices. Within the same part, CFL determines the matching order by the pre-computation of candidate vertex sets, which is implemented through several iterations of top-down construction and bottom-up refinement. By preserving all edges connecting candidate vertex sets, DAF [38] reduces computation during the search and leverages a failure set to eliminate more search space effectively.

2.3.2 GPU-based subgraph matching methods

As an NP-hard problem, subgraph matching is inherently difficult to tackle. This presents an opportunity for GPU acceleration. Consequently, an increasing number of researchers have begun investigating the potential of utilizing GPUs for addressing subgraph matching. GpSM [100] designs efficient kernels that harness massive parallelism of the GPU and prunes the search space based on a simplified query graph. GSI [116] proposes the Prealloc-Combine strategy to prevent joining-twice and develops several techniques to alleviate the issue of limited GPU memory. RPS [37] takes advantage of reusable plans arising from identical local structures in the query graph and selects a matching order that produces reusable plans as much as possible to save computation. In the context of subgraph matching problems for a certain type of query graph, PARSEC [24] first adopts a DFS-based solution and presents a range of optimization methods such as set intersections with binary-encoded induced subgraphs and smart counting. Focusing on subgraph matching on large datasets across multiple GPUs, PBE [36] partitions the data graph into various segments, enabling parallel intra-partition computation. In order to effectively conduct inter-partition computation, PBE proposes a heuristic method for the order selection.

2.3.3 Distributed subgraph matching methods

In order to accelerate subgraph matching, a number of studies [59, 60, 85] typically decompose the query graph into multiple join units to leverage parallelism of distributed systems. TwinTwigJoin [59] selects TwinTwig that is a star of at most two edges as the join unit and adopts a left-deep join order. It has been proven that TwinTwigJoin is instance optimal to StarJoin [99] when the cost is evaluated based on Erdős-Rényi random graph model [27]. To reduce execution time, CliqueJoin [60] employs clique as the join unit for dense queries and adopts the bushy join for approaching optimality. CrystalJoin [85] decomposes the query graph into a core and a set of crystals, allowing for significant compression of crystal matches to address the output crisis in subgraph matching.

2.3.4 Other related work

In order to explore potential research directions and gain a comprehensive understanding of existing works, several in-depth surveys have been conducted. [63] and [96] provide extensive reviews of subgraph matching methods on the single-machine environment, analyzing their strengths, weaknesses and potential improvements. RapidMatch [97], on the other hand, offers a comparative study of backtracking search-based methods and join-based methods. [61] focus on subgraph matching techniques in distributed systems, examining the challenges and opportunities presented by this setting.

Several noteworthy theoretical works have been presented in the context of relational join, which bear a strong connection to subgraph matching problem. Among these, AGM [9] provides a tight worst-case bound for join size, parame-

Table 2.3: A summary of subgraph matching/enumeration approaches

Methodology	**Type**	**System**	**Approach**
Backtracking	Labeled	Multicore	Ullmann [101], VF2 [22], QuickSI [90], GraphQL [42], GADDI [117], SPATH [118], Turbo$_{\text{iso}}$ [39], BoostIso [88], CFL [15], CECI [14], DAF [38]
	Labeled	GPU	GpSM [100], GSI [116]
	Unlabeled	GPU	RPS [37], PARSEC [24]
	Unlabeled	MultiGPU	PBE [36]
Join	Unlabeled	Distributed	TwinTwigJoin [59], CliqueJoin [60], CrystalJoin [85]

terized by the fractional edge cover, offering valuable insights into the complexity of relational join in the worst case. To achieve worst-case optimality, NPRR [77] and LeapFrog [102] have been proposed. Generic Join [78] highlights the common principles behind these two algorithms and presents a unified framework. RapidMatch [97] suggests that backtracking search-based methods are worst-case optimal when the set intersection satisfies the min property. This is substantiated by establishing an equivalence between backtracking search-based methods and LeapFrog.

2.3.5 Summary

Table 2.3 presents a summary of all methods. We focus on addressing the subgraph matching problem on GPUs. As such, multicore algorithms and subgraph enumeration algorithms, including those on GPUs and distributed systems, are not considered in our evaluation. As GSI has been shown to outperform GpSM, we select GSI as the baseline for our performance comparison.

Chapter 3

Matrix Factorization

In this chapter, we introduce the methodological details about our approach to matrix factorization along with the experimental results.

3.1 Preliminaries

In this section, we present the formal definition of matrix factorization and introduce background knowledge of stochastic gradient descent.

3.1.1 Matrix factorization

We consider a user-item rating matrix $R \in \mathbb{R}^{m \times n}$ where m and n are the number of rows and the number of columns of the matrix, respectively. For each $1 \leq u \leq m$ and $1 \leq v \leq n$, $r_{u,v}$ is the rating from the user u to the item v. R is normally sparse since not every element in R is explicitly reported. Matrix factorization aims to represent the matrix R as a dot product between two dense matrices $P \in \mathbb{R}^{m \times k}$ and $Q \in \mathbb{R}^{k \times n}$, where k is the number of latent factors. A

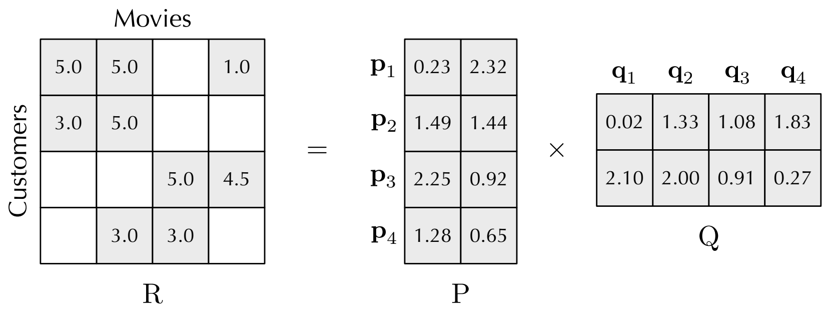


Figure 3.1: A rating matrix R and a corresponding matrix factorization

mathematical representation of the matrix factorization is shown as follows.

$$r_{u,v} \approx \mathbf{p}_u \mathbf{q}_v \tag{3.1}$$

In Equation 3.1, $\mathbf{p}_u$ is the u-th row vector of P, and $\mathbf{q}_v$ is the v-th column vector of Q. Matrix factorization achieves Equation 3.1 by minimizing the following loss function.

$$\mathcal{L} = \sum_{(u,v)\in R} (r_{u,v} - \mathbf{p}_u \mathbf{q}_v)^2 + \lambda_P \|\mathbf{p}_u\|_F^2 + \lambda_Q \|\mathbf{q}_v\|_F^2 \tag{3.2}$$

In Equation 3.2, $(r_{u,v} - \mathbf{p}_u \mathbf{q}_v)^2$ measures the gap between $r_{u,v}$ and estimated value $\mathbf{p}_u \mathbf{q}_v$. $\lambda_P \|\mathbf{p}_u\|_F^2$ and $\lambda_Q \|\mathbf{q}_v\|_F^2$ are used to avoid over-fitting. $\|.\|_F^2$ computes the Frobenius norm[1] of a vector. λ_P and λ_Q are regularization coefficients.

Example 2. *We give an example of the matrix factorization in Figure 3.1. The rating matrix R contains nine ratings for four movies given by four customers. The number of latent factors k is 2. We have $r_{1,2} = 5.0$ which means that u_1 gives a rating 5.0 to v_2. The results of R's matrix factorization are shown on the right of R. P is a user preference matrix, and Q is a movie feature matrix. The*

[1]https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm

vector $\boldsymbol{p}_1(0.23, 2.32)$ *is the preference of the user* u_1*, and* $\boldsymbol{q}_2(1.33, 2.00)^T$ *is the feature of the movie* v_2*. The estimated value of* $\boldsymbol{p}_1\boldsymbol{q}_2$ *is* 4.9459*, which is close to* $R_{1,2}$ 5.0*.*

3.1.2 Stochastic gradient descent

It is time-consuming to calculate the loss of the whole matrix R when using the loss function of Equation 3.2, especially when R contains billions of items. Several works have been done to minimize Equation 3.2 and improve the efficiency of matrix factorization [30, 58, 113]. We follow a prevalent algorithm called *stochastic gradient descent* (SGD) [30].

SGD executes several iterations. The number of iterations can be specified by users. Instead of straightforwardly applying the gradient descent to minimize Equation 3.2 in each iteration, SGD computes the gradient of every element $r_{u,v}$ in R and updates the corresponding vectors in the result matrices. Consequently, the original loss function in Equation 3.2 is reduced to the following equation.

$$\mathcal{L} = (r_{u,v} - \mathbf{p}_u\mathbf{q}_v)^2 + \lambda_P\mathbf{p}_u\mathbf{p}_u^T + \lambda_Q\mathbf{q}_v^T\mathbf{q}_v \tag{3.3}$$

The gradient of Equation 3.3 is represented as follows.

$$\frac{\partial L}{\partial \mathbf{p}_u} = 2(\mathbf{p}_u\mathbf{q}_v - r_{u,v})\mathbf{q}_v^T + 2\lambda_P\mathbf{p}_u \tag{3.4}$$

$$\frac{\partial L}{\partial \mathbf{q}_v} = 2(\mathbf{p}_u\mathbf{q}_v - r_{u,v})\mathbf{p}_u^T + 2\lambda_Q\mathbf{q}_v \tag{3.5}$$

Based on the gradient (Equation 3.4 and Equation 3.5), we train the model iteratively, and the value of the loss function decreases until it is convergent.

Algorithm 1: SGD

```
   Input: $R_{m\times n}, k, \lambda_P, \lambda_Q, \gamma, t$
   Output: $P_{m\times k}, Q_{k\times n}$
   // Data pre-processing phase
 1 Init($P_{m\times k}$, $Q_{k\times n}$);
   // Training phase
 2 foreach $iteration \leftarrow 1$ to $t$ do
 3     foreach $r_{u,v} \in R_{m\times n}$ do
 4         $e_{u,v} \leftarrow r_{u,v} - \mathbf{p}_u\mathbf{q}_v$;
 5         $\mathbf{p}_u \leftarrow \mathbf{p}_u + \gamma(e_{u,v}\mathbf{q}_v^T - \lambda_P\mathbf{p}_u)$;
 6         $\mathbf{q}_v \leftarrow \mathbf{q}_v + \gamma(e_{u,v}\mathbf{p}_u^T - \lambda_Q\mathbf{q}_v)$;
   // Data post-processing phase
 7 Save($P_{m\times k}$, $Q_{k\times n}$);
```

Equation 3.6 shows this process where γ is the learning rate.

$$\mathbf{p}_u \leftarrow \mathbf{p}_u - \gamma\frac{\partial L}{\partial \mathbf{p}_u} \qquad \mathbf{q}_v \leftarrow \mathbf{q}_v - \gamma\frac{\partial L}{\partial \mathbf{q}_v} \tag{3.6}$$

We present the pseudocode of SGD in Algorithm 1. There are several input parameters. $R_{m\times n}$ is a sparse rating matrix stored in the form of triadic tuple. k is the number of latent factors. γ is the learning rate. λ is the regularization parameter. t is the number of iterations. Algorithm 1 outputs two feature matrices $P_{m\times k}$ and $Q_{k\times n}$. The algorithm consists of three phases: data preprocessing, SGD training, and data post-processing. The data preprocessing phase initializes two resulting matrices P and Q with values generated randomly. In the training phase, in each iteration (line 3-6), every element is picked to decrease the value of loss function by using Equation 3.6. The SGD training terminates when the given number of iterations is reached (line 2) or the model converges. The feature matrices are stored after training.

Problem Definition. We aim to develop an efficient SGD-based matrix factorization algorithm on the heterogeneous CPU-GPU systems.

Remark 1. *Our research mainly focuses on the scheduling strategy for the task division and assignment between CPUs and GPUs. Our proposed techniques will not closely depend on any specific GPU or GPU SGD-based matrix factorization algorithms.*

3.2 Our approach

In this section, we first give a straightforward method and show its drawbacks. Then, we briefly introduce scheduling algorithms for heterogeneous systems and propose our method to balance workloads. Finally, an overview of our improved algorithm is provided.

3.2.1 A straightforward method

A straightforward idea to utilize both CPU and GPU resources is to treat a GPU kernel as a worker thread. Based on this idea, we can adapt FPSGD [121] by regarding a GPU as an additional worker thread. Similarly, to avoid the conflict and obtain good training quality, a worker thread receives a new block satisfying the two criteria in Section 2.1.1 once it finishes processing a block. We apply the FPSGD and CuMF_SGD algorithms to process a matrix block on a CPU thread and a GPU kernel, respectively. This straightforward algorithm shown above can be called HSGD.

Let n_c and n_g be the number of CPU threads and GPUs. Following the matrix-division rule in [121], the number of blocks should be at least $(n_c + n_g + 1) \times (n_c + n_g + 1)$. We refine this formula and give a more precise matrix-division rule.

Rule 1. *Given an input rating matrix R, n_c CPU threads, and n_g GPUs, R should be divided into at least $(n_c + n_g + 1) \times (n_c + n_g)$ blocks.*

We explain the rationale of this rule. When the number of blocks is less than $(n_c + n_g + 1) \times (n_c + n_g)$, every thread is always assigned the same block. As a result, only several specific blocks are updated during the algorithm. Obviously, this will lead to a terrible training result. Even worse, the algorithm cannot fully exploit all worker threads. For example, in Figure 2.1, assume that there are 4 threads. The block $B_{1,1}$ is assigned to thread 1, and the rest gray blocks are assigned to other threads. When thread 1 finishes processing $B_{1,1}$, the rest gray blocks may still be occupied. To avoid conflicts, thread 1 has to continually process $B_{1,1}$. By contrast, when the block number increases to $(n_c + n_g + 1) \times (n_c+n_g)$, thread 1 can always locate a spare row or column which is not occupied by other blocks.

3.2.2 Motivation

Even though the HSGD successfully combines CPU and GPU resources, we have several observations which can help (i) improve the working efficiency of GPUs and (ii) balance workloads for different hardware.

Exploiting Hardware Characteristics. GPUs and CPUs have different features. We have two observations as follows.

Observation 1. *In the context of MF, small blocks cannot saturate the GPU computing power.*

To indicate the relationship between the block size and the efficiency of the GPU, we launch a GPU kernel with the default configuration to process blocks with different sizes. The details of configuration and hardware can be found in Section 3.5. The GPU throughput is reported in Figure 3.2(a), where the labels on the x-axis represent the number of elements in a block, and the labels on the y-axis represent the average number of elements processed in every second.

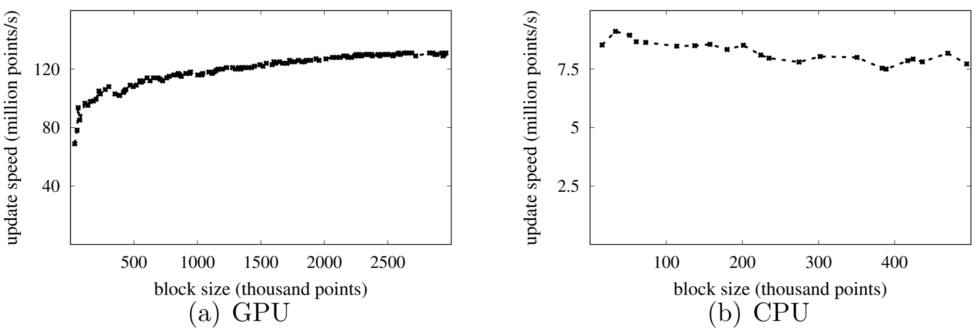


Figure 3.2: Processing speed of GPUs and CPUs on blocks with different sizes

The throughput significantly increases when the block size is relatively small. Afterward, the upward trend becomes gentle as the block size continues to grow. This phenomenon may be due to two main reasons. First, we need to transfer data via the PCI-e bus to the global memory of GPU when launching a GPU kernel. A small block cannot fully utilize the bandwidth of the bus. Second, more data can better utilize all threads and the cache mechanism of the GPU.

Observation 2. *In the context of MF, the computing power of CPU cores is not sensitive to the block size.*

The average number of elements processed by a thread of CPU in every second is shown in Figure 3.2(b). Unlike the GPU, the throughput of a CPU thread always remains stable when the block size varies. This is because the worker threads of CPU are relatively more independent than those of GPU and the computing capability of a CPU thread is not so powerful, compared with the whole GPU device.

Nonuniform Matrix Division. Based on the above two observations, an immediate idea to improve the algorithmic efficiency is to set the block size as large as possible. However, as mentioned in Section 3.2.1, the input matrix should be divided into at least $(n_c + n_g + 1) \times (n_c + n_g)$ blocks and the block size under this division strategy is still relatively small. For example, we use 16 CPU threads and a GPU in the default configuration of our experiments. Considering the real-world dataset Yahoo!Music with $1,000,990$ rows and $624,961$ columns, we divide it into at least 18×17 blocks. Consequently, the number of elements in each block is less than 1 million, which is still not large enough to saturate the GPU computing power in the light of Figure 3.2(a).

To improve the working efficiency of GPUs, we divide the rating matrix into blocks with different sizes. The large ones are assigned to GPUs, and the small ones are assigned to CPUs. Towards this end, there are several issues that we need to address. For example, we need to answer how to set suitable block sizes for both CPU and GPU, and how to divide the rating matrix in practice. We will answer these questions in the following section, and the final matrix division strategy will be given in Section 3.4.

Workload Balance. As shown above, we should make the size of blocks assigned to GPUs as large as possible in terms of improving the working efficiency of GPUs. However, an extreme nonuniform division strategy may cause a serious workload imbalance problem, which can remarkably reduce the overall efficiency when combining two hardware resources. To prevent this issue, a scheduling strategy should be considered. Many efforts have been done to balance workloads for the applications in heterogeneous systems. They can be categorized into three kinds: (1) *dynamic methods*, (2) *static methods by classifier*, and (3) *static methods by cost models*. Here, we briefly review some of representative methods among them and elaborate reasons why they cannot be straightforwardly used in our problem. Then, we propose our own method. For more details of scheduling strategies in heterogeneous systems, interested readers can refer to surveys[83, 76].

(1) Dynamic Methods. The dynamic methods assign a new task to a com-

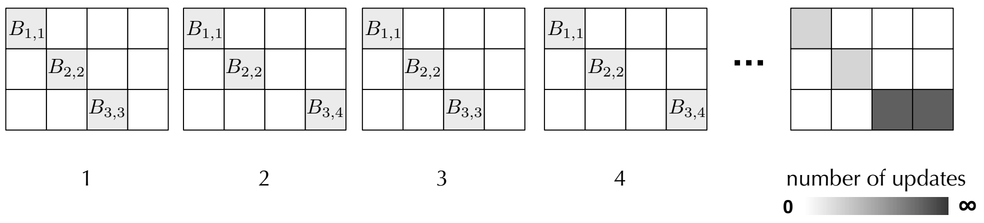


Figure 3.3: A running example of the straightforward algorithm given 2 CPU cores and 1 GPU

putational device according to the performance of devices on previous tasks [89, 49, 64, 13, 16, 21, 105, 55]. For example, [89] maintains a double-ended queue. CPU processes the tasks from the front of the queue. Simultaneously, GPU processes the tasks from the reverse direction. The algorithm naturally enables the dominant hardware resource to handle more tasks.

The baseline algorithm in Section 3.2.1 essentially adopts a dynamic strategy. If a CPU core or a GPU finishes updating a block, it is assigned a new block without incurring any conflict. The dynamic method performs well in other problems if there is not any rule to assign tasks.

However, this type of method does not work well in our problem due to the independence property of the task assignment. As discussed in Section 3.2.1, we first equally divide the matrix into a set of blocks, which is necessary to avoid conflicts. GPUs cannot achieve an ideal performance in this setting according to Observation 1. In addition to the problem of GPU working efficiency, the dynamic update method in HSGD may suffer from a poor training result. We give an example to explain this problem.

Example 3. *We consider computing the matrix factorization of a rating matrix R on a machine with two CPU threads c_1, c_2 and one GPU g_1. Assume that we divide R into 3×4 matrix blocks, which is shown in Figure 3.3. In the beginning, two CPU threads c_1, c_2 and the GPU g_1 get blocks $B_{1,1}$, $B_{2,2}$, and $B_{3,3}$,* *respectively. Normally, the computing power of a GPU is much stronger than that of a CPU thread. As a result, when g_1 has completed its task $B_{3,3}$, c_1 and c_2 are still working on their blocks $B_{1,1}$ and $B_{2,2}$. According to the scheduling policy of HSGD, g_1 will apply a new block which is independent of $B_{1,1}$ and $B_{2,2}$ and has the least number of updates. Block $B_{3,4}$ satisfies these conditions. g_1 picks $B_{3,4}$ in the second step of Figure 3.3. In the following steps, g_1 will continually update the two blocks in the lower right corner since $B_{1,1}$ and $B_{2,2}$ are always occupied by c_1 and c_2. This phenomenon makes the numbers of updates for different blocks severely unbalanced, which is demonstrated in the last matrix. The number of updates for a block is relatively large if the corresponding color is dark. Compared with the original SGD algorithm which updates matrix elements randomly, the process in this example leads to a weak training result.*

(2) Static Methods by Classifier. Static methods provide scheduling decisions for worker threads of different devices before applications start. This type of method establishes a classifier based on training datasets in the offline phase [35, 57, 107, 33]. Given a new task, the classifier identifies the class of the task and applies a corresponding strategy of task assignment derived from the offline phase. There are several drawbacks if applying the classifier-based methods in our problem. First, it is complicated and time-consuming to generate training data including static code features and optimal partition strategies. Moreover, these methods usually rely on specific frameworks such as OpenCL [94] and Insieme [1] to obtain static code features. This makes them not general. Second, they are usually designed for multi-task platforms. Consequently, The partition scheme generated by them is coarse-grained.

(3) Static Methods By Cost Model. This type of method estimates CPU's and GPU's execution time by establishing a cost model [72, 62]. A cost model is a function revealing the relationship between the input data size and the corre-

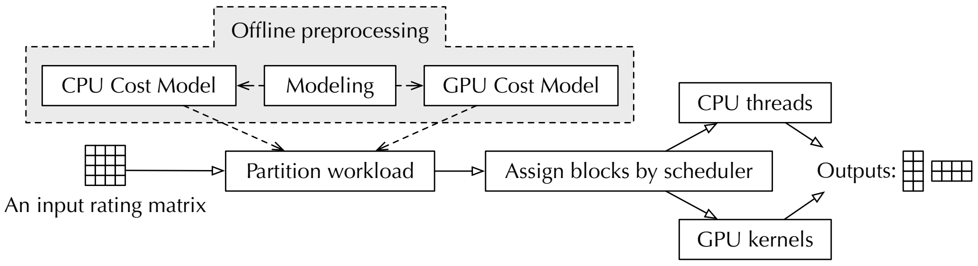


Figure 3.4: Overview of HSGD*

sponding execution time of a specific worker thread. A representative approach among them is Qilin [72]. It divides a training dataset N into two parts N_1 and N_2, which are assigned to CPUs and GPUs, respectively. N_1 is further divided into m subparts $N_{1,1}...N_{1,m}$. Each subpart $N_{1,i}$ is processed by a CPU thread, and the corresponding execution time is recorded. A similar operation is applied to N_2 by GPUs. Qilin uses the curve fitting to construct two linear equations as the projections of the execution times for CPUs and GPUs respectively.

In the context of our problem, a simple linear function in [72] is hard to accurately estimate the execution time of GPUs. Recall Figure 3.2(a), it is not a horizontal line. This proves that the execution time does not increase linearly as the number of elements in a block grows.

Our Method to Balance Workloads. We propose a hybrid method for our problem. We first customize a cost model to divide the matrix specialized for the MF problem. We improve the accuracy of the GPU cost model. The details of cost models are given in Section 3.3. Second, we have a dynamic scheduling mechanism, which allows the dominant resource to use work-stealing mechanism. This alleviates the deviation between the cost model and the practical performance.

3.2.3 The framework

In this section, we give an overview of our improved algorithm in Figure 3.4, which is called HSGD*. Our method contains an offline preprocessing phase and an online processing phase. The offline phase (the gray area in Figure 3.4) derives a cost model which estimates the hardware performance. This step can be performed only once on a machine, and the corresponding parameters are stored to support the query of any input rating matrix in the online phase.

In the online phase, a sparse rating matrix R is given. HSGD* first divides the matrix into two parts, denoted as R_1 and R_2, based on cost models of CPUs and GPUs from the offline phase. Then, R_1 and R_2 are further divided into several blocks. Finally, the scheduler assigns blocks to worker threads. The calculation process continues until the number of iterations reaches the predefined value. During most of the period when HSGD* runs, CPU threads are only allowed to process blocks in R_1, and GPUs are only allowed to process blocks in R_2. Similar to HSGD, the block assignment avoids conflicts in the same row or column. We also have a dynamic scheduling strategy to balance workloads in practice, which is not reflected in Figure 3.4. The details will be shown in Section 3.4. A formal pseudocode of the framework HSGD* is reported in Algorithm 2, which is self-explanatory.

3.3 Our cost model

Given an input matrix R, let α and $1 - \alpha$ be the proportion of the workload assigned to GPUs and CPUs, respectively, where $0 \leq \alpha \leq 1$. We use $T_g(\alpha \cdot R)$ and $T_c((1-\alpha) \cdot R)$ to denote the time spent on updating elements in $\alpha \cdot R$ and $(1-\alpha) \cdot R$ by a GPU and a CPU thread, respectively. When the context is clear, $T_g(\alpha)$ and $T_c(1-\alpha)$ are used for short. The total running time T of our

Algorithm 2: HSGD*

```
  Input: $R_{m\times n}, k, \lambda_P, \lambda_Q, \gamma, t, n_c, n_g$
  Output: $P_{m\times k}, Q_{k\times n}$
  // Offline Phase
1 generate cost models of both CPUs and GPUs;
  // Online Query Processing
2 partition $R_{m\times n}$ according to the cost models;
3 pre-process data;
4 InitScheduler($R_{m\times n}$, $n_c$, $n_g$);
5 scheduler assigns blocks to CPU threads and GPUs;
6 return $P_{m\times k}, Q_{k\times n}$;
```

algorithm is represented below.

$$T = \max(\frac{T_g(\alpha)}{n_g}, \frac{T_c(1-\alpha)}{n_c}) \tag{3.7}$$

Note that both $T_g(\alpha)$ and $T_c(1-\alpha)$ are monotonic. Based on the computed cost functions, the total running time is minimized when the load between resources keeps balancing. We set α using the following equation.

$$\alpha = argmin|\frac{T_g(\alpha)}{n_g} - \frac{T_c(1-\alpha)}{n_c}| \tag{3.8}$$

Based on discussion above, our aim is to derive cost functions $f_g(\alpha) \simeq T_g(\alpha)$ and $f_c(\alpha) \simeq T_c(\alpha)$ for a GPU and a CPU thread, respectively, where $f_g(\alpha)$ (resp. $f_c(\alpha)$) denotes the estimation of $T_g(\alpha)$ (resp. $T_c(\alpha)$). As discussed earlier, a straightforward method to establish a cost model is to follow the works [72, 62] which think that execution time is linearly related to the size of the input matrix. However, based on Observation 1, we find that the processing speed of GPUs increases when the block size increases in our problem. This makes linear regression methods for the GPU cost model inaccurate. Moreover, the computation in execution kernels of GPUs and the data transfer are not completely

Algorithm 3: Cost Estimation

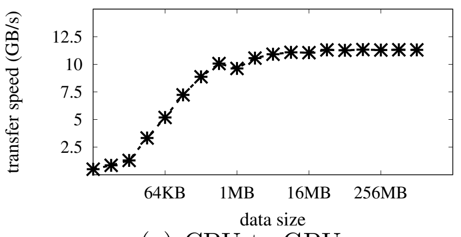
```
Input: $R_{m \times n}$
Output: $f_c$ and $f_g$
// S is an array of segments
// $P_c$ and $P_g$ are arrays of structures
// Data preprocessing phase
1 $S \leftarrow$ SampleDataset($R_{m \times n}$);
// Training cost models
2 $P_c \leftarrow$ TestCPUKernel($S$);
3 $f_c \leftarrow$ CPUModelFitting($P_c$);
4 $f_g^{transfer} \leftarrow$ TestTransferSpeed();
5 $P_g \leftarrow$ TestGPUKernel($S$);
6 $f_g^{execute} \leftarrow$ GPUModelFitting($P_g$);
7 $f_g \leftarrow$ Comebine($f_g^{transfer}$, $f_g^{execute}$);
```

serial due to CUDA stream mechanism. The total running time of GPU is not a simple sum of the kernel execution and the data transfer. This observation makes us reconsider how execution kernel and data transfer exactly influence the total running time of GPUs, which is not discussed in previous cost models.

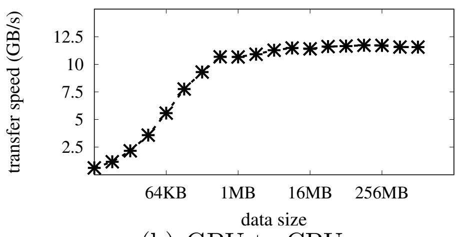
In the rest of this section, we introduce the strategy to prepare the training data in Section 3.3.1 and propose our cost model of GPUs in Section 3.3.2. For the cost model of CPUs, we use a linear function to estimate the performance similar to [72]. A formal pseudocode to compute cost models is given in Algorithm 3. We explain each step as follows.

3.3.1 Data preparation and training for CPUs

To derive the training data, we shuffle the input dataset to avoid uneven data distribution. After the data is shuffled, we equally divide input dataset into N disjoint parts $S_1, S_2, S_3 ... S_N$, stored in array S at line 1 of Algorithm 3. Then, CPU execution kernel configured with a single thread is launched to compute on datasets generated in the last step at line 2. Instead of comput-



(a) CPU to GPU



(b) GPU to CPU

Figure 3.5: Transfer speed varies with block size

ing on $S_1, S_2, S_3 ... S_N$ respectively in [72], CPU execution kernel computes on $S_1, S_1 + S_2, S_1 + S_2 + S_3 ... S_1 + S_2 + S_3 + ... + S_N$ respectively, and the corresponding execution time is recorded. After this, we get an array P_c including data size and corresponding execution time. As a training data set, this array is used to curve fitting for CPUs. Our adaptation generates a wider range of training data which can better reflect the relationship between data size and execution time. To eliminate noise, the execution time in the training data is derived from the average of multiple tests.

3.3.2 Estimating working efficiency of GPUs

Given a task, the total processing time by a GPU is spent on two parts: (1) *data transfer between the CPU and the GPU via the PCI-e bus*, and (2) *GPU execution kernels.*

Data Transfer. Data transfer has two directions — from CPU to GPU (sometimes called Host to Device) and from GPU to CPU. We denote the times spent on them by $f_g^{c \Rightarrow g}$ and $f_g^{g \Rightarrow c}$, respectively. We only discuss the process to model the data transfer from CPU to GPU, and the model for the other direction is similar.

Figure 3.5 reveals the transfer speed for data with different sizes. The transfer speed grows very fast in the beginning. After the data size is larger than a threshold, the transfer speed remains stable. Based on this phenomenon, we use a function to fit the curve when the dataset is not very large, then use the linear regression to model the rest. A formal model is expressed as follows. $|R|$ denotes the size of data transferred from CPU to GPU, and τ denotes the threshold.

$$f_g^{c \Rightarrow g} = \begin{cases} \dfrac{|R|}{a \cdot \sqrt{log|R|} + b} & \text{if } |R| \leq \tau; \\ a \cdot |R| + b & \text{otherwise.} \end{cases}$$

The rationale for $|R| \leq \tau$ is that the data transfer time can be represented by the quotient of the data size and the transfer speed. According to Figure 3.5, we use the function $a \cdot \sqrt{log|R|} + b$ to model the curve of the first stage. We select this function since the trend performs like an inverse function of the parabola. Note that the label distribution on the x-axis is not linear. This is because the logarithmic scale can make the trend clearly presented even though the data size is small. Obviously, this phenomenon shows that we cannot fully utilize the bandwidth of the PCI bus if the data is not large enough, which supports Observation 1. Then, we determine the threshold τ by the extent to transfer speed variation. Empirically, when the variation of the transfer speed is less than 2% in a time unit, we consider that the transfer speed has been stable. Finally, we fit the curve as a linear function for the second stage when $|R| > \tau$. The curve fitting can be done by using the least squares method.

GPU Execution Kernel. We design the cost model of the GPU execution kernel. Similar to Figure 3.5, the throughput of updating remains stable after the block size reaches a threshold, which means that the computing power of the GPU is saturated. To fit the curves, we use a logarithmic function when the dataset is not very large, and then use the linear regression to model the rest. The growth trend of the logarithmic function can be slower than the power

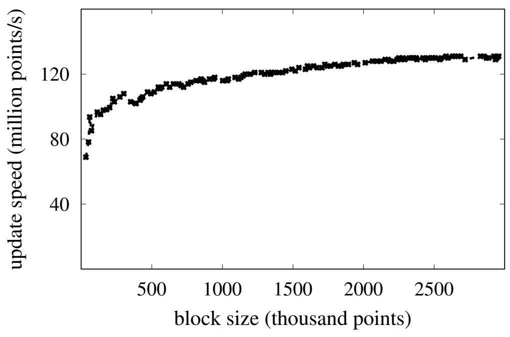


Figure 3.6: Kernel execution time by varying data size

function (e.g., $\sqrt{x}$), which is more consistent with the trend in Figure 3.6. This is why we choose it to model the curve of the first stage. A formal model is expressed as follows, where f_g^{kernel} denotes the time spent on R by the GPU execution kernel. The function $a \cdot log|R| + b$ represents the processing speed of the GPU execution kernel.

$$f_g^{kernel} = \begin{cases} \dfrac{|R|}{a \cdot log|R| + b} & \text{if } |R| \leq \tau; \\ a \cdot |R| + b & \text{Otherwise.} \end{cases}$$

Overall GPU Cost Model. We cannot simply sum the time of the kernel execution and the data transfer as our estimation for the overall execution time of the GPU, since these two parts are not absolutely serial. Specifically, to improve the overall working efficiency of GPUs, we adopt a widely used optimization based on the CUDA stream mechanism. A CUDA stream contains a list of GPU commands executed in serial, and commands in different streams are executed in parallel if hardware resources permit. At the same time, commands in different streams can be synchronized. This mechanism allows us to perform data transfer and kernel execution in parallel without breaking correctness. We explain this idea in the following example.

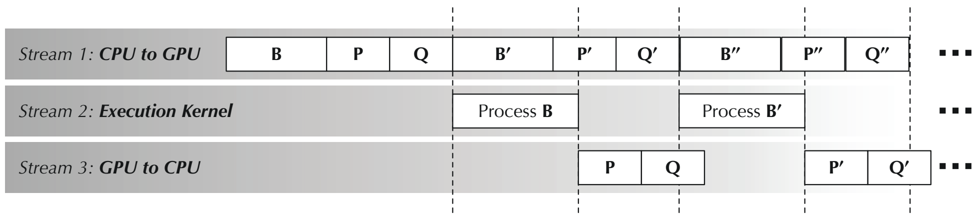


Figure 3.7: Data transfer optimization

Example 4. *As shown in Figure 3.7, we use three streams to manage the data transfer from CPU to GPU, the kernel execution, and the data transfer from GPU to CPU, respectively. Assume that a block B is assigned to the GPU. Stream 1 transfers B and corresponding P, Q to the global memory of a GPU. Then, the GPU kernel scans the block B and updates P, Q. Simultaneously, stream 1 continuously transfers the next block B' assigned to the GPU and corresponding P', Q'. When stream 2 finishes B, stream 3 transfers the updated P, Q back to CPU.*

From this example, the overall time for GPU f_g can be roughly decided by the maximum time spent among these three streams because it covers the time of the other two parts. Note that although the first and the last schemes cannot be overlapped by the maximal stream in the figure, the cost can be ignored when the number of transferred and computed blocks is very large. Note that $f_g^{g \Rightarrow c}$ is always smaller than $f_g^{c \Rightarrow g}$ since we do not need to transfer blocks back to CPU. Therefore, we define the overall cost model of a GPU as follows.

$$f_g = \max(f_g^{c \Rightarrow g}, f_g^{kernel}) \tag{3.9}$$

The overall cost model of a GPU depends on the maximum between data transfer time from CPU and GPU and execution time of the GPU kernel.

3.4 Workload balance in practice

In this section, we present techniques to further balance workload.

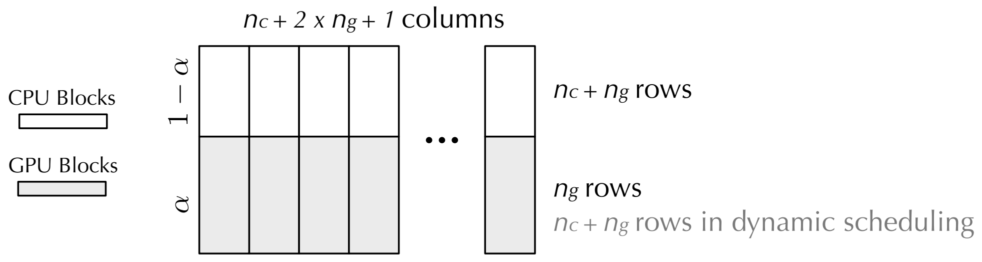
3.4.1 Dynamic scheduling

Even though we have proposed a tailored GPU cost model for MF, the estimation may be still hard to exactly reflect the computing power of devices given a different dataset. The workloads of CPU and GPU may be unbalanced if we assign blocks simply according to the cost model. To remedy this issue, we adopt a dynamic scheduling strategy when one device finishes its tasks. Specifically, assume that the GPU has finished its tasks. Instead of waiting for the tasks being processed by CPUs, we allow GPUs to pick some blocks originally assigned to CPUs. We call it *static phase* when the GPU and the CPU only process the originally assigned tasks and call it *dynamic phase* when one of them finishes its own tasks and is involved in processing tasks of the other. In *static phase*, every GPU is assigned to blocks in specific rows so that it can update one segment of one result matrix all the time and avoid the transfer of this segment.

3.4.2 Putting things together

We explain our final strategy for the matrix division. An example is shown in Figure 3.8.

Number of Columns. Based on the cost model proposed in Section 3.3, we first partition the matrix into two sub-matrices R_c and R_g for CPUs and GPUs, respectively. They are marked by white and gray in Figure 3.8. In the light of Rule 1, we further divide the matrix into $n_c + 2 \times n_g + 1$ columns. This setting guarantees two things. The first thing is that GPUs can always know



$n_c + 2 \times n_g + 1$ columns
CPU Blocks
GPU Blocks
$1-\alpha$
α
$n_c + n_g$ rows
n_g rows
$n_c + n_g$ rows in dynamic scheduling

Figure 3.8: The final division strategy

not only the current block but also the next block. This enables the overlap between computation and data transfer in Figure 3.7. The second thing is that there always exists a spare column when a GPU kernel or a CPU thread finishes processing its block.

Number of Rows for CPUs. As shown in Rule 1, we can divide the input matrix into $n_c + n_g$ rows, where there are n_c (resp. n_g) rows in R_c (resp. R_g). However, this division strategy causes a problem when the dynamic scheduling is activated. Specifically, assume that GPUs have finished their own tasks and are involved in processing blocks of CPUs. Currently, we have totally $n_c + n_g$ (CPU and GPU) threads working on $(n_c + 2 \times n_g + 1) \times n_c$ blocks. This would break Rule 1 and cannot fully exploit all worker threads. To support the assistance from GPUs, we set the number of rows of CPUs as $n_c + n_g$ based on Rule 1. The setting would not affect the CPU efficiency since the computing power of CPUs is not sensitive to the block size as shown in Observation 2.

Number of Rows for GPUs. Similarly, If CPUs first finish their tasks and apply for blocks in R_g, the number of rows in R_g should be at least $n_c + n_g$. However, compared with the row number n_g for GPUs, the row number $n_c + n_g$ leads to a smaller block size, which cannot saturate the computing power of GPUs according to Observation 1. Different from the case of CPUs, the division strategy for GPUs needs to satisfy that the block size is large enough in the beginning, while the number of rows is large enough to avoid conflicts if CPUs join. To achieve this target, we divide R_g into n_g rows. For each row R_g^i where $1 \leq i \leq n_g$, we further divide R_g^i into $\lceil \frac{n_g+n_c}{n_g} \rceil$ sub-rows. As a result, relatively large blocks with sizes $\frac{R_g}{(n_c+2\times n_g+1)\times n_g}$ are assigned to GPUs in static phase, and blocks with sizes $\frac{R_g}{(n_c+2\times n_g+1)\times n_g \times \lceil \frac{n_g+n_c}{n_g} \rceil}$ are assigned to GPUs and CPUs in dynamic phase.

Example 5. *We give an example to explain the division strategy for R_g. Assume that we have 2 GPUs and 4 CPU threads, i.e., $n_c = 4, n_g = 2$. We divide R_g into* 2 *rows, and each row is further divide into* 3 *sub-rows. In static phase, we assign a block with size $\frac{R_g}{9\times 2}$ to a GPU, and in dynamic phase, we assign a block with size $\frac{R_g}{9\times 6}$ to a GPU or a CPU thread. On the other hand, R_c is divided into* 9 *columns and* 6 *rows. This division for R_c would not change in the algorithm.*

3.5 Experiments

In this section, we conduct extensive experiments to show the efficiency and the effectiveness of our proposed algorithms. Algorithms appearing in our experiments are summarized as follows.

- CPU-Only: Only CPU works. We uniformly divide the matrix and use the strategy in [121] to assign blocks. More details can be found in Section 2.1.1. We use AVX and OpenMP for acceleration.
- GPU-Only: Only GPU works. We vary the number of rows and columns for the matrix division and adopt the best one. The ”-O3” optimization flag is supported.
- HSGD: CPU and GPU work in parallel. The algorithm is introduced in Section 3.2.1. AVX, OpenMP, and ”-O3” optimization flag are supported.

- HSGD*: CPU and GPU work in parallel. Nonuniform matrix division and dynamic strategy are used. Our cost model decides the size of blocks assigned to two hardware resources. AVX, OpenMP, and "-O3" optimization flag are supported.

Stochastic gradient methods and the parameter k used in [121] and [109] are different. To correctly combine two methods on Heterogeneous CPU-GPU Systems, we embed the core part of **LIBMF**[2] and **CuMF_SGD**[3] into our code and make minor modifications to make the stochastic gradient methods they use consistent. For stochastic gradient method, We choose to use the more concise one in [121]. For the value of parameter k, we choose the larger one in [109] because a large k value can lead to a better training result.

Datasets and Parameter Setting. We evaluate algorithms in four real-world datasets — *MovieLens*[4], *Netflix*[5], *R1*[6], and *Yahoo!Music*[7]. Statistics of the datasets are presented in Table 3.1. For reproducibility, we consider the original training/test sets in our experiments. More details about each dataset can be found on the corresponding website. We set the parameters following [20], which are also listed in Table 3.1.

Experimental Environment. We use a machine with Intel Xeon E5-2687W v3 3.10GHz processors and a Quadro P4000 GPU with 8GB global memory. The number of available cores is 20. The system interface of the GPU is PCI Express 3.0×16. The total bandwidth is 32GB/s. By default, we use 16 CPU threads and 128 GPU parallel workers. Here, we follow the definition of GPU parallel workers in [109], which means the number of elements computed simultaneously

[2]`https://github.com/cjlin1/libmf`

[3]`https://github.com/cuMF/cumf_sgd`

[4]`http://grouplens.org/datasets/movielens/`

[5]`https://www.kaggle.com/netflix-inc/netflix-prize-data`

[6]`https://webscope.sandbox.yahoo.com/catalog.php?datatype=r`

[7]`https://webscope.sandbox.yahoo.com/catalog.php?datatype=c`

Table 3.1: Network statistics and parameter settings

Datasets	MovieLens	Netflix	R1	Yahoo!Music
m	71,567	2,649,429	1,948,883	1,000,990
n	65,133	17,770	1,101,750	624,961
$\#Training$	9,301,274	99,072,112	104,215,016	252,800,275
$\#Test$	698,780	1,408,395	11,364,422	4,003,960
k	128	128	128	128
λ_P	0.05	0.05	1	1
λ_Q	0.05	0.05	1	1
γ	0.005	0.005	0.005	0.01

in the GPU kernel. All datasets can fit in memory in our experiments.

Organization. Section 3.5.1 shows the adaptiveness of our algorithms by varying the computing resources. Section 3.5.2 shows the effectiveness of our algorithm compared with the state-or-the-art competitor. Section 3.5.3 and Section 3.5.4 evaluate our optimization techniques including matrix division strategy and workload balance.

3.5.1 Overall efficiency

We evaluate the overall efficiency of our final algorithm HSGD* with CPU-Only and GPU-Only as comparisons. We use Root Mean Square Error (RMSE) [8] as a metric for the loss, which is widely used in many recommender systems. For each dataset, we terminate all algorithms and record the corresponding running time when the RMSE reaches a predefined value. Given that we use a different stochastic gradient method from [109] and a different k value from [121], the predefined loss values they used are not available. For fair comparison, we select these values that can be reached by all methods including HSGD which suffers from a weak training quality. The predefined loss values are 0.66, 0.82, 20, and 19 for MovieLens, Netflix, R1, and Yahoo!Music, respectively. The comparisons

[8] `https://en.wikipedia.org/wiki/Root-mean-square_deviation`

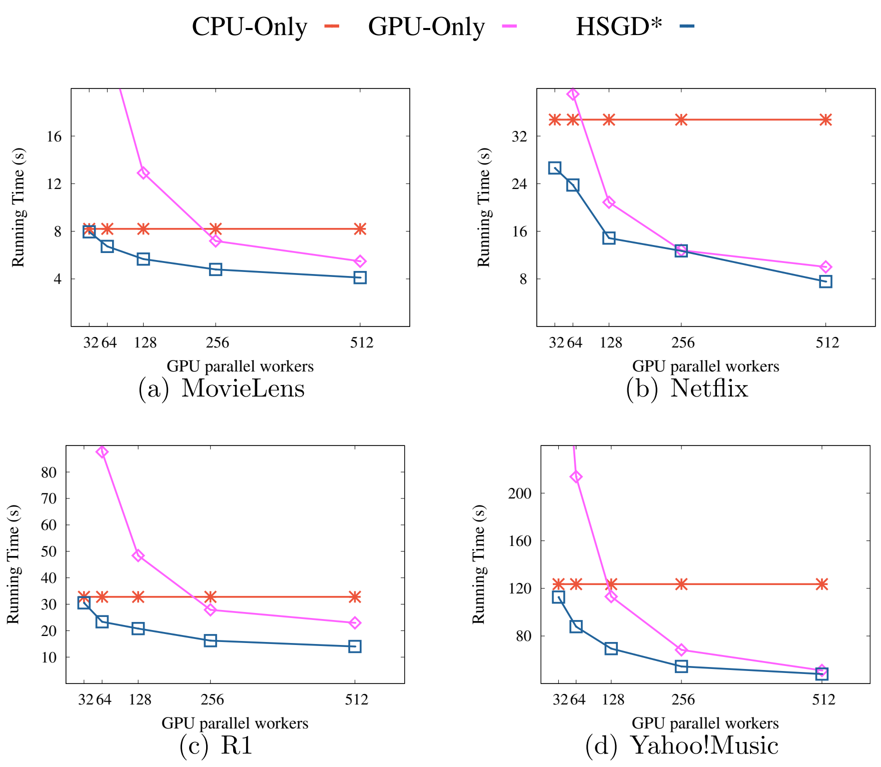


Figure 3.9: Varying GPU Threads

between HSGD and HSGD* will be shown in Section 3.5.3.

Varying GPU parallel workers

In this experiment, we evaluate the adaptiveness of our algorithm by varying the GPU parallel workers from 32 to 512. The running times of algorithms for different GPU parallel workers are reported in Figure 3.9 for four datasets. Note that the CPU thread number is fixed to the default value 16.

As a reference, the running time of CPU-Only is stable on all settings. Initially, the GPU-Only is slower than CPU-Only. When we use more GPU threads, the running time of GPU-Only decreases and overtakes that of CPU-Only. The running time of HSGD* is the smallest among all algorithms. For example, in

R1, given 32 GPU worker threads, HSGD* takes 30 seconds while CPU-Only and GPU-Only take 33 seconds and 170 seconds respectively. When the thread number increases to 512, HSGD* takes 14 seconds while GPU-Only takes 23 seconds. The decrease of the time of HSGD* also shows that our algorithm is adaptive to different GPU settings and can fully utilize the increasing computing power of GPUs.

Varying CPU thread Number

In this experiment, we evaluate the adaptiveness of our algorithm by varying the CPU thread number from 4 to 16. The GPU parallel workers are fixed to the default value 128. The running times of algorithms for different CPU thread numbers are reported in Figure 3.10.

In contrast to Figure 3.9, the running time of GPU-Only is consistent, and the running time of CPU-Only decreases when we use more CPU threads. HSGD* is the fast algorithm on all settings and all datasets. For example, in R1 when the CPU thread number is 4, HSGD* takes 29 seconds, while CPU-Only takes 109 seconds, and GPU-Only takes 48 seconds. When the CPU thread number increases to 16, HSGD* takes only 20 seconds while CPU-Only takes 33 seconds. The decrease of the time of HSGD* shows that our algorithm is adaptive to different CPU settings and can fully utilize the increasing computing power of CPUs.

Figure 3.10 and Figure 3.9 show the high efficiency of HSGD*. When the gap between the computing power of CPU and GPU is limited (default setting), HSGD* achieves a 1.4-2.3x speedup over CPU-Only and a 1.4-2.3x speedup over GPU-Only on all datasets. The experiments also show that the overhead cost of HSGD* is minor. When the gap between the computing power of CPU and GPU is large, e.g., CPU uses 16 threads and GPU uses 512 parallel workers in

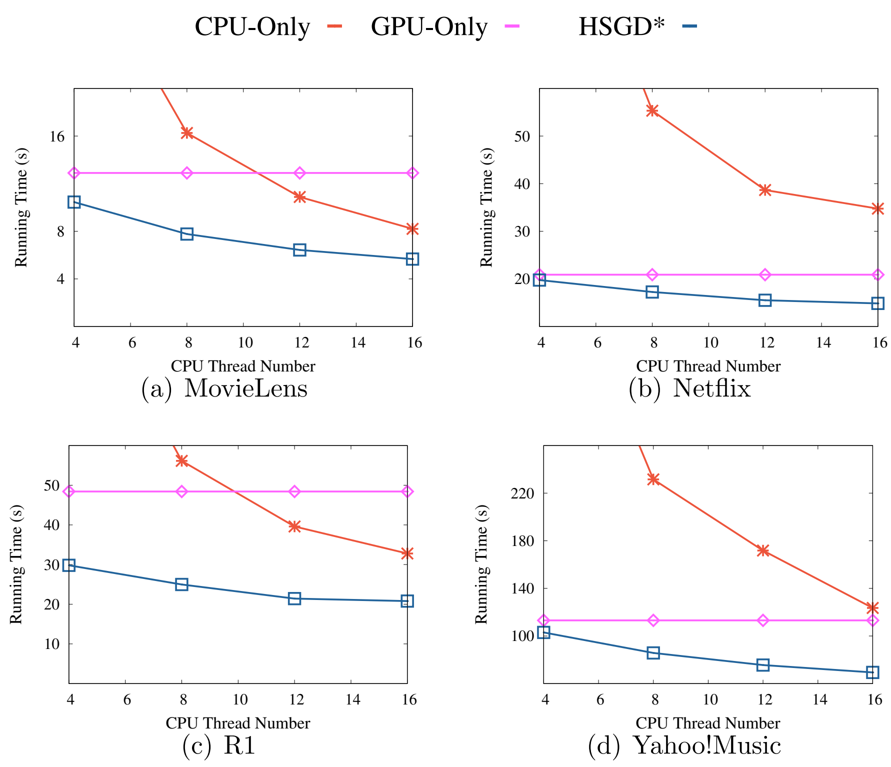


Figure 3.10: Varying CPU Threads

Figure 3.9, HSGD* still achieves a slight speedup over GPU-Only.

3.5.2 Training quality

In this experiment, we report the derived loss values (RMSE) of our algorithm HSGD* during the training process to show the effectiveness of our method. The experiment will demonstrate the loss value of our algorithm finally converges to a reasonable value. CPU-Only and GPU-Only are also compared as references.

The results are shown in Figure 3.11. The downward trends of HSGD* are obvious, and the loss of HSGD* converges in the shortest time. In addition, HSGD* achieves a similar converged loss value compared with other algorithms,

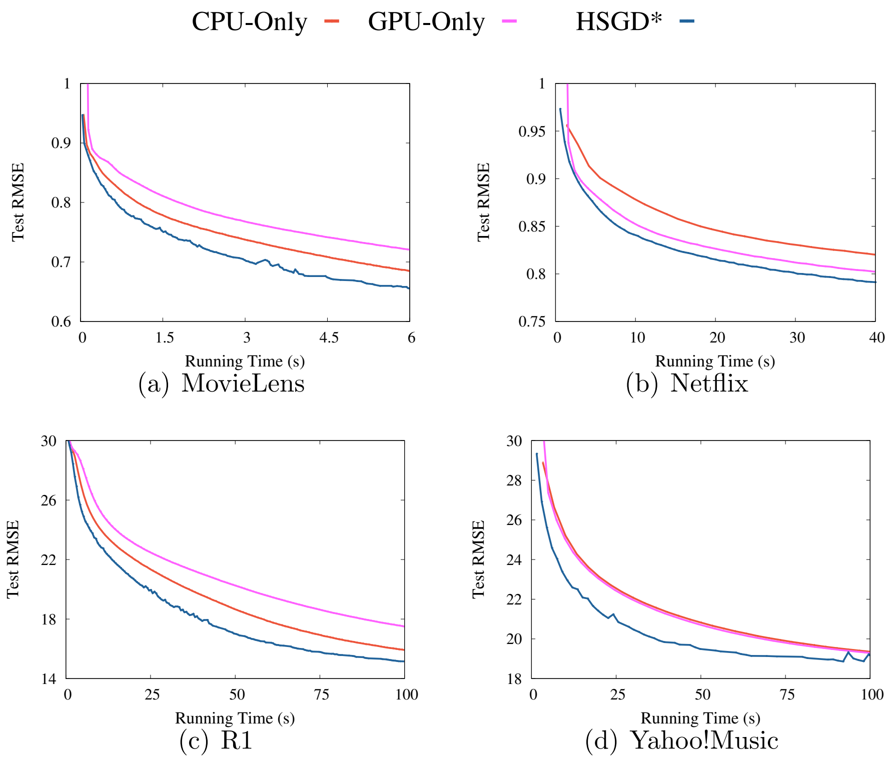


Figure 3.11: Test RMSE over training time on four datasets

which shows the effectiveness of our algorithm. For example, in Yahoo!Music, the loss of HSGD* drops to 23 in 10 seconds. At the same time, the loss values of CPU-Only and GPU-Only are 25.2 and 25, respectively. When time increases to 25 seconds, the loss of HSGD*, CPU-Only, and GPU-Only drops to 20, 22.5, and 22.3, respectively. Finally, all loss values of HSGD*, CPU-Only, and GPU-Only stay about 19.

3.5.3 Matrix division strategy

We evaluate the effectiveness of our matrix division strategy in this experiment. For each dataset, we record the loss (RMSE) value on different running times of

Table 3.2: Comparison of cost models

Datasets		MovieLens	Netflix	R1	Yahoo!Music
Workload proportion					
HSGD*-Q	C	49.56%	55.98%	56.07%	56.46%
	G	50.44%	44.02%	43.93%	43.54%
HSGD*-M	C	55.91%	49.02%	49.75%	53.61%
	G	44.09%	50.98%	50.25%	46.39%
Running time					
HSGD*-Q		0.92 s	15.87 s	13.07 s	40.88 s
HSGD*-M		**0.89** s	**13.02** s	**12.08** s	**35.41** s

HSGD* with HSGD as a comparison. The result is shown in Figure 3.12.

We can see that the training quality of HSGD is poor especially when processing relatively large datasets. This phenomenon is consistent with the discussion in Example 3. The nonuniform matrix division in HSGD* fixes this issue. Given the same running time, HSGD* derives a smaller loss value than HSGD, and the advantage of HSGD* is obvious especially in large datasets. For example, given 50 seconds in R1, the RMSE value for HSGD* reaches to 17, while that for HSGD is only 21. The result proves that nonuniform matrix division can utilize GPU resources better.

3.5.4 Workload balance

We evaluate the effectiveness of techniques used to balance workloads in this section.

Cost models

To show the effectiveness of our cost models, we report the proportion of workloads derived by our cost models with [72] as a comparison in Table 3.2. To clearly reflect the algorithmic efficiency based on two cost models, we make these

Table 3.3: Effectiveness of dynamic scheduling

Dataset	HSGD*-M	HSGD*
MovieLens	0.89 s	**0.84** s
Netflix	13.02 s	**11.42** s
R1	12.08 s	**10.58** s
Yahoo!Music	35.41 s	**30.96** s

two methods run the same number of iterations, which is 20 in this experiment.

In Table 3.2, HSGD*-Q represents the algorithm HSGD* which uses Qilin [72] to evaluate the working efficiency of hardware. HSGD*-M represents the algorithm HSGD* which uses our model in Section 3.3 to evaluate the working efficiency of hardware. Note that for fairness, both HSGD*-Q and HSGD*-M do not include the dynamic scheduling strategy in Section 3.4 to further balance workloads. "C" and "G" in the table represent the assigned proportion of workloads to CPUs and GPUs, respectively.

The practical running times of HSGD*-Q and HSGD*-M are also reported. The running time of HSGD*-M is smaller than that of HSGD*-Q on all datasets. This result proves that our cost model can derive a more accurate estimation for the working efficiency of hardware. We can find that HSGD*-M prefers to assign more work to GPU compared with HSGD*-Q on all datasets except MovieLens. For MovieLens, HSGD*-M observes that the performance of GPU is not strong when processing a small dataset (Observation 1). Therefore, it assigns less work to GPU. The effectiveness of HSGD*-M becomes considerable when the dataset is large. Note that given a smaller target loss, both algorithms will require more iterations, and the advantage of our method will become obvious. For example, to achieve the predefined loss value of Yahoo!Music in Section 3.5.1, HSGD* needs 46 iterations, which is more than twice that of this experiment.

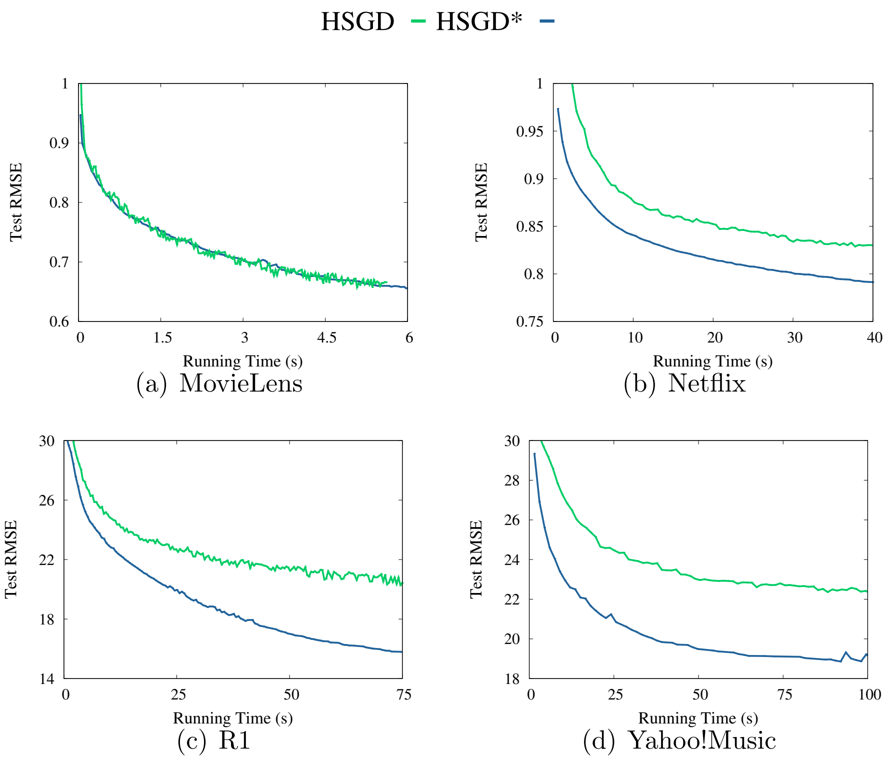


Figure 3.12: Test RMSE over training time

Dynamic scheduling

We evaluate the effectiveness of the dynamic scheduling strategy (Section 3.4). Similar to the experiment for cost models, we use HSGD*-M to denote our final algorithm without the dynamic scheduling technique to further balance workloads. The running times of HSGD*-M and HSGD* on all datasets are shown in Table 3.3.

The result shows HSGD* is faster than HSGD*-M on all datasets. Note that in MovieLens with relatively small size, the computing power of GPU cannot be saturated, which degrades the effectiveness of the dynamic scheduling. As a result, the improvement of dynamic scheduling on MovieLens is minor. By combining the new cost models and the dynamic scheduling strategy, our final algorithm achieves a significant improvement in balancing workloads of MF.

3.6 Conclusion

In this chapter, we delve into addressing the matrix factorization problem on heterogeneous systems.

We discover that a non-uniform partitioning strategy can boost GPU efficiency while sustaining high recall. Based on this key observation, we introduce a cost model to determine the sizes of large and small matrix blocks and dynamically allocate tasks at runtime, aiming to achieve load balancing across two working units.

Experimental findings demonstrate that our method yields a performance acceleration ranging from 1.4 to 2.3 times when compared with CPU-only or GPU-only methods. Furthermore, experimental results indicate that our cost model and dynamic scheduling strategy reduce running time due to their effective load balancing.

While our approach exhibits potential, it also comes with certain limitations. Our approach is tailored for discrete heterogeneous systems where the CPU and GPU have their own separate memory spaces. In the context of integrated heterogeneous systems, where data transfer between CPU and GPU memory is absent, our cost model may not hold its accuracy.

Chapter 4

k Nearest Neighbors Search

In this chapter, we introduce the methodological details about our approach to k nearest neighbors search along with the experimental results.

4.1 Preliminaries

In this section, We formalize the problem of ANN search on proximity graphs and review the search algorithm on proximity graphs along with representative proximity graphs.

4.1.1 Problem definition

Definition 1. *(**K Nearest Neighbor Search**). Given a set of points P ($|P| > k$) and a distance function δ in a space S, k nearest neighbor search for a query point $q \in S$ is to return a set of points $N \subseteq P$ ($|N| = k$) such that $\forall u \in N, \forall v \in P \setminus N$,*

$$\delta(u, q) \leq \delta(v, q). \tag{4.1}$$

Due to *the curse of dimensionality* [45], considerable research efforts turn to searching approximate k nearest neighbors (ANN) when the dimensionality of the space S is high. In this way, we might find a good trade-off between result quality and search efficiency. Let $X = \{x_i | 1 \leq i \leq k\}$ denote the result by an approximate algorithm, a common way to measure precision for the query point q is defined as $\frac{|X \cap N(q)|}{k}$ where $N(q)$ includes k nearest neighbors of the query q.

A variety of proximity graphs have been proposed in the literature to facilitate approximate nearest neighbor search. Below is a general definition of the proximity graph followed by two important properties.

Definition 2. ***(Proximity Graph)**. Given a set of points P in a multidimensional space S and a distance function δ, a proximity graph $G = (V, E)$ of P consists of a set of n vertices $V = P$ where each vertex in V is uniquely associated with a point in P, and a set of m edges E each of which connects two vertices in V.*

Without loss of generality, we assume a proximity graph G is a directed graph. Whenever there is no ambiguity, we use a point and its corresponding graph vertex exchangeably. Generally, there are two key properties for proximity graphs used for ANN search:
(1) For each vertex v, all or the majority of its outgoing neighbors are close to v in terms of the given distance function δ. A few outgoing neighbors not close to v might also be included to speed up the search in some proximity graph models (e.g., long edges in NSW graph [73]).
(2) All vertices have the same (similar) number of outgoing neighbors. We may use an upper bound d_{max} and a lower bound d_{min} to control the number of outgoing neighbors in proximity graphs. The bounded number of neighbors is friendly for GPU-based graph processing. Note that we only keep the outgoing neighbors in adjacency lists.
Problem statement. We aim to develop efficient *GPU-accelerated* algorithms for approximate nearest neighbor (ANN) search on a proximity graph as well as

constructing proximity graphs.

4.1.2 Proximity graph search and construction

Search on proximity graph. Though a variety of proximity graph models have been proposed in the literature, most of them use the *beam search* strategy which is an A^*-like search algorithm with a limited budget. The key idea of the beam search on proximity graphs is the greedy heuristic that guides the search, i.e., (1) choose the unvisited neighbor vertex which is closest to the query q in the graph search. (2) apply the backtracking to avoid a locally optimal solution under a particular budget, i.e., search more nearest neighbors than required for exploring neighbors of local optimum.

Algorithm. Given a proximity graph $G = (V, E)$, a query point q and a number k of returned nearest neighbors, Algorithm 4 searches nearest neighbors in iterations. (1) A max-heap N, a min-heap C and a hash table H are initialized (Lines 1-3). (2) A start vertex v_s is added to C and H to start iterations up (Lines 4-5). (3) As iterations begin, for each iteration, (a) pick the vertex v_c closest to q in C (Line 7). (b) check whether v_c is closer than the current k-th nearest neighbor v_f in N (Line 9). If the number of vertices in N is smaller than k, we regard $\delta(v_f, q) = INF$. (c) If the condition does not hold, the search terminates (Line 10). Otherwise, v_c is added to N (Line 14), and its unvisited outgoing neighbors are inserted into C for subsequent exploration and marked as visited (Lines 15-17).

Example 1: Given the proximity graph G_1, the query point q in Figure 1.1 and $k = 4$, Algorithm 4 returns k-nearest neighbors of q. In the beginning, N, C and H are empty. Suppose v_1 is chosen as the entry point, it is added to C and H. Then, the first iteration starts up. In iteration 1, as the closest point to q in C, v_1 is popped up from C. Afterwards, v_1 is directly pushed into N

Algorithm 4: Search on Proximity Graph

```
   Input: A proximity graph $G = (V, E)$, a query point $q$ and the number of
          returned nearest neighbors $k$
   Output: A set $N$ of $k$ nearest neighbors of $q$ in $V$
1  $C := \emptyset$; // candidate set
2  $N := \emptyset$; // top $k$ result so far
3  $H := \emptyset$; // visited points set
4  pick a start vertex $v_s$;
5  $C := C \cup \{v_s\}$; $H := H \cup \{v_s\}$;
   // search on the proximity graph
6  while $|C| > 0$ do
       // pop the point $v_c$ closest to $q$
7      $v_c := C$.Min(); $C$.Pop();
       // $k$-th closest neighbor so far
8      $v_f := N$.Max();
9      if $\delta(v_c, q) > \delta(v_f, q)$ and $|N| = k$ then
10         break;
11     if $|N| > k$ then
12         $N$.Pop();
13     $N := N \cup \{v_c\}$;
14     foreach outgoing neighbor $u$ of $v_c$ in $G$ do
15         if $u \notin H$ then
16             $C := C \cup \{u\}$; $H := H \cup \{u\}$;
17 return $N$;
```

because the number of points in N is lower than k. Last, all neighbors of v_1 are sequentially added to C and H because they are neither visited. In iteration 2, v_8 is popped up from C and pushed into N. Its unvisited neighbor v_{10} is added into C and H. In subsequent iterations, the search algorithm traverses the path $v_{10} \rightarrow v_{12} \rightarrow v_9$, thereby visiting all neighbors of points on this path. After iteration 5, the closest point to q in C is v_4, which is worse than v_{10} (the furthest point to q in N). Hence, traversal terminates, and v_{12}, v_9, v_8 and v_{10} in N are returned.

Construct proximity graph. A variety of proximity graph models have been proposed in the literature. Here, we focus on two representative models: Navigable Small World (NSW) graph and K Nearest Neighbors (KNN) graph.

Navigable Small World Graph. In an NSW graph, the outgoing edges of each vertex include short-range links and long-range links. Short-range links are regarded as an approximation of the Delaunay Graph [10], and long-range links maintain the properties of the small world [56]. The construction process is a serial insertion of all points in the dataset. For each new point, (1) d_{min} nearest neighbors are selected from points in the current graph (all points are selected if the number of points in the current graph is less than d_{min}). (2) selected nearest neighbors are *bidirectionally* linked with the new point. The insertion continues until all points have been inserted into this graph. G_1 in Figure 1.1 is constructed as an NSW graph where d_{min} is set to 2. For example, among three neighbors of v_7, v_1 and v_5 are picked because they are closest to v_7 when v_7 is inserted; v_8 is picked because v_7 is one of the closest neighbors of v_8 when v_8 is inserted.

As its important variant, the hierarchical navigable small world graph (HNSW) is a hierarchical graph where each layer is an NSW graph for a subset of points.

K Nearest Neighbors Graph. In a KNN graph, each point is connected to its k (Here, $k = d_{min} = d_{max}$) nearest neighbors. NN-Descent [25] is often used to construct the KNN graph. It first randomly picks neighbors for each point and iteratively improves the quality of neighbors of each point by exploring the neighbors of its neighbors. The process terminates when the precision improvement of the KNN graph is small enough.

4.2 Proximity graph search

First, we introduce the motivation of our GPU-based proximity graph search. We next present the search algorithm and a theoretical analysis.

4.2.1 Motivation

We note that the distance computation can be significantly accelerated by GPU-based solutions because the sub-vectors of an entire feature vector (i.e., a point in high-dimensional space) are independent and can naturally be processed simultaneously. For instance, we can simultaneously use 32 threads in a warp to compute the Euclidean distance between two points (i.e., vectors) in 640-dimensional space where each thread takes care of the 20-dimensional sub-vector, and the partial results can be easily aggregated by warp-level primitives. However, as stressed in [119] and illustrated in Figure 4.5 in Section 4.4, the bottleneck of the state-of-the-art graph search becomes the data structure operations. This is because SONG still follows the search paradigm of CPU-based solutions and the corresponding data structure operations (e.g., dynamic maintenance of the priority queue and hash table) are expensive on a GPU. Though a set of optimizations have been explored, SONG uses a single thread for data structure operations in each query to avoid irregular dependency among threads in their implementations. This inherently underutilizes the GPU computing bandwidth since multiple threads with the same instruction will be invoked at the same time even when the smallest computing unit, i.e., the warp, is used for each query.

To better maintain the data structures in the GPU solution, we apply the lazy strategy on the updating and checking operations. Though the lazy strategy has been widely adopted in the literature, we would like to point out that the existing works on proximity graph based ANN search, including the state-of-the-art GPU solution, maintain the data structures in an eager manner. To better apply a lazy strategy for the GPU-friendly solution, we need to choose new data structures and re-design the search paradigm, which, together, pose new challenges. We re-design the search algorithm so that the maintenance of

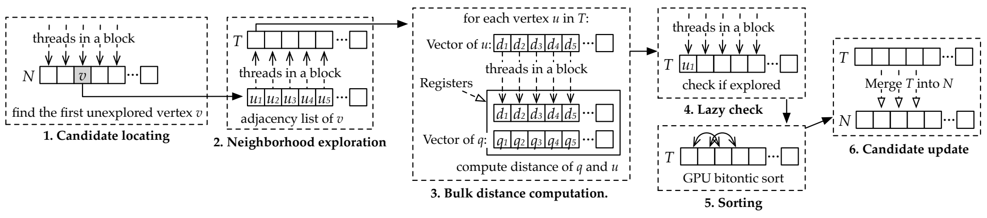

threads in a block
N
v
find the first unexplored vertex v
1. Candidate locating
T
threads in a block
u_1 u_2 u_3 u_4 u_5
adjacency list of v
2. Neighborhood exploration
for each vertex u in T:
Vector of u: d_1 d_2 d_3 d_4 d_5
threads in a block
Registers
d_1 d_2 d_3 d_4 d_5
Vector of q: q_1 q_2 q_3 q_4 q_5
compute distance of q and u
3. Bulk distance computation.
threads in a block
T u_1
check if explored
4. Lazy check
T
GPU bitontic sort
5. Sorting
T
Merge T into N
N
6. Candidate update

Figure 4.1: A diagram of search algorithm GANNS

data structures is friendly to GPU. The key challenge is to implement efficient parallelism during the update of data structures while ensuring that iterations can keep running. We focus on the following two perspectives.

Candidate selection. It is non-trivial to have an efficient GPU implementation for priority queues as stressed in [119]. A set of optimization techniques has been proposed by SONG such as bounded priority queue optimization and selected insertion optimization to accommodate the GPU computation. Nevertheless, SONG has to rely on a single thread for the above process because it extracts the closest point sequentially from a priority queue, which is efficient in terms of following the greedy heuristics and quickly approaching the target but not friendly for the parallel computation in GPU.

Lazy update. To alleviate the above issue, an alternative is to use a concurrent heap [19] that supports parallel operations. Unfortunately, it is inefficient in the context of ANN search. This is because the number of points stored in the heap is few in our problem. As a result, the depth of the heap is low. This means that we have to sequentially update almost all vertices in the heap when we insert or delete points.

Instead of dynamically maintaining the candidate set and current top k result with two priority queues, we use a *lazy update* strategy with two fixed-length arrays N and T with lengths l_n and l_t, respectively. We say that

◦ A vertex is a **visiting** vertex regarding q if its distance to q will be calculated.

∘ A vertex is an **exploring** vertex regarding q if the distances of its neighbors to q will be calculated.

N keeps the top k results and potential exploring vertices at the same time. In each iteration, a vertex in N is picked as the exploring vertex; T stores visiting vertices that are outgoing neighbors of the exploring vertex.

The selection of an exploring vertex can be based on warp-level primitives. N and T can be updated by GPU sorting and merging algorithms, i.e., the maintenance of the candidate set and current top k result can be efficiently paralleled.

Visited vertices mark. During search, a vertex might be accessed multiple times when its neighbors are explored. Thus, it is necessary to mark the visited vertices to reduce the redundant computation. In CPU-based proximity search, a hash table can be used for efficiently checking if a vertex has been visited (Line 16 of Algorithm 4). In this way, re-computation of the visited vertices can be avoided. In [119], two possible alternatives are discussed for this purpose: opening addressing hash table and bloom filter, followed by visited deletion optimization. The open addressing hash table with a single thread is used in the implementation of SONG because the overhead of synchronization does not payoff. As an alternative, one may wonder if the well-known bitmap hashing can be applied since it can be easily paralleled without any synchronization cost. Unfortunately, this is not efficient on the GPU because of the high latency of the random memory accesses involved in the warp threads and the limited on-chip memory [66].

Lazy check. Based on the above observation and the fact that the distance computation in the context of GPU computation is very efficient, we do not check if a neighbor of v has been visited. Consequently, we can avoid the use of a hash table at the cost of some redundant distance computations. Note that we still assess if a vertex has been explored by checking the array N to avoid the propagation of this redundant distance re-computation.

4.2.2 GPU-based proximity graph search

We first show the data structures used in the search algorithm and corresponding memory space allocation. The search algorithm is presented next.

Data Structures and Memory Allocation. Data structures in our search algorithm, as well as corresponding GPU memory allocation are as follows. (1) The proximity graph G, high-dimensional points (i.e., features), and high-dimensional queries, are kept in the GPU global memory. (2) N and T are allocated in the shared memory. (3) The coordinate values (i.e., feature) of each point, including the query q, will be assigned to the registers.

Search Algorithm. Note that we allocate a thread block for one query. Since ANN search of each query points is independent, the *inter-block level parallelism* is immediate by utilizing multiple thread blocks. Next, we illustrate our GPU-based ANN search algorithm in one thread block.

Initially, T is empty, and N includes the start vertex v_s. We use $N[v]$ (resp. $T[v]$) to denote the array element associated with the vertex v in N (resp. T), and $N[v]$.explored (resp. $T[v]$.explored) to indicate if v has not been explored. Then, search starts up. Each iteration of the search is illustrated in Figure 4.1. It consists of 6 phases as follows.

(1) *Candidate locating.* When each iteration starts, the first vertices v with $N[v]$.explored = false in N will be identified. In particular, threads access the flag *explored* of vertices in parallel. Warp-level primitives *__ballot_sync* and *__ffs* are used to aggregate these flags to reveal the first unexplored vertex. If these vertices are all explored, subsequent vertices are processed in the same way. The

search will be terminated if all vertices in N are already explored.

(2) *Neighborhood exploration.* Let $\{u_i\}$ denote outgoing neighbors of v. They are loaded into T by collaboration of threads in the thread block, and we set $T[u_i]$.explored as false. Then, v is marked as explored.

(3) *Bulk distance computation.* Distances between vertices in T and q are computed one by one. For each vertex in T, its corresponding d-dimensional vector (point) is transferred from the GPU global memory to the registers of n_t threads. Each thread is responsible for the computation of its corresponding sub-vector, and the partial results are aggregated by the warp-level primitive *__shfl_down_sync*.

(4) *Lazy check.* As discussed in Section 4.2.1, there might be redundant distance computation because we do not check whether a vertex has been visited when it is inserted to T. Nevertheless, we will exam if vertices in T have been explored before they are merged into N to prevent the propagation of redundant computation. Given that vertices in N are sorted by their distances w.r.t q, we perform a parallel binary search for vertices in T. $T[v]$.explored is set to true if v is already in N because we do not need to consider v for neighborhood exploration again.

(5) *Sorting.* The well-known GPU algorithm bitonic sort [12] is employed to sort vertices in T based on their distances w.r.t q and the flag *explored*, and ties are broken by vertex ID.

(6) *Candidate update.* Since vertices in T and N have been ordered, we apply the bitonic sorting-based merge [53] algorithm to retrieve l_n closest vertices among $T \cup N$, and keep results in N for further processing. Note that it is possible that v has been explored in N, then discarded from N. In this scenario, v will not appear in N again since the distance of the l_n-th vertex of N will keep decreasing during the search. In this way, we can avoid exploring the neighbors of a vertex

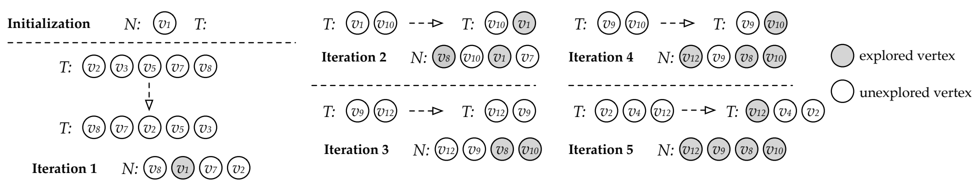


Figure 4.2: An example of search algorithm GANNS

multiple times though its distance to q might be re-computed.

Example 2: Given $k = 4$, the proximity graph G_1 and the query point q in Figure 1.1, our search algorithm returns k-nearest neighbors of q. Similarly, we assume that the entry point is v_1. In the beginning, it is loaded into N, and its flag *explored* is initialized as false.

In iteration 1, the first point not yet explored in N is the point v_1. Therefore, its neighbors v_2, v_3, v_5, v_7 and v_8 are loaded into T, and the flag *explored* of v_1 in N is set to true. Then, distances between vertices in T and q are computed. During lazy check, no point has a duplication in N. As a result, none of the points are set as explored. After sorting, the order of vertices in T becomes v_8, v_7, v_2, v_5 and v_3. Last, we update N by merging it with T. In iteration 2, v_8 is recognized as the next explored point and its neighbors are loaded, computed, checked, sorted and merged. Subsequently, the path $v_{10} \rightarrow v_{12} \rightarrow v_9$ is traversed. In iteration 5, the only unexplored point in N, v_9 is chosen as the next explored point. However, its neighbors make N unchanged. Meanwhile, points in N are all explored. Hence, the process terminates, returning v_{12}, v_9, v_8 and v_{10}.

As shown, though our search algorithm has the same search path as Algorithm 4 and SONG, neighbors of the exploring point in each iteration are considered in a batch rather than one by one. This increases parallelism and reduces the cost of maintaining data structures. Moreover, compared with Algorithm 4 and SONG, our search algorithm consumes less memory because it aborts the maintenance of visited points.

Remarks. To justify the practicality of solving ANN search on the GPU, we also investigate the impact of data transfer between CPU and GPU on total time cost. Specifically, compared with the time of querying on the GPU, the time of data transfer between CPU and GPU is negligible. This is because (1) data transfer occurs only at the beginning and end of search, and the size of data is minor compared with bandwidth. For instance, the size of results is around 1MB when there are 2000 queries in one batch and k is set to 100. However, the bandwidth of data transfer (PCI Express 3.0×16) is around 10GB/s; (2) CUDA provides a stream mechanism that supports asynchronous processing of kernel computation and data transfer. That is to say, data transfer can be overlapped with querying on the GPU even when several batches of points need to be processed on the GPU.

4.2.3 Analysis

Memory Usage. Memory usage is of importance in GPU-based algorithms. The proposed search algorithm makes much effort to (1) avoid additional buffer such as auxiliary arrays; (2) reduce the usage of shared memory of one thread block to enhance potential parallelism, i.e. the sizes of N and T are profitable. For instance, l_n usually takes 32, 64 or 128 while l_t, is set to d_{max} which is 32 by default; and (3) utilize registers. As reported in [47], the register is the largest SRAM structure on the chip, usually around 256KB per SM. Instead of ignoring this important memory structure, we load the vector of q and points into registers for the distance computation.

Time Complexity. Given the number n_t of threads in a thread block, the complexity of phases (1) and (2) is $O(\frac{l_n}{n_t})$ and $O(\frac{l_t}{n_t})$, respectively; for phase (3), the complexity is $O(l_t \times \frac{n_d}{n_t})$ where n_d is the dimensionality of the points;

for phase (4), the complexity is $O(\log l_n \times \frac{l_t}{n_t})$; for phase (5), the complexity is $O(\log^2 l_t \times \frac{l_t}{n_t})$; and for phase (6), the complexity is $O(\log l_n \times \frac{l_n}{n_t})$. Putting them together, the overall cost is $O(\log l_n \times \frac{l_t+l_n}{n_t} + l_t \times \frac{n_d}{n_t} + \log^2 l_t \times \frac{l_t}{n_t})$ time for each iteration of the search. As a reference, the time complexity of three phases of SONG is $O(l_t)$, $O(l_t \times \frac{n_d}{n_t})$ and $O(l_t \times \log l_n)$, respectively. Theoretically, the speedup of both bulk distance computation and data structure operation is linear to the number of threads n_t within a thread block.

4.3 Proximity graph construction

We first present the motivation of our GPU-based NSW graph construction algorithm. We next describe the algorithm and give an analysis. Then, we briefly show the techniques for the construction of two other popular proximity graphs: HNSW and KNN graphs.

4.3.1 Motivation

Intuitively, the improvement of the search performance by SONG and our search algorithm can immediately accelerate NSW graph construction. However, it is nontrivial due to the sequential insertions during construction.

Straightforward methods. Generally, we have two straightforward GPU implementations for NSW graph construction as follows.

Sequential method. We can strictly follow the *sequential processing* of the points $\{v_1, v_2, \ldots\}$. Specifically, for a new point v_i, we conduct ANN search on the current NSW graph G, put its k nearest neighbors obtained from ANN search $\{u_1, \ldots, u_k\}$ as the adjacency list of v_i, and update the adjacency list of u_i with v_i as well. Then v_{i+1} will be processed on the new graph G', including edges created by the insertion of v_i.

Parallel method. We can simply design a parallel method by *parallel processing* of the points $\{v_i, v_{i+1}, \ldots\}$. In particular, ANN search on the current NSW graph G is performed in parallel for each point in the batch, and results are aggregated to update edges together.

Remarks. (1) The sequential method is inefficient because of the waste of the GPU computing bandwidth, i.e., there is no inter-block level parallelism. (2) The parallel method suffers from the quality of graphs. This is because all other points in the same batch are ignored during the graph construction for each point v_i, e.g., they will not appear in its adjacency list.

The pitfalls of the above two methods prompt us to design a new NSW graph construction algorithm such that we can fully exploit the GPU computing bandwidth without sacrificing the quality of the constructed graphs. In a nutshell, we use the divide-and-conquer strategy. As shown in Figure 4.3, we partition the points P into a set of disjoint groups $\{P_0, \ldots, P_t\}$ of the same size. In the first phase, we process each group P_i by one thread block and build a *local* NSW graph G_i by sequentially processing the points. Then the local graphs will be successively merged to the first local graph G_0. We carefully design the implementation to ensure that (1) all operations in the graph construction are friendly to GPU; (2) both inter-block level parallelism and intra-block level parallelism are achieved; and (3) the quality of the resulting graphs is the same as the NSW graph constructed by sequential insertions.

4.3.2 GPU-based NSW graph construction

We first demonstrate data structures as well as their locations in the memory hierarchy. Notice that the search process is also included in the construction process, and we only introduce new data structures here.

Data Structures and Memory Allocation. (1) The proximity graph G

is pre-allocated in the global memory to store neighborhood information of all points. (2) Graph G' is also pre-allocated in the global memory. It is used to store intermediate neighborhood information for the optimization of our GPU solution. (3) An edge list E is located in the global memory, and is used to record nearest neighbors for inserted points. (4) An array I is used for the processing of the edge list E, which is allocated in the global memory.

Construction Algorithm. Without loss of generality, we assume that all points are assigned a unique ID, representing the order in which they are inserted. Let $v.N$ denote the adjacency list of the vertex (point) v in the NSW graph. for each vertex $u \in v.N$, we say the edge $v \to u$ is a "forward edge" if u.ID ¡ v.ID. Otherwise, $v \to u$ is a "backward edge".

Generally, a forward edge $v \to u$ is generated in an aggressive way when v issues a KNN search in NSW graph construction, and u is included in the search result. A backward edge $v \to u$ is created in a passive way when u issues the kNN search, and v is one of the search results.

Algorithm 5 presents the pseudo-code of the NSW graph construction.

The term **parallel do across blocks** indicates the *inter-block level parallelism*, i.e., the task is parallel processed by multiple thread blocks. We first partition points in P into $t + 1$ disjoint groups P_0, P_1, ..., P_t (Line 1).

The following construction algorithm consists of two phases: *local graphs construction* and *local graphs merge*.

(1) *Local graph construction.* (Lines 2-8) We assign each group P_i to a thread block for local graph construction (Line 2). The computation of each local graph is independent so the inter-block level parallelism can be immediately achieved. These points in the group P_i will be sequentially inserted into the local graph G_i using the following two steps (Lines 4-8).

Step 1. For each inserted vertex v_{ij} in P_i, we search d_{min} nearest neighbors on

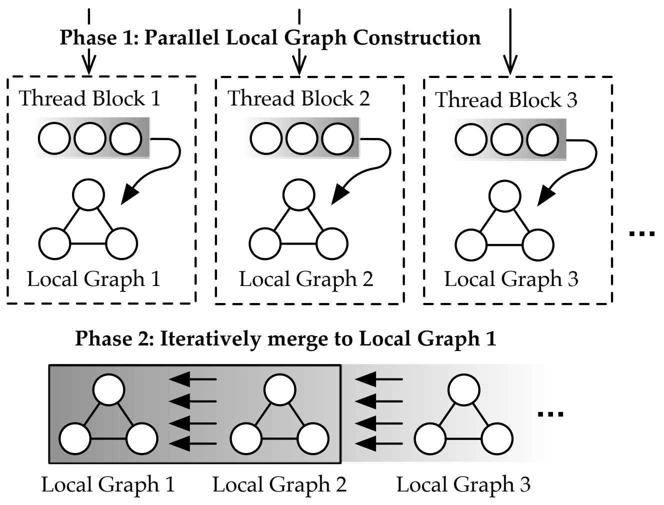


Figure 4.3: Our Strategy

G_i. The search results of each new point v_{ij} are stored in their corresponding adjacency lists in both G and G', denoted by $v_{ij}.N$ and $v_{ij}.N'$ respectively. Note that $v_{ij}.N'$ uses memory space allocated for G'.

Step 2. For each point u_{ij} in the adjacency list of v_{ij}, its adjacency list in G is updated. Specifically, we insert the vertex v_{ij} into the adjacency list of u_{ij}. We then locate the position by binary search and write the vertex id of v_{ij} and the distance between u_{ij} and v_{ij}. Note that the adjacency list of each vertex is an array with fixed size d_{max} where elements are ordered by distance. The last element is discarded if the list is already full.

(2) *Local graphs merge.* (Lines 9-22) After constructing local graphs, the last t graphs are sequentially merged into the first local graph G_0. It requires t iterations, and there are three steps in each iteration. Assume we are in the i-th iteration.

Step 1. For each vertex v_{ij} in G_i, it is processed by one thread block. The computations of these vertices are independent of each other in this step, hence the inter-block level parallelism is immediate.

First, d_{min} nearest neighbors of v_{ij} is retrieved against the graph G_0 which includes points in $P_0 \dots P_{i-1}$ (Line 12), and these nearest neighbors will merge with $v_{ij}.N'$ in G' to form new $v_{ij}.N$ (Line 14). Recall that $v_{ij}.N'$ stores nearest neighbors of v_{ij} among points in P_i which have smaller ID than v_{ij}. Consequently, the d_{min} nearest neighbors of v_{ij} among points accessed before v_{ij} are kept in $v_{ij}.N$, i.e., the forward edges of v_{ij} are now readily available.

Next, we update the backward edges. Note that the update of backward edges for points u_{ij} in $\bigcup_{v_{ij} \in P_i} v_{ij}.N$ is a bit tricky because the vertex u_{ij} might be included as one of the nearest neighbors in several adjacency lists, i.e., it could appear across multiple thread blocks. Hence, it might lead to inconsistent results if we do not have any concurrency control among blocks. Therefore, we put these backward edges into an edge list E for subsequent processing (Line 17).

Step 2. Here, we aim to organize these backward edges in compressed sparse raw (CSR) format ((Line 19)). (1) We employ bitonic sorting to organize edges in E by the IDs of the starting vertices (i.e., u_{ij}), with the ties broken by the distances. (2) $I[i]$ is set to 1 if the i-th edge in E is the first edge of a particular starting vertex; otherwise, $I[i] = 0$. (3) The prefix sum of I is computed, and we update I such that $I[i]$ is the beginning position of i-th starting vertex in E.

Step 3. Now, we update the backward edges of starting vertices in E. We assign each starting vertex to one thread block. Assume the vertex u_{ij} is assigned to the i-th thread block. (1) We use $I[i]$ and $I[i+1]-1$ to obtain the corresponding edges of u_{ij} in E, and load these edges from the global memory to the shared memory. (2) We load the current adjacency list of u_{ij} in G_0 from the global memory to the shared memory. (3) These two adjacency lists are merged, and we use the first d_{max} elements as the adjacency list of u_{ij}.

Consequently, we update G_0 by merging G_i, and G_0 is returned after all local

Algorithm 5: GPU-based NSW Graph Construction

```
Input: A point set $P$, the minimum degree $d_{min}$ in $G$ and the maximum
       degree $d_{max}$ in $G$
Output: A proximity graph $G = (V, E)$
1  partition $P$ into disjoint sets $P_0, P_1, \ldots, P_t$;
2  parallel do across blocks
3      $G_i := \emptyset$;
4      foreach $v_{ij}$ in $P_i$ do
5          $v_{ij}.N$, $v_{ij}.N'$ := Search($G_i$, $v_{ij}$, $d_{min}$);
6          foreach $u_{ij}$ in $v_{ij}.N$ do
7              parallel do within block
8                  $u_{ij}.N := u_{ij}.N \cup \{v_{ij}\}$;
9  for $i = 1$ to $t$ do
10     $E := \emptyset$; $I := \emptyset$;
11     parallel do across blocks
           // $v_{ij} \in G_i$
12         $v_{ij}.N :=$ Search($G_0, v_{ij}, d_{min}$);
13         parallel do within block
14             $v_{ij}.N := v_{ij}.N \cup v_{ij}.N'$;
15         foreach $u_{ij} \in v_{ij}.N$ do
16             parallel do within block
17                 $E := E \cup (u_{ij} \to v_{ij}, \delta(u_{ij}, v_{ij}))$;
18     parallel do across blocks
19         $E, I :=$ GatherScatter($E$);
20     parallel do across blocks
21         parallel do within block
22             $u_{ij}.N := u_{ij}.N \cup \{E[I[i]], E[I[i]+1], \ldots\}$;
23 return $G_0$;
```

graphs are processed.

Remarks. Though we propose Algorithm 5 in the context of GPU, it is essentially independent of hardware substrate. That is to say, it can also be applied to other system settings that have multiple working units such as multi-core CPU systems and distributed systems. In these system settings, each working unit can be individually responsible for the construction of one local graph and the search of nearest neighbors of one point in the merged local graph in each iteration during the merging of local graphs.

4.3.3 Analysis

Quality of graphs. Given exact nearest neighbors, Algorithm 5 can generate the NSW graph, which is the same as that constructed by sequential insertions.

Proof sketch. Suppose the NSW graph is constructed by the sequential insertion, the outgoing edges of a vertex v consists of forward edges and backward edges, denoted by E_f^v and E_b^v, respectively. Specifically, the ending vertices in E_f^v are nearest neighbors ahead of v, and v is one of the nearest neighbors of each ending point u in E_b^v which is behind v.

We can prove that if nearest neighbors set η_v among points ahead of v can be found for each point v, we can construct NSW graphs. It is self-explanatory that E_f^v has been found. Consider any u in E_b^v, v must appear in η_u. Otherwise, u will not appear in E_b^v because we can find the nearest neighbors set η for each point including u, we can update E_b^v after u searches its nearest neighbors.

In Algorithm 5, $v.N$ initially records the nearest neighbors of v within $P_0 \cup \dots P_{i-1}$, and $v.N'$ keeps the nearest neighbors of v from P_i which are ranked before v. Therefore, we have $v.N \cup v.N' = \eta_v$ for each vertex v. $\square$

Memory Usage. Global memory and shared memory are used in other parts besides search. (1) The size of G is $O(n_p \times d_{max})$ where n_p is the number of points in P and the user-defined parameter d_{max} represents the maximum degree in G. (2) The size of G' is $O(n_p \times d_{min})$ where d_{min} denotes the number of returned nearest neighbors while searching. (3) The size of E is $O(n_b \times d_{min})$ where n_b denotes the number of points in a batch. (4) The size of I is also $O(n_b \times d_{min})$. (5) For each thread block, the usage of shared memory is $O(d_{max})$ for merging.

Time complexity. We analyze the time complexity of steps excluding search. Let n_t and n_b be the number of threads in one thread block and the number of thread blocks respectively. Note that n_t may vary in different kernels. (1) The complexity of Step 2 in the local graph construction phase is $O(d_{max} \times \frac{t}{n_b} \times \frac{d_{min}}{n_t})$ where t represents the number of local graphs. Specifically, there is the factor d_{max} because we must place neighbors in new positions. (2) For Step 2 in the local graph merge phase, the complexity is $O(\log^2 |E| \times \frac{|E|}{n_b \times n_t})$ and $O(\log |E| \times \frac{|E|}{n_b \times n_t})$ for bitonic sorting and prefix sum computation, respectively. (3) For Step 3 in the local graph merge phase, the complexity is $O(\log d_{max} \times \frac{|P_i|}{n_b} \times \frac{d_{max}}{n_t})$ where $|P_i|$ represents the number of points in the local graph P_i. This implies that, theoretically, the speedup of the NSW graph construction is linear to the number of threads within a thread block (n_t) as well as the number of thread blocks (n_b).

4.3.4 Extension

Though the proposed construction algorithm (Algorithm 5) is tailored for the NSW graph, with minor modifications it can also be used to construct other proximity graphs. Next, we show how to extend the above NSW graph construction algorithm to support two popular proximity graphs: the HNSW and KNN graphs.

HNSW graph. HNSW graph [74] is a hierarchical organization of the NSW graph where each layer is an NSW graph for a subset of the point set P that is randomly selected. The higher the level of a layer is, the fewer points it contains. The bottom layer includes all points in P.

The adaptation of Algorithm 5 to support the construction of the HNSW graph is natural. A straightforward method is to construct an NSW graph for the subset of points on each layer respectively. However, when the construction of each NSW graph is independent of each other, search of nearest neighbors of points during construction can not benefit from the hierarchical structure that can shorten search path. To avoid this drawback, we determine to construct HNSW graph level-by-level.

However, there is still one problem even though we sequentially construct NSW graph on each layer: the random selection of subsets on each layer is such that we can not directly access adjacency lists of points on some layers according to their vertex IDs because some points might not be on this layer. A possible method is to maintain an index for each layer that records the position of each vertex it has. However, it assigns an index for each vertex on each layer, which consumes additional memory. A better method might be that (1) we shuffle IDs of vertices and record the mapping. (2) during construction, these vertices are inserted into each layer in turn, i.e., vertices with smaller IDs can reach higher levels. This means that we can access the adjacency list of one vertex according to its vertex ID because points that have a smaller ID than this must be also on this layer. (3) vertex IDs are recovered based on the stored mapping after construction. Consequently, we only need to maintain a shuffled order while keeping the random selection of points on each layer.

KNN Graph. The difference between a KNN graph and an NSW graph is that the former needs to maintain global nearest neighbors for each point. That is to say, when a set of new points P' are added into a KNN graph constructed on the point set P, we need to find not only the nearest neighbors on P for each point in P', but also the nearest neighbors on P' for each point in P. Then, for each point in P (resp. P'), returned nearest neighbors on P' (resp. P) are merged into its original adjacency list on P (resp. P').

A straightforward adaptation of Algorithm 5 is that (1) during local graph construction, we maintain nearest neighbors in the current local graph for each point. (2) while merging local graphs, we search the nearest neighbors for vertices in both G_0 and G_i. (3) we update the corresponding adjacency lists in both graphs, comparable to the update of the backward edges. The main drawback of this solution is that multiple NN searches will be invoked for each point.

Table 4.1: Real-life Datasets

Dataset	**Type**	**Dimension**	**Vertices**	**Metric**
SIFT1M [5]	Image	128	1M	Euclidean
GIST [5]	Image	960	1M	Euclidean
NYTimes [26]	Text	256	0.29M	Cosine Similarity
GloVe200 [82]	Text	200	1.18M	Cosine Similarity
UQ_V [92]	Video	256	3.03M	Euclidean
MSong [2]	Audio	420	0.99M	Euclidean
Notre [3]	Image	128	0.33M	Euclidean
UKBench [79]	Image	128	1.1M	Euclidean
DEEP [112]	Image	96	8M	Euclidean
SIFT10M [5]	Image	32	10M	Euclidean

To overcome it, we turn to an iterative method proposed by [25]. This method follows the property that the neighbors of neighbors are likely to be neighbors. Initially, this method randomly generates the adjacency list of each point. Then, it is iteratively improved. In each iteration, each pair of neighbors $\{u_1, u_2\}$ of each vertex v will form two new edges $u_1 \rightarrow u_2$ and $u_2 \rightarrow u_1$ for u_1 and u_2, respectively. Then, these newly generated edges are used to update the adjacency lists of vertices. This process terminates when the adjacency lists of all points cease to change. We can see that the key to this framework is distance computation between each pair of neighbors of each vertex and the update of adjacency lists. They can be implemented naturally as shown in Figure 4.1 (BulkDistanceComputation) and Algorithm 5 (Step 3 of local graphs merge phase).

4.4 Experiments

We conduct experiments to evaluate (1) the efficiency of our GPU-accelerated nearest neighbors search algorithm GANNS; (2) the impact of the number k of returned nearest neighbors and the number e of explored vertices; (3) the efficiency of our GPU-based graph construction framework GGraphCon; and (4) scalability.

Datasets. We use ten real-world datasets as summarized in Table 4.1. In particular, (1) NYTimes and GloVe200 are heavily skewed while the dimension of GIST is relatively high. This makes them hard, compared to other datasets. (2) SIFT10M consists of ten million vectors randomly selected from SIFT1B [5]. Here, we only use the first 32 dimensions of each vector. Similarly, DEEP comprises eight millions points randomly selected from the original dataset DEEP1B[1].

Algorithms. We implement the following algorithms in C++ and CUDA C. (1) GANNS is shown in Section 4.2.2 for GPU-based nearest neighbors search. In our implementation, we set l_n to the power of 2 for ease of GPU memory management. Here, we introduce another parameter e to achieve a fine-grained trade-off between efficiency and accuracy, where we only considere the first e vertices in N for exploration. (2) GSerial and GNaiveParallel (Section 4.3.1) are straightforward GPU-based graph construction algorithms that use SONG for searching. (3) The GPU-based graph construction framework GGraphCon is shown in Section 4.3.2. It includes (3.1) GGraphCon$_{\text{GANNS}}$ that uses GANNS for searching, (3.2) GGraphCon$_{\text{SONG}}$ that uses SONG for searching.

Baselines. (1) For ANN search, we compare with the state-of-the-art graph-based ANN search algorithm SONG [119]. The code is from original authors[2]. (2) For graph construction, apart from straightforward methods GSerial and GNaiveParallel, we also compare with algorithms GraphCon$_{\text{NSW}}$ and GraphCon$_{\text{HNSW}}$ on CPU. GraphCon$_{\text{NSW}}$ establishes navigable small world graphs with degree limitation, which is implemented by [119][2]. GraphCon$_{\text{HNSW}}$ constructs hierarchical

[1]`https://research.yandex.com/datasets/biganns`
[2]`https://github.com/sunbelbd/song`

navigable small world graphs [74], which is publicly available[3].

Evaluation. (1) For nearest neighbors search, we measured the accuracy by recall, which is defined as the ratio of correct nearest neighbors to returned neighbors. The accuracy is evaluated over the test set. Specifically, each test set comprises 2000 vertices. Search time is denoted by "Queries Per Second" which represents the average number of completed queries per second. (2) For graph construction, we report running time and measure the quality of graphs by recall, which can be achieved given the same search algorithm.

Configuration. The experiments are conducted on a Linux Server powered by a 26-core Intel Xeon Gold 6238R 2.2GHz CPU and NVIDIA Quadro P5000 GPU with 2560 cores and 16GB memory. We compile all codes with NVCC (CUDA 10.0), GCC 5.4.0 and the -O3 flag.

4.4.1 Search performance

Efficiency. Fixing $k = 10$, we evaluate the efficiency of GANNS and SONG. We test the number of queries per second when varying the recall. As shown in Figure 4.4, we have the following observations. (1) The ranges of recall achieved by GANNS and SONG are the same on all datasets. This shows that the parallelization scheme of GANNS does not change the quality of results. (2) High recall values (larger than 0.95) can be achieved on all datasets except GloVe200. For instance, the highest recall on the hard dataset GIST is 0.97 while the highest recall values on some datasets like UKBench and UQ_V are close to 1. This validates the superior search accuracy of the graph-based search methods. (3) GANNS consistently outperforms SONG on all datasets. When the recall is not very high (around 0.8), GANNS is 1.5-5 times faster than SONG. (a) On some datasets, GANNS can achieve about 5x speedup. For instance, the throughput

[3]`https://github.com/nmslib/nmslib`

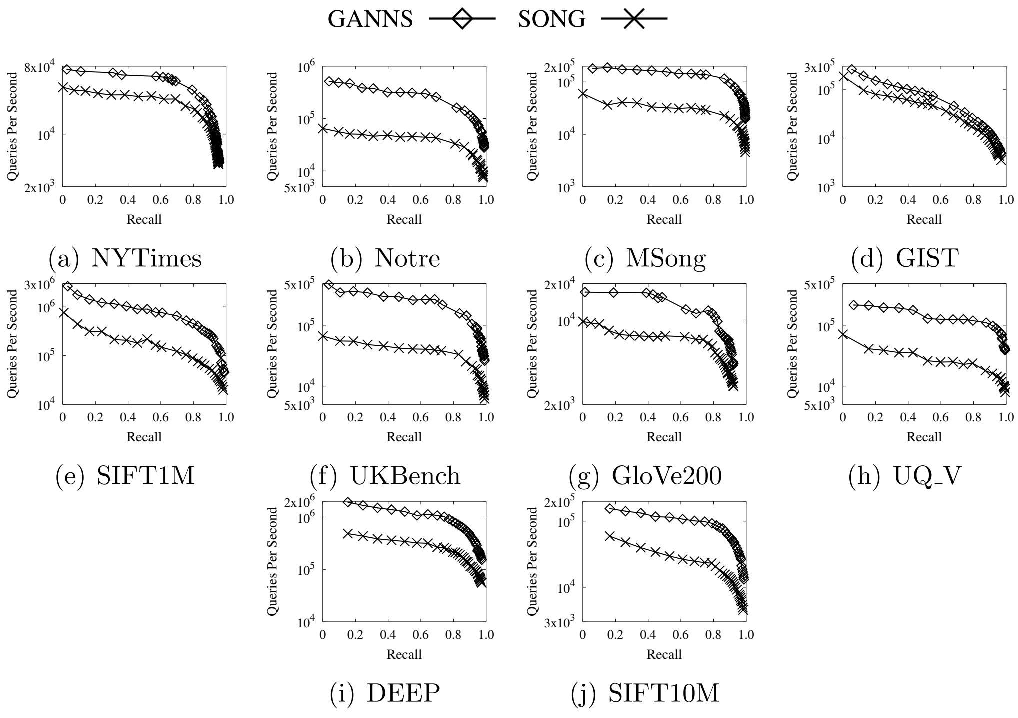


Figure 4.4: Throughput on different recall

of GANNS is 458.5k queries per second on SIFT1M when the recall is 0.795. By contrast, the throughput of SONG is 88.5k queries per second while achieving the same recall. (b) On the hard datasets with moderate dimension NYTimes and GloVe200, the speedup is reduced to around 2. (c) For the hard dataset GIST with a high dimension, GANNS achieves around 1.5x speedup.

A time breakdown of GANNS and SONG is presented in Figure 4.5 when the recall is around 0.8. It demonstrates that compared with SONG, the cost of data maintenance of GANNS decreases. This is consistent with Figure 4.4. In particular, on hard datasets, the cost of data maintenance of GANNS is little higher than that on other datasets. This is because more vertices as candidates need to be maintained on hard datasets.

Effect of Parameters. We evaluate the impact of the number k of returned

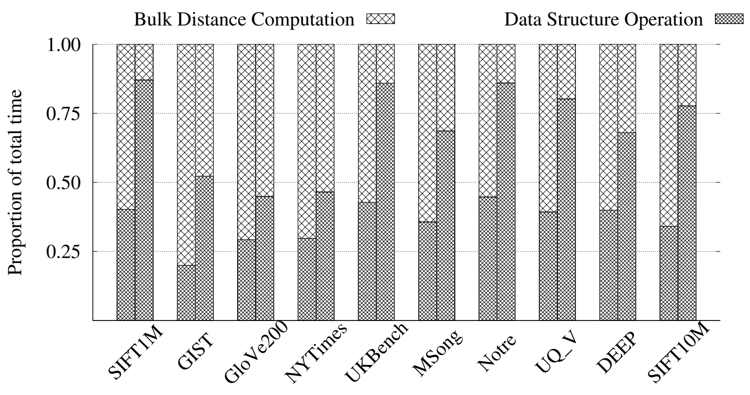
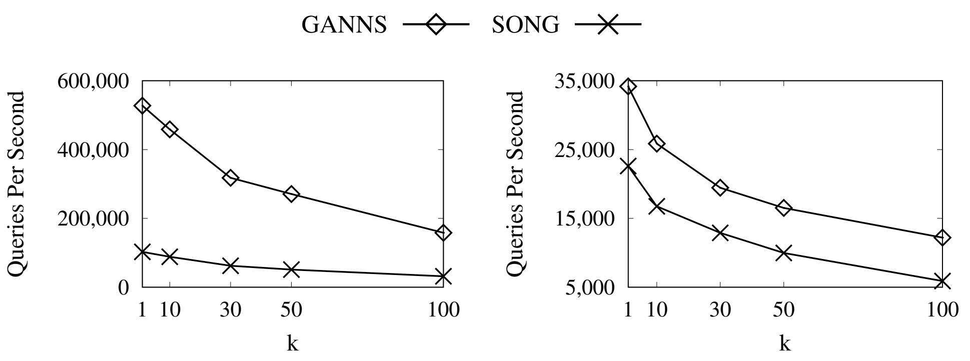
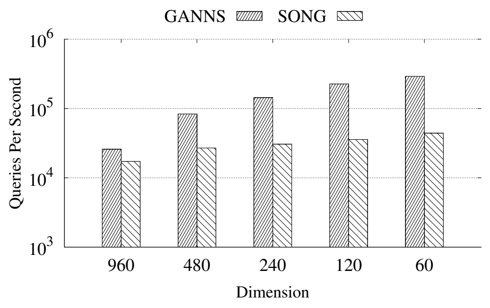


Figure 4.5: Execution time breakdown of GANNS (left) and SONG (right)



(a) SIFT1M (b) GIST

Figure 4.6: Throughput on different parameter k



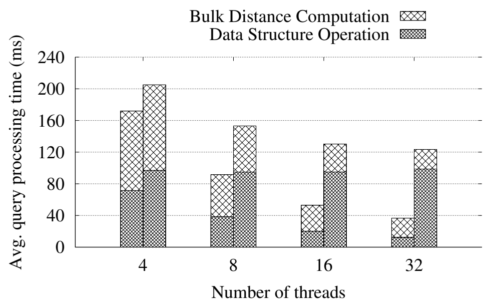


Figure 4.7: The effect of n_d Figure 4.8: The effect of n_t

vertices and the number n_d of dimension.

Varying k. Fixing $recall = 0.8$, we vary k from 1 to 100. We report the results on

SIFT1M and GIST here. As shown in Figure 4.6, the speedup remains relatively stable as k increases. On SIFT1M, the largest speedup is 5.3, and the smallest speedup is about 5; On GIST, the largest speedup is 2, and the smallest speedup is 1.5.

Varying n_d. Fixing $k = 10$, we vary n_d from 960 to 60 on dataset GIST to demonstrate the effect of dimension on query performance of GANNS and SONG. Here, we report throughput when $recall = 0.8$. As shown in Figure 4.7, query performance of both algorithms improve when the dimensionality n_d decreases mainly because the cost of bulk distance computation becomes low for low dimensional data. It is shown that the performance gap between GANNS and SONG becomes more significant when the dimensionality is low, e.g., the speedup enlarges from 1.5x to 6x as dimension n_d decreases from 960 to 60. This is because the percentage of the data structure operation cost becomes larger when the dimensionality decreases, and SONG cannot take advantages of the parallelism of GPU for this cost within a thread block.

Parallelism. To evaluate parallelism of GANNS and SONG, we vary the number of threads in each thread block (i.e., query) from 4 to 32, and report the average distance computation time and data structure operation time of two algorithms, SONG and GANNS, on SIFT1M dataset in Figure 4.8. Regarding the distance computation, two algorithms take similar time and both enjoy a significant speedup. For instance, they spend around 100 ms and 24 ms when the number of thread is 4 and 32, respectively. Regarding the data structure operation, GANNS still demonstrates a good speedup, e.g., takes around 71 ms and 12.3 ms when the number of threads is 4 and 32 respectively, while SONG cannot take advantage of the parallelism within the thread block.

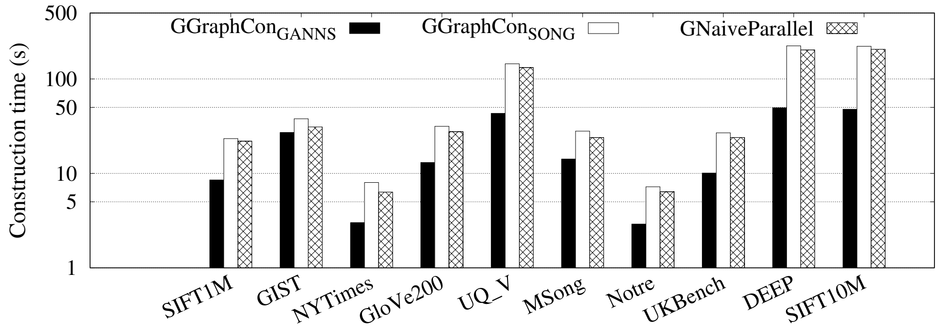


Figure 4.9: Graph construction time

4.4.2 Construction performance

Parallelization scheme. We evaluate our GPU-based graph construction. The upper bound d_{max} and the lower bound d_{min} are fixed as 32 and 16, respectively, unless otherwise stated. CPU algorithms run in a single thread.

Efficiency. We compare the running time of GGraphCon$_{\text{GANNS}}$, GGraphCon$_{\text{SONG}}$, GNaiveParallel and GSerial. As shown in Figure 4.9, (1) Given the same search kernel, the straightforward method GNaiveParallel only slightly outperforms our proposed scheme GGraphCon$_{\text{SONG}}$. This shows that the overhead of our scheme is minor, which is derived from considering links between vertices in the same batch. (2) The search process dominates graph construction. By using GANNS, GGraphCon$_{\text{GANNS}}$ achieves the apparent speedup over GGraphCon$_{\text{SONG}}$. On some datasets, GGraphCon$_{\text{GANNS}}$ has a 2x-3.3x speedup. For instance, the running time of GGraphCon$_{\text{GANNS}}$ on UQ_V is only 43s, while GGraphCon$_{\text{SONG}}$ needs to spend 145s. On hard datasets, the speedup is between 1.4-2.2. (3) We also conduct experiments to evaluate GSerial. The result tells us that its running time is very long, e.g. 3810s on SIFT1M (not shown).

For reference, we also compare with GraphCon$_{\text{NSW}}$. The results are reported in Table 4.2 (GGC denotes GGraphCon). GGraphCon$_{\text{GANNS}}$

Table 4.2: Comparison with CPU algorithm (NSW)

Dataset	GraphCon$_{\text{NSW}}$	GGC$_{\text{GANNS}}$	GGC$_{\text{SONG}}$
SIFT1M	355s	8.5s (41.8x)	23s (15.4x)
GIST	1335s	27s (49.4x)	38s (35.1x)
NYTimes	249s	3s (83x)	8s (31.1x)
GloVe200	531s	13s (41x)	31.5s (16.9x)
UQ_V	1720s	43s (40x)	145s (11.9x)
MSong	620s	14s (44.3x)	28s (22.1x)
Notre	87s	3s (29x)	7s (12.4x)
UKBench	375s	10s (37.5x)	27s (13.9x)
DEEP	4135s	49.5s (83.5x)	224s (18.5x)
SIFT10M	2986s	48s (62x)	222s (13.5x)

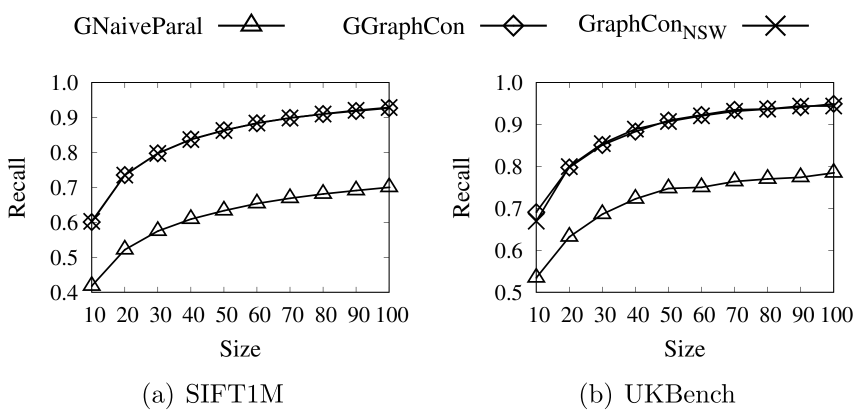
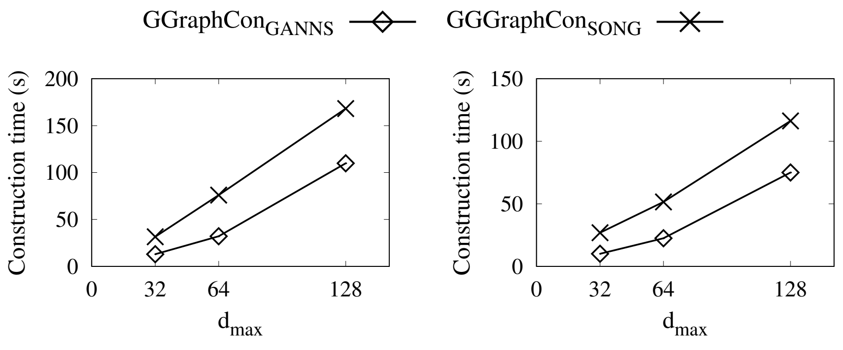


Figure 4.10: Graph quality

and GGraphCon$_{\text{SONG}}$ both have significant speedup. In particular, GGraphCon$_{\text{GANNS}}$ could achieve 40-50x speedup on most datasets.

Quality. We evaluate the quality of proximity graphs from the straightforward parallelization scheme GNaiveParallel, our proposed scheme GGraphCon and the serial CPU method GraphCon$_{\text{NSW}}$. We report the results on SIFT1M and UKBench. We form the following observations from Figure 4.10. (1) The recall achieved on the graph by GNaiveParallel is much lower than that of other graphs. For instance, on SIFT1M, the recall of GNaiveParallel is only 0.7 even though



(a) GloVe200 (b) UKBench

Figure 4.11: Construction time by varying d_{max}

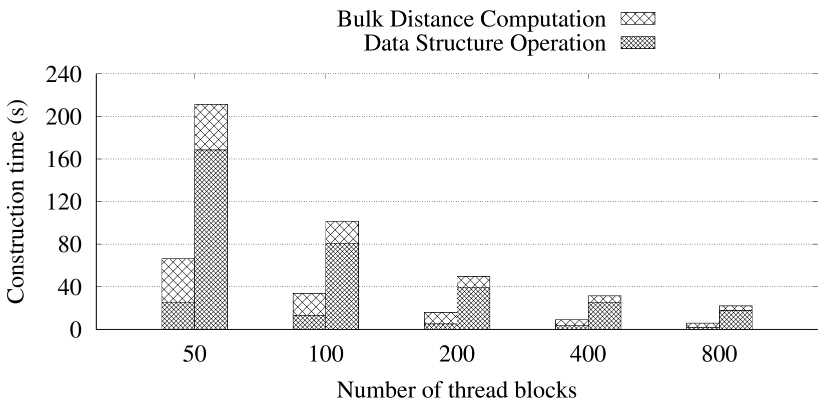


Figure 4.12: Performance scaling of GGC$_{\text{GANNS}}$ (left) and GGC$_{\text{SONG}}$ (right) while varying the number of thread blocks

e increases to 100, whereas recalls of the other two algorithms could be 0.92. This demonstrates that the graph quality constructed by GNaiveParallel is poor. (2) The graph quality produced by GGraphCon is almost the same as that by GraphCon$_{\text{NSW}}$ among all values of e on two datasets. This shows that the quality of the graph from GGraphCon is as good as the graph from the serial algorithm. The result is consistent with our analysis in Section 4.3.3.

Scalability. We evaluate the scalability of our proposed scheme GGraphCon. We vary the bound d_{max} from 32 to 128. Correspondingly, d_{min} varies from 16

Table 4.3: Comparison with CPU algorithm (HNSW)

Dataset	**GraphCon$_{\textbf{HNSW}}$**	**GGC$_{\textbf{GANNS}}$**	**GGC$_{\textbf{SONG}}$**
SIFT1M	313s	11s (28.5x)	37s (8.5x)
GIST	2138s	48s (44.5x)	68s (31.4x)
NYTimes	324s	4s (81x)	12s (27x)
GloVe200	5255s	17s (309x)	52s (101x)
UQ_V	1737s	47s (37x)	215s (8x)
MSong	823s	20s (41x)	48s (17.1x)
Notre	85s	3.2s (26.6x)	11s (7.7x)
UKBench	342s	11s (31.1x)	38s (9x)
DEEP	4550s	70.2s (65x)	308s (15x)
SIFT10M	2823s	82s (34.4x)	338s (8.4x)

to 64. We report the results on GloVe200 and UKBench here. The results are shown in Figure 4.11. The running time gently increase when increasing d_{max}. The increase of running times of GGraphCon$_{\text{GANNS}}$ and GGraphCon$_{\text{SONG}}$ are both almost linear.

Parallelism. We evaluate parallelism of GGraphCon$_{\text{GANNS}}$ and GGraphCon$_{\text{SONG}}$ on the dataset SIFT1M. Setting d_{max} and d_{min} to 32 and 16 respectively, we vary the number of thread blocks from 50 to 800 for the construction of NSW graphs where the number of threads per thread block is set to the default value (32). Here, we report the graph construction time of two algorithms: GGraphCon$_{\text{GANNS}}$ and GGraphCon$_{\text{SONG}}$. It includes distance computation time and data structure operation time. As shown in Figure 4.12, Though we still cannot achieve the theoretical maximal speedup, it is reported that around 10x-13x speedup can be achieved for the distance computation and data structure operations of two algorithms when the number of thread blocks grows from 50 to 800 (i.e., 16x speedup theoretically). Note that although SONG cannot utilize the parallelism of the threads within the thread blocks, it can immediately take advantage of multiple thread blocks since the search processes are independent to each other.

Extension. We show that our proposed scheme can be extended to construct graphs with other formats. Here, we implement the construction of HNSW where d_{max} and d_{min} are 32 and 16 respectively. The running time is shown in Table 4.3 (GGC denotes GGraphCon). The result is consistent with Table 4.2.

4.5 Conclusion

In this chapter, we present a graph-based method for the k nearest neighbors search problem on the GPU.

We find that data structure operations emerge as the new bottleneck when distance calculations are accelerated on the GPU. Building on this observation, we employ a simple yet GPU-friendly data structure lists instead of heaps that are challenging to parallelize on the GPU. Leveraging lists, we parallelize all steps of the search process. Additionally, we propose a divide-and-conquer approach for efficient graph construction on the GPU.

Experimental findings reveal that our search method accelerates data structure operations and achieves a performance boost of 1.5 to 5 times compared to SONG at a high recall level (recall = 0.8). Moreover, due to the parallelization of the search process, our method exhibits near-linear scalability with respect to the number of threads and dimensions. Extensive experiments also demonstrate that our graph construction algorithm delivers a 40 to 60 times performance improvement over the single-thread CPU implementation and scales well with the size of graphs.

When the search progresses along a single long path, the efficiency gains of our data structure operations diminish relative to heap updates. This may occur when dealing with especially complex datasets.

Chapter 5

Subgraph Matching

In this chapter, we introduce the methodological details about our approach to subgraph matching along with the experimental results.

5.1 Preliminaries

In this section, we define subgraph matching problem and present the state-of-the-art solution.

5.1.1 Problem definition

We start with basic notations.

Labeled Graph. Assume an infinite label set Σ, a graph $g = (V, E, \mathcal{L})$ is a vertex-labeled graph if

∘ V is a finite set of vertices.

∘ $E \subseteq V \times V$ is a finite set of edges.

∘ there is a label function $\mathcal{L}$ that associates the vertex $u \in V$ with a label $l \in \Sigma$.

We present the definition of subgraph isomorphism.

Subgraph Isomorphism. Given a query graph $q = (V_q, E_q, \mathcal{L})$ and a data graph $G = (V_G, E_G, \mathcal{L})$ that are both vertex-labeled graphs, a subgraph isomorphism of q in G is an *injective* function f that maps V_q to V_G such that

- $\circ$ $\forall u \in V_q, \mathcal{L}(u) = \mathcal{L}(f(u))$.
- $\circ$ $\forall (u, u') \in E_q, (f(u), f(u')) \in E_G$.

For brevity, a subgraph isomorphism of q in G is also called a *match* of q in G. When the context is clear, we may directly write a partial match (the length is less than $|V_q|$) as a match.

We formulate subgraph matching problem.

Problem Statement. Given a query graph q and a data graph G, subgraph matching is to find all subgraph isomorphisms M of q in G.

Due to significance of subgraph matching problem, there are many existing works in the literature. In these works, some notations are widely used, which can be introduced as follows. For readability, the frequently used notations are summarized in Table 5.1.

Matching Order. A matching order φ of a graph g is a permutation of its vertex set V. To specify vertices in φ, we denote by $\varphi[i]$ the ith vertex in φ. Similarly, $\varphi[i : j]$ is a set of vertices the index of which in φ is between i and j $(1 \leq i \leq j \leq |V|)$.

Backward/Forward Neighbors. Given a graph g and a matching order φ of g, the backward neighbors $N_+^{\varphi}(u)$ of vertex $u \in V$ are the neighbors of u located before u in φ. Likewise, the forward neighbors $N_-^{\varphi}(u)$ of vertex $u \in V$ are the neighbors of u located after u in φ.

Induced Subgraph. Given a graph $g = (V, E)$ and a subset $V' \subset V$, the induced subgraph $g[V']$ of g on V' consists of the vertex set V' and the edge set $E' = \{(u, u') \mid u, u' \in V, (u, u') \in E\}$. Given a matching order φ of graph g, there exist a sequence of induced subgraphs $\{g_i\}$ where $g_i = g[\varphi[1 : i]]$ and

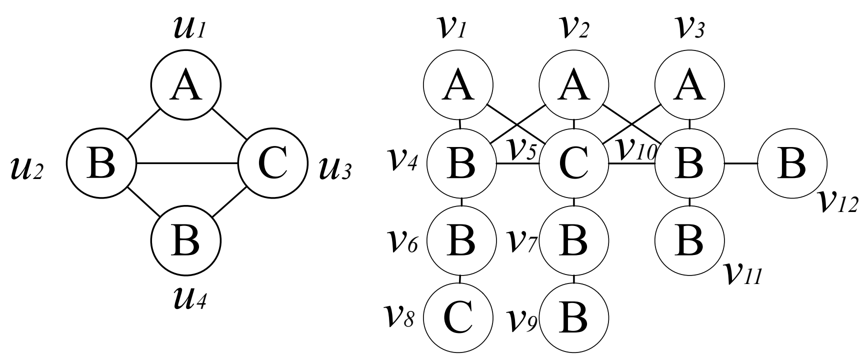


Figure 5.1: An example of query graph q and data graph G

$1 \leq i \leq |V|$.

Candidate Vertex Set. Given a query graph $q = (V_q, E_q, \mathcal{L})$ and a data graph $G = (V_G, E_G, \mathcal{L})$, a candidate vertex set $C(u)$ of $u \in V_q$ on G is a subset of V_G, excluding data vertices in V_G that cannot be mapped to u. Generally, heuristic strategies are applied to generate candidate vertex sets $\{C(u)\}$ for all $u \in V_q$.

Neighbor Label Frequency Filtering. A common heuristic strategy to generate candidate vertex sets is neighbor label frequency filtering (NLF). $\forall u \in V_q$, $v \in V_G$, NLF produces $C(u)$ by excluding data vertices that violate either of the following two criteria where $N(u, l)$ denotes the set $\{u' \in N(u) \mid \mathcal{L}(u') = l\}$:

- ◦ $|N(v, l)| \geq |N(u, l)|, \forall l \in \mathcal{L}$.
- ◦ $\mathcal{L}(u) = \mathcal{L}(v)$.

Example 6. *Given the query graph q and the data graph G in Figure 5.1, the candidate vertex set $C(u_1)$ generated by NLF is $\{v_1, v_2, v_3\}$ because their neighbors contains vertex v_5 and vertex $v_4(v_{10})$. other candidate vertex sets are as follows: $C(u_2) = \{v_4, v_{10}\}$; $C(u_3) = \{v_5\}$; $C(u_4) = \{v_4, v_6, v_7, v_{10}\}$.*

Table 5.1: The summary of notations

Notation	**Description**
g, q and G	graph, query graph and data graph
V, E and $\mathcal{L}$	vertex set, edge set and label set
$d(u)$, $N(u)$ and $\mathcal{L}(u)$	degree, neighbors and labels of u
$g[V]$	vertex-induced subgraph of g on V
$C(u)$	candidate vertex set of query vertex u
f and φ	match and matching order
$N_+^\varphi(u)/N_-^\varphi(u)$	backward/forward neighbors of u given φ
M/M_i	the set of matches/partial matches with length i

5.1.2 State-of-the-art GPU solution

We introduce GSI [116], which is the state-of-the-art GPU solution to subgraph matching.

GSI follows the filtering-and-enumeration framework. The filtering phase of GSI is a lightweight NLF method. It encodes the neighborhood structure and label information of a vertex v as a bit-vector signature $S(v)$. To further save memory, GSI utilizes a hash function H to transform the labels of neighbors of v such that the size of the neighborhood structure can be bounded. That is, l is replaced by $H(l)$ when calculating neighbor frequency. For each frequency, its state $S(v, l)$ is denoted by two bits as follows:

$$S(v,l) = \begin{cases} 00 & |N(v, H(l))| = 0 \\ 01 & |N(v, H(l))| = 1 \\ 11 & |N(v, H(l))| > 1 \end{cases}$$

Combining label information and neighborhood structure, $S(v)$ becomes a fixed-length vector as illustrated in Figure 5.2. After encoding all vertices in V_q and V_G, GSI uses the bitwise AND operation & to filter out data vertex v for query vertex u if $S(u)\&S(v) \neq S(u)$.

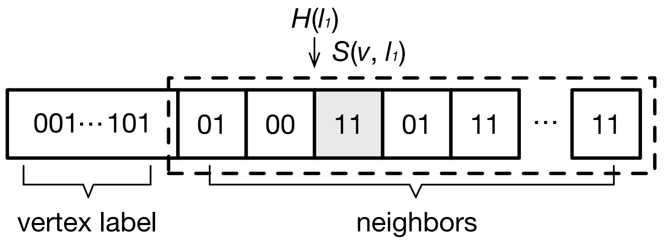


Figure 5.2: A bit-vector signature S

During the enumeration, GSI adopts a BFS-based method that extends all matches M_i of the induced subgraph q_i of q in G to all matches M_{i+1} of the induced subgraph q_{i+1} of q in G. To improve the performance of enumeration, GSI proposes two optimization techniques.

The first technique is the Prealloc-Combine strategy. Due to that dynamic structures are not supported on the GPU, the memory storing matches M_{i+1} must be allocated before matches M_{i+1} are written. GSI pre-allocates the memory of M_{i+1} based on the upper bound of M_{i+1}, which is computed as follows:

$$\sum_{\forall f \in M_i, v=f(u)} |N(v, \mathcal{L}(\varphi[i+1]))|$$

where u is a chosen backward neighbor of $\varphi[i+1]$. Consequently, the redundant computation of two-phase output scheme can be avoided.

The second technique is the PCSR structure. It is used to save memory when there are tons of labels in graphs. It divides the data graph G into several partitions $P(G,l)$ for all $l \in \mathcal{L}_{\mathcal{E}}$. In each partition, edges with label l are reorganised, and the index is stored into a hash table. PCSR can avoid memory consumption caused by a large number of empty neighbor lists in CSR.

5.2 Framework

In this section, we elaborate on our motivations and develop our method. Following this, an overview is given.

5.2.1 Motivation

As a lightweight method, the filtering of GSI can be easily parallelized. This makes it friendly to the GPU. At the same time, GSI can prune numerous search branches on data graphs with tons of labels. However, it experiences a decrease in the pruning ability when the number of labels in the data graph is moderate. Specifically, it can only identify three states (i.e., 0, 1 and others). This leads that it cannot distinguish between different cases when $|N(v, l)| > 1$. For example, a data vertex v cannot be excluded from $C(u)$ by GSI if $|N(v, l)| = 2$ and $|N(u, l)| = 3$ because the states of $S(u, l)$ and $S(v, l)$ are both 11. However, $N(v, l) > 1$ of a vertex v is common on real-world graphs.

We examine the pruning ability of GSI and NLF on the dataset US Patents. The result is shown as Figure 5.3. It can be observed that the pruning ability of GSI in comparison to NLF degrades when the number of labels decreases. This is because its accuracy is limited. Extensive experiments [96, 63] have shown that methods with strong pruning ability can achieve superior performance. This motivates us to develop a method with high pruning efficiency on the GPU.

The Prealloc-Combine strategy allows GSI to reduce computation, thereby improving the performance of enumeration. Meanwhile, we notice that the number of matches grows sharply when the depth of the search tree increases. The size of M can be up to 10^{12} while $|V_q| = 8$. As a result, this involves a significant number of memory transactions. However, GSI overlooks this aspect, leading to uncoalesced memory transactions during the writing of M_{i+1}. This results in

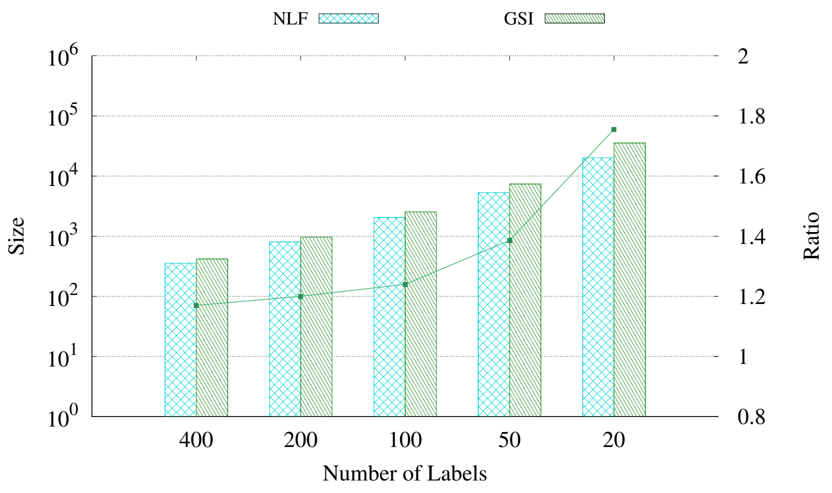


Figure 5.3: Pruning Ability Comparison

inefficiencies in memory throughput and reduces occupancy. Inspired by this, we intend to design an efficient output scheme.

Though GSI proposes PCSR structure to save memory consumption, the memory required to store M_i can be substantial, making it infeasible to fit into the global memory. It is necessary to propose a method that can efficiently manage global memory such that the scalability is enhanced. Otherwise, the number of solved queries on the GPU will be limited. Therefore, we intend to propose a solution to alleviate this issue.

5.2.2 Solution

Filtering methods aims to save computation of enumeration by pre-processing the data graph. Therefore. we start with delving into the enumeration computation for seeking avenues to reduce the computational overhead. During the enumeration, the extension of a match can be formulated as follows:

Equation 1. *For a match f_i, enumeration extends it to a set of matches $\{f_{i+1}\}$*

as follows:

$$\{f_i, (\varphi[i+1], v) \mid v \in \bigcap_{u \in N_+^{\varphi}(\varphi[i+1])} N(f_i(u)) \wedge v \neq f_i(u), \forall u \in \varphi[1:i]\} \tag{5.1}$$

The intersection in Equation 5.1 is related with lists $N(f(u))$ for all $u \in N_+^{\varphi}(\varphi[i+1])$. This may lead that the computation of enumeration of two different matches f_i, f_i' is the same when the following case happens:

$$\forall u \in N_+^{\varphi}(\varphi[i+1]), f_i(u) = f_i'(u) \wedge \exists u \in \varphi[1:i] \setminus N_+^{\varphi}(\varphi[i+1]), f_i(u) \neq f_i'(u)$$

That is, the projections of f_i and f_i' on the set $N_+^{\varphi}(\varphi[i+1])$ are the same. This consequently ensures that the intersection in Equation 5.1 is the same for f_i and f_i'. Due to the explosive growth of matches when being extended, a lot of same computation is naturally conducted during the enumeration. This presents an opportunity for the optimization. An instant idea is to store intersection results such that repetitive computation can be avoided. However, it is infeasible to store all intersection results on the GPU due to its large size. As a reference, one can observe the number of triangles in a data graph, which represents the size of intersection results of two lists [114]. Therefore, a trade-off between efficiency and memory consumption is required. This leads us to focus on the following situation:

$$\bigcap_{u \in N_+^{\varphi}(\varphi[i+1]} N(f_i(u)) = \emptyset \tag{5.2}$$

The case of Equation 5.2 bypasses the storage of neighbor lists, significantly reducing memory consumption. We only need to maintain the set of data vertices $\{v \mid v = f_i(u) \wedge u \in N_+^{\varphi}(\varphi[i+1]\}$ for a match f_i. However, the size of the recorded set is not fixed for different matches, which makes it challenging to store and access on the GPU. To alleviate this issue, we first generate candidate vertex sets $\{C(u)\}$ by NLF. Then, the following computation is conducted:

$$\forall u \in N_{+}^{\varphi}(\varphi[i+1]), N(f_i(u)) \cap C(\varphi[i+1]) \tag{5.3}$$

If the result of Equation 5.3 is empty, we can remove $f_i(u)$ from $C(u)$ such that some redundant computation can be avoided. This can be shown in Example 7.

Example 7. *Consider the query graph q and the data graph G in Figure 5.1, if the matching order φ is (u_1, u_2, u_3, u_4), the match $f_3 = \{(u_1, v_2), (u_2, v_{10}), (u_3, v_5)\}$ finds that $C(u_4) \cap N(v_{10}) = \emptyset$. Hence, v_{10} is excluded from $C(u_2)$. The match $f_3' = \{(u_1, v_3), (u_2, v_{10}), (u_3, v_5)\}$ is no longer generated, thereby avoiding the computation $C(u_4) \cap N(v_{10})$ of f_3'.*

Compared with the previous idea, this method only involves simple data structure, i.e., a few arrays of $\{C(u)\}$. This makes it concise to maintain and parallelize. Thus far, we have carved out a preliminary solution. But there are still two underlying issues that should be discussed. The first one is that the candidate vertex set $C(u')$ could be affected when $C(u)$ changes where $u' \in N_{+}^{\varphi}(u)$. The second problem is that the current idea we have considered so far is built upon an existing matching order φ. To solve these problems, we propose an iterative filtering method where the computation of Equation 5.3 is applied to $N(u)$ for all $u \in V_q$ to consider all possible matching orders.

In addition to the filtering phase, we also design some optimization techniques for the enumeration phase. Remind the issues of write throughput and scalability mentioned in Section 5.2.1. We propose 1*-phase output scheme that increases coalesced memory transactions by utilizing shared memory for higher write throughput. In terms of superior scalability, we design a pipeline method that can split M_i the extension M_{i+1} of which cannot fit into global memory

```
Algorithm 6: FSIG ($q$, $G$, $t_{max}$)
1 $\{C_0(u)\} \leftarrow$ NLF ($q$, $G$);
2 for $t \leftarrow 1$ to $t_{max}$ do
3     $\{C_t(u)\} \leftarrow$ Filter ($\{C_{t-1}(u)\}$, $q$, $G$);
4 $\mathcal{A} \leftarrow$ Collect ($\{C_{t_{max}}(u)\}$, $q$, $G$);
5 $\varphi \leftarrow$ GetOrder ($\mathcal{A}$, $q$);
6 $M \leftarrow$ Enumeration ($\varphi$, $C_{t_{max}}(\varphi[1])$, $\mathcal{A}$, $q$);
7 return $M$;
```

into several blocks such that we can switch from BFS to DFS to continue the computation.

5.2.3 Overview

We briefly introduce our method *Filtered Subgraph Isomorphism on the GPU* (FSIG). As shown in Algorithm 6, FSIG follows the filtering-and-enumeration framework.

We first generate the initial candidate vertex sets $\{C_0(u)\}$ by NLF in line 1. Each candidate vertex set $C_0(u)$ is stored in the form of a bitmap. Then, we conduct our filtering method for t_{max} rounds to shrink candidate vertex sets in lines 2-3 where t_{max} is a pre-defined value. In line 4, edges (v, v') are collected for all $(u, u') \in E_q$ and $v \in C(u) \wedge v' \in C(u')$. Afterwards, these edges are re-organized to form an auxiliary structure $\mathcal{A}$. $\mathcal{A}(u, u')$ keeps the data edges corresponding to the query edge $(u, u') \in E_q$ in the form of CSR (*Offsets*, *Edges*). Notice that the size of the array *offsets* is $|V_G|$. Though it has numerous same values, this implementation allows us to directly access the index of the neighbor list of a vertex in the array *Edges* by its vertex id.

Based on the strong pruning ability of our filtering method, we can obtain a decent matching order φ by the auxiliary structure $\mathcal{A}$, which contains $|C(u)|$ for all query vertices and the size of the array *Edges* for all query edges. Here, we employ the strategy of CFL [15] to generate φ, which follows a core-forest-leaf order. Last, our enumeration method is performed to find all matches M in line 6. The details of Filter and Enumeration will be given in later sections.

5.3 Implementation

In this section, we present the details of our implementation.

5.3.1 Filtering

We show the implementation of our filtering method, as demonstrated in Algorithm 7.

In lines 1-3, the required structure is initialized. For each $u \in V_q$, we store two copies of the bitmap $C(u)$ as an array $C'(u)$ and a bitmap $\bar{C}(u)$. To generate $C'(u)$, the prefix sum of $C(u)$ is computed by an exclusive scan. Then, for all $v \in C(u)$, they are written into $C'(u)$ based on the index indicated by the prefix sum and its value in $C(u)$.

In lines 4-5, the computation of Equation 5.3 is conducted, and the result is written into $\bar{C}(u)$. For all $u \in V_q$, $C'(u)$ is utilized to guarantee that each warp can be assigned to an available data vertex. For a vertex $u \in V_q$, the computation of a warp is as follows. (1) $N(v)$ is read from global memory into registers; (2) For each neighbor $u' \in N(u)$, the intersection between $N(v)$ and $C(u')$ is performed. Notice that $C(u')$ is a bitmap. As a result, the computation of intersection can be accelerated; (3) If $\exists u' \in N(u), N(v) \cap C(u') = \emptyset$, the state of v in $\bar{C}(u)$ is set to 0. The reason that we cannot write the state of v into $C(u)$ is that $C(u)$ can be accessed by a data vertex v' where $v' \in C(u')$. This leads inconsistent results arising from read-write conflicts.

It can be seen that both critical kernels can achieve full parallelism, i.e.,

Algorithm 7: Filter $(\{C(u)\}, q, G)$

```
1 for $u \in V_q$ do in parallel
2     $C'(u) \leftarrow$ Compact $(C(u))$;
3     $\bar{C}(u) \leftarrow C(u)$;
4 for $u \in V_q$ do in parallel
5     Check $(C(u), C'(u), \bar{C}(u), q)$;
6 return $\bar{C}(u)$;
```

intra-kernel level and inter-kernel level, by the support from the CUDA stream mechanism.

5.3.2 1*-phase output scheme

We propose 1*-phase output scheme to reduce the extra memory consumption of the Prealloc-Combine strategy and enhance the write throughput.

The Prealloc-Combine strategy proposed in [116] can avoid the redundant computation of two-phase output scheme at the cost of extra memory cost because it allocates space for extended matches according to a upper bound. When the gap between the bound and the actual memory cost is significant, the pre-allocated space could exceed the memory limitation. Though the data transfer between CPU and GPU can solve this problem, the additional expense can result in a decrease in performance.

To alleviate the issue, we derive a tighter upper bound to pre-allocate memory for extended matches M_{i+1}. GSI utilizes the following equation to determine the upper bound where u is a backward neighbor of $\varphi[i+1]$:

$$\sum_{f_i \in M_i} |N(f_i(u), \mathcal{L}(\varphi[i+1]))| \tag{5.4}$$

we establish an upper bound as follows:

$$\sum_{f_i \in M_i} |\mathcal{A}(u, \varphi[i+1], f_i(u))| \tag{5.5}$$

where u is a backward neighbor of $\varphi[i+1]$ and $\mathcal{A}(u, \varphi[i+1], f_i(u))$ represents the list *Edges* of $f_i(u)$ in $\mathcal{A}(u, \varphi[i+1])$. It can be proven that the upper bound derived from Equation 5.5 is tighter than Equation 5.4.

Proof. Let $f_i(u)$ be v, imagine that S is the set $\{w \mid \mathcal{L}(w) = \mathcal{L}(\varphi[i+1])\}$, it is clear that $S \cap N(v) = N(v, \mathcal{L}(\varphi[i+1]))$. Let $C^*(\varphi[i+1])$ be the initial set that contains data vertices after the NLF, $C^*(\varphi[i+1]) \subseteq S$. At the same time, $C(\varphi[i+1]) \subseteq C^*(\varphi[i+1])$ because some data vertices may be removed after filtering. It can be inferred that $C(\varphi[i+1]) \cap N(v) \subseteq C^*(\varphi[i+1]) \cap N(v) \subseteq S \cap N(v) = N(v, \mathcal{L}(\varphi[i+1]))$.

Hence, $\sum_{f_i \in M_i} |\mathcal{A}(u, \varphi[i+1], f_i(u))| = \sum_{f_i \in M_i} |C(\varphi[i+1]) \cap N(f_i(u))| \leq \sum_{f_i \in M_i} |N(f_i(u), \mathcal{L}(\varphi[i+1]))|$. □

In addition, the write throughput of GSI is limited because its output scheme. As shown in Figure 5.4, a warp is responsible for the extension of a match $f_i \in M_i$. When locating the intersection results *res* by arrays *ub* and *cnt*, the warp distributes the write tasks among the threads. For each thread, it is responsible for writing an extended match f_{i+1} of f_i. This leads uncoalesced memory transactions as illustrated in the bottom-left part of Figure 5.4. To improve the write throughput, we load the intersection results and the match f_i in the shared memory. Then, we assign the write tasks to threads in a consecutive manner as illustrated in the bottom-right part of Figure 5.4. That is, multiple threads collaborate to write extended matches. The value that a thread should write into M_{i+1} can be computed by the thread id and the length $i+1$ and read from shared memory.

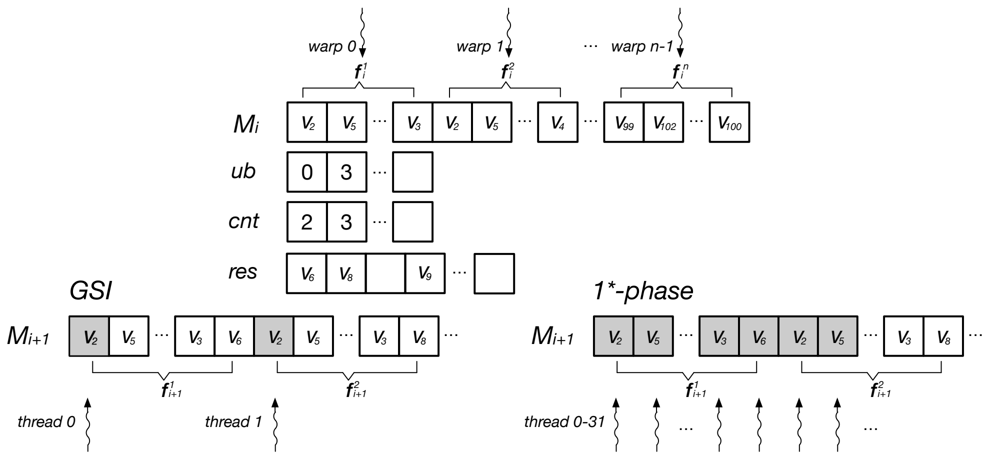


Figure 5.4: An example of output scheme

We present our enumeration method in Algorithm 8. M_1 is initialized as an array of the bitmap $C_{t_{max}}(\varphi[1])$ in line 1. Afterwards, M_1 is iteratively extended until M is generated in lines 2-12. In each round, M_{i-1} is extended to M_i. A backward neighbor w of $u = \varphi[i]$ with the smallest $|\mathcal{A}(w, u)|$ is selected in lines 3-4. In lines 5-7, the upper bound ub of M_i is computed based on Equation 5.5, and the array cnt that records the size of intersection results is initialized. The array res that stores intersection is allocated based on the upper bound and lists in $\mathcal{A}(w, u)$ are written into res in line 8. Here, neighbors in list that have existed in f_i are removed, and cnt is updated.

Then, the intersection continues to be performed for other backward neighbors of u in lines 9-10. res and cnt are updated. In line 11, the size of M_i is computed based on cnt. And, matches in M_{i-1} are extended to matches in M_i in line 12 as shown in Figure 5.4. During the process, match f_i, neighbor list, and intersection result res are all loaded into shared memory. The intersection for other backward neighbors in line 10 is implemented in the form of parallel binary search.

Algorithm 8: Enumeration (φ, $C_{t_{max}}(\varphi[1])$, $\mathcal{A}$, q)

```
1  $M_1 \leftarrow$ Compact ($C_{t_{max}}(\varphi[1])$);
2  for $i \leftarrow 2$ to $|V_q|$ do
3      $u \leftarrow \varphi[i]$;
4      $w \leftarrow$ SelectBN ($N_+^{\varphi}(u)$);
5      $ub \leftarrow$ ComputeUB ($w$, $\mathcal{A}(w,u)$, $M_{i-1}$);
6      $cnt \leftarrow ub$;
7      $ub \leftarrow$ ExclusiveScan ($ub$);
8      $res, cnt \leftarrow$ Load ($w$, $ub$, $\mathcal{A}(w,u)$, $M_{i-1}$);
9      for $w' \in N_+^{\varphi}(u) \setminus \{w\}$ do
10         $res, cnt \leftarrow$ Intersect ($w'$, $res$, $\mathcal{A}(w',u)$, $M_{i-1}$);
11     $cnt \leftarrow$ ExclusiveScan ($cnt$);
12     $M_i \leftarrow$ Extend ($res$, $cnt$, $ub$, $M_{i-1}$);
13 return $M$;
```

5.3.3 Scalability

We propose a pipeline method to enhance the scalability. As shown in Figure 5.5.

We divide global memory into two parts, which have the same space in our implementation. The first part stores M_{i-1} and all auxiliary arrays, and the second part stores M_i and its array ub'. We notice that the size of auxiliary array ub that pre-allocates memory can be large. Instead of dividing M_{i-1} and auxiliary arrays after finishing all computation, we determine to first split M_{i-1} and ub into several blocks such as B_1, B_2 and other blocks.

To split M_{i-1}, we intend to estimate the memory cost of M_i such that all structures in a block during the computation can fit into global memory. The estimation of memory cost $\mathcal{ES}$ of M_i is as follows:

$$\mathcal{ES}(M_i) = 2 \times |M_{i-1}| + \mathcal{UB} \times (1 + \alpha \times (i+1)) \tag{5.6}$$

where $|M_i|$ is the number of matches in M_i and $\mathcal{UB}$ is the upper bound derived from the array ub. Here, we need to consider the memory consumption of three

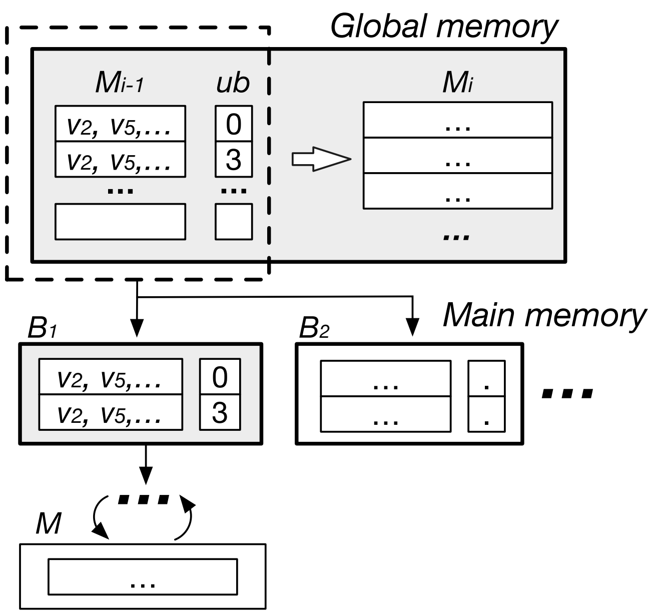


Figure 5.5: Pipeline Framework

auxiliary structures *ub*, *cnt* and *res* and matches M_i. Notice that M_{i-1} is not required to consider because it has been on the global memory. As a comparison, the segments of *ub* in blocks is required to rewrite for re-locate positions of *res* in blocks, though it is also on the global memory.

In Equation 5.6, $2 \times |M_{i-1}|$ is the memory cost of arrays *ub* and *cnt*. $\mathcal{UB}$ is the memory cost of array *res*. $\mathcal{UB} \times \alpha \times i$ is the memory cost of matches M_i. α is a pre-defined parameter, which represents the ratio between $|M_i|$ and $\mathcal{UB}$. $\mathcal{UB} \times \alpha$ is the memory cost of ub' in the next round.

When $\mathcal{ES}(M_i)$ exceeds available memory, we separate them into blocks such that $\mathcal{ES}(B_i)$ satisfy the following conditions: (1) $\mathcal{ES}(B_i)$ is less than available memory. (2) The first part that includes *ub*, *cnt* and *res* and the second part that contains M_i and ub' both consume under half of the global memory. This can avoid that matches in the next round is split into too many blocks with small size.

After the segmentation is completed, these blocks and subsequent blocks are computed in the depth-first order. That is, B_1 resides on the GPU, and the remaining segments such as B_2 are temporarily transferred to main memory. After all matches related to B_1 are found, other segments are loaded into global memory and computed in turn. To enhance robustness, we allocate unified memory when the size of a structure slightly exceeds the available memory in the implementation.

5.4 Experiments

In this section, we would evaluate the effectiveness and efficiency of our proposed method FSIG by conducting extensive experiments.

5.4.1 Experimental Setup

Baselines. To evaluate the performance of our proposed approach, GSI [116] serves as the baseline that is the state-of-the-art GPU-based solution for subgraph matching on labeled graphs. The implementation of GSI[1] is from its original authors. We modify it to support undirected labeled graphs.

Datasets and Queries. In the experiments, we use six common real-world datasets [63, 15, 96][2]. The details of datasets are shown in Table 5.2. *hu* is a protein interaction network [63]. *wn* is a lexical network of words[3]. *db*, *yt*, and *up* are from SNAP [65]. *eu* is a network of webs [4].

Following previous works [14, 38, 15, 39], we generate query sets by randomly extracting subgraphs from each dataset. For all datasets, their query sets contain 100 generated queries. For *db*, *yt*, and *up*, the size $|V_q|$ of each query is 10. For

[1]https://github.com/pkumod/GSI
[2]https://github.com/RapidsAtHKUST/SubgraphMatching
[3]http://vlado.fmf.uni-lj.si/pub/networks/data/

Table 5.2: Properties of datasets

Dataset	*abbr.*	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{L}\rvert$	$\mathcal{D}_{max}$	$\overline{\mathcal{D}}$
Human	*hu*	4,674	86,282	44	771	36.9
WordNet	*wn*	76,583	120,399	5	543	3.1
DBLP	*db*	317,080	1,049,866	15	343	6.6
Youtube	*yt*	1,134,890	2,987,624	25	28,754	5.3
US Patents	*up*	3,774,768	16,518,947	20	793	8.8
Eu2005	*eu*	862,664	16,138,468	40	68,963	37.4

[*] $\mathcal{D}_{max}$ and $\overline{\mathcal{D}}$ denote maximum degree and average degree respectively.

hu, *wn*, and *eu*, the size $|V_q|$ of each query is 8 because it is more challenging to solve subgraph matching problem on these datasets. Specifically, *hu* and *eu* both have high average degree, and most vertices in *wn* have the same label.

Metrics and Parameters. To evaluate the performance, we consider running time. It is measured in seconds (s). We set the maximal running time to a reasonable value, i.e., 10^5s. For unsolved queries, running times are regarded as the maximal running time. The parameter α is set to 0.6 on *db*, *yt*, and *up*. For *hu*, *wn*, and *eu*, α is set to 0.9. The default value of t is 3.

Experimental Environment. Experiments are conducted on a machine equipped with 88GB main memory, one NVIDIA Quadro RTX 5000 GPU (16GB device memory), and one Intel Xeon Gold 6238R CPU processor (2.2GHz, 26 cores). Two programs are both compiled with -O2 flag.

5.4.2 Overall Performance

In this subsection, we showcase the efficiency of FSIG in comparison with GSI. Figure 5.6 captures the performance of our method against GSI across all datasets. For each dataset, the average running time of all 100 queries is reported. Evidently, our method consistently outperforms the baseline.

From Figure 5.6, we can draw several key insights. FSIG achieves the most

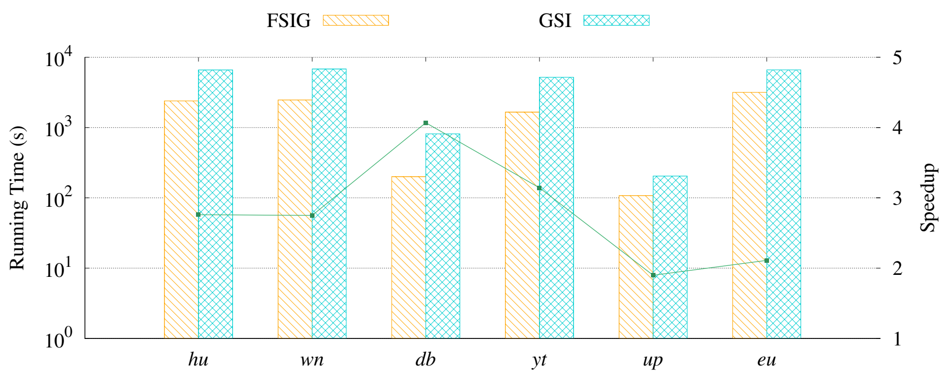


Figure 5.6: Overall Performance

significant lead on the dataset *db*. In comparison to the average running time 813.5s of GSI, the average running time of FSIG is around 200s. This shows a 4x speedup. Even in the worst-case scenario (e.g., *up*), our method maintains a discernible edge over the baseline. Our method demonstrates a 1.9x acceleration. This is attributed to the easy-to-handle characteristic of the dataset. Even with a higher number of vertices included in the candidate vertex set C, a significant number of branches can be still pruned when the depth of the search tree increases. In terms of other datasets, an around 3x speedup is achieved on *hu*, *wn*, and *yt*, while a 2.1x speedup is observed on *eu*.

Given the consistent performance on various datasets, it is clear that our approach presents tangible performance gains compared to the existing work.

5.4.3 Filtering Performance

In this subsection, we evaluate the effectiveness of the filtering technique that we propose. Figure 5.7(a) and Figure 5.7(b) show the performance.

Figure 5.7(a) reports the running time of FSIG and GSI in the filtering phase. Note that the y-axis scale is in milliseconds. As shown in the figure, the filtering times for our method and GSI both increase with the size of V_G.

Compared with GSI, the filtering time of our method increases. The maximum value can be 31.5ms on *up*. FSIG accesses the adjacency lists of vertices in candidate vertex sets, which leads to random memory reads. This diminishes the memory read efficiency of our method. As a comparison, GSI performs bitwise operations between vectors, which significantly boosts its efficiency. Notice that this increase is negligible when considering the overall running time that spans seconds.

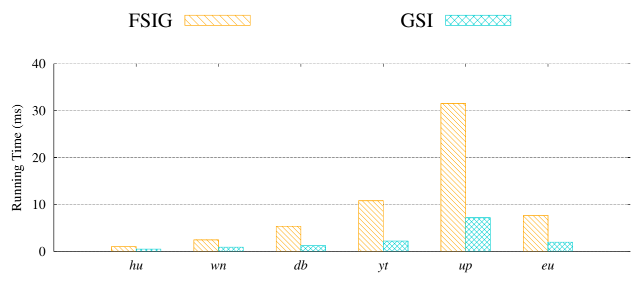
Figure 5.7(b) demonstrates the pruning ability of FSIG and GSI. It can be seen that our method can obtain a smaller $|\overline{C}|$ across all datasets. On *db*, *yt*, and *up*, we can observe a substantial decrease in the size of C ranging from 3-fold to 10-fold. Even on challenging datasets *hu*, *wn*, and *eu*, the size of C is still reduced by $25 - 35\%$. This highlights that our filtering method, compared to the lightweight approach of GSI, can effectively avoid the traversal of redundant search branches (albeit not entirely), thereby leading to superior performance.

In summary, it can be indicated that the slight delay of our method in the filtering phase is a negligible cost of the gains in efficient pruning of the search tree.

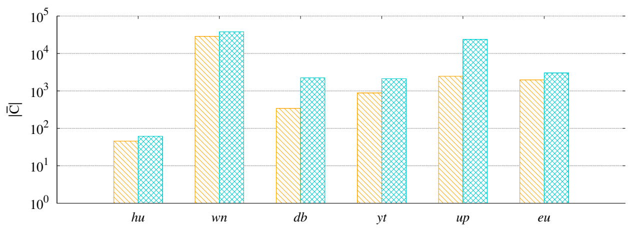
5.4.4 Enumeration Performance

In this subsection, we report the performance of enumeration. The results are displayed in Figure 5.8.

Figure 5.8(a) reports the memory usage in the Pre-alloc phase. It can be seen that our method consistently consumes less memory than GSI. Compared to GSI, our method conserves around half the memory on datasets *wn*, *yt*, and *eu*. This can be up to around 420GB on *eu*. Similarly, the memory consumption of our method is slightly less than that of GSI, i.e., $5 - 10\%$, on other datasets. Efficient memory usage enables FSIG to support larger datasets and



(a) Filtering Time



(b) Pruning Ability

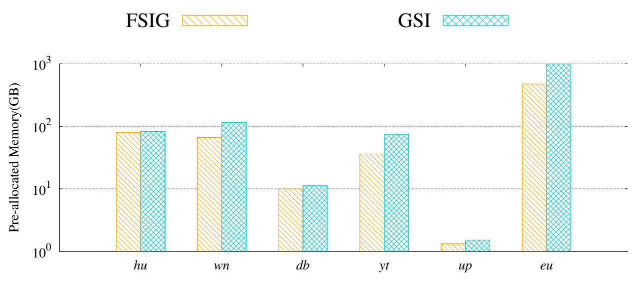
Figure 5.7: Filtering Performance

* $|\overline{C}|$ denotes the average size of C of all query vertices

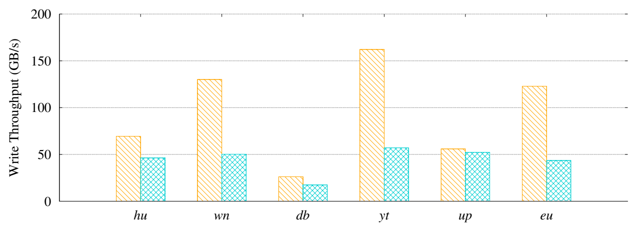
more intricate queries, which can enhance scalability performance.

Figure 5.8(b) evaluates the effectiveness of our optimization technique in the enumeration. It depicts the write throughput of FSIG anf GSI during the enumeration phase. It can be observed that FSIG exhibits a better write performance. FSIG offers a speedup of 2.5 to 3 times over GSI on datasets *wn*, *yt*, and *eu*. For example, the write throughput of FSIG is 122.89GB/s on *eu* while the write performance of GSI is 43.512GB/s. On other datasets, our method also has better performance, which exhibits a writing speed increment of $3-23$GB/s. Such enhancement in write throughput ensures that we can traverse the search tree more efficiently.

Conclusively, the empirical results reinforce the superiority of FSIG against GSI in memory conservation and write throughput during the enumeration phase.



(a) Pre-allocated Memory Size



(b) Write Throughput

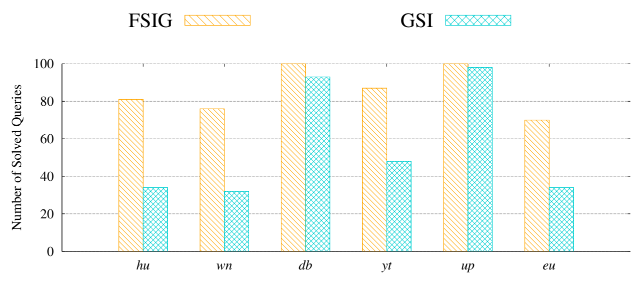
Figure 5.8: 1*-phase output scheme

5.4.5 Scalability

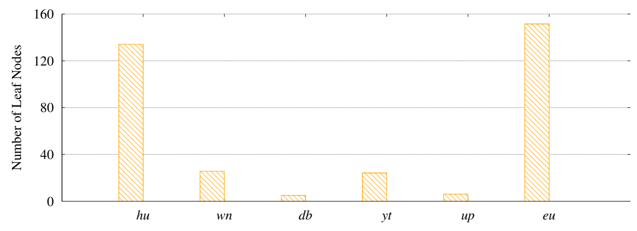
In this subsection, we evaluate the scalability of our method in contrast to GSI, Figure 5.9(a) and Figure 5.9(b) show the performance.

Figure 5.9(a) presents the comparison of FSIG against GSI in terms of the number of solved queries. It can be seen that our method consistently outperforms the baseline. We can glean several observations from Figure 5.9(a). (1) FSIG solves all queries on datasets *db* and *up*. (2) FSIG solves around 90 queries on *yt* while GSI finishes 48 queries. (3) FSIG addresses $70 - 80$ queries, whereas GSI solves only around 30. The reasons FSIG falls short with a handful of queries include timeouts caused by their inherent complexity and the restriction of available host memory. However, GSI faces difficulties because of its inability to efficiently manage GPU memory and the "out of time" issue.

To provide more details, Figure 5.9(b) illustrates the frequency of data trans-



(a) Number of Solved Queries



(b) Number of Transfers

Figure 5.9: Scalability

* Leaf nodes are leaves of search tree

fers between the GPU and CPU for our method. It is measured by the number of leaf nodes in the search tree because there is a heightened tendency for search node to be divided into several blocks when the traversal reaches the deepest level of the search tree. It can be seen that the number of leaf nodes in the search tree of FSIG is moderate, i.e., $5 - 150$ during the enumeration. For *db* and *up*, the value is around 5. For *wn* and *yt*, it is around 25. For *hu* and *eu*, the average numbers of leaf nodes are 134.16 and 151.48 respectively. This shows the adaptability of our method, which can maintain a manageable search tree irrespective of the dataset complexities.

In summary, it can be shown that FSIG demonstrates superior scalability when compared to GSI. This advantage is seen in its capability to handle more diverse queries and adapt its search tree.

5.5 Conclusion

In this chapter, we propose a backtracking-based approach for the subgraph matching problem on the GPU.

We note that the existing method compromises its pruning efficiency due to the compression of neighborhood information. To devise an effective and highly parallel filtering method, we discuss scenarios that can expedit enumeration and select one friendly to memory capacity and memory transaction on the GPU. We observe a rapid increase in the number of matches throughout the enumeration process, making memory transactions a substantial part of the runtime. In light of this, we propose a new write scheme that can increase coalesced memory transactions. Additionally, a pipelining technique is applied to enhance scalability.

Experimental results validate that our filtering approach effectively reduces the candidate sets, achieving a performance improvement of 2 to 4 times compared to GSI. Moreover, it realizes up to a threefold increase in memory throughput during the enumeration process. Meanwhile, extensive experiments demonstrate that our approach has robust scalability, with the capability to resolve between 70% to 100% of the queries.

Though our method effectively reduces candidate, there are scenarios where further pruning is possible as discussed. However, the cost and benefits of such pruning remain unknown on the GPU, meriting further exploration.

Chapter 6

Epilogue

This research endeavor has been driven by the overarching objective to augment the efficacy of hardware acceleration for graph data processing. In pursuit of this aim, we have systematically tackled three interrelated computational challenges: matrix factorization, k nearest neighbors search, and subgraph matching, each presenting unique opportunities for performance optimization.

In the realm of matrix factorization detailed in Chapter 3, we demonstrate that a non-uniform data block allocation strategy improves GPU efficiency without sacrificing recall. Our proposed cost model and dynamic workload distribution optimize resource utilization and ensure a balanced load across working units, thereby enhancing overall system performance.

Continuing our exploration, we present an efficient graph-based algorithm designed to address the k nearest neighbors search problem on the GPU in Chapter 4. This approach significantly enhances the search process by parallelizing data structure operations. Furthermore, we employ a divide-and-conquer strategy that enables the efficient construction of graphs on the GPU.

Finally, in Chapter 5, we introduce a backtracking-based method for subgraph matching on the GPU. This approach surpasses the existing method by enhancing pruning efficiency. Additionally, we optimize memory read-write operations to further improve the performance. Impressively, our method shows remarkable scalability, affirming its strength in handling large-scale graph data.

Together, these contributions mark a meaningful advancement in applying hardware acceleration to complex data processing tasks. This thesis emphasizes the interplay between algorithmic development and hardware capabilities. It not only enhances current computational performance but also provides a versatile foundation for diverse applications ranging from big data analytics to artificial intelligence. Specifically, the matrix factorization techniques could be pivotal in advancing recommender systems, while our kNN and subgraph matching algorithms may set new benchmarks in pattern recognition and database searching. Additionally, it is my aspiration that these endeavors will serve as a catalyst for future research in the realm of hardware acceleration for graph data processing.

Looking ahead, this thesis sets the stage for a range of exciting opportunities in hardware-accelerated graph data processing. In the area of matrix factorization, it is possible to leverage integrated heterogeneous systems equipped with multiple GPUs to accelerate the processing of large-scale datasets. This introduces a set of novel challenges that warrant further investigation. For the k nearest neighbors search problem, we are looking at creative ways to process massive datasets on GPUs, which suffer from their limited memory. And as for subgraph matching, we are keen on developing even more effective pruning techniques to speed up the enumeration process. These are real, tangible issues that the next wave of research will tackle, and we are optimistic that the community will come together to push these innovations even further.

Bibliography

[1] "Insieme compiler runtime framework," http://insieme-compiler.org.

[2] "MSong," http://www.ifs.tuwien.ac.at/mir/msd/download.html.

[3] "Notre," http://phototour.cs.washington.edu/datasets/.

[4] "The webgraph framework i: compression techniques," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 595–602.

[5] "SIFT and GIST," http://corpus-texmex.irisa.fr/, 2010.

[6] "Proxima," "https://www.alibabacloud.com/blog/proxima-a-vector-retrieval-engine-independently-developed-by-alibaba-damo-academy_597699", 2021.

[7] M. T. Al Amin, C. Aggarwal, S. Yao, T. Abdelzaher, and L. Kaplan, "Unveiling polarization in social networks: A matrix factorization approach," in *INFOCOM*, 2017, pp. 1–9.

[8] K. Aoyama, K. Saito, H. Sawada, and N. Ueda, "Fast approximate similarity search based on degree-reduced neighborhood graphs," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2011, pp. 1055–1063.

[9] A. Atserias, M. Grohe, and D. Marx, "Size bounds and query plans for relational joins," in *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 2008, pp. 739–748.

[10] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," *ACM Computing Surveys (CSUR)*, vol. 23, no. 3, pp. 345–405, 1991.

[11] A. Babenko and V. Lempitsky, "Additive quantization for extreme vector compression," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 931–938.

[12] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 1968, pp. 307–314.

[13] M. E. Belviranli, L. N. Bhuyan, and R. Gupta, "A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures," *TACO*, vol. 9, no. 4, p. 57, 2013.

[14] B. Bhattarai, H. Liu, and H. H. Huang, "Ceci: Compact embedding cluster index for scalable subgraph matching," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1447–1462.

[15] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1199–1214.

[16] M. Boyer, K. Skadron, S. Che, and N. Jayasena, "Load balancing in a changing world: dealing with heterogeneity and performance variability," in *CF*, 2013, p. 21.

[17] J. Canny and H. Zhao, “Bidmach: Large-scale learning with zero memory allocation,” in *BigLearning, NIPS Workshop*, 2013.

[18] W. Chen, J. Chen, F. Zou, Y.-F. Li, P. Lu, and W. Zhao, “Robustiq: A robust ann search method for billion-scale similarity search on gpus,” in *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, 2019, pp. 132–140.

[19] Y. Chen, F. Hua, C. Huang, J. Bierema, C. Zhang, and E. Z. Zhang, “Accelerating concurrent heap on gpus,” *arXiv preprint arXiv:1906.06504*, 2019.

[20] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin, “A learning-rate schedule for stochastic gradient methods to matrix factorization,” in *PAKDD*, 2015, pp. 442–455.

[21] H. J. Choi, D. O. Son, S. G. Kang, J. M. Kim, H.-H. Lee, and C. H. Kim, “An efficient scheduling scheme using estimated execution time for heterogeneous computing systems,” *The Journal of Supercomputing*, vol. 65, no. 2, pp. 886–902, 2013.

[22] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, “A (sub) graph isomorphism algorithm for matching large graphs,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[23] P. Cui, X. Wang, J. Pei, and W. Zhu, “A survey on network embedding,” *TKDE*, vol. 31, no. 5, pp. 833–852, 2019.

[24] V. Dodeja, M. Almasri, R. Nagi, J. Xiong, and W.-m. Hwu, “Parsec: Parallel subgraph enumeration in cuda,” in *2022 IEEE International Parallel*

and Distributed Processing Symposium (IPDPS). IEEE, 2022, pp. 168–178.

[25] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th international conference on World wide web*, 2011, pp. 577–586.

[26] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[27] P. Erdős, A. Rényi *et al.*, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[28] C. Fu and D. Cai, "Efanna: An extremely fast approximate nearest neighbor search algorithm based on knn graph," *arXiv preprint arXiv:1609.07228*, 2016.

[29] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," *Proceedings of the VLDB Endowment*, vol. 12, no. 5.

[30] S. Funk, "Netflix update: Try this at home," https://sifter.org/~simon/journal/20061211.html, 2006.

[31] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 4, pp. 744–755, 2013.

[32] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *KDD*, 2011, pp. 69–77.

[33] A. Ghose, S. Dey, P. Mitra, and M. Chaudhuri, "Divergence aware automated partitioning of opencl workloads," in *ISEC*, 2016, pp. 131–135.

[34] L. Gong, H. Wang, M. Ogihara, and J. Xu, "idec: indexable distance estimating codes for approximate nearest neighbor search," *Proceedings of the VLDB Endowment*, vol. 13, no. 9, pp. 1483–1497, 2020.

[35] D. Grewe and M. F. O'Boyle, "A static task partitioning approach for heterogeneous systems using opencl," in *CC*, 2011, pp. 286–305.

[36] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan, "Gpu-accelerated subgraph enumeration on partitioned graphs," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1067–1082.

[37] W. Guo, Y. Li, and K.-L. Tan, "Exploiting reuse for gpu subgraph enumeration," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[38] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han, "Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1429–1446.

[39] W.-S. Han, J. Lee, and J.-H. Lee, "Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 337–348.

[40] B. Harwood and T. Drummond, "Fanng: Fast approximate nearest neighbour graphs," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 5713–5722.

[41] T. Hastie, R. Mazumder, J. D. Lee, and R. Zadeh, “Matrix completion and low-rank svd via fast alternating least squares,” *The Journal of Machine Learning Research*, vol. 16, no. 1, pp. 3367–3402, 2015.

[42] H. He and A. K. Singh, “Graphs-at-a-time: query language and access methods for graph databases,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 405–418.

[43] X. He, H. Zhang, M.-Y. Kan, and T.-S. Chua, “Fast matrix factorization for online recommendation with implicit feedback,” in *SIGIR*, 2016, pp. 549–558.

[44] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng, “Query-aware locality-sensitive hashing for approximate nearest neighbor search,” *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 1–12, 2015.

[45] P. Indyk and R. Motwani, “Approximate nearest neighbors: towards removing the curse of dimensionality,” in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604–613.

[46] H. Jegou, M. Douze, and C. Schmid, “Product quantization for nearest neighbor search,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 1, pp. 117–128, 2010.

[47] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, “Gpu register file virtualization,” in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 420–432.

[48] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, “Dissecting the nvidia volta gpu architecture via microbenchmarking,” *arXiv preprint arXiv:1804.06826*, 2018.

[49] V. J. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro, "Predictive runtime code scheduling for heterogeneous architectures," in *HiPEAC*, 2009, pp. 19–33.

[50] J. Jin, S. Lai, S. Hu, J. Lin, and X. Lin, "Gpusgd: A gpu-accelerated stochastic gradient descent algorithm for matrix factorization," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 14, pp. 3844–3865, 2016.

[51] Z. Jin, C. Li, Y. Lin, and D. Cai, "Density sensitive hashing," *IEEE transactions on cybernetics*, vol. 44, no. 8, pp. 1362–1371, 2013.

[52] D. S. Johnson and M. R. Garey, *Computers and intractability: A guide to the theory of NP-completeness.* WH Freeman, 1979.

[53] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, 2019.

[54] Y. Kalantidis and Y. Avrithis, "Locally optimized product quantization for approximate nearest neighbor search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 2321–2328.

[55] R. Kaleem, R. Barik, T. Shpeisman, C. Hu, B. T. Lewis, and K. Pingali, "Adaptive heterogeneous scheduling for integrated gpus," in *PACT*. IEEE, 2014, pp. 151–162.

[56] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, 2000, pp. 163–170.

[57] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer, "An automatic input-sensitive approach for heterogeneous task partitioning," in *ICS*, 2013, pp. 149–160.

[58] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, no. 8, pp. 30–37, 2009.

[59] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce," *Proceedings of the VLDB Endowment*, vol. 8, no. 10, pp. 974–985, 2015.

[60] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 217–228, 2016.

[61] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang *et al.*, "A survey and experimental analysis of distributed subgraph matching," *arXiv preprint arXiv:1906.11518*, 2019.

[62] J. Lee, M. Samadi, and S. Mahlke, "Orchestrating multiple data-parallel kernels on multiple devices," in *PACT*, 2015, pp. 355–366.

[63] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *Proceedings of the VLDB Endowment*, vol. 6, no. 2, pp. 133–144, 2012.

[64] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age," in *SIGMOD*, 2014, pp. 743–754.

[65] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[66] B. Lessley and H. Childs, “Data-parallel hashing techniques for gpu architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 237–250, 2019.

[67] B. Li, S. Tata, and Y. Sismanis, “Sparkler: Supporting large-scale matrix factorization,” in *EDBT*, 2013, pp. 625–636.

[68] S. Li, J. Kawale, and Y. Fu, “Predicting user behavior in display advertising via dynamic collective matrix factorization,” in *SIGIR*, 2015, pp. 875–878.

[69] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin, “Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 8, pp. 1475–1488, 2019.

[70] D. Lian, C. Zhao, X. Xie, G. Sun, E. Chen, and Y. Rui, “Geomf: joint geographical modeling and matrix factorization for point-of-interest recommendation,” in *KDD*, 2014, pp. 831–840.

[71] W. Liu, C. Mu, S. Kumar, and S.-F. Chang, “Discrete graph hashing,” in *Proceedings of the 27th International Conference on Neural Information Processing Systems-Volume 2*, 2014, pp. 3419–3427.

[72] C.-K. Luk, S. Hong, and H. Kim, “Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping,” in *MICRO*, 2009, pp. 45–55.

[73] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, “Approximate nearest neighbor algorithm based on navigable small world graphs,” *Information Systems*, vol. 45, pp. 61–68, 2014.

[74] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[75] J. Martinez, J. Clement, H. H. Hoos, and J. J. Little, "Revisiting additive quantization," in *European Conference on Computer Vision*. Springer, 2016, pp. 137–153.

[76] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *CSUR*, vol. 47, no. 4, p. 69, 2015.

[77] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, 2012, pp. 37–48.

[78] H. Q. Ngo, C. Ré, and A. Rudra, "Skew strikes back: New developments in the theory of join algorithms," *ACM SIGMOD Record*, vol. 42, no. 4, pp. 5–16, 2014.

[79] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2. Ieee, 2006, pp. 2161–2168.

[80] J. Oh, W.-S. Han, H. Yu, and X. Jiang, "Fast and robust parallel sgd matrix factorization," in *KDD*, 2015, pp. 865–874.

[81] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *CGO*. ACM, 2014, p. 273.

[82] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *EMNLP*, 2014, pp. 1532–1543.

[83] M. Pradeep, J. M. Navin, and B. Kannappan, "A review of scheduling mechanisms for heterogeneous multi-core machines," *International Journal of Pure and Applied Mathematics*, vol. 120, no. 6, pp. 13–24, 2018.

[84] N. Pržulj, D. G. Corneil, and I. Jurisica, "Efficient estimation of graphlet frequency distributions in protein–protein interaction networks," *Bioinformatics*, vol. 22, no. 8, pp. 974–980, 2006.

[85] M. Qiao, H. Zhang, and H. Cheng, "Subgraph matching: on compression and computation," *Proceedings of the VLDB Endowment*, vol. 11, no. 2, pp. 176–188, 2017.

[86] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, and J. Tang, "Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec," in *WSDM*, 2018, pp. 459–467.

[87] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *NIPS*, 2011, pp. 693–701.

[88] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 617–628, 2015.

[89] R. A. Rossi and R. Zhou, "Leveraging multiple gpus and cpus for graphlet counting in large networks," in *CIKM*, 2016, pp. 1783–1792.

[90] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 364–375, 2008.

[91] T. A. Snijders, P. E. Pattison, G. L. Robins, and M. S. Handcock, "New specifications for exponential random graph models," *Sociological methodology*, vol. 36, no. 1, pp. 99–153, 2006.

[92] J. Song, Y. Yang, Z. Huang, H. T. Shen, and R. Hong, "Multiple feature hashing for real-time large scale near-duplicate video retrieval," in *Proceedings of the 19th ACM international conference on Multimedia*, 2011, pp. 423–432.

[93] E. Stehle and H.-A. Jacobsen, "A memory bandwidth-efficient hybrid radix sort on gpus," in *SIGMOD*, 2017, pp. 417–432.

[94] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[95] S. J. Subramanya, F. Devvrit, H. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," *Advances in Neural Information Processing Systems*, vol. 32, pp. 13 771–13 781, 2019.

[96] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1083–1098.

[97] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He, "Rapidmatch: a holistic approach to subgraph query processing," *Proceedings of the VLDB Endowment*, vol. 14, no. 2, pp. 176–188, 2020.

[98] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin, "Srs: solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index," *Proceedings of the VLDB Endowment*, 2014.

[99] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *arXiv preprint arXiv:1205.6691*, 2012.

[100] H.-N. Tran, J.-j. Kim, and B. He, "Fast subgraph matching on large graphs using graphics processors," in *International Conference on Database Systems for Advanced Applications.* Springer, 2015, pp. 299–315.

[101] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[102] T. L. Veldhuizen, "Leapfrog triejoin: a worst-case optimal join algorithm," *arXiv preprint arXiv:1210.0481*, 2012.

[103] J. Wang, J. Wang, J. Song, X.-S. Xu, H. T. Shen, and S. Li, "Optimized cartesian k-means," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 1, pp. 180–192, 2014.

[104] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *arXiv preprint arXiv:2101.12631*, 2021.

[105] Z. Wang, L. Zheng, Q. Chen, and M. Guo, "Cap: co-scheduling based on asymptotic profiling in cpu+ gpu hybrid systems," in *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, 2013, pp. 107–114.

[106] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," in *VLDB*, vol. 98, 1998, pp. 194–205.

[107] Y. Wen, Z. Wang, and M. F. O'boyle, "Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms," in *HiPC*, 2014, pp. 1–10.

[108] P. Wieschollek, O. Wang, A. Sorkine-Hornung, and H. Lensch, "Efficient large-scale approximate nearest neighbor search on the gpu," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2027–2035.

[109] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "Cumf_sgd: Parallelized stochastic gradient descent for matrix factorization on gpus," in *HPDC*, 2017, pp. 79–92.

[110] D. Xu, I. W. Tsang, and Y. Zhang, "Online product quantization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 11, pp. 2185–2198, 2018.

[111] X. Yan, P. S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, 2004, pp. 335–346.

[112] A. B. Yandex and V. Lempitsky, "Efficient indexing of billion-scale datasets of deep descriptors," in *IEEE Conference on Computer Vision & Pattern Recognition*, 2016, pp. 2055–2063.

[113] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *ICDM*, 2012, pp. 765–774.

[114] M. Yu, L. Qin, Y. Zhang, W. Zhang, and X. Lin, "Aot: Pushing the efficiency boundary of main-memory triangle listing," in *Database Systems for Advanced Applications: 25th International Conference, DASFAA*

2020, Jeju, South Korea, September 24–27, 2020, Proceedings, Part II 25. Springer, 2020, pp. 516–533.

[115] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *PVLDB*, vol. 7, no. 11, pp. 975–986, 2014.

[116] L. Zeng, L. Zou, M. T. Özsu, L. Hu, and F. Zhang, "Gsi: Gpu-friendly subgraph isomorphism," in *2020 IEEE 36th International Conference on Data Engineering (ICDE).* IEEE, 2020, pp. 1249–1260.

[117] S. Zhang, S. Li, and J. Yang, "Gaddi: distance index based subgraph matching in biological networks," in *Proceedings of the 12th international conference on extending database technology: advances in database technology*, 2009, pp. 192–203.

[118] P. Zhao and J. Han, "On graph query optimization in large networks," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 340–351, 2010.

[119] W. Zhao, S. Tan, and P. Li, "Song: Approximate nearest neighbor search on gpu," in *2020 IEEE 36th International Conference on Data Engineering (ICDE).* IEEE, 2020, pp. 1033–1044.

[120] E. Zhong, Y. Shi, N. Liu, and S. Rajan, "Scaling factorization machines with parameter server," in *CIKM*, 2016, pp. 1583–1592.

[121] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin, "A fast parallel sgd for matrix factorization in shared memory systems," in *ACM RecSys*, 2013, pp. 249–256.