

國立臺灣大學電機資訊學院電機工程學系

碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

在圖形處理器上之並行極大二分團列舉

Parallel Maximal Biclique Enumeration on GPUs

張家鳴

Chia-Ming Chang

指導教授: 郭斯彥 博士

Advisor: Sy-Yen Kuo, Ph.D.

中華民國 112 年 7 月

July, 2023

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

在圖形處理器上之並行極大二分圖列舉

Parallel Maximal Biclique Enumeration on GPUs

本論文係張家鳴(姓名)R10921101(學號)在國立臺灣大學電機工程學系完成之碩士學位論文，於民國 112 年 7 月 12 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Electrical Engineering on 12(date) 7(month) 2023(year) have examined a Master's thesis entitled above presented by Chia-Ming Chang (name) R10921101 (student ID) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

鄧斯序

(指導教授 Advisor)

雷欽隆

林振峰

陳英一

郭冠羽

系主任 Director:

李建模



Acknowledgements

隨著碩士論文的最後幾次編輯，我的研究所生活也即將結束了。在分散式系統與網路實驗室的兩年間成長了許多，謹以此致謝表達自己萬分的謝意。

首先要謝謝郭斯彥教授的指導，在我找不到實驗室而徬徨無助的時候，選擇擔任我的指導教授。在我的研究方向上給了我很大的自由度，並且替分散式系統與網路實驗室營造了良好的研究氛圍，使得每次都能開心的前往實驗室，心靈富足的離開。也要感謝每週五不遠千里前來的袁世一教授，在每一次報告的提問都讓我更客觀、更批判性的去看待自己的研究題目，並且在為研究感到迷惘時，總能給我清楚且實際的指引與建議。

接下來要感謝的是與我共同研究的謝宙穎學長與鄭博修同學。由衷感謝謝宙穎學長的細心指導，在每一個我自己對研究感到挫折無力不想向前的時候，總是精力滿滿的提供不同的看法與可能性，讓優化的方向有更全面的考量，也讓論文的論述更加完整。此外也要感謝我的研究夥伴鄭博修，有他共同參與實作的開發與討論，不只減少了一個人獨自解決問題的無力感，也在每個研究卡關的時候，提供抽象又具體的舉例說明，讓原本枯燥惱人的實作，增添了幾份快活的氣氛。除此之外也要感謝宇宙 team 的其他人，謝謝朱祐葳學長與賴明緯學長在碩一時的指導，為組內會議提供充沛的知識與溫暖的支持，每次會議後的休閒愉樂至今讓我回味無窮。也謝謝劉彥甫在會議中提出獨到且實務的見解，並展現他對學術與

專業上的熱忱與不懈，為宇宙 team 樹立榜樣。最後謝謝碩一的冠榮、冠宇和柏瑜，在組內會議上提供的問題與意見，都讓我在準備口試的路上更加順利。此外，也要感謝分散式系統與網路實驗室的其他夥伴，mtk 姐妹會的各位，碩二一起當離散助教和準備口試事宜的子楷，到現在還沒跟我確定關係的田鈺光……。

最後，也感謝我的家人與好友，給我無盡的支持與鼓勵。在每個懷疑自己的時候，都能用正向的話語和務實的解決方法，讓我重拾力量繼續向前。

二零二三年七月三日

張家鳴

於博理 607



摘要

本論文透過圖形處理器優化極大二分團列舉 (Maximal Biclique Enumeration) 的執行效能。過往最大二分團列舉之演算法皆採用深度優先搜尋技巧，同時需要大量的遞迴呼叫與集合操作。然而，因為圖形處理器在硬體上的限制，使得深度優先搜尋與遞迴呼叫並不可行，並且圖形處理器上的記憶體限制也使得過往的資料結構，會導致記憶體不足的問題。同時，在解決前述之硬體限制之後，也因為圖形處理器本身高平化度的特性，使得如何平均有效率的將工作量分配給每個執行單元，變成影響效能的最大問題之一。

本研究改寫過往之極大二分團列舉演算法，將遞迴呼叫改為迴圈形式，減緩在圖形處理器上呼叫函式所造成的高負擔；同時，也使用新的資料結構，以解決原集合在圖形處理器上不可用之問題，進而減少了宣告集合對記憶體的需求。後續進一步使用了輕量化候選項選擇、工作竊取等技巧，緩解在圖形處理器上執行緒區塊之間工作量不平衡之情形；並使用平行交集計算與雙層平行計算，進一步利用執行緒區塊內之平行度。在提出極大二分團列舉在圖形處理器上之可能性的同時，提高此問題在圖形處理器上之效能。

關鍵字：二分團、圖形處理器、平行運算



Abstract

Maximal biclique is a crucial property in many applications, such as social network analysis and computational biology. By Maximal Biclique Enumeration (MBE), we can list all the maximal biclique in a given graph and thus extract the insight of it. Along with the growth of graph size, previous studies have introduced different algorithms to solve the MBE problem with parallelism on CPUs to improve the scalability ; however, to the best of author's knowledge, the problem has not been implemented on Graphic Processing Unit (GPU).

In this study, it aims to improve scalability and runtime of MBE through the parallelism provided by GPUs. First, to successfully migrate the MBE algorithm to the GPUs, it proposes a loop-based algorithm to replace the recursion, and then uses a new data structure to solve the memory explosion problem. Lastly, profiting from the parallelism of GPUs, it develops lightweight candidate selection and work-stealing techniques to alleviate the workload imbalance between thread blocks. Furthermore, it leverages parallel

intersection computation and two-level parallelism within each thread block to effectively utilize the inherent parallelism of these blocks.

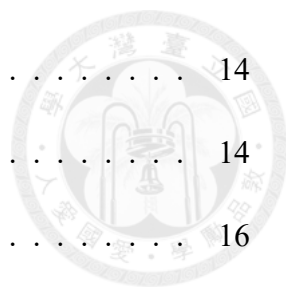
Keywords: biclique, GPU, parallel computing





Contents

	Page
Verification Letter from the Oral Examination Committee	i
Acknowledgements	ii
摘要	iv
Abstract	v
Contents	vii
Chapter 1 Introduction	1
Chapter 2 Background	5
2.1 GPU architecture	5
2.2 GPU memory hierarchy	7
2.3 Compute Unified Device Architecture	7
2.3.1 CUDA synchronization granularity	7
2.3.2 CUDA Dynamic Parallelism	8
2.4 Maximal Biclique Enumeration	9
Chapter 3 Related work	10
3.1 iMBEA	10
3.1.1 Tree Pruning	13
3.1.2 Candidate Selection	13



3.2	Share-Memory parMBE	14
3.2.1	First level recursion tree	14
3.2.2	Memory consumption	16
3.2.3	Parallel Intersection Computation	16
3.3	PMCE on GPUs	17
3.3.1	Compact Representation	17
Chapter 4	Migration the parallel MBE Algorithm to GPUs	18
4.1	Obstacles and Overview of Implementation	18
4.2	Loop-based DFS Control Flow on GPUs	19
4.3	Compact Representation in cuMBE	21
Chapter 5	Optimization of Parallel MBE Algorithm on GPUs	22
5.1	Workload Imbalance	22
5.2	Lightweight Candidate Selection	23
5.2.1	First Level Early Stop	24
5.2.2	Second Level Early Stop	25
5.3	Work Stealing	26
5.4	Parallel Intersection Computation	27
5.5	Two Level Parallelism	28
Chapter 6	Evaluation	30
6.1	Configurations	30
6.1.1	Platforms and Datasets	30
6.1.2	CPU Baselines	32
6.2	Overall Performance	32

6.3	Workload Balance	32
References		35



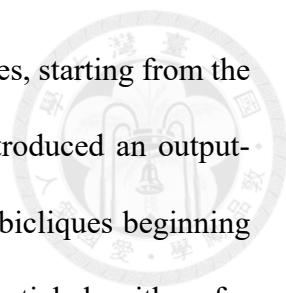


Chapter 1 Introduction

This paper studies the Maximal Biclique Enumeration (MBE) problem. With a given bipartite graph, we need to enumerate all the maximal biclique of the graph. A bipartite graph $G = (U, V, E)$ consists of two disjoint vertex sets, L and R ; also, $\forall (u, v) \in E, (u, v) \in LXR$. If there are two vertex sets named A, B , which $A \subseteq L$ and $B \subseteq R$, and $\forall a \in A \forall b \in B, (a, b) \in E$, then we call $G = (A, B, E)$ is a biclique. Besides, if a biclique B is not a proper sub-graph of any other larger biclique, we call B is a maximal biclique.

Maximal bicliques have proven useful in solving the a wide range of other practical applications, such as web community analysis, graph neural network aggregation acceleration, computational biology and so on. In practical scenarios, such as the aforementioned applications, bicliques with a small number of vertices are often uninteresting. Thus, it is preferable to extract only large or even maximal bicliques. However, as the quantity of maximal bicliques can grow exponentially[14], the computational complexity of MBE is considerable.

Research on sequential algorithms for solving the MBE problem has been ongoing for over a decade. iMBEA[16] leverages the fact that all maximal bicliques in a bipartite graph G can be listed by exploring the sets in either 2^L or 2^R (the power set for L or



R). It suggests a recursive approach for enumerating maximal bicliques, starting from the vertex set with the smallest number of vertices. Alexe et al.[3] introduced an output-sensitive algorithm that constructs large bicliques by merging small bicliques beginning with stars. Numerous other studies have focused on developing sequential algorithms for solving MBE in static graphs. On the other hands, parallel algorithms for solving MBE problem are not investigated as much as sequential ones.

The first parallel algorithm POP-MBC was proposed by [10] in 2009. It utilizes a round-robin strategy to achieve high load balancing among different processors and reduced the overhead of synchronization with reduced vertex set. Besides, Arko et al.[9] devised a parallel algorithm on MapReduce platform, which clustering the input graph into separate graphs, and solve the sub-problem independently. ParMBE[6], on the other hand, introduced another parallel algorithm based on the MineLMBC[8]. It leverages memory shared by multiple cores, and is also the state of the art on the problem of MBE.

Previous studies have shown that they made good use of the power of parallelism between multiples instances and cores. However the current methods do not migrate the MBE problem to GPUs. As the size of graph growing up, we need more parallelism to improve to scalability of parallel algorithm on solving MBE. Hence, GPUs could be a reasonable solution. There are several difficulties when we try to solve the MBE problem on GPUs. First, most algorithms of the MBE problem are depth first search (DFS) manner, which means they need a lot of recursion function call during the execution of algorithm. However, due to the hardware limitation, recursion is not suitable for GPU architecture. Furthermore, because the previous parallel algorithms were running on the CPU, they had larger memory space compared to GPUs. This allowed them to declare new data structures with each recursive call and each core. However, when it comes to GPUs, this solution

does not work on GPU. When we want to use tens of thousands of thread on GPU, it is not reasonable for us to allocate a group of data structure for each layer of recursion and each thread, especially under the situation that the GPUs own fewer DRAM than CPUs do.

As a result, to the best of author's knowledge, we present the first parallel algorithms to solve MBE on GPUs. In order to successfully transition the MBE algorithm to GPUs, we replace the recursion structure into loop to meet the architecture of GPU; also, due to the limitation of device memory on GPUs, we adopted the compact representation method proposed in parMCE [4] to reduce the usage of device memory. Besides, in order to deal with the workload imbalance problems between thread blocks, We propose a lightweight candidate selection approach to reduce the overall workload and achieve a more balanced recursion tree. Furthermore, we introduce a work-stealing mechanism, which allows each thread block to have a fair distribution of tasks. Furthermore, to better utilize the parallelism within each thread block, we adopt the Parallel Intersection Computation (PIC) method proposed in [6]. Additionally, we introduce a two-level parallelism scheme, which enables us to distribute the workload to threads within each thread block more flexibly.

To sum up, the contributions of this paper are:

- We present the first parallel algorithms to solve MBE on GPUs.
- We transform the recursion structure into loop-based structure and adopt compact representation to migrate MBE algorithm to GPUs.
- We propose the lightweight candidate selection and work stealing technique to solving the unbalancing problem inter thread block.
- We adopt parallel intersection computation and introduce the two-level parallelism

skill to make good use the parallelism within each thread block.

- The results show that our approach can achieve up to 3.83x overall speedup in the social network datasets, and scale up the size of graph dramatically.

The rest of the thesis is organized as follow. Chapter 2 introduces the GPU architecture manufactured by NVIDIA and the CUDA programming model as well. Besides, it gives the definition and applications of Maximal Biclique Enumeration (MBE) problem. Chapter 3 discusses the related works and some optimizations they proposed. Chapter 4 describes the challenges we faced when migrating the MBE algorithm to GPU, and the implementation details of our solution. Chapter 5 will delve into the methods we propose to address the inter-thread block imbalance issue, as well as explain how to better utilize the parallelism within a single thread block. Chapter 6 evaluates different parallel strategies for MBE and compares them with prior studies. Chapter 7 concludes the study, covering the future works as well.



Chapter 2 Background

This section introduces the architecture of Nvidia's Graphics Processing Unit (GPU), as well as the Compute Unified Device Architecture (CUDA) [12] programming model. It also defines the problem of maximal biclique enumeration (MBE) and the application of it.

2.1 GPU architecture

There are 2 perspectives of GPU architecture, the CUDA programming one and the hardware one. Each of them can be mapped from one to another. In figure 2.1, it shows the mapping relation between CUDA programming interface and hardware. From the CUDA programming aspect, we can see that a group of thread can construct a thread block, and a group of thread blocks is called a CUDA kernel grid. On the other hand, a GPU hardware execution resource can be separated into several Streaming Multiprocessor (SM), and each SM is consisted of a group of thread. Each thread block is executed on a single SM, and will not be migrated to other SMs. Each grid is executed by one GPU, and a GPU can execute multiple kernel grids simultaneously.

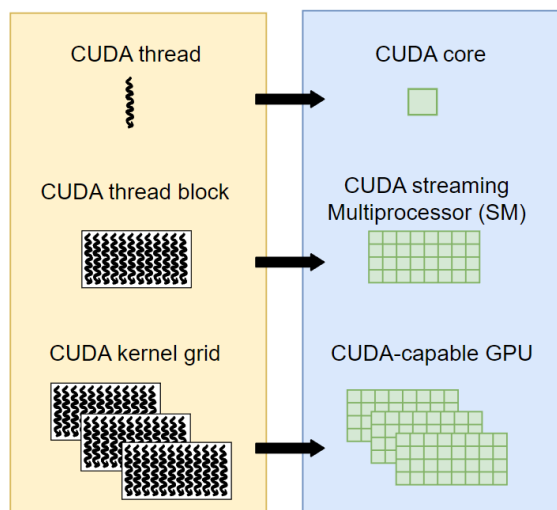


Figure 2.1: GPU architecture

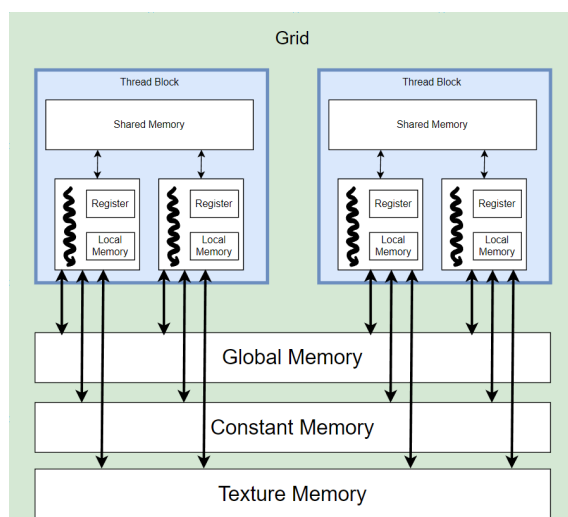
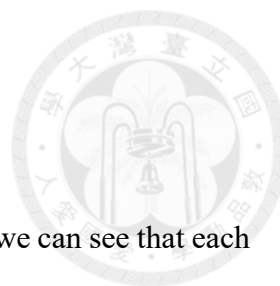


Figure 2.2: GPU architecture



2.2 GPU memory hierarchy

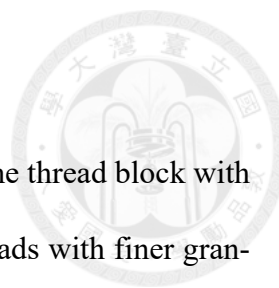
The memory hierarchy of GPU has several levels. In figure 2.2, we can see that each thread has its own register file and local memory. Besides, each thread in a thread block shares the shared memory, which is also the L1 cache. The thread in thread block A cannot access the data in the shared memory of thread block B; also, when we declare a shared variable, this variable will only be allocated once, and all the threads in thread block will share the variable. Finally, there is a global device memory which is available to all the threads in all thread block. Because the location of shared memory is on-chip, the latency of shared memory is roughly 100x lower than the local and global memory [7].

2.3 Compute Unified Device Architecture

The Compute unified device architecture (CUDA) is proposed by Nvidia, which make programmers and researchers implement the function they want with a C-like programming language, executing on the GPUs provided by Nvidia. The functions executed on GPUs are called device code or kernel function, while the instructions executed on CPUs are call host code. All the kernel functions need be launched by CPU and specify the size of grid and block when the kernel is launched.

2.3.1 CUDA synchronization granularity

When we try to implement our function and arrange the behavior of massive thread on the GPUs, the synchronization granularity plays an important role. If we synchronize with wrong or improper granularity, the program might be not efficient, or even wrong.

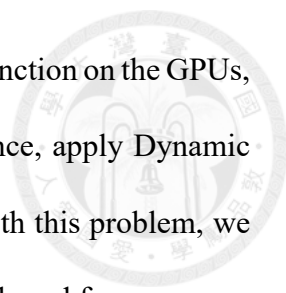


Before CUDA 9, we can only synchronize the threads in the same thread block with `__syncthreads()` function. Hence, if we can to synchronize the threads with finer granularity, we can only decrease the size of thread block. On the other hand, if we want to synchronize the whole grid, we can only finish the current kernel function, relaunching the next kernel function to achieve the global synchronization. which is suffered from extremely heavy overhead of function launching.

In CUDA 9 [11], Nvidia proposed two features: `cooperativeKernelLaunch` and `__syncwarp()` functions. With the `cooperativeKernelLaunch` function, we can synchronize the whole grid in the kernel function by `grid.sync()` function without kernel launching overhead. Because of the function, our the overhead of work stealing features could be relieved. Besides, the `__syncwarp()` function offers us the opportunity to synchronize the threads in the granularity of warp, which is the basic scheduling unit for GPUs. With `__syncwarp()` function, we can leverage another parallelism in the Parallel Intersection Computation.

2.3.2 CUDA Dynamic Parallelism

As we mentioned in chapter 2.1, a kernel function need be launched from the host. Under this circumstance, the recursive algorithm is not achievable by kernel function, for it cannot call itself on the device side. Hence, in CUDA 6, Dynamic Parallelism [2] was introduced by Nvidia, which offers programmers the ability to launch another kernel function from the original one. This seems to give programmers the opportunity to implement the recursive function on the GPUs. However, [15] indicated that the overhead of dynamic



parallelism is considerable. Even if we could implement a recursive function on the GPUs, it suffers from the launching overhead which is not acceptable. Hence, apply Dynamic Parallelism naively is obviously not suitable for our task. To deal with this problem, we rewrite the iMBEA algorithm from a recursive algorithm into a loop-based form.

2.4 Maximal Biclique Enumeration

A bipartite graph is a mathematical structure that consists of two disjoint sets of vertices, denoted as U and V , such that all edges in the graph connect a vertex from U to a vertex from V . In other word, there isn't any edge connect two vertices in U or in V at the same time.

Given a bipartite graph $G = (U, V, E)$, a biclique is a complete subgraph of G , where every vertex in U is connected to every vertex in V . If a biclique, denoted as B , is not proper contained by any other biclique, B is defined as a Maximal Biclique.

Maximal Biclique Enumeration refers to the process of enumerating all the maximal biclique in a given bipartite graph. The enumeration of maximal bicliques is critical in various applications such as social network analysis, neural network acceleration, data mining, and pattern recognition. It gives us the insights into the densest and largest complete bipartite subgraphs within a graph. However, this problem is proved to be NP-complete [13]. Hence, because of the value of the insights MBE provide, how to speedup the runtime of MBE is an important issue, especially in large graph, for the graph constructed from social network and biological information are complicated and large.



Chapter 3 Related work

The recent studies [1, 5, 6] are constructed using the iMBEA algorithm [16], which is specifically designed for bipartite graphs. In this chapter, we discuss the iMBEA algorithm [16] in detail, including the optimized techniques it introduced. Also, we describe other optimized techniques proposed by ParMBE[6]. Finally, we also introduce a relevant work in this field is the GPU-based solution for maximal clique enumeration (MCE) proposed by [4]. The compact representation of the vertex sets and the workload balancing methods it proposed inspire us quite much.

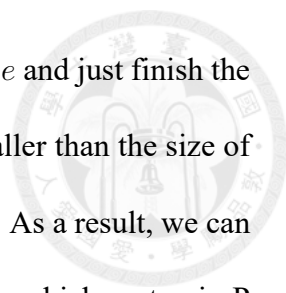
3.1 iMBEA

The iMBEA [16] (Improved Maximal Biclques Enumeration Algorithm) algorithm is designed for enumerate all maximal bicliques in a given bipartite graph efficiently. In the beginning of iMBEA, it select a vertex x from the P set, adding x into R . Based on the vertices in R , it then decides which vertices in U can remain in L' set.

Based on the L' , it scan the whole Q set afterward. for each vertex v in Q , it checks the number of neighbors of v $N[v]$ in L' . If $N[v]$ is equal to the size of L' , it means that the vertex v should be added into R as well. However, based on the principle of Q , we cannot add any vertex in Q into R , which means the current biclique cannot be one of the

Algorithm 1 iMBEA

1: **procedure** iMBEA(G, L, P, Q, R)
2: G : a bipartite graph $G = (U \cup V, E)$
3: L : vertex subset of U consisting of common neighbors of vertices in R .
4: R : vertex subset of V consisting the vertices of current maximal biclique in V set.
5: P : vertex subset of V consisting the candidates selected from to be added to R .
6: Q : vertex subset of V consisting the vertices which have been added into P and cannot be added into R again.
7: **while** $|P| \neq 0$ **do**
8: $P = P - \{x\}$
9: $R' = R \cup \{x\}$
10: $L' = \{u \in L \mid (u, x) \in E(G)\}$
11: $P' = \emptyset$
12: $Q' = \emptyset$
13: $is_maximal = true$
14: $\bar{L}' = L \setminus L'$
15: $C = \{x\}$
16: **for all** $v \in Q$ **do**
17: $N[v] = \{u \in L' \mid (u, v) \in E(G)\}$
18: **if** $|N[v]| == |L'|$ **then**
19: $is_maximal = false$
20: **break**
21: **else if** $N[v] > 0$ **then**
22: $Q' = Q' \cup \{v\}$
23: **end if**
24: **end for**
25: **if** $is_maximal == TRUE$ **then**
26: **for all** $v \in P$ **do**
27: $N[v] = \{u \in L' \mid (u, v) \in E(G)\}$
28: **if** $|N[v]| == |L'|$ **then**
29: $R' = R' \cup \{v\}$
30: $S = \{u \in \bar{L}' \mid (u, v) \in E(G)\}$
31: **if** $|S| == 0$ **then**
32: $C = C \cup \{v\}$
33: **end if**
34: **else if** $N[v] > 0$ **then**
35: $P' = P' \cup \{v\}$
36: **end if**
37: **end for**
38: PRINT(L', R')
39: **if** $P' \neq \emptyset$ **then**
40: iMBEA(G, L', R', P', Q')
41: **end if**
42: **end if**
43: $Q = Q \cup C$
44: $P = P \setminus C$
45: **end while**
46: **end procedure**



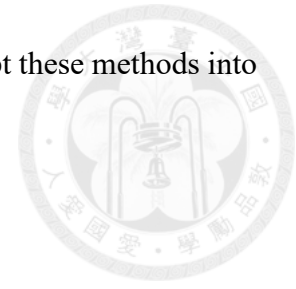
maximal biclique. Hence, we can simply set the *is_maximal* to *false* and just finish the current branch. On the other hand, if the $N[v]$ of each v in Q are smaller than the size of L' , we can conclude that the current biclique satisfies the maximality. As a result, we can move to the next step: Iterating all the vertices in P and determining which vertex in P can be added to R without removing any vertex in L' . It's also worth mentioning that if we find the $N[v]$ is equal to 0, which means the vertex v connects to no vertex in L' , we can directly remove the vertex from Q . This is because we are no longer need to check the vertex anymore. This property is also hold when we check the vertices in P .

Next, Since we have known the maximality of current biclique is satisfied, we can now try to add the vertices in P into R to form R' without reducing the size of L' . To make sure that the vertex added into R from P will not shrink L' , for each vertex v in P , we need to calculate the $N[v]$ as well. If $N[v]$ is equal to the size of L' , it means that we can add v into R "pain-freely". Similarly, in this step we check whether the $N[v]$ is larger than 0. If it is, which means v has the potential to be added R again to construct another maximal biclique. Consequently, We need to add it into P' . Otherwise, we can simply remove v from P' , for there is no opportunity for v to be added into R' while maintaining the size of L' larger than 1.

Finally, we can conclude that the (L', R') is a maximal biclique. At the same time, if there is at least 1 vertex in P' , we can base on the state of current biclique, recursively finding another maximal biclique by adding the vertex from P' into R' and generating the corresponding L'' . Otherwise, if the P' is an empty set, we can finish the current branch, moving the x from P to Q , and step to the next iteration of while loop.

After describing the algorithm of iMBEA, we would discuss some optimization pro-

posed by it. Meanwhile, we would also explore whether we can adopt these methods into our algorithm.



3.1.1 Tree Pruning

As seen in lines 30-32 and lines 43-44 of algorithm 1, iMBEA tried to maintain a set S to contain the vertices in P whose neighbor is totally equal to the L' . With S , we can prune these vertices from the P set into Q set, which reduce the number of iterations of while loop in lines 7 of algorithm 1. While this optimization could effectively reduce the redundant work of finding process, it will also disrupt the order of while loop, which originally remove the vertex from P one by one. This is acceptable under the sequential execution scenario. However, when it comes to the parallel one, we tried to execute each recursion tree in parallel. In other words, we tried to process the while loop in parallel, which means we need to know the status of P set and Q set in advance. Hence, this optimization is not suitable for us.

3.1.2 Candidate Selection

In lines 7 to 8 of Algorithm 1, we can notice that this algorithm try to move a vertex x from P into R to generate the next potential maximal biclique. Naively, we could arbitrarily select any vertex in P , and it would still lead to a correct result. However, drawing from the findings in the [16], if we select the x in P in non-decreasing order, we could not only lead to more balance recursion tree, but also avoid the production of numerous subsets that are not maximal.

As mentioned in section 3.1.1, the parallel algorithm we proposed try to process the

recursion trees in parallel. Under this situation, whether the workload between each execution unit is balance or not is highly related to the balance level of recursion tree. Consequently, the candidate selection technique is quite important for our algorithm.

As described in iMBEA[16], it select the x which minimizes the size of L' from P by implementing the insertion sort in lines 35 of algorithm 1. Similarly, ParLMBC [6] also apply this technique by sorting the vertices in tail set before the recursive exploration. Both of their methods to achieve the non-decreasing order are depending on sorting the entire P set or tail set. Before they try to launch another recursive function call, they store the P or tail set, and recover it after returning from the recursive function call.

3.2 Share-Memory parMBE

The main function block2 of ParMBE proposed by [6] is named ParLMBC. Instead of using L , P , Q , and R set to describe its algorithm, it uses X , $\Gamma(X)$, and $\text{tail}(x)$. X , $\Gamma(X)$ and $\text{tail}(X)$ is similar to R , L and P correspondingly. In the following of section3.2, we will discuss the main challenge we face while trying to migrate this ParMBE algorithm on CPUs to GPUs. Besides, We also describe parallel intersection computation, which is a common neighbor optimization techniques we adopt from parMBE [6] into our own parallel algorithm on GPUs.

3.2.1 First level recursion tree

As seen in 3.1, after entering the entry point, there are lots of recursion tree to be calculate. In the parMBE algorithm, it assigns a core the solve the separate and independent recursion tree to achieve the parallelism. This inspires us to dispatch the thread blocks to



Algorithm 2 parLMBC

```

procedure parLMBC( $X, \Gamma(X), tail(X)$ )
  for  $v \in tail(X)$  do
    if  $|\Gamma(X \cup \{v\})| < 1$  then
       $tail(X) = tail \setminus \{v\}$ 
    end if
  end for
  if  $|X| + |tail(X)| < 1$  then
    return
  end if
  parallel sort vertices of  $tail(X)$  into ascending order of  $|\Gamma(X \cup \{v\})|$ 
  Let the elements of sorted  $tail(X)$  be presented in the order  $0 \dots k$ 
  for  $i \in [0 \dots k]$  in parallel do
     $ntail(X) = tail[i + 1 \dots k]$ 
    if  $|X \cup \{v\}| + |ntail(X)| > 1$  then then
       $Y = \Gamma(\Gamma((X \cup \{v\})))$  in parallel
      if  $Y \setminus (X \cup \{v\})$  is a subset of  $ntail(X)$  then
        if  $|Y| \geq 1$  then
           $B = B \cup \langle Y, \Gamma(X \cup \{v\}) \rangle$ 
        end if
      end if
    end if
    ParLMBC( $Y, \Gamma(X \cup \{v\}), ntail(X) \setminus Y$ )
  end if
end for
end procedure

```

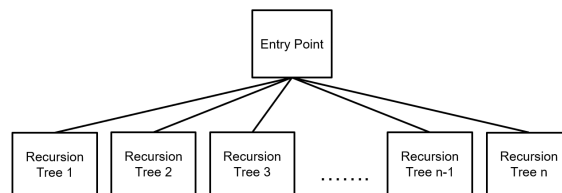


Figure 3.1: Recursion tree of MBE algorithm

solve each recursion tree in the same manner.



3.2.2 Memory consumption

As seen in Algorithm 2, the process of it will launch lots of recursion, and the depth of the recursion tree can reach up to the maximum degree of a vertex in the graph. At the same time, because the algorithm needs to maintain X , $\Gamma(X)$, and $\text{tail}(x)$, the memory consumption will grow up linearly with the depth of recursion. This overhead get even worse when we need to take the advantage of parallelism. In other words, if we execution the algorithm in parallel with 16 cores (like parMBE 2 does), it needs really lots of memory space to avoid out of memory situation. As a result, parMBE [6] selected an experimental environment with 3TBs of RAM. However, when we want to migrate this algorithm to GPUs, we face a vert intuitive question that the device memory of GPUs is 24GBs only. Meanwhile, the number of threads on GPUs exceeds that on CPUs by hundreds to thousands of times. Hence, how to maintain a data structure to describe X , $\Gamma(X)$, and $\text{tail}(x)$ (or L , P , Q , and R) in algorithm is a critical issue to be solved.

3.2.3 Parallel Intersection Computation

Traditionally, the vertex set used in MBE algorithms are stores in the unordered set, which is suitable to perform some set operation on it. However, when we possess a larger number of execution units, to perform the set operation in a sequential manner seems to no longer be as suitable. As a consequence, parMBE [6] proposed the Parallel Intersection Computation (PIC) technique, which perform the intersecting operation from $\Gamma(X \cup v)$ to generate the $\Gamma(\Gamma(X \cup v))$. We adopt the concept to perform the computation of $N[v]$ in

lines 17 and 27 in algorithm 1. We will explain it in detail in chapter 5.



3.3 PMCE on GPUs

PMCE[4] is the first paper who proposed the algorithm to solve the Maximal Clique Enumeration (MCE) problem on GPUs. While there are certain differences in the algorithms and implementation details when enumerating maximal biclique and maximal clique, PMCE[4] has provided us with substantial insights regarding how to maintain the state of vertex sets and how to solve the workload imbalance problem. In the following sections, we will introduce the compact representation of the vertex set proposed in PMCE[4], and explain the workload balance method they used.

3.3.1 Compact Representation

As mentioned in chapter 3.2.2, the memory consumption of MBE algorithm is considerable. Hence, how to maintain the status of vertex set with limited memory space is a necessary issue to be solved. In the PMCE, it leverage the compact representation to store the status of vertex in a single array. With this data structure, the memory consumption is no longer related with the depth. In other words, the function in all levels will share the same single array. While the original usage of this compact representation is to represent the induced graph for the optimization in PMCE, this compact representation also works for us in representing to L,P,Q and R set.



Chapter 4 Migration the parallel MBE Algorithm to GPUs

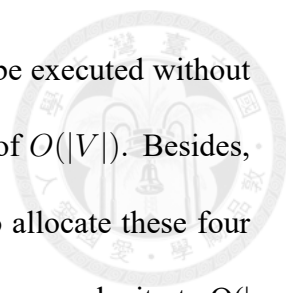
In this chapter, we will describe the challenges faced when attempting to migrate the MBE algorithm from CPU to GPU. Subsequently, we will propose corresponding solutions to these difficulties and provide an in-depth discussion on the details of their implementations.

4.1 Obstacles and Overview of Implementation

- Recursion Function Call

As seen in 1, the algorithm operates on a recursive principle, causing lots of function call during execution. this is acceptable on CPU because the function launching overhead on CPU is relatively small. However, when it comes to GPUs, the situation changes dramatically. Referencing 2.3.2, the dynamic parallelism offers us the opportunity the execute recursive algorithm on GPU, while suffering from serious overhead. Therefore, implementing the iMBEA algorithm without resorting to recursion becomes an issue that must be addressed.

- Limitation of Device Memory



As we observed in Chapter 3.1, the iMBEA algorithm cannot be executed without for vertex sets - L , P , Q and R , leading to a space complexity of $O(|V|)$. Besides, because of the intrinsic nature of recursion, we would need to allocate these four vertex sets with each recursive call, thereby escalating the space complexity to $O(|V| * \text{depth})$. Moreover, because we aim for parallel execution of this algorithm, each degree of parallelism would require four dedicated vertex set. This in turn amplifies the space complexity to $O(|V| * \text{depth} * \text{degree_parallelism})$, resulting out-of-memory issue. Consequently, how to maintain the status of those for vertex sets emerges as a crucial matter.

4.2 Loop-based DFS Control Flow on GPUs

In Algorithm 3, we can observe that we've added an outer loop with the condition *while lvl* ≥ 0 around the original iMBEA algorithm. Moreover, the places in the code where recursion was required have been modified by setting *is_recursive* to true, subsequently breaking the current *while |P| > 0* loop. Following this, the instruction pointer moves to *line 44* to determine whether the reason for breaking out of the loop was due to a need for recursion or a normal condition where $|P| == 0$. Furthermore, it's worth noting that in Algorithm 3, we only describe the process at *line 8* using pseudocode, without delving into the detailed explanation of how we maintain the L , P , Q , and R sets across different levels. This issue will be thoroughly discussed in Chapter 4.3, where we explain how to manage the contents of a single L , P , Q , R set across different levels without significantly affecting performance.

Algorithm 3 cuMBE

```
1: procedure cuMBE( $G, L, P, Q, R$ )
2:    $G$ : a bipartite graph  $G = (U \cup V, E)$ 
3:    $L$ : vertex subset of  $U$  consisting of common neighbors of vertices in  $R$ .
4:    $R$ : vertex subset of  $V$  consisting the vertices of current maximal biclique in  $V$  set.
5:    $P$ : vertex subset of  $V$  consisting the candidates selected from to be added to  $R$ .
6:    $Q$ : vertex subset of  $V$  consisting the vertices which have been added into  $P$  and cannot be
   added into  $R$  again.
7:   while  $lvl \geq 0$  do
8:     maintain  $L, P, Q, R$  for current level
9:     while  $|P| \neq 0$  do
10:       $P = P - \{x\}$ 
11:       $R' = R \cup \{x\}$ 
12:       $L' = \{u \in L \mid (u, x) \in E(G)\}$ 
13:       $P' = \emptyset, Q' = \emptyset$ 
14:       $is\_maximal = true$ 
15:      for all  $v \in Q$  do
16:         $N[v] = \{u \in L' \mid (u, v) \in E(G)\}$ 
17:        if  $|N[v]| == |L'|$  then
18:           $is\_maximal = false$ 
19:          break
20:        else if  $N[v] > 0$  then
21:           $Q' = Q' \cup \{v\}$ 
22:        end if
23:      end for
24:      if  $is\_maximal == TRUE$  then
25:        for all  $v \in P$  do
26:           $N[v] = \{u \in L' \mid (u, v) \in E(G)\}$ 
27:          if  $|N[v]| == |L'|$  then
28:             $P = P - \{v\}$ 
29:             $R' = R' \cup \{v\}$ 
30:          else if  $N[v] > 0$  then
31:             $P' = P' \cup \{v\}$ 
32:          end if
33:        end for
34:        PRINT( $L', R'$ )
35:        if  $P' \neq \emptyset$  then
36:           $Q = Q \cup \{x\}$ 
37:           $is\_recursive = true$ 
38:          break
39:        end if
40:      end if
41:       $Q = Q \cup \{x\}$ 
42:    end while
43:     $is\_maximal = true$ 
44:    if  $is\_recursive == TRUE$  then
45:       $lvl ++$ 
46:    else
47:       $lvl --$ 
48:    end if
49:  end while
50: end procedure
```

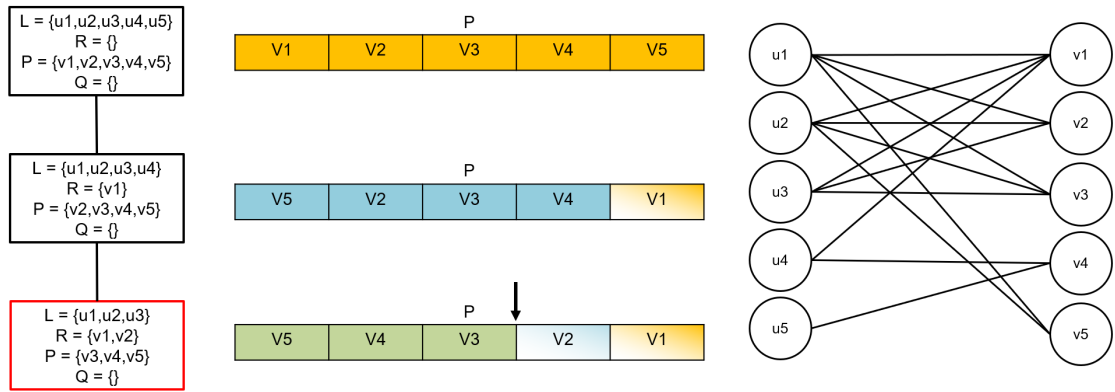


Figure 4.1: Compact representation

4.3 Compact Representation in cuMBE

In Chapter 3.3.1, we mentioned that compact representation was initially used to describe induced graphs. However, we found that this data structure is indeed highly compatible with our MBE algorithm. Firstly, we need to declare an integer array whose size is equal to depth to serve as the pointer for each level (we name it $lp[lvl]$). In Figure 4.1, we can see that each time we need to move a point from P to R, there's no necessity to declare a new array. Instead, we can use an *atomic_swap* operation to swap it to the end of the entire array, and then decrement $lp[lvl + 1]$. In this way, we can determine the size of P for the next level. If we need a recursive call afterward, we can then use $lp[lvl + 1]$ as the size of P in the next level. When we finish the recursion and return, we only need to refer to $lp[lvl]$. Through this method, we can keep all the points we hope to remain in P at the next level compact at the front of the array. Additionally, by observing the value of $lp[lvl]$, we can determine the size of P when we are at different levels. This representation can also be adopted on L, Q and R set.



Chapter 5 Optimization of Parallel MBE Algorithm on GPUs

In this chapter, after successfully migrating the MBE algorithm to GPUs, we encountered a significant issue. Once each thread block is assigned its own recursion tree, workload imbalance arises due to the varying depths of the trees. Therefore, in this chapter, we strive to address this inter-thread block problem. In addition, there is a corresponding issue within each thread block (the intra-thread block problem). Given the hundreds of threads available within a thread block, properly and efficiently assigning work to each thread is a challenge. This task will be elaborated in detail in this chapter.

5.1 Workload Imbalance

As we mentioned in chapter 3.2.1, our intention is to assign each thread block to execute its own independent recursion tree. This idea is indeed feasible, but it confronts a significant challenge - workload imbalance. Since a graph is an unorganized data structure, when we receive a bipartite graph, we have no way of predicting the depth of each recursion tree in advance. As a result, when we want to randomly distribute the recursion trees to different thread blocks, it may occur that some thread blocks end up with several

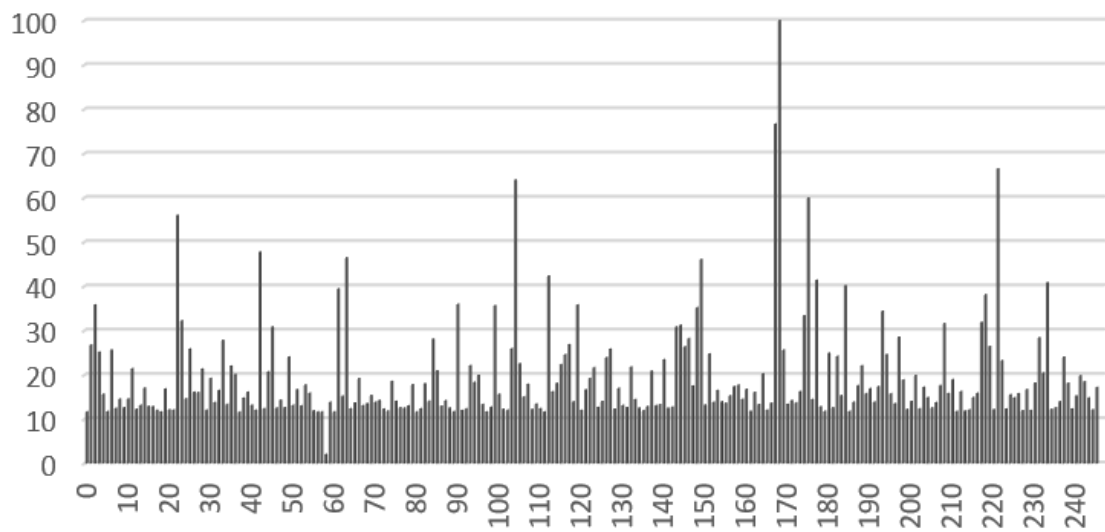


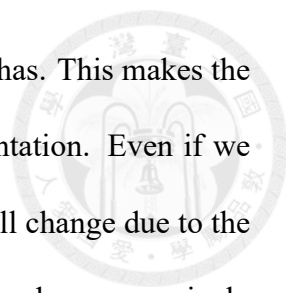
Figure 5.1: Normalized execution time of each thread block

deep recursion trees while others get relatively shallow ones. In such a scenario, the thread blocks that receive the smaller recursion trees enter an idle state prematurely, thereby failing to fully exploit the benefits of parallelism provided by the entire GPU. As illustrated in Figure 5.1, we can observe that without any optimization in distributing recursion trees, most of the thread blocks run for only about 10% to 20% of the time compared to the longest-running thread block. Ideally, we would hope for the completion times of each thread block to be as close as possible.

5.2 Lightweight Candidate Selection

In Chapter 3.1.2, we mentioned that when we want to select vertex v from P to join R , we hope that the size of L for this v can be as small as possible. This not only makes the recursion tree more balanced, but also reduces the overall workload.

In previous studies, both iMBEA and parMBE sorted P according to the size of L before choosing v , and then sequentially popped points from P to join R . However, as we can see in Figure 4.1, when the algorithm goes down recursively and returns to the current



level, though the members of our P set have not changed, their order has. This makes the simple method of sorting P infeasible after we use compact representation. Even if we have sorted the current P according to the size of L , the order of P will change due to the `atomic_swap` operation during the recursion when the algorithm goes down recursively and then returns.

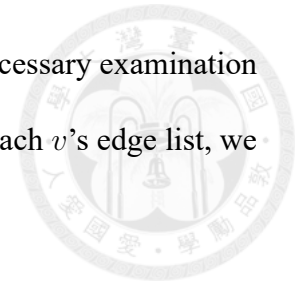
Therefore, we propose another idea, which is to use a linear search method to spend $O(|P|)$ each time to find the vertex in P that can make L the smallest to join R . Although this method is feasible, spending $O(|P|)$ each time brings a non-negligible overhead. Therefore, we propose a two-level early stop mechanism for the linear search operation:

5.2.1 First Level Early Stop

When we wish to ascertain the size of L after each vertex in P has joined R , we must compute this in real time. Unlike the typical linear search where the size of L after the addition of v_1 into R is directly known, this process requires traversal through all edges of v . During the traversal, we have to verify if each neighbour n of v is in the original L . If it is, n can remain in L ; otherwise, it must be removed.

Suppose we have vertices v_1 to v_p in P today. If we find that the size of L is 2 after v_1 joins R , we need to check v_2 to v_p . Suppose v_2 has many neighbours, from n_1 to n_m . If we find that both n_1 and n_2 neighbours are in the original L , it indicates that the size of L will be at least 2 after v_2 joins R . But we aim to find v that minimizes L , so we know we no longer need to examine n_3 to n_m as, even if they are not in the original L , the resulting size of L would still be the same as if v_1 had been added. This idea can be applied to the remaining vertices in P , from v_3 to v_p .

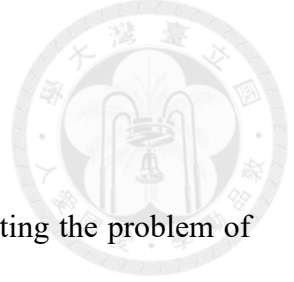
In other words, the first level early stop aims to reduce the unnecessary examination of edge lists while evaluating each v in P . While we need to check each v 's edge list, we don't necessarily need to go through the entire edge list.



5.2.2 Second Level Early Stop

As mentioned in Chapter 5.2.1, there are v_1 to v_p vertices in P for which we need to compute the size of L after they have been added to R . While the first level early stop allows us to avoid having to examine all edges in each vertex's edge list, we can save even more time in candidate selection by not having to inspect each vertex in P . Imagine that we found a vertex in the previous candidate selection that reduced the size of L to 1. When selecting the next vertex from P to add to R , if we find another vertex that also reduces the size of L to 1, we can stop inspecting the other vertices in P . That is to say, we always remember the size of L caused by the last selected vertex, and use that size as our target for the next selection. As soon as we find a vertex that matches the previous minimum, we can stop scanning the vertices in array P .

In other words, the goal of the second level early stop is to avoid going through the entire set P . Of course, we might not always find a vertex v that reduces the size of L to the same minimum as the previous selection. However, the overhead of maintaining the previous minimum is almost negligible, and subsequent experiments have shown that this modification significantly optimizes the process.

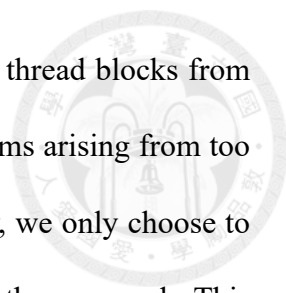


5.3 Work Stealing

While candidate selection has already done a fair job of alleviating the problem of workload imbalance, we observed from experiments that merely relying on the first level recursion tree as a unit of work distribution is still too coarse, even with the optimization of candidate selection. Hence, we propose a work-stealing mechanism that allows thread blocks that have run out of first level recursion trees to execute in the later stages of execution, to help other executing thread blocks deal with their second level recursion trees (RT_2) within their assigned first level recursion trees ($RT_1(blockID)$).

In the beginning, we allow each thread block to handle its own $RT_1(blockID)$ until any of them finds no more RT_1 to process. At this point, it issues a signal along with a `grid.sync()`, and asks all other thread blocks to record which RT_2 within their own $RT_1(blockID)$ they are currently handling. The idle thread block then starts checking from $RT_1(blockID + 1)$ all the way to $RT_1((blockID + NUM_{block} - 1) \bmod (NUM_{block}))$, to see if there are any unfinished RT_2 in the RT_1 that all other thread blocks are responsible for. If there are, it helps process them. Similarly, if other thread blocks also find themselves becoming idle during this process, they also start checking sequentially from $RT_1(blockID + 1)$. The entire algorithm only ends when all thread blocks realize they have become idle.

Such a work-stealing mechanism has several advantages. First, we chose to let idle thread blocks actively seek out other work to do, rather than passively waiting for the busy thread blocks to distribute their work, meaning that the overhead required for work reallocation is carried by the idle thread blocks, not the originally busy ones. Second, every idle thread block starts checking from its own $RT_1(blockID + 1)$, rather than uniformly



starting the check from $RT_1(0)$. This design prevents all newly idle thread blocks from repeatedly checking the same RT_1 and also avoids contention problems arising from too many thread blocks simultaneously accessing the same RT_1 . Finally, we only choose to distribute work to different thread blocks at a granularity of RT_2 at the very end. This allows us to enjoy the benefits of lower distribution overhead that comes with distributing at the granularity of RT_1 for most of the time, while still being able to benefit from the workload balance offered by distributing at a granularity of RT_2 when necessary.

5.4 Parallel Intersection Computation

As can be seen in Algorithm 3, we often need to know the value of $N[v]$ (like line 16, line 26). When calculating $N[v]$, we have two options. Let's take the for loop from line 25 to line 33 as an example.

First, we can traverse all the points in the P set. Suppose for a certain point v , we examine whether its edge list can fully encompass the current L set. If so, this means point v is connected with all points in L, and we can add this point to R. This method essentially starts from the P set, examining whether the size of $N[v]$ for all points v in P is the same as the size of L.

Second, we can alternatively start from L. We initially declare an array the same size as V. For each point's edge list in L, we perform an atomic add operation on the value of the array with the vertex number as the array index. After this process, we traverse the entire array. If someone's value is the same size as L, then we can also say that this point is connected with all points in L. The second method essentially starts from L and calculates the number of connections between each person in the V set and L based on L's edge list.

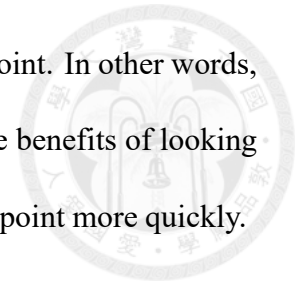
The second method is the parallel intersection computation we said.

As we closely observe the MBE algorithm, we will find that the rate at which size of L decreases is significantly greater than the rate at which size of P decreases. Moreover, size of Q may even grow larger. Hence, choosing the method of parallel intersection computation intuitively reduces the number of points we need to traverse, thereby accelerating the overall time for calculating $N[v]$. Simultaneously, since we always decide on L starting from R , we will always make V the point with fewer nodes in the bipartite graph. Under such circumstances, the degree of each node in U will usually be smaller than that of the nodes in V . In other words, when we traverse each node's edge list, the edge list of the nodes in L will also be shorter than the node in V be. Due to these two reasons, we choose the method of parallel intersection computation, and our experiments have confirmed that parallel intersection computation can effectively enhance our performance.

5.5 Two Level Parallelism

Given that we have 256 threads in a single thread block, determining how to effectively distribute tasks to each thread is crucial. As we mentioned in Chapter 5.4, we will need to traverse the edge list of each point in L . At this moment, if we have 256 threads, intuitively we have two methods of assigning threads. First, we allow each thread to look at its own points, meaning we can review 256 points at a time. The second method is to look at just one point at a time, but let 256 threads traverse the edge list of that point together, allowing us to complete each point's edge list more quickly. The downside is that if the degree of the point is less than 256, some threads will be idle. Based on our observation, the degree of each point usually does not reach 256. Therefore, we chose to

use a warp (32 threads) as a unit to traverse the edge list of a single point. In other words, we can look at 8 points simultaneously. In this way, we can enjoy the benefits of looking at multiple points at the same time and finish the edge list of a single point more quickly.





Chapter 6 Evaluation

In this chapter, we will first introduce the experimental environment and the datasets we used. Then, we will discuss the state-of-the-art (SoTA) benchmark we compare against and why we chose it.

Following that, we will examine the overall speedup in runtime after our optimizations compared to the SoTA. Additionally, we will investigate whether our proposed solutions indeed equalize the workload across different thread blocks to address the issue of workload imbalance.

Finally, we will delve deeper to see how much performance improvement each individual optimization can bring to the overall runtime.

6.1 Configurations

6.1.1 Platforms and Datasets

In terms of experimental environment, we conducted our experiments on an Nvidia RTX3090 graphics processing unit, choosing to implement on the stable version of CUDA: 11.6. As for the dataset, since we decided to compare with parMBE, we selected the same four datasets used in that paper. The property of these datasets are described in table 6.2,

Table 6.1: Experiment Environment

CPU	Intel i9 10900k
Main memory	128GB
GPU	RTX 3090
Device memory	24GB
Operating system	Ubuntu 20.04
CUDA version	11.6



Table 6.2: Experiment Environment

Dataset	#Vertices	#Edges	#Maximal Bicliques
Youtube	124325	293360	1826587
IMDB	1199919	3782463	5160061
Stack Overflow	641873	1301942	3320824
BookCrossing	445801	1149739	54458953

and the meaning of vertices and nodes the the graph are explained in following:

YouTube: This network represents the bipartite relationship between YouTube users and the groups they belong to. The nodes represent users and groups, while an edge connecting a user and a group indicates the user's membership in that group.

IMDB: This network illustrates the relationship between individuals and movies or television programs. It is a bipartite network, with individuals (such as actors and directors) represented as nodes on the left, and works (such as films and television programs) represented as nodes on the right. An edge indicates that a person was involved in a particular work.

Stack Overflow: This network depicts the bipartite relationship of the Stack Overflow favorite system. Stack Overflow is a prominent question and answer website within the Stack Exchange Network. The nodes in the network represent users and posts. An undirected and unweighted edge indicates that a user has designated a post as a favorite.

BookCrossing: This network captures data related to the reading habits of members within the BookCrossing community. The nodes in the network represent users and books,

while an edge signifies an interaction between a user and a book, indicating that the user has engaged with or interacted in some way with the book.



6.1.2 CPU Baselines

To the best of the authors’ knowledge, this paper proposes the first algorithm to solve MBE on a GPU. Therefore, we do not have other GPU MBE algorithms for comparison. As a result, we selected parMBE, which is parallelized on a CPU and is considered state-of-the-art in terms of execution time, as our benchmark. Although this comparison is not entirely fair, we hope this experiment will show that implementing on a GPU can lead to a certain degree of performance improvement.

6.2 Overall Performance

In Figure 6.1, we can see that our algorithm on the GPU achieves a certain speedup on all datasets. The maximum speedup we can reach is up to 3.6x.

6.3 Workload Balance

$$\text{Busy Ratio} = \frac{\sum_{n=0}^{\#\text{blk}-1} \text{runtime}(n)}{\text{max_runtime} \times \#\text{blk}} \quad (6.1)$$

Table 6.3: Busy Ratio

	Youtube	IMDB	Stack Overflow	BookCrossing
CS:X WS:X	0.203	0.710	0.285	0.144
CS:O WS:X	0.523	0.991	0.664	0.720
CS:X WS:O	0.608	0.982	0.633	0.413
CS:O WS:O	0.902	0.992	0.999	0.994

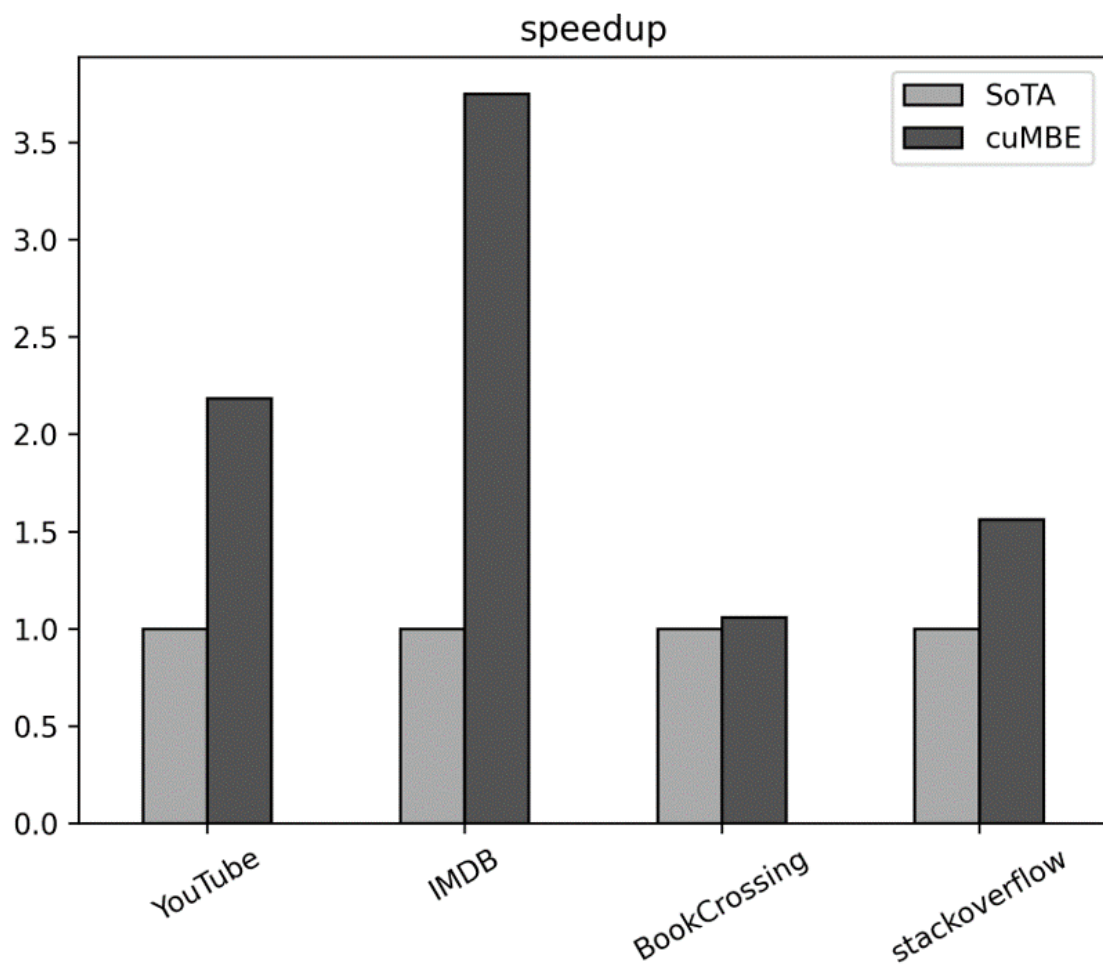


Figure 6.1: Speedup compared to parMBE

Table 6.4: Runtime with CS & WS optimization

	Youtube	IMDB	Stack Overflow	BookCrossing
CS:X WS:X	10.573	52.516	2757.73	4940.08
CS:O WS:X	3.331	31.213	489.65	713.175
CS:X WS:O	7.695	39.308	1925.35	4021.7
CS:O WS:O	2.221	31.357	349.274	689.811

In order to evaluate the effectiveness of our optimization methods for workload imbalance, we define the busy ratio as the total time spent by all thread blocks divided by the product of the maximum time spent by all thread blocks and the number of blocks. In other words, we take the maximum runtime among all thread blocks as the goal, hoping that each thread block's busy time can approach this target. That is, all of them enter the idle state at similar times, and the ideal value of the busy ratio is 100

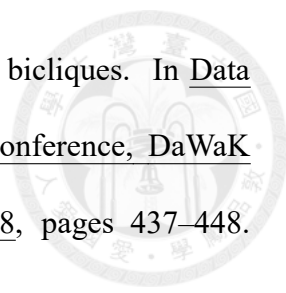
In Table 6.4, we can observe that when no optimization is applied, our busy ratio is generally less than 30%. Only the IMDB dataset shows a better performance due to its inherently balanced nature. Furthermore, we can also see that when we try to add either the Candidate Selection (CS) or Work Stealing (WS) technique, our busy ratio shows significant improvement, but it is still not enough. Finally, when both techniques are simultaneously incorporated, the busy ratio can reach more than 99%. YouTube, due to its relatively smaller graph size resulting in fewer workloads, fails to reach a 99% busy ratio. However, a 90.2% busy ratio is still considerably better than the initial 20.3%.

It is noteworthy that the busy ratio of IMDB after the addition of CS optimization is almost as good as when both CS and WS are both employed. We believe this is because IMDB is already well-balanced after adding CS, thus it cannot benefit from the advantages brought by WS. However, this indirectly shows that WS does not introduce excessive overhead.



References

- [1] A. Abidi, R. Zhou, L. Chen, and C. Liu. Pivot-based maximal biclique enumeration. In IJCAI, pages 3558–3564, 2020.
- [2] A. Adinets. Cuda dynamic parallelism api and principles, 2014. Accessed June. 3, 2023.
- [3] G. Alexe, S. Alexe, Y. Crama, S. Foldes, P. L. Hammer, and B. Simeone. Consensus algorithms for the generation of all maximal bicliques. Discrete Applied Mathematics, 145(1):11–21, 2004.
- [4] M. Almasri, Y.-H. Chang, I. E. Hajj, R. Nagi, J. Xiong, and W.-m. Hwu. Parallelizing maximal clique enumeration on gpus. arXiv preprint arXiv:2212.01473, 2022.
- [5] L. Chen, C. Liu, R. Zhou, J. Xu, and J. Li. Efficient maximal biclique enumeration for large sparse bipartite graphs. Proceedings of the VLDB Endowment, 15(8):1559–1571, 2022.
- [6] A. Das, S.-V. Sanei-Mehri, and S. Tirthapura. Shared-memory parallel maximal clique enumeration. In 2018 IEEE 25th International Conference on High Performance Computing (HiPC), pages 62–71. IEEE, 2018.
- [7] M. Harris. Using shared memory in cuda c/c++, 2013. Accessed June. 1, 2023.

- 
- [8] G. Liu, K. Sim, and J. Li. Efficient mining of large maximal bicliques. In Data Warehousing and Knowledge Discovery: 8th International Conference, DaWaK 2006, Krakow, Poland, September 4-8, 2006. Proceedings 8, pages 437–448. Springer, 2006.
- [9] A. P. Mukherjee and S. Tirthapura. Enumerating maximal bicliques from a large graph using mapreduce. IEEE Transactions on Services Computing, 10(5):771–784, 2016.
- [10] R. Nataraj and S. Selvan. Parallel mining of large maximal bicliques using order preserving generators. 2009.
- [11] Nvidia. <https://docs.nvidia.com/cuda/archive/9.0/>, 2018. Accessed June. 1, 2023.
- [12] Nvidia. Cuda toolkit, 2023. Accessed June. 3, 2023.
- [13] R. Peeters. The maximum edge biclique problem is np-complete. Discrete Applied Mathematics, 131(3):651–654, 2003.
- [14] E. Prisner. Bicliques in graphs i: Bounds on their number. Combinatorica, 20(1):109–117, 2000.
- [15] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das. Controlled kernel launch for dynamic parallelism in gpus. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 649–660. IEEE, 2017.
- [16] Y. Zhang, C. A. Phillips, G. L. Rogers, E. J. Baker, E. J. Chesler, and M. A. Langston. On finding bicliques in bipartite graphs: a novel algorithm and its application to the integration of diverse biological data types. BMC bioinformatics, 15:1–18, 2014.