université
PARIS-SACLAY

NNT : 2017SACLV069

UNIVERSITÉ DE VERSAILLES
ST-QUENTIN-EN-YVELINES
université PARIS-SACLAY

Parallel solutions for large-scale eigenvalue problems arising in graph analytics

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université de Versailles Saint-Quentin-en-Yvelines

École doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Saclay, le 13 décembre 2017, par

Alexandre Fender

Composition du Jury :

M. Jean-Marc Delosme Professeur à l'Université d'Evry-Val d'Essonne	Président
M. Michel Daydé Professeur à l'ENSEEIHT	Rapporteur
Mme. Katherine Yelick Professeur à l'University of California Berkeley, USA	Rapporteur
M. Joe Eaton Technical lead à Nvidia, USA	Examinateur
M. Jean-Luc Gaudiot Professeur à l'University of California Irvine, USA	Examinateur
Mme. Nahid Emad Professeur à l'Université de Versailles	Directrice de thèse
M. Serge Petiton Professeur à l'Université de Lille	Co-Directeur de thèse

Thèse de doctorat

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Solutions parallèles pour les grands problèmes de valeurs propres issus de l'analyse de graphe

Mots clés : méthodes numériques, parallélisme, science de données, GPU, PageRank, groupement spectral

Résumé : Les graphes, ou réseaux, sont des structures mathématiques représentant des relations entre des éléments. Ces systèmes peuvent être analysés dans le but d'extraire des informations sur la structure globale ou sur des composants individuels. L'analyse de graphe conduit souvent à des problèmes hautement complexes à résoudre. À grande échelle, le coût de calcul de la solution exacte est prohibitif. Heureusement, il est possible d'utiliser des méthodes d'approximations itératives pour parvenir à des estimations précises. Les méthodes historiques adaptées à un petit nombre de variables ne conviennent pas aux matrices creuses de grande taille provenant des graphes. Par conséquent, la conception de solveurs fiables, évolutifs, et efficaces demeure un problème essentiel.
L'émergence d'architectures parallèles telles que le GPU ouvre également de nouvelles perspectives avec des progrès concernant à la fois la puissance de calcul et l'efficacité énergétique.
Nos travaux ciblent la résolution de problèmes de valeurs propres de grande taille provenant des méthodes d'analyse de graphe dans le but d'utiliser efficacement les architectures parallèles. Nous présentons le domaine de l'analyse spectrale de grands réseaux puis proposons de nouveaux algorithmes et implémentations parallèles.
Les résultats expérimentaux indiquent des améliorations conséquentes dans des applications réelles comme la détection de communautés et les indicateurs de popularité.

Title : Parallel solutions for large-scale eigenvalue problems arising in graph analytics

Keywords : numerical methods, parallelism, data science, GPU, PageRank, spectral clustering

Abstract : Graphs, or networks, are mathematical structures to represent relations between elements. These systems can be analyzed to extract information upon the comprehensive structure or the nature of individual components. The analysis of networks often results in problems of high complexity. At large scale, the exact solution is prohibitively expensive to compute. Fortunately, this is an area where iterative approximation methods can be employed to find accurate estimations. Historical methods suitable for a small number of variables could not scale to large and sparse matrices arising in graph applications. Therefore, the design of scalable and efficient solvers remains an essential problem.
Simultaneously, the emergence of parallel architecture such as GPU revealed remarkable ameliorations regarding performances and power efficiency.

In this dissertation, we focus on solving large eigenvalue problems arising in network analytics with the goal of efficiently utilizing parallel architectures. We revisit the spectral graph analysis theory and propose novel parallel algorithms and implementations.
Experimental results indicate improvements on real and large applications in the context of ranking and clustering problems.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Acknowledgements

C'est en français que j'ai le plaisir d'écrire ces quelques lignes pour remercier celles et ceux qui ont le plus compté au cours de mon doctorat. Pour moi la thèse fut une expérience exceptionnelle autant d'un point de vue professionnel que personnel et c'est en grande partie lié à toutes les rencontres et les échanges qu'elle a entrainés.

C'est pourquoi je souhaite tout d'abord remercier ma directrice de thèse pour ces trois années passionnantes. Je remercie Nahid Emad pour sa disponibilité, son enthousiasme et ses précieux conseils qui ont accompagné l'ensemble de mes recherches. Je tiens également à remercier Serge Petiton, mon Co-Directeur de thèse, pour ses encouragements et ses recommandations. Merci à tous les deux pour cette aventure particulièrement enrichissante qui n'aurait pas été possible sans vous. Je n'oublierai pas tous ces moments passés à discuter de l'avenir dans lequel je saurai mieux avancer grâce à vous.

Je reremercie chaleureusement Michel Daydé, Professeur à l'ENSEEIHT, et Katherine Yelick Professeur à l'University of California Berkeley, qui m'ont fait l'honneur d'être les rapporteurs de ma thèse. Merci infiniment pour vos recommandations et votre aide.

J'exprime mes remerciements à Jean-Marc Delosme, Professeur à l'Université d'Evry-Val d'Essonne, pour avoir accepté d'être président du jury. Vos questions et vos remarques ont contribué à enrichir mon travail. Je remercie vivement Joe Eaton, Technical lead à Nvidia, pour être présent à mon jury et pour avoir soutenu ce projet dès le début en faisant le lien entre la partie théorique et applicative de mes recherches. Je reremercie Jean-Luc Gaudiot, Professeur à l'University of California Irvine, pour

avoir accepté de faire partie du jury et pour vos conseils.

Ce travail n'aurait pu être mené à bien sans Nvidia, qui m'a permis, grâce à son soutien matériel, de me consacrer sereinement à l'élaboration de ma thèse. Je remercie particulièrement Maxim Naumov, pour tous nos échanges sur mes recherches. Je tiens aussi à remercier mes collègues du bureau de Paris pour tous ces moments riches aussi bien professionnellement que personnellement.

Je suis reconnaissant de l'accueil chaleureux que m'a témoigné la Maison de la Simulation. Je remercie l'ensemble des personnes que j'ai pu croiser durant ces trois ans pour l'agréable atmosphère qu'ils ont su installer. En particulier, je remercie Édouard Audit, Martial Mancip et Valérie Belle qui ont rendu possible ma soutenance à la Maison de la Simulation. Je n'oublie pas les membres du laboratoire Li-Parad, merci de m'avoir fait découvrir le monde de la recherche et de m'avoir encouragé à y participer au cours de ces huit années à l'Université de Versailles.

Je tiens finalement à remercier celles et ceux qui me sont chers. Leurs attentions et encouragements m'ont accompagné tout au long de ce parcours. Je remercie ma mère, pour son soutien et sa confiance infaillible. Merci à Many, Luc, Jeannette et Patrick d'avoir été présent à chaque instant.

Au terme de ce parcours, je remercie Océane pour avoir su m'accompagner de la meilleure des façons et pour continuer à m'inspirer quotidiennement.

Table of contents

List of figures ix

List of tables xiii

Nomenclature xv

1 Introduction 1

1.1 Motivations . . . 1

1.1.1 Graph analytics . . . 2

1.1.2 Eigenvalue problems . . . 2

1.1.3 Hardware acceleration . . . 3

1.2 Structure of this thesis and core contributions . . . 4

2 Accelerated spectral graph analytics 7

2.1 From networks to linear algebra . . . 7

2.1.1 Structure of graphs and mathematical representation . . . 8

2.1.2 Compressed matrix representation for computers . . . 11

2.1.3 Graph algorithms in linear algebra . . . 14

2.2 Eigenvalue problems arising in graphs . . . 16

2.2.1 PageRank . . . 16

2.2.2 Clustering . . . 18

2.3 GPU accelerators . . . 21

2.3.1 Architecture and programming . . . 21

2.3.2 Eigenvalue methods for networks and accelerators . . . 24

2.3.3 Accelerated basic operations . . . 32

3 Accelerated multiple implicitly restarted Arnoldi method with nested subspaces 39
3.1 Introduction 39
3.2 Implicitly restarted Arnoldi solver with PageRank applications on GPU 40
3.2.1 Hybrid GPU approach 42
3.3 Accelerated multiple IRAM with nested subspaces 45
3.3.1 Enabling nested subspaces in the hybrid IRAM algorithm 45
3.3.2 Synchronous auto-tuning 46
3.3.3 Implementation 47
3.3.4 Distributed considerations 49
3.4 Experimental results 50
3.4.1 Power method on GPUs 50
3.4.2 Implicitly restarted Arnoldi for networks on GPU 53
3.4.3 Accelerated multiple IRAM with nested subspaces 56
3.5 Conclusion and future works 60

4 Spectral modularity clustering 61
4.1 Introduction 61
4.1.1 Modularity 63
4.2 Spectral modularity maximization 67
4.2.1 Algorithm 67
4.2.2 Eigenvalue Problem 68
4.2.3 Clustering Problem 68
4.2.4 Parallelism and Energy Efficiency 70
4.3 Numerical Experiments 71
4.3.1 Context 71
4.3.2 Clustering and Effects of Precision 72
4.3.3 Adaptive Clustering 75
4.3.4 Related Work 78
4.3.5 Modularity and Spectral Clustering 80
4.4 Conclusion and Future Work 83

5 Jaccard and PageRank weights in spectral clustering 85
5.1 Introduction 85

5.2 Jaccard Weights . . . 87
5.2.1 Jaccard and Related Coefficients . . . 87
5.2.2 Jaccard and Related Edge Weights . . . 88
5.3 Implementation . . . 91
5.3.1 Parallel Algorithm . . . 91
5.3.2 PageRank and Vertex Weights . . . 94
5.4 Graph Clustering . . . 96
5.4.1 Jaccard Spectral Clustering . . . 96
5.4.2 Tversky Spectral Clustering . . . 97
5.4.3 Profiling . . . 98
5.5 Numerical Experiments . . . 99
5.5.1 Multi-level Schemes (CPU) . . . 100
5.5.2 Spectral Schemes (GPU) . . . 101
5.5.3 Quality Across Many Samples . . . 102
5.6 Conclusion and Future Work . . . 104

6 Multiple implicitly restarted Lanczos with nested subspaces **105**
6.1 Introduction . . . 105
6.1.1 Spectral graph analysis and clustering . . . 106
6.2 Multiple implicitly restarted Lanczos with nested subspaces . . . 107
6.2.1 Proposed approach . . . 107
6.2.2 Hybrid acceleration . . . 108
6.2.3 Profile . . . 111
6.3 Experiments . . . 112
6.3.1 Modularity . . . 113
6.3.2 Minimum balanced cut . . . 118
6.3.3 Different architectures . . . 120
6.3.4 Variation of the Krylov subspace size . . . 121
6.4 Conclusion . . . 123

7 Conclusions and perspectives **125**

Communications **129**

References **131**

Appendix A Resumé en Français **139**

A.1 Motivations 139

A.1.1 Analyse de graphe 140

A.1.2 Problèmes de valeurs propres 140

A.1.3 Accélération matérielle 141

A.2 Structure de la thèse et contributions 142

List of figures

2.1 Directed graph with 6 vertices and 10 edges 7
2.2 Amazon book co-purchasing data set with PageRank vertex weights, Jaccard edge weights and spectral cluster assignments in two groups . . 8
2.3 Power law distribution 10
2.4 Adjacency matrix representation of Figure 2.1 11
2.5 Multidimensional CSR representation 14
2.6 PageRank visualization, α =0.9 16
2.7 The Google matrix of Figure 2.1 18
2.8 Hierarchical clustering coupled with spectral clustering 20
2.9 Diagram of Nivida GP100 with 60 SM and 3840 cores 21
2.10 Streaming multi-processor (SM) of Nvidia GP100 22
2.11 Parallel tree-based warp reduction 34
2.12 Parallel SpMV - CSR scalar 35
2.13 Parallel SpMV - CSR vectorized 35
2.14 Parallel SpMV - CSR merge-path 36
2.15 CSRMV-MP vs CUSPARSE on the entire matrix collection of the University of Florida (Merrill and Garland, 2016). 38
2.16 CSRMV-MP vs MKL on the entire matrix collection of the University of Florida (Merrill and Garland, 2016). 38

3.1 Network representing 105 books sold by Amazon connected by 441 frequent co-purchasing by the same buyers. 41
3.2 Graph of Figure 3.1 where color and diameter of vertices vary according to the PageRank. 41
3.3 Accelerated hybrid approach of IRAM and MIRAMns 45

3.4 Speedup of the power method implementation on GPU vs. CPU (parallel) for PageRank applications . . . 51
3.5 Speedup of the power method on several GPU for PageRank applications 52
3.6 Speedup of IRAM on GPU vs. power method on GPU with different damping factors . . . 54
3.7 Impact on time and memory when changing the Krylov subspace size on com-Orkut . . . 55
3.8 Profiling of IRAM on GPU . . . 55
3.9 Speedup and cycles saved in MIRAMns vs. IRAM. . . . 57
3.10 Comparison of the residuals of MIRAMns and IRAM on Cage15. . . . 58
3.11 Selected subspace size in MIRAMns vs. IRAM on af23560 . . . 59
3.12 Variation of the subspace frequency in MIRAMns on cage14 . . . 59

4.1 Same graph as in Figure 3.2 where vertices are coloured according to their ground truth cluster. . . . 62
4.2 Same graph as in Figure 3.2 where vertices are coloured according to their cluster found by our spectral modularity maximization. . . . 62
4.3 Profiling of the modularity algorithm . . . 68
4.4 The time achieved for 64 and 32 bit precision, when splitting the graph into 7 clusters . . . 72
4.5 The number of iterations achieved for 64 and 32 bit precision, when splitting the graph into 7 clusters . . . 73
4.6 The modularity score achieved for 64 and 32 bit precision, when splitting the graph into 7 clusters . . . 74
4.7 Comparing the impact of varying the number of clusters used for assignment for different number of computed eigenvectors . . . 76
4.8 The modularity score achieved when changing the number of clusters for citationCiteseer network in 64 bit precision . . . 77
4.9 The speedup and relative quality when compared to the reference results for large data sets on GPU in (Auer, 2013) . . . 78
4.10 Zachary Karate Club network where vertices are coloured according to their faction. . . . 79
4.11 The speedup and relative quality when compared to the reference results for large data sets on CPU in (Lasalle and Karypis, 2015) . . . 81

4.12 The modularity score obtained on large cases by using assignment to clusters generated by modularity and minimum balanced cut algorithms for 7 clusters in 64 bit precision 82

5.1 Amazon book co-purchasing original graph 86
5.2 Amazon book co-purchasing graph with Jaccard 86
5.3 Graph example, G = (V, E) 89
5.4 Speedup of the GPU implementation vs. 1 and 12 CPU threads when computing Jaccard Weights 94
5.5 Speedup when computing PageRank 95
5.6 Profile of spectral clustering with PageRank vertex and Jaccard edge weights 99
5.7 Improvement in the quality of partitioning obtained by METIS, with Jaccard and Jaccard-PageRank for `coPapersCitseer` graph 101
5.8 Improvement in the quality of partitioning obtained by nvGRAPH, with Jaccard and Jaccard-PageRank for `coPaperDBLP` graph 102
5.9 Improvement in the quality of partitioning obtained by nvGRAPH and METIS, with Jaccard and Jaccard-PageRank weights 103

6.1 Overview of the accelerated MIRLns solver. Green is associated to the device and blue corresponds to the host 109
6.2 Profiling of the implicitly restarted Lanczos eigensolver 111
6.3 The number of iterations achieved for 64 and 32 bit precision for spectral modularity maximization 114
6.4 The modularity score achieved for 64 and 32-bit precision for spectral modularity maximization (Titan X). 116
6.5 The speedup of MIRLns over IRL on Tesla P100 for 64 and 32-bit precision in the context of spectral modularity maximization. 117
6.6 The number of iterations achieved for 64 and 32-bit precision for spectral balanced cut minimization (Titan X). 118
6.7 The ratio edge cut score achieved for 64 and 32-bit precision for spectral balanced cut minimization (Titan X). 119
6.8 The number of iterations achieved for 64 and 32-bit precision for spectral modularity maximization on k6000 and Titan X hardware. 121

6.9 The convergence of IRL and MIRLns on hollywood-2009 with an eye on the selected subspace size $m_{max} = 22$ (Tesla K20c). 122

List of tables

2.1 COO format 12
2.2 CSR format 12
2.3 CSC format 13
2.4 Ellpack format 13

3.1 General information on networks 51
3.2 IRAM solver on GPU vs. power method on GPU in PageRank applications ($\alpha = 0.85$, $m = 4$, 64 bit precision) 53
3.3 General information on matrices 56

4.1 General information on networks 71
4.2 The modularity (Mod), time (T) and # of iterations (It) achieved for 64 and 32 bit precision, when splitting the graph into 7 clusters 75
4.3 The modularity (Mod), best number of clusters (Clu) according to modularity, time (T) and number of iterations (It) achieved in 64 bit precision 77
4.4 The modularity for a given # of clusters when compared to the reference results for large data sets in (Auer, 2013) 80
4.5 The modularity for a given # of clusters when compared to the reference results for small data sets in (Auer, 2013) 80
4.6 The modularity score (Mod) and time (T) obtained by our spectral modularity maximization (GPU) and reference results from (Lasalle and Karypis, 2015) on hierarchical modularity maximization (CPU) . . 81
4.7 The modularity score (Mod) and time (T) obtained by using the assignment to partitions generated by modularity and minimum balanced cut algorithm for 7 clusters in 64 bit precision 82

5.1 Time(ms) needed to compute Jaccard weights 94

5.2 Time(ms) needed to compute PageRank 96
5.3 General information on networks 99
5.4 Improvement in the quality of partitioning obtained by nvGRAPH (Spect) and METIS (M-L), with Jaccard (J) and Jaccard-PageRank (J+P) weights 103

6.1 The number of iterations (It) and modularity score (Mod) achieved for 64 and 32-bit precision for spectral modularity maximization (Titan X). 115
6.2 The time (T) in millisecond, modularity score (Mod) and speedup (SU) achieved for 32 and 64-bit precision for spectral modularity maximization on Tesla P100. 117
6.3 The number of iterations (It) and ratio edge cut score (ECR) achieved for 64 and 32-bit precision for balanced cut minimization. 119

Nomenclature

Acronyms / Abbreviations

API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CCDF	Complementary cumulative distribution function
COO	Coordinate matrix format
CPU	Central Processing Unit
CREW	Concurrent Read Exclusive Write
CSC	Compressed Sparse Column matrix format
CSR	Compressed Sparse Row matrix format
CSRMV	Compressed Sparse Row Matrix Vector multiplication
CUDA	Compute Unified Device Architecture
ECC	Error Correcting Code
ERAM	Explicitly Restarted Arnoldi Method
FLOPS	FLoating point Operations Per Second
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
HPC	High Performance Computing

IRAM	Implicitly Restarted Arnoldi Method
IRL	Implicitly Restarted Lanczos Method
MERAM	Multiple Explicitly Restarted Arnoldi Method
MIRAMns	Multiple Implicitly Restarted Arnoldi Method with nested subspaces
MIRLns	Multiple Implicitly Restarted Lanczos method with nested subspaces
MKL	Maths Kernel Library
NNZ	Number of non-zeroes
NP	Non-deterministic Polynomial-time problem
PCI	Peripheral Component Interconnect
PRAM	Parallel Random-Access Machine
QR	QR factorization or the QR method
RAM	Random-Access Memory
RMAT	Recursive MATrix graph generator
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor
SPMV	SParse Matrix Vector multiplication
SSSP	Single Source Shortest Path algorithm
SSWP	Single Source Widest Path algorithm
STL	Standard Template Library
TDP	Thermal Design Power
TEPS	Traversed Edges Per Second
UVM	Unified Virtual Memory

Chapter 1

Introduction

1.1 Motivations

For the past 30 years, the world's technological capacity to store information has roughly doubled every 3 years, resulting in an immense amount of heterogeneous data (Hilbert and Lopez, 2011). Moreover, the recent shift in human communication coupled with technical progress has triggered an extensive growth of data volume. Yet, it is estimated that only 1% of the information in the digital universe is currently analysed (Gantz and Reinsel, 2012).
Simultaneously, the CPU clock speed has stopped increasing causing the democratization of parallel architectures and the need for and a new generation of software (Sutter, 2005).
Since then, high performance computing (HPC) and parallel techniques for solving complex computational problems have reached unprecedented levels. Most powerful machines (Dongarra et al., 2017) now have over a million cores and approach the exaFLOPS corresponding to a billion-billion calculations per second.
We are now at the edge of a new era of science where data and hardware open new perspectives transforming societies through the next level of analytics and artificial intelligence.

1.1.1 Graph analytics

Many recent applications model information as relations between abstract entities. An intuitive structure for this model is a graph, also called a network. Each node can represent a person, place, object and each relationship represents how two nodes are associated. This allows the modeling of all kinds of data from social networks to roads, to neural networks, to the Internet, to populations or anything else defined by relationships (Newman, 2010).
Graph analytics is the science of analysing the comprehensive structure or the nature of individual components in a network. It finds key information such as communities, important entities, and paths in the graph.
These graph problems can be defined in terms of linear operations over arrays of data (e.g. vectors and matrices). Most elements of the system are actually zero because an element is generally connected to a fraction of all network entities. These problems are known as sparse linear algebra problems. Effective sparse methods and algorithms often balance storage, computational cost, and stability (Kepner, 2011). Most advanced structural graph analysis problems are categorized as non-deterministic polynomial-time hard problem (NP-hard). The NP-Hard theoretical complexity class contains problems that are especially difficult to solve by nature. Thus, the formal problem definition is often reduced to another simpler problem which can be solved in reasonable time. Our work targets two important graph analytics topics which are ranking (Page et al., 1998) and clustering (Schaeffer, 2007). The former being the problem of finding the importance of each node in a graph and the latter being the problem of finding similar subsets.

1.1.2 Eigenvalue problems

In linear algebra, an eigenvector v of a linear transformation is a vector whose direction does not change when that linear transformation is applied to it. The linear transformation can be represented as a square matrix A so this condition can be written as the equation $Av = \lambda v$ where λ is a scalar known as the eigenvalue, associated with the eigenvector v.
Many applications in the fields of health, agriculture, advertising, electromagnetic, energy, optimal control, finance lead to eigenvalue problems of very large size. Each problem has its own specificity which opens vast possibilities to propose new scientific

high-performance methods.
The eigenpairs of networks contain key information which can be leveraged from several different perspectives such as ranking and clustering (Chung, 1997). Graphs correspond to large and sparse matrices and Krylov methods are therefore indicated because they can quickly find accurate approximations of the eigenpairs by reducing very large problems to small ones (Bai et al., 2000). Furthermore, Krylov eigenvalue methods have been recognized for their scalability (Maschhoff and Sorensen, 1996). In these solvers, many parameters have a direct impact on efficiency, resilience, and energy consumption. Best parameters differ for one case to another and, in general, many optimal parameters remain unknown in advance. To overcome this issue, the idea of automatically adapting parameters at run-time has shown promising results (Shahzadeh Fazeli et al., 2015). In this thesis, we focus on adaptive strategies to improve the implicitly restarted Arnoldi and Lanczos methods (Sorensen, 1997) for network analytics.

1.1.3 Hardware acceleration

The graphics processing unit (GPU) is a hardware acceleration circuit which is now one of the most accessible high-performance computational platforms. Intended initially for performing graphics related computations for applications such as computer games and visualization, GPUs have evolved into general-purpose and economic parallel processing units (Owens et al., 2007). GPUs have a significantly higher degree of parallelism than multi-core CPUs but GPU cores are simpler. In comparison to CPUs, GPUs dedicate more transistors to arithmetic logic units and fewer to caches and control flow. GPUs are ideal for parallel problems because they have a high computational throughput potential. GPUs are also energy efficient since most cores are dedicated to actual computation.
For many applications, including sparse Krylov eigensolvers, the memory bandwidth is the performance bottleneck. In GPUs, memory takes a special role as it is designed to access and modify as much memory as possible as quickly as possible. This design directly benefits to graph analytics and sparse linear algebra where algorithms are often bounded by memory accesses. In this thesis, we propose to study the spectral

graph analytics for the GPU architecture.

1.2 Structure of this thesis and core contributions

This thesis connects sparse numerical linear algebra to data science through spectral graph analytics. We study two important problems on networks from the spectral perspective which are ranking and clustering. We propose novel techniques for solving large eigenvalue problems arising in graph analytics with the goal of efficiently utilizing current and next generations of parallel architectures. For each proposed solution, we present a GPU implementation and experiments on large data sets. Our results improve the performances of linear eigenvalue solvers and graph analytics.

This thesis is built upon decades of scientific progress. First, we revisit the spectral graph analytics theory and identify how graphs connect to sparse eigenvalue problems on GPUs. Chapter 2 is dedicated to a presentation of the research topic and the context.

In Chapter 3, we revisit the spectral graph analytics theory and propose to leverage accelerators in the implicitly restarted Arnoldi method with nested subspaces (MIRAMns). We present a fast parallel solver to compute the dominant eigenpairs of directed networks such as Markov chains. We explain the first implementation on accelerator and the optimizations for graphs. Experiments in the context of PageRank applications showed faster convergence compared to the traditional power iteration method.

In Chapter 4, we develop a parallel approach for computing the modularity clustering often used to identify and analyse communities in social networks. We show that modularity can be approximated by looking at the largest eigenpairs of the weighted graph adjacency matrix that has been perturbed by a rank one update. We generalize this formulation to identify multiple clusters at once and propose a way to detect the number of natural clusters. We develop a fast parallel implementation for it

that takes advantage of the Lanczos eigenvalue solver and k-means algorithm on GPUs.

In Chapter 5, we define Jaccard edge weights on a graph and generalize them to account for vertex weights, such as the PageRank. We use these weights to minimize the sum of ratios of the intersection and union of nodes on the boundary of clusters. We construct a Laplacian matrix and show how finding a minimum balanced cut for it can be formulated as an eigenvalue problem. Also, we develop a fast parallel implementation on GPUs. Finally, we compare the quality of the obtained clustering on large networks.

In Chapter 6, we present a novel method for solving large and sparse symmetric eigenvalue problems based on the implicitly restarted Lanczos method coupled with subspace size auto-tuning (MIRLns), which is inspired by MIRAMns for the Arnoldi method. Our implementation combines CPU and GPU strengths to compute the invariant subspace of real scale-free undirected networks. Experiments indicate convergence improvements on real graphs with million entities in the context of clustering problems.

In Chapter 7, we summarize the key results obtained in this thesis and present our concluding remarks. Finally, we suggest some possible paths to future reflections.

Chapter 2

Accelerated spectral graph analytics

2.1 From networks to linear algebra

A graph is a mathematical structure to represent relations between elements, it can be visualized as a set of point (vertices) connected together by lines (edges). Mathematically, a graph is defined by its vertex V and edge E sets :

$$G = (V, E) \tag{2.1}$$

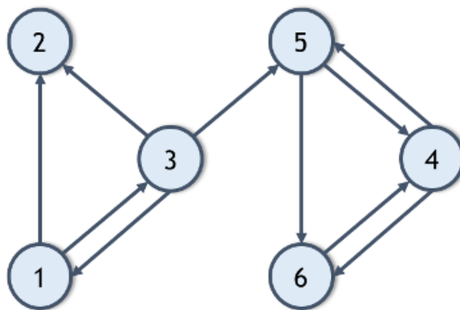


Fig. 2.1 Directed graph with 6 vertices and 10 edges

The vertex set $V = \{1, \ldots, n\}$ represents n nodes in a graph, with each node identified by a unique identifier $i \in V$. The edges may be directed (one-way) to indicate that there is a connection from a vertex i to a vertex j or undirected (two-way) to represent the reciprocity in the connection. A graph is weighted if a number (weight) is assigned to each edge. Weights often represent probabilities, distances, capacities, cost or any

other concept with numerical value. For instance, the edge set $E = \{w_{i_1,j_1}, ..., w_{i_m,j_m}\}$ represents m weighted edges in a graph, with each edge identified by $w_{i,j} \in E$. Figure 2.1 shows a simple directed graph with 6 vertices and 10 edges.

2.1.1 Structure of graphs and mathematical representation

Figure 2.2 is an example of how graph representation helps interpreting data. It shows the network of Amazon frequent book co-purchasing (Bader et al., 2013; Bastian et al., 2009). There are 105 books about US politics connected by 441 frequent co-purchasing by the same buyers. The vertex diameter varies according to the PageRank as presented in Section 3.1. The vertex color indicates the assignment of nodes into clusters, it is done with spectral clustering (see Section 2.2.2 and Chapter 4). Those clusters correspond to liberal and conservative topics. The edge thickness is based on Jaccard and PageRank weights as presented in Section 5.2.1.
Notice that the combination of different structural spectral analysis leads to a visually intuitive discovery of clusters and makes frequently co-purchased books stand out while giving indications on their strength of connections to other similar books.

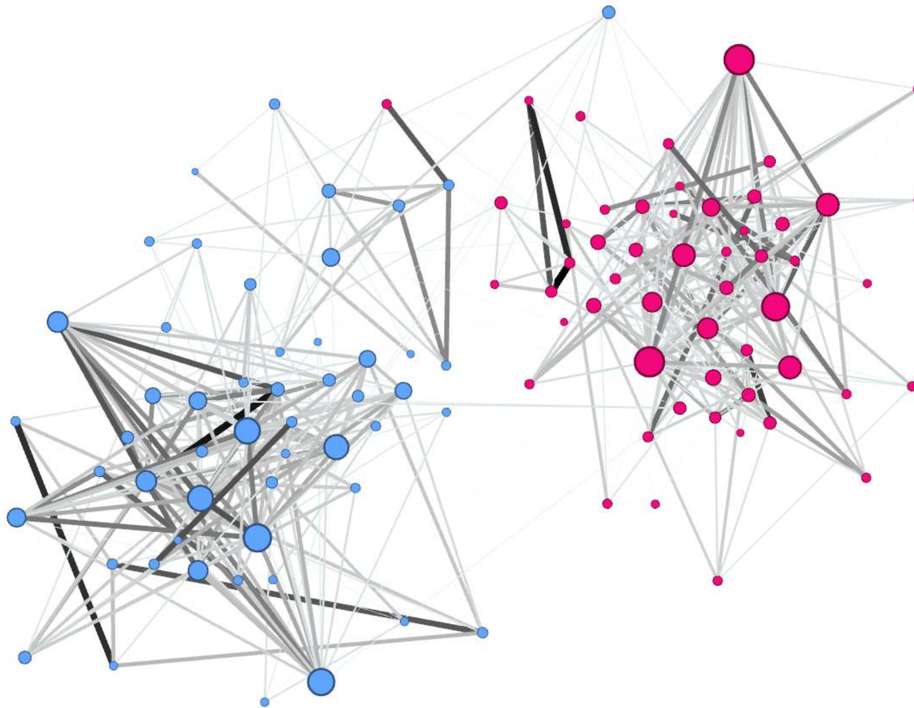
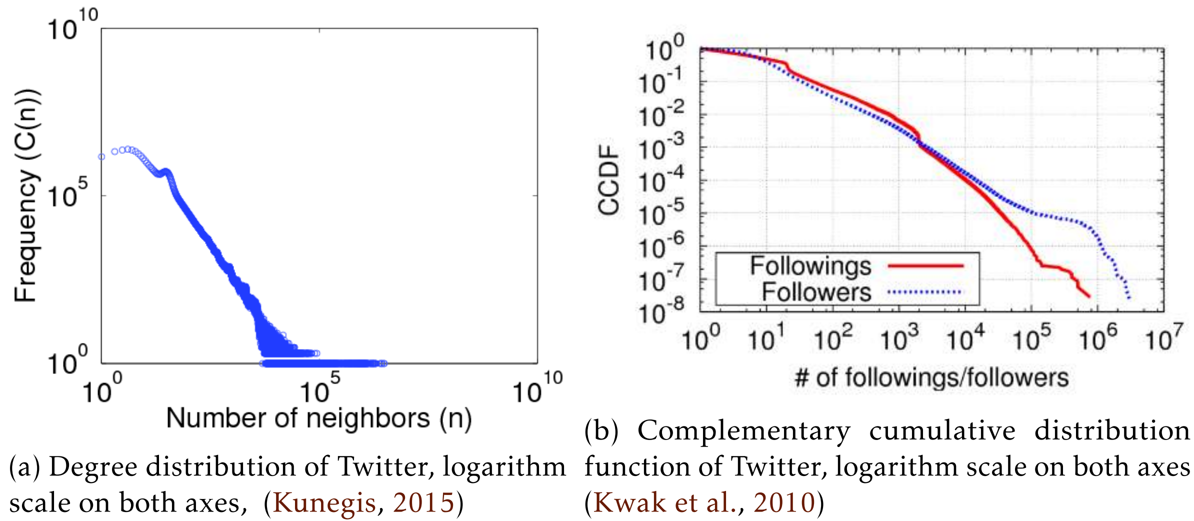


Fig. 2.2 Amazon book co-purchasing data set with PageRank vertex weights, Jaccard edge weights and spectral cluster assignments in two groups

All information is computed based on the graph structure only. Hence, no meta-data or labels were used. Everything was computed with parallel algorithms and implementations developed in this thesis. The visualization tool is Gephi (Bastian et al., 2009), an opensource software for exploring networks.

At large scale, analyzing the graph structure become a complex and expensive operation. Let us now take the follower network from Twitter as example of this problematic (Kwak et al., 2010). Twitter is a directed network as following a user does not imply any reciprocity. Still, 22.1% of connections are reciprocal. Twitter contains 1.4 billion social relations between 41 million users. The compressed adjacency list represents 12GB in memory in 32 bits. Thus, the first obstacle is the size.
One of the most important trait of the structure of a network is how edges connect vertices, which is known as the vertex degree distribution as illustrated in Figure 2.3a. Most web and social networks have a power law distribution (exponent between 2 and 3) which means that the majority of the vertices have a few connections compared to the size of the graph, but some are super-connected. This can be inspected visually by plotting the degree distribution on a logarithmic scale, on which a power law renders as a straight line as shown on Figure 2.3a. Twitter fits to a power-law distribution with the exponent of 2.276 till 10^5, beyond that users have more followers than predicted. For this data set, the maximum degree is $3,081,112$ and 40 users have more than $1,000,000$ followers whereas the average degree is 35 with an average deviation of 354. The power law distribution generates technical issue that need to be considered for the compression format of the sparse matrix and for parallel load balancing.
Another way to detect the power law is the complementary cumulative distribution function (CCDF) as shown in Figure 2.3b. The CCDF corresponds to the probability that the degree of a node picked at random is larger than x in function of x. Notice the glitch at 2.10^2 on the x-axis which is due to the upper limit on the number of persons a user could follow before 2009. Another glitch occurs around 20, this is because twitter suggests 20 people to follow in a single click to newcomers. Many other distribution indicators exist such as the Geni Coefficient which is a measure of inequality from economics and the edge distribution entropy. It is fundamental to be aware of these measures when designing algorithms for specific applications or drawing conclusions from experimental results.

One important property is the existence of short paths between any randomly-chosen pair of vertices which is known as the small-world phenomenon (Watts and Strogatz, 1998). For Twitter, 70.5% people could connect in less than 4 hops and 97.6% in less than 6 hops. An empirical study of networks for several real applications can be found in (Newman, 2010).



(a) Degree distribution of Twitter, logarithm scale on both axes, (Kunegis, 2015)

(b) Complementary cumulative distribution function of Twitter, logarithm scale on both axes (Kwak et al., 2010)

Fig. 2.3 Power law distribution

In order to address graph analytics in linear algebra the graph is represented by its adjacency matrix. The graph adjacency matrix is defined by :

$$a_{i,j} = \begin{cases} w_{i,j} & \in E \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

An example of adjacency matrix is given in Figure 2.4 for the graph presented in Figure 2.1. In Figure 2.4a each edge weight is assumed to be one for all the graph. As shown in Figure 2.4b, it is also possible to generate values such as the equiprobability of following an edge from each vertex (eg. Markov chain). On real networks the adjacency matrix is usually very sparse (ie. with many 0 entries) because each individual is only connected to number of other individuals that is small compared to the size of the network. The size of these matrices can be over a billion rows (vertices) with a dozen of non-zero elements per row in average.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

(a) Adjacency matrix, $weight_{i,j} = 1$

$$\begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

(b) $weight_{i,j} = 1/out_degree_i$

Fig. 2.4 Adjacency matrix representation of Figure 2.1

2.1.2 Compressed matrix representation for computers

An n by n matrix can be stored in a compressed format by storing only non-zeroes values. If the matrix is sparse, the memory and computational improvements are huge. In this section we present four famous existing compression formats. The goal is to measure pros and cons by considering variables such as performances for accelerators, flexibility of the format, and compression rate for real networks.

COO and Edge List

The most natural way to represent a network is probably the edge list. In a directed graph, the notation (v_i, v_j) represents a link from v_i to v_j, thus, the complete graph consists in a simple list of couple (v_x, v_y). Weighted graphs can be represented using a list of triples (v_i, v_j, w_k) where w is the weight from v_i to v_j. For a graph, the COO compression of its adjacency matrix corresponds exactly to the edge list.

In the Compressed coordinate format (COO) each entry has its value and its absolute coordinate in the matrix. For instance, Table 2.1 corresponds to the compressed version of the adjacency matrix of Figure 2.4b.

The COO format is probably the most flexible in the sense that it allows independent insertion and deletion, and can be partitioned without effort. We also notice the fast transpose operation which consists in swapping the row and column indices. These advantages come with a direct cost in memory and more indirect one in term of computation. Regarding the memory, it requires $3{*}nnz$ ($row_indices+column_indices+values$). Accessing an element requires to load two coordinates and one value. The overall memory access pattern is random (if the list is not sorted). Those are two important disadvantages on architectures such as modern CPUs and GPUs. If the network is

undirected (ie. the matrix is symmetric) the format is often optimized by storing only the lower triangular part of the adjacency matrix. In this case (v_i, v_j) represent a link from v_i to v_j and from v_j to v_i.

Row (Source)	0	0	2	2	2	3	3	4	4	5
Column (Destination)	1	2	0	1	4	4	5	3	5	3
Values (Weights)	0.50	0.50	0.33	0.33	0.33	0.50	0.50	0.50	0.50	1.00

Table 2.1 COO format

CSR and Adjacency List

A graph can also be stored as an adjacency list, where for each vertex v_i, the neighbours $v_j, ..., v_k$ and the weights $w_j, ..., w_k$ (if they exist) are listed. For instance, Table 2.2 corresponds to the compressed version of the adjacency matrix of Figure 2.4b.

The adjacency list is often preferred because it improves the edge list by avoiding redundancy of information. For example, $(v_i, v_j), (v_i, v_k)$ can be written $v_i : v_j, v_k$. The compressed sparse row format (CSR) of the adjacency matrix of a graph is a way to represent its adjacency list. The CSR format consists in three arrays : the row pointer of size n, the column indices of size nnz and the values of size nnz. This compression improves both weaknesses of the COO format since it requires less memory $(2 * nnz + n)$ and allows a predictable memory access pattern with fewer data to load per element to process. This comes with a loss of flexibility as the insertion of an element requires to shift all three arrays. CSR is widely used as sparse matrix format for linear algebra and supported in most libraries. It is also good for power-law networks since it is completely independent of the sparsity pattern. If the network is undirected (ie. the matrix is symmetric) the format is often optimized by storing only the lower triangular part of the adjacency matrix.

Row pointers	0	2	2	5	7	9	10			
Column indices	1	2	0	1	4	4	5	3	5	3
Values	0.50	0.50	0.33	0.33	0.33	0.50	0.50	0.50	0.50	1.00

Table 2.2 CSR format

CSC

The compressed sparse column format (CSC) is composed by three arrays: the column pointer of size n, the row indices of size nnz and the values of size nnz Table 2.2 corresponds to the compressed version of the adjacency matrix of Figure 2.4b.
It is exactly the same as CSR but from the column perspective, so advantages and disadvantages, are similar to the CSR format. Notice that the transposed of a matrix in CSR is equal to this same matrix compressed in CSC and interpreted as CSR.

Columm pointers	0	1	3	4	6	8	10			
Row indices	3	1	3	1	5	6	3	4	4	5
Values	0.33	0.50	0.33	0.50	0.50	1.00	0.33	0.50	0.50	1.00

Table 2.3 CSC format

Ellpack

Ellpack compression attempts to reduce the original matrix size and keep a regular structure at the same time. Hence, the number of columns of the compressed matrix is reduced to the size of the largest row (in term of $non-zeroes$). Ellpack stores 2 matrices : one for the values and the other for the columns indices, as shown in Table 2.4. Since Ellpack is very regular, it is well suited for accelerators and efficient for regular sparsity pattern (Bell and Garland, 2008) but not for networks with power law distribution, especially in term of storage. For those cases, Ellpack can be improved like in SGP (Petiton and Emad, 1996) at the cost of a higher pre-processing and/or additional memory requirements.

0.50	0.50	*
*	*	*
0.33	0.33	0.33
0.50	0.50	*
0.50	0.50	*
1.00	*	*

(a) Values

1	2	*
*	*	*
0	1	4
4	5	*
3	5	*
3	*	*

(b) Indices

Table 2.4 Ellpack format

In the context of accelerated graph analysis, we think the best trade-off is the CSR format. This is the format we used for all codes described in this thesis.

Property graphs

Graphs can store and combine several layers of information, with multiple dimensions of vertexes and edges, which is often called a property graph. The representation can be similar to the CSR format with multiple value dimensions attached to vertices and edges as shown in Figure 2.5. Algorithms can adapt to take advantage of this structure to enable new analytics. For instance, for spectral analysis this feature allows running multiple instances of the solver, potentially in parallel, on different edge dimensions of a single network. Typically, several matrices can share the same topology with different sets of weights.

Multiple vertices dimensions can be used to attach different sets of values to them. Those dimensions are independent and can be combined for analysis purposes. This format is also helpful in situations where we could take advantage of interlaced inputs.

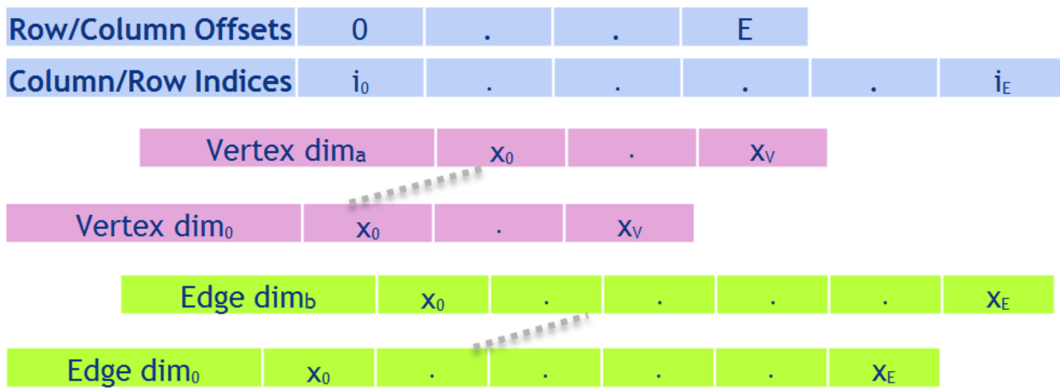


Fig. 2.5 Multidimensional CSR representation

2.1.3 Graph algorithms in linear algebra

The matrix representation allows to apply basic linear algebra to graphs and take advantage of the existing research in mathematics and computer science. Arithmetic operations such as addition and multiplication can be generalized and applied to matrices and vectors, this semi-ring approach can be leveraged to solve many graphs problems (Seshadhri et al., 2011). More precisely, semi-rings operate on a set R with two binary operators: $+$ and $*$ that satisfy:

- $(R,+)$ is associative, commutative with additive identity $(additive_identity + a = a)$
- $(R,*)$ is associative with multiplicative identity $(multiplicative_identity * a = a)$

- Left and Right multiplication is distributive over addition
- Additive identity = multiplicative null operator ($null_op * a = a * null_op = null_op$)

Single source shortest path (SSSP) or Single source widest path (SSWP) are good examples of this approach. For example, on a directed graph A, the SSSP consists in finding the shortest distance between a vertex $source$ and every other vertices in A. We define a semi-ring equipped with the binary operations $\oplus$ as $min(a,b)$ the minimum of a and b, and $\otimes$ as the regular addition of a and b (Mohri, 2002). Then, the SSSP problem can be solved using Algorithm 1.

Algorithm 1 SSSP algorithm

$sssp_0 = indentity(\otimes) = \infty$
$sssp_0[source] = indentity(\oplus) = 0$
repeat
  $sssp_{i+1} = A^T \otimes sssp_i \oplus sssp_i$
until $sssp_{i+1} = sssp_i$

The single source widest path consists in finding the maximal capacity path from a vertex $source$ to every other vertices. It works the same way than SSSP with a different semi ring, where $\oplus$ returns $max(a,b)$ the maximum of a and b, and $\otimes$ returns $min(a,b)$ the minimum of a and b (Hardouin et al., 2008). Then, the SSWP problem can be solved using Algorithm 2.

Algorithm 2 Single Source Widest Path

$sswp_0 = indentity(\otimes) = -\infty$
$sswp_0[source] = indentity(\oplus) = \infty$
repeat
  $sswp_{i+1} = A^T \otimes sswp_i \oplus sswp_i$
until $sswp_{i+1} = sswp_i$

During this thesis we implemented those algorithms to find Single Source Shortest/Widest Path distances on GPU. It was released in nvGRAPH, as part of NVIDIA's CUDA Toolkit 8.0.

2.2 Eigenvalue problems arising in graphs

The study of the eigenvalues of graphs is called spectral graph analysis and often used for vertex ranking, graph clustering and partitioning. In its simplest form an eigensolver computes the eigenpair λ_i and v_i such as

$$A * v_i = \lambda * x_i, \tag{2.3}$$

where λ_i is an eigenvalue of A and v_i is the eigenvector corresponding to λ_i. In graph theory, an eigenpair of a graph corresponds to the eigenpair of its adjacency matrix A.

2.2.1 PageRank

PageRank (Page et al., 1998) is a ranking method which measures the relative importance of elements in a graph by creating a score of importance based on topological dependencies and propagation of influence between vertices. The underlying assumption is that more important vertices are likely to receive more links from other vertices. Figure 2.6 shows a visualization of the graph in 2.1 where PageRank information is used. The vertex diameter and the edge width vary based on the PageRank score which is shown in the center of the vertices.

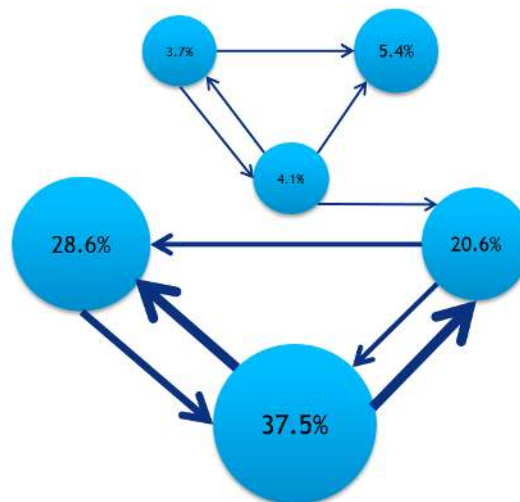


Fig. 2.6 PageRank visualization, α =0.9

PageRank was introduced in 1997 and named after Larry Page who co-founded Google. Hence, it is the historical algorithm used by Google Search to rank websites in their search engine results (Bryan and Leise, 2006; Langville and Meyer, 2003). In this application, the vertices of the graph are web pages and edges are hyperlink. Nowadays, Google combines the PageRank with several other metrics to rank websites. The PageRank can also be used to predict the evolution of an ecosystem, to anticipate human movement and traffic flow or in recommendation engines.
PageRank is based on a Markov model enriched with a damping factor ($\alpha \in [0,1]$), to represent the probability to follow an outgoing edge. For example, on the graph of the web, Google estimated that $\alpha = 0.85$ considerering that a user has 85% of chance to follow edges (ie. click on links) and 15% to open a new window and jump to a random web page.
A Markov chain is a mathematical system that undergoes transitions from one state to another where future states depend only on the current state. It can be seen as a weighted graph where vertices represent states and edges are the probabilities of transition. A stationary distribution (or equilibrium) represents a steady state in the chain's behavior. This equilibrium can be seen as a vector where the ith component represents the probability to be in the ith state. Hence, the stationary distribution of a Markov Chain is the vector w such that $w * H = w$, where H is the transition matrix. PageRank assumes the equiprobability of following an outgoing link, as a result H is defined as :

$$h_{ij} = \begin{cases} 1/\mathrm{d}(i) & \text{if } \mathrm{d}(i) \neq 0 \text{ and there is an outgoing link from } i \text{ to } j \\ 0 & \text{otherwise,} \end{cases} \tag{2.4}$$

The equilibrium vector has a large range of applications since it is used to predict the most probable future state based on present observations. (Langville and Meyer, 2006)

The Google Matrix G (Eq. 2.5) and the PageRank vector p are formed from H and α using:

$$G = \alpha H^T + b(\alpha a + (1-\alpha)e)^T \tag{2.5}$$

Where a is the bookmark vector of leafs, formed using Eq. 2.6, and b is the constant and uniform vector $1/n$.

$$a_i = \begin{cases} 1 & \text{if there are no outgoing links from } i \\ 0 & \text{otherwise} \end{cases} \tag{2.6}$$

The PageRank vector is the eigenvector corresponding to the dominant eigenvalue of G. Notice that matrix G is row-stochastic. Therefore, it has non-negative elements and satisfies $Ge = e$ with $e = (1,...,1)^T$. In order to support large datasets, we do not form the Google matrix G because it is dense (Eq. 2.5). Instead, we formulate the problem to work on the sparse and irreducible matrix H as shown in Algorithm 3. Indeed, G is obtained by applying a rank-one update to H, so the main cost per iteration can be reduced to one sparse matrix vector multiplication. The number of iterations depends on the gap between λ_{max} and the second largest eigenvalue which is bounded by the damping factor $\alpha \in [0,1]$. Many methods can solve the PageRank problem and a comprehensive survey was done in (Langville and Meyer, 2006). Fig 2.7 shows the Google matrix of Fig 2.1 with α =0.9.

$$\begin{pmatrix} \frac{1}{60} & \frac{7}{15} & \frac{7}{15} & \frac{1}{60} & \frac{1}{60} & \frac{1}{60} \\ \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{19}{60} & \frac{19}{60} & \frac{1}{60} & \frac{1}{60} & \frac{19}{60} & \frac{1}{60} \\ \frac{1}{60} & \frac{1}{60} & \frac{1}{60} & \frac{1}{60} & \frac{7}{15} & \frac{7}{15} \\ \frac{1}{60} & \frac{1}{60} & \frac{1}{60} & \frac{7}{15} & \frac{1}{60} & \frac{7}{15} \\ \frac{1}{60} & \frac{1}{60} & \frac{1}{60} & \frac{11}{12} & \frac{1}{60} & \frac{1}{60} \end{pmatrix}$$

Fig. 2.7 The Google matrix of Figure 2.1

2.2.2 Clustering

Graph clustering and partitioning problems consist in dividing a graph into k smaller components. The result is k separated groups where the number of edges between them is minimized and/or the connection inside each group is maximized. For the partitioning case, partitions are expected to have similar sizes while clustering focuses on finding tightly connected groups without constraints on the size. The idea is simple but this is actually a Non-deterministic Polynomial-time hard problem (NP-hard) which is expensive to solve. A comprehensive review of clustering techniques is given in (Fortunato, 2010). In this thesis we focus on spectral approaches (Newman, 2006;

Ng et al., 2001).

Spectral minimum balanced cut clustering

A common approach for the clustering problem is to estimate how well clusters are independent by looking the number of edges between them. A famous technique to minimize the number of edges between clusters is based on the graph Laplacian matrix (Ng et al., 2001). The Laplacian matrix is a sparse positive semi-definite matrix defined as:

$$L = D - A \tag{2.7}$$

where A is the symmetric adjacency matrix of the undirected graph, $D = diag(Ae)$, and $e = (1, ..., 1)^T$. The relationship between the eigenpairs of a Laplacian matrix and the connectivity of a graph was first noted by Donath (Donath and Hoffman, 1973) and Fiedler (Fiedler, 1973). The minimum balanced cut problem is NP-complete but by the Courant-Fischer theorem (Horn and Johnson, 1986) it can be approximated by the smallest eigenpairs of the Laplacian matrix.

Spectral modularity clustering

The modularity metric is based on the idea that similar vertices are connected by more edges in the current graph than if they were randomly connected. It measures how well a given clustering applies to a particular graph versus a random graph (Newman, 2003, 2006; Newman and Girvan, 2004). The modularity metric has been used in practice to study different disease epidemics (Newman, 2003). This metric is closely related to the assortativity coefficient (Newman, 2002) as well as the algebraic connectivity of graphs (Chung, 1997; Donath and Hoffman, 1973; Fiedler, 1973), with a comprehensive scientific literature review given in (Chen et al., 2014; Newman, 2010).

The clustering is achieved by finding vertex assignments into clusters such that the modularity is maximized. The modularity maximization problem is NP-complete but by the Courant-Fischer theorem (Horn and Johnson, 1986) it can be approximated by the largest eigenpairs of the modularity matrix. Many other methods have been developed for maximizing modularity based on an agglomerative approach such as

(Clauset et al., 2004) and (Blondel et al., 2008).

Hierarchical clustering

Although hierarchical clustering is not a spectral technique by nature, it is widely used in practice (Karypis and Kumar, 1998). The technique arranges the network into a hierarchy of groups according to a specified cost function. This can be achieved by an agglomerative or divisive approach.

In the agglomerative, or bottom-up approach, each vertex starts in its own cluster, and clusters are merged based on a set of heuristics. Each level of the hierarchy is constructed from the previous level by collapsing vertices and edges together. This is similar to the agglomerative technique used in algebraic multi-grid. In the divisive or top-down approach, all vertices start in one cluster and splits are performed recursively.

This multi-level scheme can be a clustering technique by itself or it can also be combined with other approaches. As shown on Figure 2.8 the clustering problem can be solved on the coarsest level using spectral clustering, and the results propagated back to the fine level.

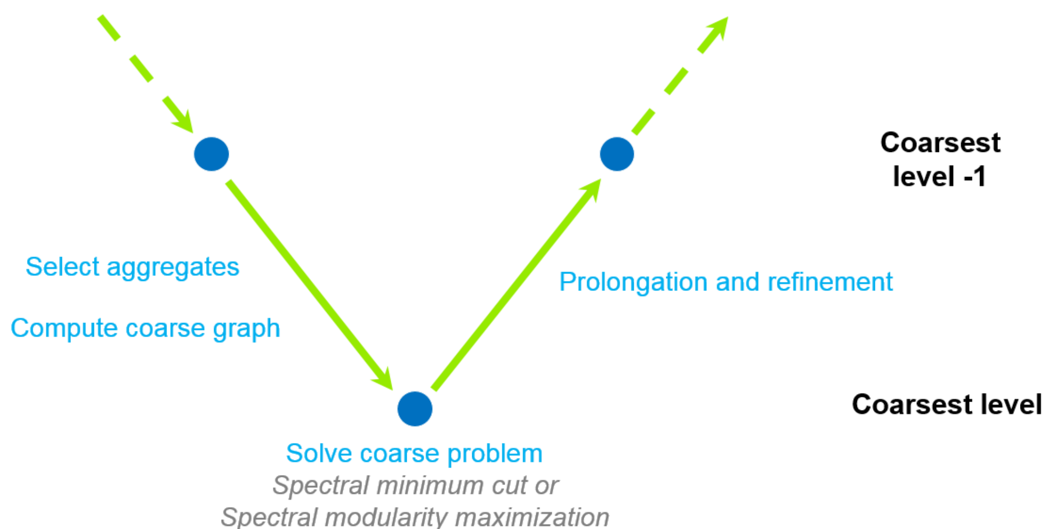


Fig. 2.8 Hierarchical clustering coupled with spectral clustering

2.3 GPU accelerators

2.3.1 Architecture and programming

A graphics processor unit (GPU) Figure 2.9 is an electronic circuit consisting in a massively parallel architecture with thousands of cores designed to quickly access and modify memory. Typical use cases involve memory-intensive and compute-intensive tasks with a Single Instruction Multiple Data (SIMD) parallel pattern (Flynn, 1972).

Fig. 2.9 Diagram of Nivida GP100 with 60 SM and 3840 cores

Architecture overview and history

At the beginning, GPUs were designed as graphics accelerators, supporting only specific functions but the hardware became increasingly programmable. It turned out that most of these graphics computations involved matrix and vector operations, and quickly people started to look at GPUs for non-graphical calculations. A software concept called General-purpose computing on graphics processing units (GPGPU) was born.

In general we refer to the CPU as host and to the GPU as device. The device has its own memory, with a fast connection between the memory and cores. There are often

other hierarchical layers such as the cache, shared memory and registers. The cores are grouped in order to form streaming multiprocessors (SM) as shown in Figure 2.10, allowing the sharing of resources such shared memory, caches, registers and scheduler. Inside a SM, one instruction is executed at the time and it runs in parallel on the CUDA cores. For a long time data transfers between host and device were explicit and under the developer's responsibility but recent architectures support a unified virtual memory address space between CPU and GPU.

Fig. 2.10 Streaming multi-processor (SM) of Nvidia GP100

Programming

In 2006 NVIDIA invented the Compute Unified Device Architecture (CUDA), a parallel computing platform and programming model for general-computing on GPUs. Today, the CUDA Toolkit (NVIDIA, 2017) includes a compiler, tools for debugging, profiling, and many libraries covering a wide range of applications. In 2009, an open framework for writing programs that execute across heterogeneous platforms called Open Computing Language (OpenCL) was developed by Apple Inc in collaboration with AMD, IBM, Qualcomm, Intel, and Nvidia.

The CUDA programming model involves writing parallel device code called *kernel*. This code is compiled with a dedicated compiler. There are bindings in many languages such as C, C++, Java, Python, Fortran etc. A kernel is written to be executed in parallel on many cores by design. There are different parameters that the user can set to specify how the kernel is going to be executed. For example, how many blocks, threads and items per thread will be used. Typically, the developer set those parameters based on the architecture specifications and the application workload.
The following example shows a simple kernel computing $z = \alpha * x + y$, also known as AXPY, in C/CUDA. Here, x, y, z are vectors and α is a scalar. In this code the kernel is declared using the keyword *__global__*. This keyword indicates that the function can be called from the host and executed on the device. The variable $threadIdx.x$ indicates the index of the thread executing the kernel.

```
__global__ void axpy(float alpha, float* x, float* y, float* z)
{
    int i = threadIdx.x;
    z[i] = alpha * x[i] + y[i];
}
```

The kernel is called with parameters provided in <<< ... >>>. In this example we assume x, y, z are accessible on the device and point to N real elements. The following call assign a thread per element of the vector:

```
axpy<<<1, N>>>(2.0, x, y, z);
```

Programming GPUs with this approach is powerful but requires a good understanding of parallel computing and of the accelerator's architecture in order to leverage the full potential of the device. A comprehensive documentation and user's guide can be found in (NVIDIA, 2017). Finally, notice that it is possible to combine the computational power of both CPU and GPU. This can be done by using a design pattern where small independent sequential tasks are left to the host while the device performs large parallel operations. This is often referred to as hybrid host/device algorithms. It can also be automated by using a dedicated framework (Lee et al., 2014).

Alternatives

GPU acceleration can be obtained by using compiler directives to automatically gener-

ate device code. For instance, OpenACC and OpenMP directives. There are general purpose libraries with a high-level interface such as Trust, a C++ template library based on the C++ STL. In addition, there is a large ecosystem of tools and libraries growing around GPUs, from basic routines to industrial software. In the field of graph analysis, GPU have proved to be efficient, leding to products such as GunRock (Wang et al., 2016), MapGaph (Fu et al., 2014) and cuStinger (Green and Bader, 2016) for instance. Most of numerical linear algebra already benefits from GPU acceleration with libraries such as Magma, Cublas, Cusparse, Cusolver, SparseSuite, AmgX, among others (NVIDIA, 2017).

2.3.2 Eigenvalue methods for networks and accelerators

Accelerated sparse eigenvalue solvers represent an active research field. For example, the Power Method and Krylov methods such as the Arnoldi method (Golub and Greif, 2006), and its explicitly restarted variant (Dubois et al., 2011; Emad et al., 2005). In particular, the implicitly restarted Arnoldi methods (IRAM) (Sorensen, 1997) showed good improvements compared to the power method.
Symmetric problems often use the accelerated version of Lanczos (Matam and Kothapalli, 2011) which can be seen as a special case of the Arnoldi method. The implicitly restarted variant Lanczos (IRL) is known for its stability and good convergence with constant and reasonable memory requirements (Lehoucq, 1995; Sorensen, 1998).
We point out that efficient dense eigenvalue solvers are out of the scope of this thesis since real networks lead to large sparse eigenvalue problems.
Let us now present several numerical sparse methods of interest for network analytics.

Power iteration method

The power iteration is a straightforward iterative method to compute an eigenpair composed by the eigenvalue λ_{max} and the corresponding eigenvector v_{max} of a general matrix A by solving iteratively

$$v_{i+1} = \frac{Av_i}{\|Av_i\|} \tag{2.8}$$

Notice that the main cost of an iteration is one sparse matrix vector multiplication (SPMV). This method works well on sparse matrices since it does not require any additional memory. However, it computes only one eigenvalue which is the dominant

one by default. It is possible to use a shift μ to converge to another eigenvalue though. This is based on the idea that if λ_i is an eigenvalue of A, a constant μ can be chosen such as $\sigma = 1/(\lambda_i - \mu)$ is the dominant eigenvalue of $(A-\mu I)^{-1}$. This method is called the inverse iteration:

$$v_{i+1} = \frac{(A-\mu I)^{-1} v_i}{\|(A-\mu I)^{-1} v_i\|} \tag{2.9}$$

Since the power iteration method overwrites the vector v at each iteration it looses useful information that could be used to speed up the convergence. Hence, the main drawback of the power method is the slow convergence compared to more advanced methods, especially when eigenvalues are clustered (Golub and Greif, 2006).

Notice that the PageRank problem can be solved using the power iteration method as follows:

Repeat $p_{i+1} = Gp_i/\|Gp_i\|$
Until $p_{i+1} = p_i$

where G is a google matrix (Eq. 2.5) and p is the PageRank vector. For this problem it is known that the number of iterations depends on the gap between λ_{max} and the second largest eigenvalue which is bounded by the damping factor $\alpha \in [0,1]$ (Brody, 1997). In order to support large datasets we do not form the Google matrix. Instead, we formulate the problem in order to work on the sparse and irreducible matrix H as shown in Algorithm 3.

Algorithm 3 PageRank

for $i = 1,2,...,n$ **do**
  $a_i = 1$ if V_i is a leaf, 0 otherwise
  $b_i = 1/n$
  $p_i \in \mathbb{R}^{*+}$
end for
repeat
  $p_{i+1} = \alpha H^T p_i + b(\alpha a + (1-\alpha)e)^T p_i$
  $p_{i+1} = p_{i+1}/\|\alpha H^T p_i + b(\alpha a + (1-\alpha)e)^T p_i\|$
until $p_{i+1} \simeq p_i$

This approach is efficient on sparse matrices since it does not require any additional memory. However, the power method is only able to compute one eigenpair, which is the dominant one by default.

Arnoldi method

The Arnoldi method is an iterative method based on Krylov subspace theory to compute an approximation of k eigenpairs of a general matrix A of size n by those of a much smaller Hessenberg matrix H_m of size m, with $k < m \ll n$. This matrix is obtained by an orthogonal reduction of A onto an m-dimensional Krylov subspace $K_m(A,v)$ (Eq. 2.10).

$$K_m(A,v) = span\{v, Av, A^2v, ...A^{m-1}v\} \tag{2.10}$$

The core part of this technique is the m-step Arnoldi factorization which forms H_m, V_m, f_m such that $AV_m = V_m H_m + f_m e_m^T$, with the process described in Algorithm 4.

Algorithm 4 m-step Arnoldi factorization

1: **Input:** A, v_i, i, m
2: **Output:** V_m, H_m, f_m
3: **for** $j = i, ..., m$ **do**
4:     **if** $j = 1$ **then**
5:         $w = A * v_1$
6:         $V_m(:,j) = v_1$
7:         $H_m(1,1) = v_1^T * w$
8:         $f_m = w - v_1 * v_1^T * w$
9:     **else**
10:         $v = f_m/||f_m||; w = A * v$
11:         $V_m(:,j) = v$; $H_m(j,j-1) = ||f_m||$
12:         $h = V_m(:,1:j)^T * w$
13:         $f_m = w - V_m(:,1:j) * h$
14:         $H_m(1:j,j) = h$
15:     **end if**
16: **end for**

The Arnoldi method starts similarly to the power iteration but instead of rewriting v at each iteration the information is kept by forming a matrix $[v, Av, A^2v, ...A^{m-1}v]$

where those vectors are orthonormalized using a modified Gram-Schmidt process to form the Krylov basis. Those vectors are often called Arnoldi vectors.
Current computers and standards only offer approximations of floating point arithmetic, thus we prefer to use a modified Garrm-Schmitt process to improve the accuracy. This problem was discussed in (Saad, 1992).

$$H_m = V_m^* A V_m \Leftrightarrow A V_m = V_m H_m + f_m e_m^*, \quad f_m = H_{m+1,m} v_{m+1} \tag{2.11}$$

Implicitly restarted Arnoldi method

The Arnoldi factorization (Algorithm 4) is based on a random initial guess, thus by improving this initial vector it is possible to speedup the convergence. Moreover, if Arnoldi takes too much time to converge, m becomes large and generate high memory requirements. A method known as the Explicitly Restarted Arnoldi Method (ERAM) was proposed in (Saad, 1980). ERAM takes advantage of spectral information of H_m and of the orthonormal basis to do a polynomial preconditioning on the initial vector v_1. The results is a new, better Krylov subspace for the new cycle. This method showed good improvements in term of convergence but requires an explicit linear combination of Ritz elements.

An approach based on implicitly shifted QR factorizations on H_m, called Implicitly Restarted Arnoldi Method (IRAM) (Lehoucq, 1995; Sorensen, 1997, 1998), uses the unwanted eigenvalues as shifts. Algorithm 6 illustrates this method; it starts with the Arnoldi factorization to build the initial subspace of size m. Then at each restart cycle all eigenpairs of H_m are computed and the residual norm is estimated (Saad, 1992). The QR step concentrates the information of interest in the upper-left part of H_m and the orthonormal matrix Q_m is used to update the matrices V_m, H_m and f_m. Thus, the initial problem is implicitly updated. This process is done iteratively, at a cost dominated by $m-k$ matrix-vector multiplications per restart cycle. When the approximation is good enough, the eigenvectors of H_m can be projected using V_m to get the Ritz eigenvectors of A.
IRAM can operate in parallel (Emad et al., 2005; Shahzadeh Fazeli et al., 2015), handles large problems (Liu et al., 2013), and does not require extra memory.

This approach is efficient for spectral graph analysis (Lehoucq, 1995), in addition it is possible to compute more than one eigenpair, which can be useful for spectral clustering applications (Ng et al., 2001).

Algorithm 5 Implicitly restarted Arnoldi algorithm

1: **Input:** A, v_1, k, m
2: **Output:** k wanted eigenpairs
3: $[V_m, H_m, f_m] = \mathit{Arnoldi\text{-}Factorization}(A, v_1, 1, m)$
4: **while** *Convergence is not reached* **do**
5: *Compute the eigenpairs of* H_m
6: *Compute the residual and stop if converged*
7: *Select set of* $p = m - k$ *shifts* $\mu_1, ..., \mu_p$ *based on unwanted eigenvalues*
8: $Q_m = I$
9: **for** $j = p, ..., 2, 1$ **do**
10: $[Q_j, R_j] = \mathit{QR\text{-}Factorization}(H_m - \mu_j I)$
11: $H_m = Q_j^H H_m Q_j$
12: $Q_m = Q_m^H Q_j$
13: **end for**
14: $f_m = V_{m_{(:,:)}} Q_{m_{(:,k+1)}} * H_{m_{(k+1,k)}} + f_m * Q_{m_{(m,k)}}$
15: $V_{m_{(:,1:k)}} = V_{m_{(:,:)}} Q_{m_{(:,1:k)}}$
16: $[V_m, H_m, f_m] = \mathit{Arnoldi\text{-}Factorization}(A, V_k, k, m)$
17: **end while**

The IRAM method can benefit from accelerators for the sparse matrix vector multiplications and the dot products. However, inside the Krylov subspace, the eigenvalue problem becomes small and cannot really take advantage of the acceleration. In this case it is possible to offload the small matrix H_m on the CPU to find the eigenpairs using Lapack (Anderson et al., 1999) and return the result to the device.

Multiple implicitly restarted Arnoldi method with nested subspace

In IRAM, the choice of the Krylov subspace size remains empirical but still has an important impact on the overall success of the method. A shown in Algorithm 6, the idea of MIRAMns (Shahzadeh Fazeli et al., 2015) is to improve this point by computing l subspaces of different sizes and select the best one for each restart.

A subspace s_1 is considered to be better than another subspace s_2 if its residual is smaller : $P(s_1) < P(s_2)$, (ie. the approximation is better at the current cycle). In practice the residual is calculated using the Ritz approximation technique and $P(s_i) = max(\rho_{(\lambda_1,w_1)},...,\rho_{(\lambda_k,w_k)})$, where $(\lambda_1,w_1),...,(\lambda_k,w_k)$ are the eigenpairs of H_m. The residual $\rho_{(\lambda_i,w_i)}$ is the residual norm associated to the Ritz eigenpair (λ_i,w_i), calculated using the Ritz estimate $| \, \|f_{s_i}\|_2 e_i^T w_i \, |$.

Algorithm 6 Multiple implicitly restarted Arnoldi with nested subspaces

Input: A, v_1, k, m_{max}
Output: k wanted eigenpairs
$[V_{m_i}, H_{m_i}, f_{m_i}]$ =*Arnoldi-Factorizations*$(A, v_1, 1, m_{max})$
where m_i *refers to Arnoldi Factorization of size* m_i
while *Convergence is not reached* **do**
  Compute the eigenpairs of H_{m_i}
  Compute the residuals, stop if one subspace converged
  Select the best subspace size
  Set m, H_m, V_m, f_m *accordingly*
  Select p=m-k shifts $\mu_1,...,\mu_p$ *from unwanted eigenvalues*
  $Q_m = I$
  for $j = p,...,2,1$ **do**
    $[Q_j, R_j] = QR\text{-}Factorization(H_m - \mu_j I)$
    $H_m = Q_j^H H_m Q_j$
    $Q_m = Q_m^H Q_j$
  end for
  $f_k = H_m(k+1,k) V_m(1:n,1:m) Q_m(1:m,j+1) + Q_m(m,k) f_m$
  $V_m(1:n,1:k) = V_m(1:n,1:m) Q_m(1:m,1:k)$
  $[V_{m_i}, H_{m_i}, f_{m_i}]$ =*Arnoldi-Factorizations*(A, V_k, k, m_{max})
end while

Lanczos and implicitly restarted Lanczos method

The Lanczos method (Calvetti et al., 1994; Lanczos, 1950) is an iterative method based on Krylov subspace theory to compute an approximation of k eigenpairs of an Hermitian matrix A of size n by those of a much smaller symmetric tridiagonal matrix T_m of size m, with $k \leq m \ll n$. It can compute multiple eigenpairs at once and keeps track of previous vector directions. The matrix T_m is obtained by an orthogonal reduction of A onto an m-dimensional Krylov subspace $K_m(A,v)$ (Equation 2.12))

$$K_m(A,v) = span\{v, Av, A^2v, ...A^{m-1}v\} \tag{2.12}$$

The Lanczos method can be seen as a special case of the Arnoldi method for symmetric matrices. Hence, the Lanczos method is also similar to the power iteration method but instead of rewriting v at each iteration the information is kept by forming a matrix $[v, Av, A^2v, ...A^{m-1}v]$ where those vectors are orthonormalized using a modified Gram-Schmidt process to form the Krylov basis. Those vectors are often called Lanczos vectors. The m-step Lanczos algorithm forms T_m, V_m, f_m according to the formula in Equation 2.13. This process is described in Algorithm 7.

$$AV_m = V_m T_m + f_m e_m^T \tag{2.13}$$

In Algorithm 7, off-diagonal elements T_m are denoted $\beta_j = T_{j-1,j}$ for notational convenience. Also notice that $T_{j-1,j} = T_{j,j-1}$ due to the symmetry. By construction, the eigenvalues of the small tridiagonal matrix T_m are good approximations of the extreme eigenvalues of A. This fundamental property leads to the Lanczos eigenvalue method.

Algorithm 7 m-step Lanczos factorization

1: **Input:** A, v_i, i, m
2: **Output:** V_m, T_m, f_m
3: $\beta_1 \leftarrow 0$
4: **for** $j = i, i+1, \cdots, m-1$ **do**
5: $\quad w'_j \leftarrow Av_j$
6: $\quad T_{j,j} \leftarrow w'^*_j v_j$
7: $\quad w_j \leftarrow w'_j - T_{j,j}v_j - \beta_j v_{j-1}$
8: $\quad \beta_{j+1} \leftarrow \|w_j\|$
9: $\quad v_{j+1} \leftarrow w_j/\beta_{j+1}$
10: **end for**
11: $w_m \leftarrow Av_m$
12: $T_{m,m} \leftarrow w^*_m v_m$

In order to limit the size of the basis (m), the method can be restarted, which has a direct impact on memory requirements. Restarting is helpful for large cases where each vector of size n is expensive to store and numerical imprecision arises quickly due to the scale of the operations. Indeed, it is possible to lose accuracy when the number of Lanczos vectors is large because of repeated roundoff resulting from numerical floating point arithmetic. The problem is caused by the loss of orthogonality between the Lanczos vectors, which happens as soon as Ritz values stabilize.

In this section we focus on the implicitly restarted Lanczos method (IRL) which compresses the information into a k−dimensional Krylov subspace (Lehoucq, 1995; Sorensen, 1997). This is achieved by applying an implicitly shifted QR scheme (Bai and Demmel, 1989) where the unwanted eigenvalues are used as shifts as shown in Algorithm 8. Initially $v_0 = [0,...,0]$ and v_1 can be initialized with a random vector with norm 1. IRL starts with the m-step Lanczos factorization of Algorithm 7 to build the initial Krylov subspace of size m. Then at each restart cycle all eigenpairs of T_m are computed and the residual norm is estimated. The QR step concentrates the information of interest in the k−dimensional subspace, which corresponds to the upper left part of T_m and the orthonormal matrix Q_m is used to update V_m, T_m and f_m. Once this step is completed the initial problem is implicitly updated and $m-k$ SPMVs are required for the next cycle. When the approximation is good enough it exits the main loop and V_m is multiplied by the matrix formed by the eigenvectors of T_m to approximate the eigenvectors of A.
Let us now discuss the complexity of this method, let nrc be the number of restart cycles. The cost of IRL in terms of matrix-vector multiplications is $m + p \times (nrc - 1)$. Indeed, in the first cycle the number of matrix-vector multiplications is m and for each of the restart cycles, the number of matrix-vector multiplications is $p = m - k$.
Notice that when A is sparse and n is large, the dot products may represent a significant part of the computation. The computation involving elements of size m is relatively cheap compared to the rest of the operations because $m \ll n$. The eigenpairs of the tridiagonal matrix T_m can be obtained in $O(m^2)$ per step with the QR method.
For a dense matrix the space complexity of IRL is $n^2 + O(m \times n)$. In the context of graphs, the complexity is $O(|E| + m * |V|)$.
The implicitly restarted technique is efficient for large sparse matrices (Lehoucq, 1995), in addition it allows to compute multiple eigenpairs at once which is useful for spectral clustering (Ng et al., 2001). As a result, IRL is a good candidate to handle large graph problems.
Notice that an alternative to the implicit restart is the explicit restart. The idea is to compute a new starting vector based on all information and start again from the beginning with this improved initial vector (Saad, 1992). In this thesis we focus on the implicit restart.

Algorithm 8 Implicitly restarted Lanczos algorithm

1: **Input:** A, v_1, k, m
2: **Output:** k wanted eigenpairs
3: $[V_m, T_m, f_m] \leftarrow$ *Lanczos-Factorization*$(A, v_1, 1, m)$
4: **while** *Convergence is not reached* **do**
5: *Compute the eigenpairs of* T_m
6: *Compute the residual and stop if converged*
7: *Select set of* $p = m - k$ *shifts* $\mu_1, ..., \mu_p$ *based on unwanted eigenvalues*
8: $Q_m \leftarrow I$
9: **for** $j = p, ..., 2, 1$ **do**
10: $[Q_j, R_j] \leftarrow$ *QR-Factorization*$(T_m - \mu_j I)$
11: $T_m \leftarrow Q_j^T T_m Q_j$
12: $Q_m \leftarrow Q_m^T Q_j$
13: **end for**
14: $f_m \leftarrow V_{m_{(:,:)}} Q_{m_{(:,k+1)}} * T_{m_{(k+1,k)}} + f_m * Q_{m_{(m,k)}}$
15: $V_{m_{(:,1:k)}} \leftarrow V_{m_{(:,:)}} Q_{m_{(:,1:k)}}$
16: $[V_m, T_m, f_m] \leftarrow$ *Lanczos-Factorization*(A, V_k, k, m)
17: **end while**

2.3.3 Accelerated basic operations

Solving a sparse eigenvalue problem requires many basic linear algebra operations. The basic linear algebra operations are commonly categorized in three levels. The first level performs scalar and vector operations, the second performs matrix-vector operations, and the third performs matrix-matrix operations. Let us now present parallel solutions and obstacles for several of them in the context of sparse eigensolvers for graph analysis on GPUs.

Vector-vector operations

Vector-vector operations are known as level 1 routines and functions. For instance, in eigensolvers, the addition of two vectors ($y = x + y$) or the scaling of a vector by a constant ($y = \alpha y$) appear frequently. Whenever possible, those operations are fused since GPUs have a hardware unit to compute fused multiply–add (FMA) using a single floating point rounding. Notice that in parallel this type of operation is straightfor-

ward and is expected to scale linearly with the number of threads.
Another frequent vector operation is the dot product ($\beta = x.y$) which appears in vector normalization for instance. On GPUs, the dot product involves an embarrassingly parallel pairwise multiplication followed by a reduction from vector to a scalar. In parallel, the reduction cannot be safely achieved in a single parallel step since the local reduction of each thread must reconcile with the current global sum before adding its contribution. On GPU, a tree-based approach within each warp is generally preferred. In this case, the reduction requires $log(t)$ parallel steps with t the number of threads as illustrated in Figure 2.11 for 8 threads. Moreover, in recent GPU architectures, the shuffle instruction enables a thread to directly read a register from another thread in the same warp. This instruction allows fast parallel tree-based reductions within a warp without temporary memory or synchronizations. The $warpReduceSum$ kernel bellow corresponds to the illustration of Figure 2.11 for $warpSize = 8$.

```
int warpReduceSum(int val) {
  for (int offset = warpSize/2; offset > 0; offset /= 2)
    val += __shfl_down(val, offset);
  return val;
}
```

Now that local warp sums exist they can be accumulated over the entire grid using atomics. Notice that using atomics at this level has a light impact on performances because there are few collisions. However, this can be an issue for reproducibility since accumulations are done in unpredictable order.
In linear algebra problems derived from graph analysis, the complexity of level 1 operations depends on the number of vertices. As a result, level 1 computations are relatively cheap compared to level 2 and 3 operations. There are efficient existing GPU implementations in libraries such as CUBLAS (NVIDIA, 2017).
An implicitly restarted eigenvalue solver runs dozens of level 1 operation per iteration. There are few cases where level 1 operations can dominate the execution time. For instance, this can occur when the number of edges is close to the number of vertices and the dot operation is not optimized.

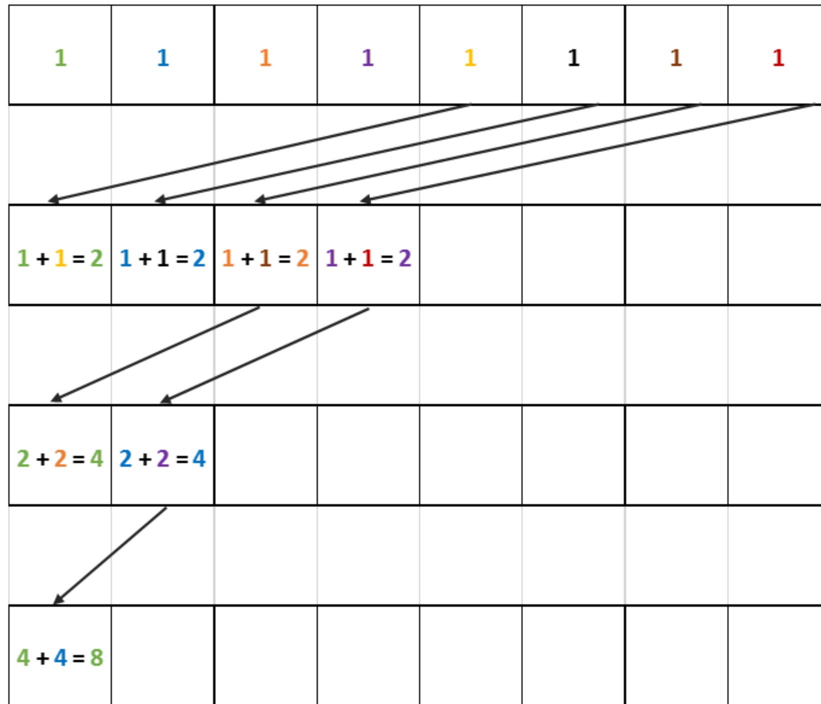


Fig. 2.11 Parallel tree-based warp reduction

Sparse matrix-vector operations

Matrix-vector operations are known as level 2 routines and functions. In practice, the sparse matrix vector multiplication (SpMV) is the most expensive part in all eigensolvers presented in this thesis. The complexity of this operation depends on the number of edges. Indeed, this matrix vector multiplication is at the heart of Krylov methods since it occurs at each iteration. In PageRank for instance, the time spent in the matrix-vector multiplication varies between 80 and 95 percent of the total time, depending on the method. As a result, it is critical to understand the details of this particular operation on GPU, especially for networks. There are over 100 specialized SpMVs for GPUs (Bell and Garland, 2008). Regarding the CSR compression format, which is popular and provides good compression for irregular networks, there are basically four common solutions:

Scalar. Since each row can be computed independently, one thread can be assigned to one row. The CSR-scalar is relatively simple to implement and performs well when using a few fast cores but this does not scale on GPUs. First because memory accesses per warp diverge and also because the total time depends on the largest rows which

are very large compared to the rest of the data set in scale-free networks.
Figure 2.12 shows the threads mapping for the sparse matrix vector multiplication in CSR with the vectorized approach for the graph of Figure 2.1 and its CSR representation in Table 2.2.

Row pointers	0	2	2			5		7		9	10
Column indices	1	2	0	1	4	4	5	3	5	3	
Values	0.50	0.50	0.33	0.33	0.33	0.50	0.50	0.50	0.50	1.00	
Active treads (t_0)	↑		↑			↑		↑		↑	
Active treads (t_1)		↑		↑			↑		↑		
Active treads (t_2)					↑						
Thread #	0		1			2		3		4	

Fig. 2.12 Parallel SpMV - CSR scalar

Vectorized. The memory access pattern is improved as well as the time to process large rows by using a vectorized approach where a group of threads of fixed size is assigned to each row. This technique performs better on GPUs, especially on regular sparsity patterns but still suffers from load balancing problems on real networks.
Figure 2.13 shows the threads mapping for the sparse matrix vector multiplication in CSR with the vectorized approach for the graph of Figure 2.1 and its CSR representation in Table 2.2.

Row pointers	0	2	2			5		7		9	10
Column indices	1	2	0	1	4	4	5	3	5	3	
Values	0.50	0.50	0.33	0.33	0.33	0.50	0.50	0.50	0.50	1.00	
Active treads (t_0)	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	
Thread #	0	1	0	1	2	0	1	0	1	0	

Fig. 2.13 Parallel SpMV - CSR vectorized

Hybrid. The hybrid technique is an adaptive combination of previous approaches. Typically, it considers a group of vertex (rows) and selects the best kernel depending on the number of edges (non-zeroes). If the group is composed of small rows, it selects a CSR scalar kernel. If the group is composed of large rows, the CSR vectorized kernel is selected. For scale-free network applications, it is helpful to have a third kernel that

splits very large rows in order to process them using several groups of threads.

Pre-processed and post-processed. Pre-processing and post-processing steps can be applied to reorder, sort, classify, or break rows and compute auxiliary information which helps to speedup the multiplication (Greathouse and Daga, 2014). This is particularly useful for iterative methods since, in general, the pre-processing step is done only once as the sparsity pattern of the matrix does not change.
For example, the CSRMV merge-path (CSRMV-MP) (Merrill and Garland, 2016), offers a perfect workload balance on GPUs, which is a primary concern in our case. At a high level it starts by computing partitions using the 2D merge-path (Deo et al., 1994; Odeh et al., 2012), the output of this step is the starting offsets of each, balanced, partition (in term of coordinates in row offsets and values or column indices as illustrated in Figure 2.14). Since multiple rows can belong to the same partition or a single row can cross partitions, a key value array is used to keep track of that. Then, the actual multiplication is performed in two steps. First, local multiplications and accumulations are computed. Since some rows may cross partitions, the second step is to use a key-value reduction to accumulate partial sums.

Row ptrs	0	2	2	5	7	9	10

Val Idx	Col Idx	Val	Thread #
0	1	0.50	0
1	2	0.50	
2	0	0.33	
3	1	0.33	1
4	4	0.33	
5	4	0.50	
6	5	0.50	
7	3	0.50	2
8	5	0.50	
9	3	1.00	

Fig. 2.14 Parallel SpMV - CSR merge-path

Figure 2.14 shows the threads mapping for the sparse matrix vector multiplication in CSR with the merge-path approach (Merrill and Garland, 2016) for the graph of Figure 2.1 and its CSR representation in Table 2.2. The merge-path CSR decomposition can be illustrated as a 2D path based on the row_offsets and the sequence of indices of column_indices or values arrays. The path begins in the top-left corner, it moves downward when consuming indices (accumulating matrix-vector dot-products within a row) and rightward when consuming row offsets. Finally, diagonals are used to split this decision path into equal-lengths. In Figure 2.14 the decomposition is done for three threads and the workload is optimally balanced.
Figures 2.15 and 2.16 show an experiment from (Merrill and Garland, 2016) on the performance of CSRMV-MP vs. Cusparse (Tesla K40, 64 bit) and MKL (dual-socket Intel e5-2695, 48-threads) on the entire matrix collection of the University of Florida (4200 matrices). Notice that CSRMV-MP has a better consistency between size and time than Cusparse because it is more resilient to row-length variations. In average, the CSRMV-MP is 1.1× and 2.9× faster on the CPUs and GPUs respectively.

Matrix-matrix operations

Matrix-matrix operations are known as level 3 routines and functions. The matrix-matrix multiplication is also known to be an expensive operation with almost cubic complexity. In implicitly restarted Krylov solvers, it appears during the implicit update of the Krylov vectors and at the very end of the method when computing the eigenvectors. Although, in practice, it is not a primary concern for three reasons :

- First, it is a very special type of matrix-matrix multiplication called tall skinny matrix matrix multiply. Indeed, notice in Algorithms 5, 6, and 8 that the matrix V of size (n,k) is always multiplied by a matrix of size (k,k) where only n is large and k is very small.
- Second, it is not in the inner loop, it happens only during the restart and at the end of the method (Algorithms 5, 6, and 8).
- Third, the dense matrix-matrix multiply is intensive, parallel, and well balanced, so this is one of the best scenarios to take advantage of GPU performances. We use the CUBLAS library for this operation (NVIDIA, 2017).

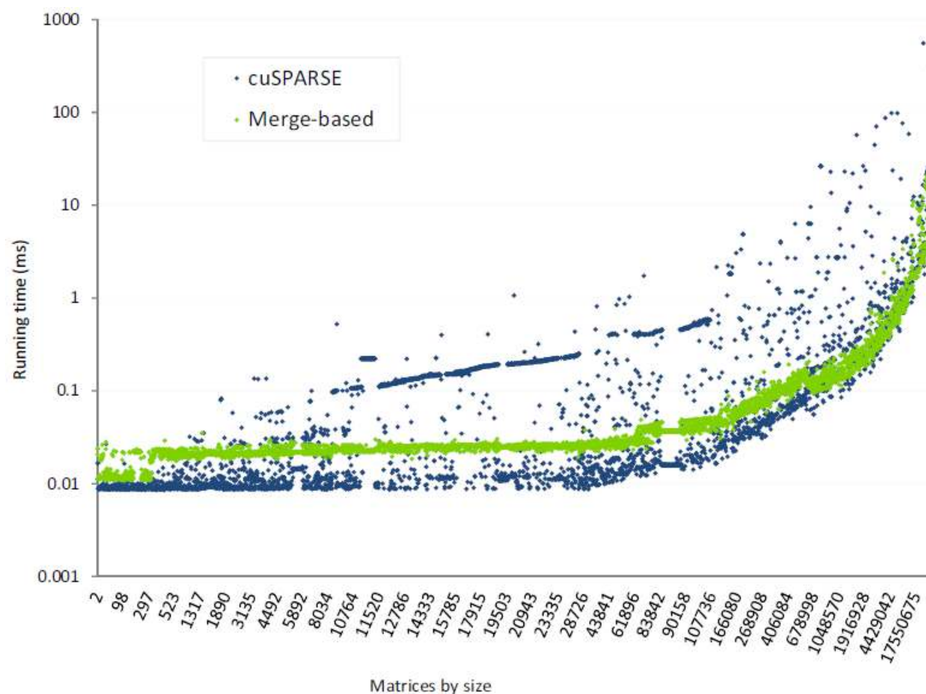


Fig. 2.15 CSRMV-MP vs CUSPARSE on the entire matrix collection of the University of Florida (Merrill and Garland, 2016).

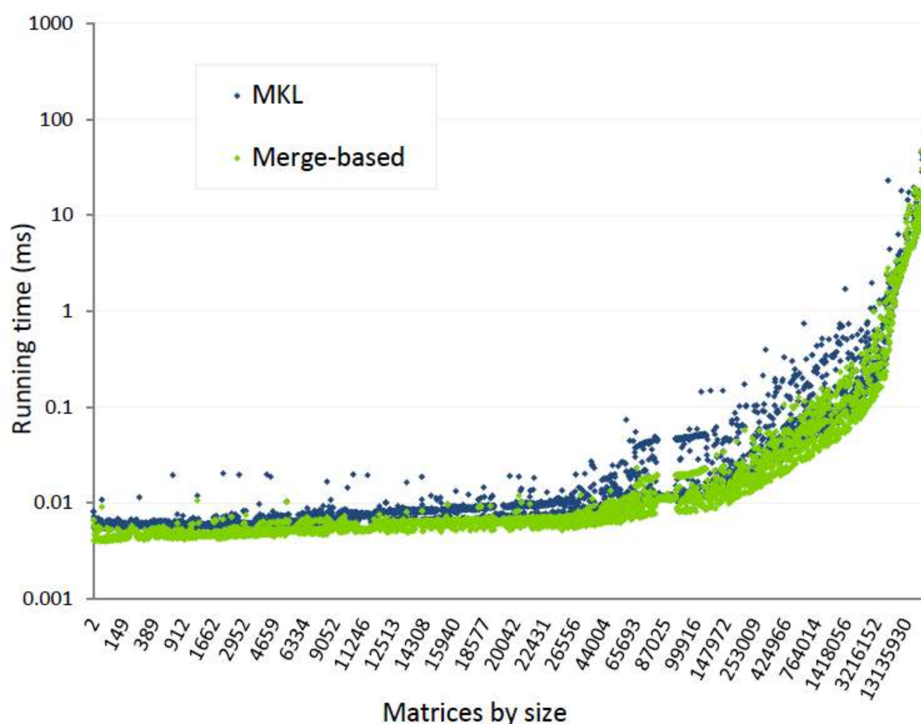


Fig. 2.16 CSRMV-MP vs MKL on the entire matrix collection of the University of Florida (Merrill and Garland, 2016).

Chapter 3

Accelerated multiple implicitly restarted Arnoldi method with nested subspaces

3.1 Introduction

The PageRank and the equilibrium of a Markov chain are famous topics of graph analytics and they lead to large, sparse, nonsymmetric, eigenvalue problems. The PageRank algorithm is used in many applications such as Web (Bryan and Leise, 2006), epidemiology (Liu et al., 2013), or finance (Ermann et al., 2015). The idea is based on a Markov model to represent transition probabilities from one vertex to another, so the stationary distribution (or the equilibrium) represents a steady state in the chain's behavior (Langville and Meyer, 2003). This equilibrium can be seen as a vector where the ith component represents the probability to be in the ith state. In other words, the stationary distribution of a Markov Chain is the vector x such that $A * x = x$, where A the transition matrix. This vector has a large range of applications since it is used to predict the most probable future state based on present observations.

For instance, the graph showed in Figure 3.1 represents frequent co-purchasing of books on Amazon. In Figure 3.2 the PageRank is used to enhance the visualization. In this example, large dark vertices have a high PageRank so we can visually detect popular books that are frequently bought with other books.

In these applications we look for the dominant eigenpair, which can be approximated with methods like the power method as shown in Algorithm 3. We intially implemented a GPU solver for this method and results showed a very fast time per iteration compared to CPU alternatives (Fender, 2014). This is expected since the main cost per iteration is a sparse matrix-vector multiplication which can be efficiently parallelized on GPU. We used this power method solver as a reference to measure improvements offered by Arnoldi methods in the context of accelerated network analysis.

For many real applications the largest eigenvalues are clustered, which slows down the convergence. As an example, with a damping factor close to 1 and a tolerance of 10^{-9} the power method requires hundreds of iterations to converge. Recall that the number of iterations depends on the gap between λ_{max} and the second largest eigenvalue which is bounded by the damping factor $\alpha \in [0,1]$ (Brody, 1997). In this situation more advanced methods,like restarted Arnoldi methods, should be considered in order to obtain the result in reasonable time. We found that even on simple PageRank cases the implicitly restarted Arnoldi method leads to a reduction of the number of matrix-vector multiplications.

In this chapter we present a parallel implicitly restarted Arnoldi solver on the GPU especially designed for fast network analysis. Algorithms and implementations are explained and experiments showed a speedup between $2\times$ and $15\times$ compared to the power method on GPU and an additional improvement up to $8\times$ by using an auto-tuning technique for the subspace size. Our solver is also efficient as we could solve the PageRank problem on networks with hundred millions edges in a few seconds by using a single GPU. The work presented in this chapter has been published as Fender et al., 2016a and Fender et al., 2016b.

3.2 Implicitly restarted Arnoldi solver with PageRank applications on GPU

In Chapter 2 we showed that GPUs have a high parallel throughput and a good power efficiency. Thus, it makes sense to use them for the largest and most computationally intensive part. However, Krylov methods like ERAM and IRAM project the problem

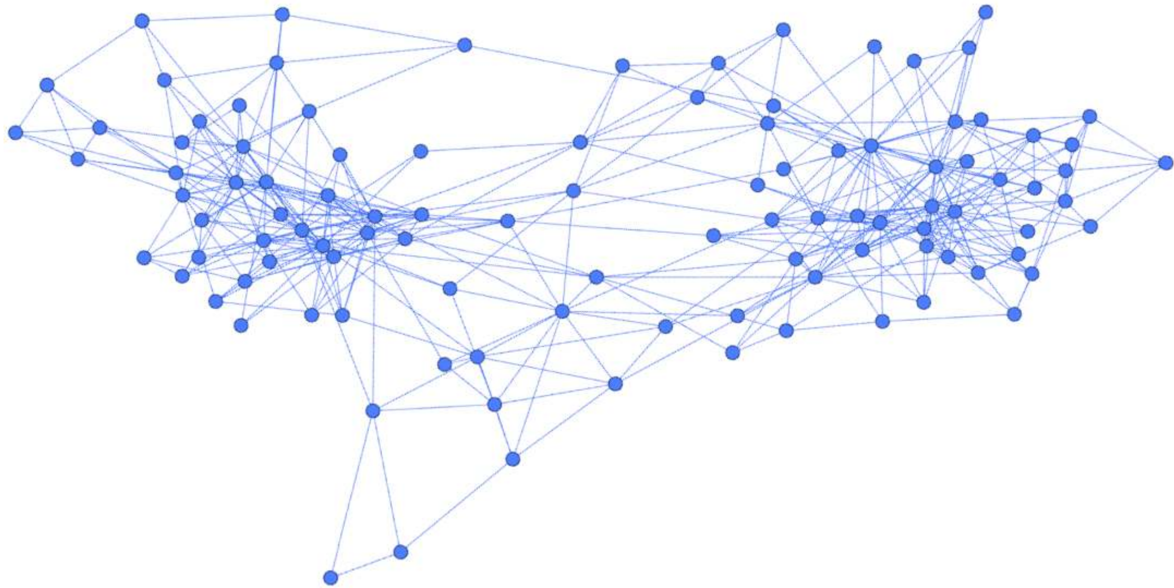


Fig. 3.1 Network representing 105 books sold by Amazon connected by 441 frequent co-purchasing by the same buyers.

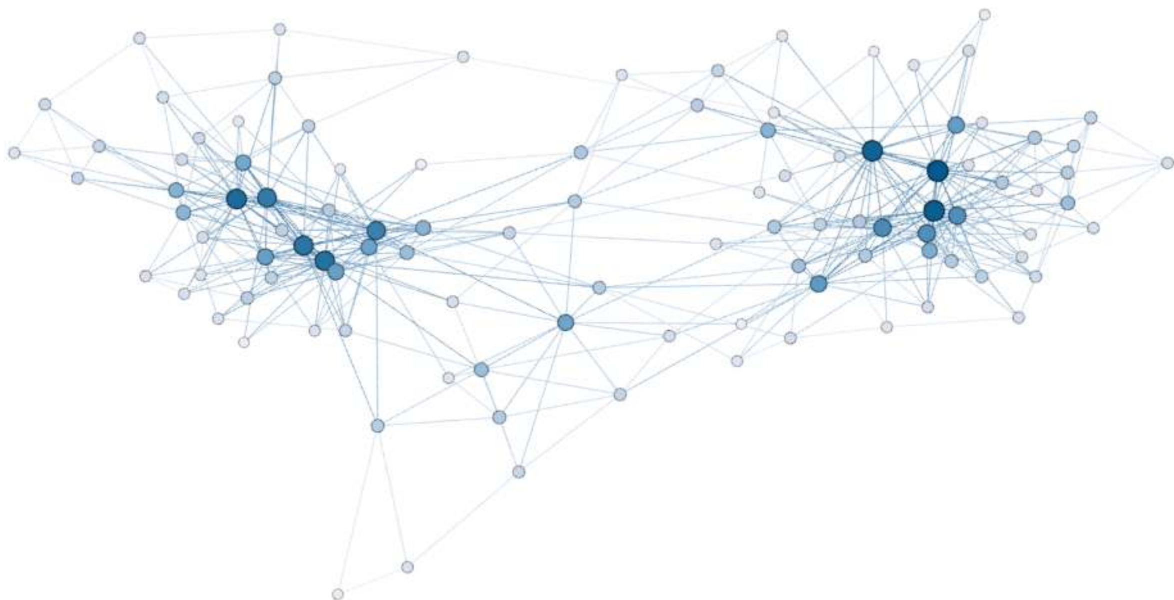


Fig. 3.2 Graph of Figure 3.1 where color and diameter of vertices vary according to the PageRank.

into a small subspace. In the PageRank case a sufficient subspace size is between 4 and 10 (depending on the network), which is too small to fully take advantage of the GPU. Fortunately we can leverage the low latency of the CPU for those operations and build a hybrid strategy based on CPU and GPU cooperation.

The host-device transfers minimization and GPU usage maximization leads to focus on the implicitly restarted Arnoldi method which is known for improving convergence, and thus, overall time of nonsymetric sparse eigenvalue problems (Lehoucq, 1995; Sorensen, 1997, 1998).

In this section we describe a new approach to combine both CPUs and GPUs strengths in order to solve those problems faster and in an efficient way. The concepts presented here can be generalized and transposed to other methods based on projections or coarsening of large sparse problems into small subspaces. We first present the hybrid acceleration of IRAM designed for large networks with power-law distribution and clustered eigenvalues. We explain how famous graph problems such as PageRank directly benefit from this solver and discuss optimizations.

3.2.1 Hybrid GPU approach

In most sparse eigenvalue solvers, the critical part is the parallel sparse matrix-vector multiplication (Bell and Garland, 2008; Greathouse and Daga, 2014).

The SPMV is a memory bounded application, which is good for GPU since the device memory bandwidth is several times larger than the host bandwidth. Nevertheless, accelerating this operation is not straightforward and several points have to be carefully designed, such as the compression format and the load balancing.

The bottleneck of the initial host to device data transfer is not relevant in the context of iterative methods since it is done only once, before the iterative process. Nevertheless, the sparse matrix compression format should still have a good compression rate. Indeed, networks have a very irregular sparsity pattern, so a bad compression can lead to huge memory costs, and thus expensive transfers. On this type of dataset CSR is 30% smaller than COO and 80% smaller than ELLPACK in average (Bell and Garland, 2008). This is the main reason why we chose the compressed sparse row (CSR) format, the other reason being the wide adoption of this format. However, this representation generates an irregular workload for the GPU and introduces important unoccupancy which impacts performances.

Another issue comes from small tasks, which can lead to important loss of resources on GPU. So, if a small task is on the critical path, it will not take advantage of the high throughput of the GPU.

At a high level, a cycle of IRAM is composed by three main steps as shown in Figure 3.3a. First the Arnoldi factorization (Algorithm 4). Second, solving the problem in the small subspace. Third, implicitly update in the original problem or compute the Ritz vectors if converged. This naturally matches a hybrid approach where the GPU would be in charge of steps 1 and 3 and the would CPU handle the second step. Notice that it is possible to hide transfers and latency in both directions by using asynchronous transfers and overlap them with computation. Hence, we propose to combine CPU and GPU to accelerate the implicitly restarted Arnoldi method.

Algorithm 9 Accelerated IRAM

Notation: D stands for *Device* and H for *Host*
Initially: A and v_1 are on D
$[H_m]_H[V_m, f_m]_D = \textit{Arnoldi-factorization}(A, v_1, 1, m)$
while *Convergence is not reached* **do**
  H: *Compute the eigenpairs of* H_m
  H: *Compute the residual and stop if converged*
  H: *Select* $p = m - k$ *shifts* $\mu_1, ..., \mu_p$ *based on unwanted eigenvalues*
  H: $Q_m = I$
  for $j = p, ..., 2, 1$ **do**
    H: $[Q_j, R_j] = \textit{QR-Factorization}(H_m - \mu_j I)$
    H: $H_m = Q_j^H H_m Q_j$
    H: $Q_m = Q_m^H Q_j$
  end for
  Transfer: H_m and Q_m from H to D
  D: $f_m = V_{m_{(:,:)}} Q_{m_{(:,k+1)}} * H_{m_{(k+1,k)}} + f_m * Q_{m_{(m,k)}}$
  D: $V_{m_{(:,1:k)}} = V_{m_{(:,:)}} Q_{m_{(:,1:k)}}$
  $[H_m]_H[V_m, f_m]_D = \textit{Arnoldi-factorization}(A, V_k, k, m)$
end while

Initially, the network is assumed to be in the device memory in CSR representation as shown in Algorithm 9. With recent unified virtual memory (UVM), the graph can also be streamed to the device on-demand as well. Every other data structure of the size of the graph, such as additional vertex or edge information are also on the device. The reason for that is the assumption that host-device and host bandwidth are considerably slower than the device bandwidth, hence, it is primordial to avoid transfers at this scale inside the iterative process.

In Algorithm 9, the SPMV and the Gram–Schmidt process are done on the device, thus we form the matrix V_m from the Arnoldi factorization (Algorithm 4) directly on the device, each column j of V_m corresponds to the result Av_i orthogonalized with the previous vectors. Notice that V_m is a dense matrix which is stored in column major order by nature. The small matrix H_m is formed at the same time directly on the host in column major order as well. The size (m) of this matrix is between 4 and 20 for most graph applications. There is no explicit or blocking transfer of the matrix H_m at this point, since the elements $H_{i,j}$ are the result of $level_1$ operations happening between the SpMVs of the Arnoldi factorization (Algorithm 4). Results are just written on the host one by one without Read After Write issues since H_m is used at the very end of the factorization.

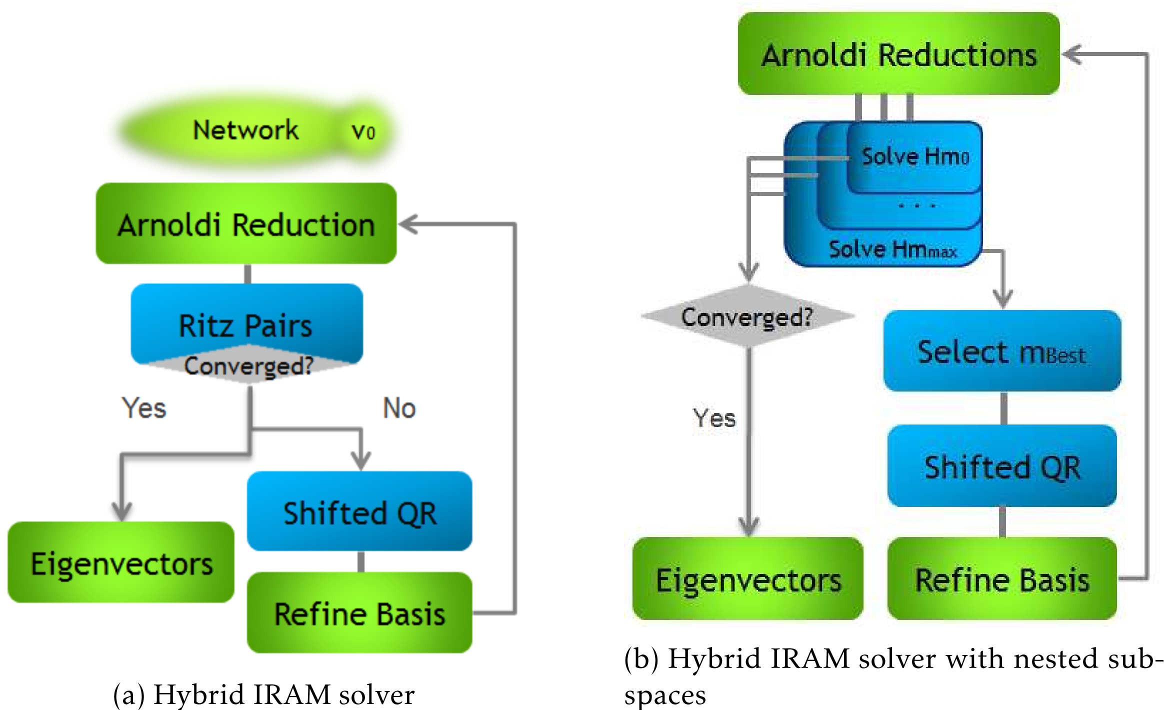
Once the desired size is reached, the eigenpairs of H_m can be computed on the host, we select the unwanted eigenvalues to perform the shifted QR factorizations there. At this point the problem is solved into the subspace and the next step is to update the basis. This is done by using the new matrix H_m^+ and Q_m^+ of Algorithm 9. However, this part involves V_m which is composed by vectors of the size of the graph. This step corresponds to tall skinny dense matrix-matrix multiplications. Fortunately, this type of operation are good use cases for accelerators as shown in Section 2.3.3, plus the large matrix A is already on the device. We only need to transfer the new subspace information which is a couple of matrices of size m.

Finally, we are ready for a new cycle with the same strategy starting by the Arnoldi factorization (Algorithm 4) at the $k+1^{th}$ step.

3.3 Accelerated multiple IRAM with nested subspaces

3.3.1 Enabling nested subspaces in the hybrid IRAM algorithm

The subspace size has an important impact on the performances, however, it is selected empirically in advance in the IRAM method. MIRAMns, generates multiple subspaces in a nested fashion in order to dynamically pick the best one inside each restart cycle. The parallel overall hybrid (host/device) algorithm is described the in Algorithm 10. Device (D) operations are performed in parallel on the throughput oriented architecture, host (H) steps are processed sequentially at high frequency. An illustration is provided in Figure 3.3b where green steps are on the device, and blue ones on the host.



(a) Hybrid IRAM solver

(b) Hybrid IRAM solver with nested subspaces

Fig. 3.3 Accelerated hybrid approach of IRAM and MIRAMns

Algorithm 10 Parallel multiple implicitly restarted Arnoldi with nested subspaces

Notation: D stand for *Device* and H for *Host*
Initially: A and v_1 are on D
$[H_{m_i}]_H[V_{m_i}, f_{m_i}]_D = Arnoldi\text{-}factorization(A, v_1, 1, m_{max})$
while *Convergence is not reached* **do**
  H: *Compute the eigenpairs of* H_{m_i}
  H: *Compute the residuals, stop if converged*
  H: *Select the best subspace size*
  H: *Set* m, H_m, V_m, f_m *accordingly*
  H: *Select* $p = m - k$ *shifts* $\mu_1, ..., \mu_p$
   based on unwanted eigenvalues
  for $j = p, ..., 2, 1$ **do**
   H: $[Q_j, R_j] = QR\text{-}Factorization(H_m - \mu_j I)$
   H: $H_m = Q_j^H H_m Q_j$
   H: $Q_m = Q_m^H Q_j$
  end for
  Transfer: H_m and Q_m from H to D
  D: $f_k = H_m(k+1, k)V_m(1:n, 1:m)Q_m(1:m, j+1) + Q_m(m, k)f_m$
  D: $V_m(1:n, 1:k) = V_m(1:n, 1:m)Q_m(1:m, 1:k)$
  $[H_{m_i}]_H[V_{m_i}, f_{m_i}]_D = Arnoldi\text{-}factorizations(A, V_k, k, m_{max})$
end while

3.3.2 Synchronous auto-tuning

The initial idea of MIRAMns (Shahzadeh Fazeli et al., 2015) is to compute l nested subspaces of different sizes and select the best one for each restart. In this chapter we introduce a way to automatically set the nested subspace sizes based on the number of desired eigenvalues and the maximum subspace size. The smallest subspace size is $k+C$, where C is a constant to make sure the minimum subspace size is large enough to capture the desired eigenvalues. In order to limit the number of subspaces of interest between $k+C$ and m_{max} we perform the quality check at a given frequency ϕ so we look at the subspaces of size $k+C, k+C+\phi, k+C+2\phi, ..., m_{max}$.
For each size we evaluate the quality of the newly generated subspace. This is done by computing the eigenpairs of H_m on the host for the residual approximation. In

the meantime we can continue the next Arnoldi factorization (Algorithm 4) on the device or wait for the residual to perform an early exit if the residual is lower than the tolerance.

When the maximum size m_{max} is reached, we compare all residuals and keep only the best subspace of size m_{select} (with $H_{m_{select}}, V_{m_{select}}, f_{m_{select}}$) on the host and discard other subspaces. Then we proceed exactly like for IRAM, we select the unwanted eigenvalues to perform the shifted QR factorizations on the host. At this point the problem is solved into the subspace and the next step is to update the basis, this is done by using the new matrices $H+$ and $Q+$. Again, since this part involves V which is composed by vectors of size n this step is done on the GPU. The transfers involve the new subspace information which is a couple of matrices of size m_{select}. Finally, we are ready for a new cycle with the exact same strategy, starting by the Arnoldi factorization (Algorithm 4) at the kth step.

3.3.3 Implementation

SpMV

The approach we chose is based on Merrill's solution which efficiently handles arbitrary sparsity pattern based on a 2D merge-path decomposition (Deo et al., 1994; Odeh et al., 2012). This approach presented in Section 2.3.3 is called CSRMV merge-path (Merrill and Garland, 2016). It offers a perfect workload balance on GPU for networks.

Complex eigenvalues

Real non-symmetric matrices often have real eigenvalues but it is very common to have complex conjugate eigenvalues as well. In this case we adjust the shifted QR step of algorithm 9 in order to avoid dealing with complex arithmetic. Typically, if μ_j has an imaginary part we consume two shifts at once (μ_j and μ_{j+1}) and compute the QR factorization of $(H_m - \mathfrak{Re}(mu_j) * I)^2 + \mathfrak{Im}(mu_j)^2 * I$. For this reason, when the user asks for k eigenvalues we actually compute $k+1$ eigenvalues, to make sure we never consume the kth eigenvalue in the double shifted QR (if $k+1$ is a complex conjugate shift).

GPU Memory Manager

GPU memory allocation and deallocation have a cost in term of performance. In methods like IRAM and MIRAMns such operations may happen at each restart. Fortunately the device memory can be managed by the application itself. In other words, instead of explicitly allocating and freeing device memory in the code we take advantage of a memory management layer in charge of managing used and unused memory blocks on the device. Hence, a large block of memory can be allocated once during the initialization of the library and distributed in an on-demand fashion by a memory manager. This manager also has the ability to allocate more GPU memory if needed.

Unified Virtual Memory

For a long time the user had to explicitly transfer the data to the accelerator and the size of the data set was bounded by the device memory capacity. With recent NVIDIA's GPU architectures and CUDA it is now possible to use a unified virtual memory address space between host and device. It is based on a page fault mechanism between the CPU and the GPU. Data are automatically properly transferred when accessed either from host or device. This enables GPU over-subscription and out-of-core algorithms to process graphs that could not fit into the local memory a single GPU.

PageRank support

Our implementation of the IRAM solver can find the equilibrium of a Markov chain or compute the PageRank. Based on the formulation of Section 2.2.1, we added support for the damping factor and the rank-one update in order to artificially represent the Google matrix (Eq. 2.5) during the Arnoldi-factorization (Algorithm 4). Hence, the largest eigenpair of the generated subspace is the equilibrium.

Design

The solver is implemented in an experimental branch of nvGRAPH, which is a CUDA library at Nvidia. It is written in C++/CUDA with C API. For many low level primitives on the GPU we leverage other parts of the CUDA Toolkit libraries such as Cublas, Cusparse and Thrust (NVIDIA, 2017). On the host we used Lapack for QR factorizations. Internally the code leverages object oriented design. Each step of the solver is independent and was validated and tested separately. We built a test suite based on

GTEST, a Google open-source framework for C++ tests.

Parameters

The IRAM solver takes a graph and an initial guess (which can be random) as input. The user also provides the desired Krylov subspace size and the number of desired eigenpairs. As output, the solver returns those eigenpairs.
The solver can artificially represent Google matrices to solve the PageRank problem. This mechanism can also be applied to guaranty a unique eigenvector corresponding to the largest eigenvalue of a stochastic matrix (Krieger, 1974). In this case the user should provide the bookmark of dangling nodes and the damping factor (ie. the probability to follow each edge). Internally this has almost no computational cost since it is done on the device during the Arnoldi factorization with the rank-one update. Finally, it is possible to print statistics like time, number of iteration, memory used, best Krylov subspace size and residual at each cycle.

3.3.4 Distributed considerations

Multiple asynchronous solvers

It is possible to take advantage of multiple devices by running an instance of a solver on each GPU. A different initial guess and various nested subspaces sizes should be used for each instance. Increasing the number of instances also increases the chances to find better parameters. Thus, the time to find a solution can be reduced. Strong scaling can be achieved with this technique for data sets of moderate size which can fit in a single device.

Distributed synchronous auto-tuning

The primary concern of a distributed version is to handle problems that cannot fit into the memory of a single device. In order to bypass this limitation, the problem can be split into several partitions. Still, the multi-GPU implementation of an implicitly restarted eigensolver will operate on data of different orders of magnitudes at all times.

Large operations involving the size of the graph are done in parallel on the device. There is one sparse matrix vector multiplication (spmv) per iteration and it is the most expensive step, since it is the only one involving the matrix. The performance

of this operation highly depends on how the matrix has been distributed among accelerators (eg. simple, 2D partitioning, random, according to a specific optimized partition vector, etc.). This partitioning directly impacts the implementation of the SPMV because, for instance, the factorization step is done differently depending on the distributed data layout.

Other operations of large size involve dense data. They are dot, axpy and gemm (tall-skinny). When targeting a large number of accelerators each operation should be replaced by its distributed version as well. In a single device those operations are quickly done. However, if the data are distributed across multiple devices, intra-node communications and host-device transfers will become increasingly expensive. This should be optimized as much as possible because those operations are called many time per iteration and can potentially end by dominating the total execution time.

Small operations involving the size of the subspace are done on the host. The reduced problem is too small to benefit from accelerated or distributed computing. Hence, it should be gathered on a single host in order to be solved, then the result can be sent to all nodes and a new cycle can start. The communication cost should be relatively low as long as the number of partitions is reasonable. The only problematic part being the implicit synchronization before each restart. As a result, the solution is obtained only when the slowest node completed all tasks.

3.4 Experimental results

We implemented and optimized the power method on GPU in order to have a fair and useful way to estimate the speedup offered by the hybrid IRAM solver. Based on that we can also quantify the improvements offered by the hybrid MIRAMns solver.

3.4.1 Power method on GPUs

The experiment was performed on a CentOS 7.2, x86-64, with 128GB of memory. We implemented the power method in nvGraph (NVIDIA, 2017) as a PageRank solver and ran it on K80, M40, P100 with ECC ON and using an Intel Xeon Haswell single-socket 16-core E5-2698 v3, base clocks. We ran GraphMat (Sundaram et al., 2015) and Galois (Nguyen et al., 2013) in parallel on Intel Xeon Broadwell dual-socket 44-core E5-2699

v4 @ 2.22GHz, 3.6GHz Turbo which was the most recent CPU architecture at the time we did the experiment. We selected a relevant sample of graphs from the University of Florida collection (Davis and Hu, 2011), shown in Table 3.1.

Name	$n = \|V\|$	$m = \|E\|$	Application
web-BerkStan	685230	7600595	Web
web-Google	916428	5105039	Web
Webbase-1M	1000005	3105536	Web
Wiki-Talk	2394385	5021410	Comm
ctiPatents	3774768	16518948	Citations
soc-LiveJournal	4847571	86220856	Social
Live Journal	5363260	79023142	Social
Twitter	41652230	1468365182	Social

Table 3.1 General information on networks

In Figure 3.4 we compare nvgraph (accelerated power method) to famous competitors which are GraphMat (Sundaram et al., 2015) and Galois (Nguyen et al., 2013). We compare the average time per iteration. Galois and GraphMat run in parallel on all available cores of the machine (2 sockets with 44 CPU threads). We run nvgraph on Maxwell (M40) and Pascal (P100 PCIe). In nvgraph, the input and output data are assumed to be on the device. The graph is transferred only once in an other API call which allows applications to chain multiple nvgraph algorithms without re-transferring the graph. For the Pagerank case, the transfer overhead is diluted over a large number of SpMV operations, rendering it negligible.

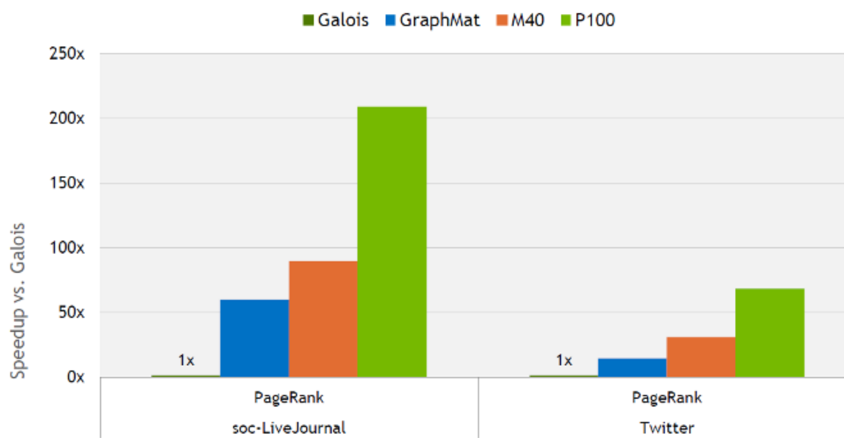


Fig. 3.4 Speedup of the power method implementation on GPU vs. CPU (parallel) for PageRank applications

First, notice that the speedup over Galois is better on live journal than on Twitter. This is because Galois performs better on larger data sets.

Second, GraphMat outperforms Galois by 5× to 50×. A reason for that is because Galois targets more general purpose parallel computing while GraphMat is an optimized software by Intel for graph analysis.

Third, notice that graphs as large as `Twitter` with 1.4 Billion edges can be handled by a single GPU with nvgraph's power method.

Fourth, nvgraph outperforms Galois by 200× and 70× on `soc-LiveJournal` and `Twitter` respectively. It also outperforms GraphMat by 4× and 5× on `soc-LiveJournal` and `Twitter` respectively.

As the cost of one iteration is dominated by the cost of the sparse matrix vector multiplication, the speedup is mostly resulting from the faster sparse matrix vector multiplication on GPU than on CPU.

Since our implementation of the power method in nvgraph is the fastest implementation, it will be our reference baseline for further comparison against the implicitly restarted Arnoldi method.

Figure 3.5 shows PageRank performances on GPU on the three latest Nvidia architectures : Kepler (K80), Maxwell (M40) and Pascal (P100 PCIe).

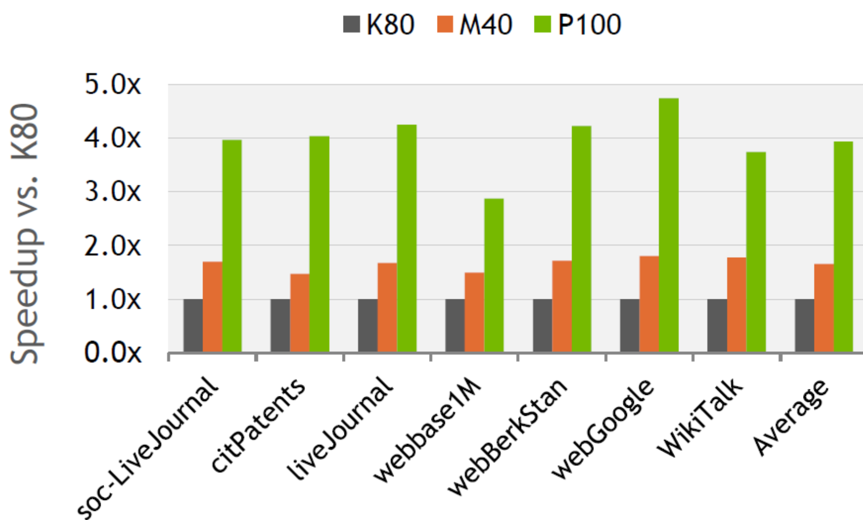


Fig. 3.5 Speedup of the power method on several GPU for PageRank applications

Graph	Type	V	E	Power SPMV	T(s)	IRAM SPMV	T(s)	Speedup
com-Youtube	Social	1,134,890	5,975,248*	65	0.20	22	0.09	2.13
rmat-1M	Artificial	1,000,000	41,237,691	18	0.41	10	0.24	1.74
wiki-2011	Citations	3,721,339	66,454,329	40	1.85	25	1.26	1.47
com-Orkut	Social	3,072,441	234,370,166*	46	6.37	25	3.55	1.79

Table 3.2 IRAM solver on GPU vs. power method on GPU in PageRank applications ($\alpha = 0.85$, $m = 4$, 64 bit precision)

Notice that P100 is 2.3× faster than M40 and 4× faster than K80 while the speedup between M40 and K80 is 1.7×. This trend seems to indicate that GPU architectures are becoming increasingly better for graph analysis workloads.

3.4.2 Implicitly restarted Arnoldi for networks on GPU

We use an NVIDIA Tesla K40m for this experiment with the driver version 352.68 and CUDA7.5. The test machine OS is RedHat6.5 and is equipped with an Intel Xeon CPU E5-2698 v3 at 2.30GHz and 256GB of memory. In Table 3.2 we selected real networks from SNAP collection (Leskovec and Krevl, 2014) and KONECT (Kunegis, 2015), the star (*) on undirected graphs indicates that an undirected connection is represented by a directed edge in each direction. Column V represents the number of vertices (size of the matrix) and column E shows the number of edges of the graph (ie. the number of non-zero entries in the matrix). `com-Youtube` and `com-Orkut` are social networks where vertices are users and edges are friendship links. `Rmat-1M` is an artificial graph generated using Boost Graph Library RMAT generator, and `wiki-2011` represents the citations (edges) between Wikipedia articles (vertices) (Eom et al., 2013).

Accelerated IRAM versus power method

In general, PageRank is a good use case for the power method solver, especially with a reasonably low damping factor and in single precision mode. Nevertheless, IRAM performs better in Table 3.2, with an average speedup of $1.8x$ for a standard damping factor of 0.85. There are simple cases where the power method can beat the hybrid IRAM solver in term of total time, for example if the two dominant eigenvalues are not clustered (at least $|\lambda_1 - \lambda_2| > 10^{-1}$) and if we do not care much about the quality of the approximation (tolerance under 10^{-4}).

Typically, the power method can reach the desired tolerance in less than 9 iterations on these cases. Ideally, a general adaptive solver could dynamically choose between the two solvers based on input information.

In the PageRank example, the damping factor has a direct impact on the gap between the two largest eigenvalues, and thus on the convergence rate. We use this property to show the speedup variation between IRAM and the power method in Figure 3.6. Notice that when the problem becomes harder to solve, the IRAM solver performs better by dividing the number of SPMV by 18.3× in average when eigenvalues are clustered (eg. $\alpha = 0.999$). For those cases the resulting average speedup in term of time is 14.4×, compared to the power method on GPU.

The difference between the reduction of the number of SpMV and the actual time speedup corresponds to the cost of the implicit restarts of IRAM.

The choice of the subspace size can be independent of the graph size, for the experiment in Figure 3.6 the subspace size is always 4, which means that the theoretical cost for solving the problem in the subspace is independent of the size of the network. Hence, increasing the network size tends to dilute the cost of the implicit restarts. For example, on a small graph like `com-Youtube` the speedup is 28.08% under the reduction of the number of SpMVs, while on com-Orkut, the largest data set of Table 3.2 , this difference is only about 2.46%.

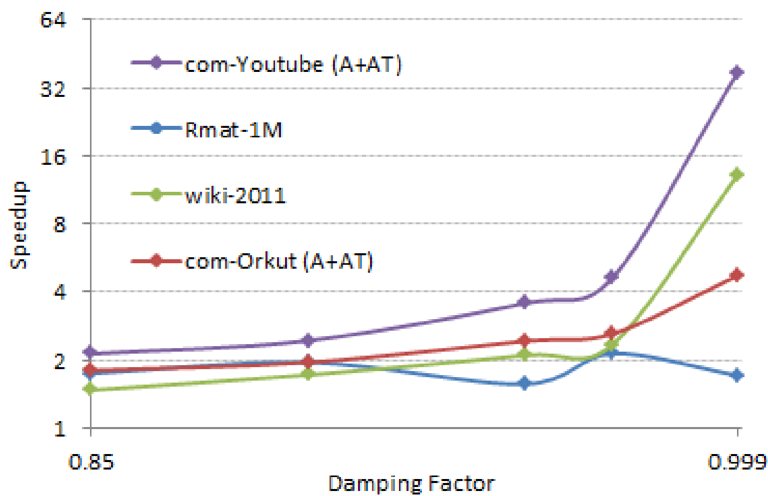


Fig. 3.6 Speedup of IRAM on GPU vs. power method on GPU with different damping factors

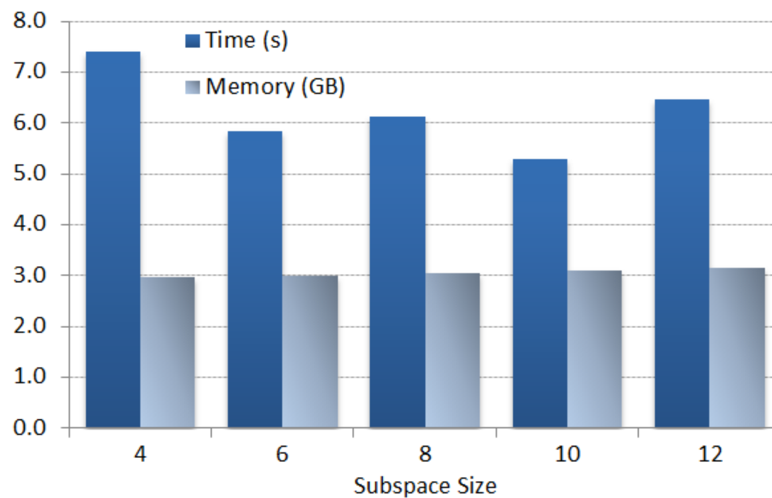


Fig. 3.7 Impact on time and memory when changing the Krylov subspace size on com-Orkut

Figure 3.8 shows a time profile of IRAM on the GPU (Algorithm 9), when looking for one eigenvalue in a subspace of size four on `wiki-2011`. This profile shows that most of the time is spent in the CSRMV ($37ms$ per CSRMV in average in this example). Notice that level 1 BLAS routines such as AXPY, DOT and SCAL are cheap compared to the CSRMV. At the end of each cycle there is a tall skinny matrix multiplication for the projection. In this PageRank example, we are looking for one eigenvector so the final projection is only one tall-skinny matrix-vector multiply. Transfers of Algorithm 9 take one to five micro-second each, and are too small to be visualized on Figure 3.8.

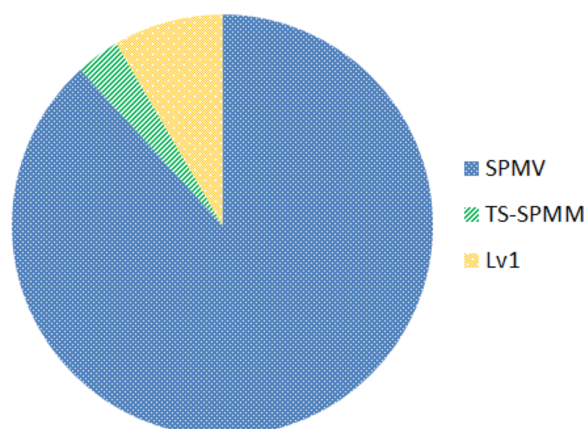


Fig. 3.8 Profiling of IRAM on GPU

3.4.3 Accelerated multiple IRAM with nested subspaces

We validated the results by comparing to the CPU reference code of this method in Matlab, we were also able to check the results of the accelerated version of IRAM and MIRAMns on the af23560 matrix by comparing with results from the literature.

The matrix af23560 (Table 3.3) is non-symmetric and comes from a computational fluid dynamics problem. In addition, we selected three large matrices from (Davis and Hu, 2011) and made sure we always converged to the same eigenvalues with both methods. Cage14 and cage15 (Table 3.3) are non symmetric matrices with real values from DNA electrophoresis and Hook1498 is symmetric, from a 3D mechanical problem. The tolerance of the quality of the approximation is $1E{-}12$ for the following experiments. Initial vectors are always identical.

Name	N	NNZ	Application
af23560	23,560	484,256	CFD
Hook1498	1,498,023	59,374,451	Structural
cage14	1,505,785	27,130,349	DNA electrophoresis
cage15	5,154,859	99,199,551	DNA electrophoresis

Table 3.3 General information on matrices

Comparison between two IRAM and MIRAMns on GPU

The SPMV is the most expensive operation per cycle, thus the number of SPMV (or the number of restart cycles, which contain a fixed number of SPMV) is a generic metric to measure the efficiency of Krylov methods. We also compare the total time since both solvers has host-device transfers inside each cycle.

MIRAMns requires to transfer p subspaces per cycle so it is critical to make sure that this extra cost is balanced by the improvement offered by the reduction of the total number of SPMV. In order to capture the same subspace information, we compare MIRAMns $(m_{k+C}, m_{k+C+\phi}, ..., m_{max})$ against IRAM(m_{max}).

Figure 3.9 shows ratios of speedup (dark blue) and reduction of the number of cycles (light blue) of MIRAMns vs. IRAM. For each matrix we have results for $k = 2$ and $k = 4$, $m = 25$. Half of our experiments show a speedup from 2× and 8×, directly related to the reduction of the number of restart cycles.

On Figure 3.9, the case Hook1498_2 illustrates an interesting detail of MIRAMns solver. Notice that, MIRAMns does not reduce the number of cycles but there is still a small speedup. What we see is a successful early exit of MIRAMns because a smaller subspace than m_{max} converged. This explains why sometimes the speedup can be slightly higher than the ratio of the reduction of the number of restarts.

Notice that as the speedup comes closer to the factor of saved cycles, the size of matrix increases. The solver is able to compute the 4 dominant eigenvalues of af23560 in $0.07s$, Hook1498 in $1.95s$, cage14 in $43.6s$, and cage15 in $235.6s$. Recall that cage15 has almost 100 million entries, the matrix market file is about $2.6GB$ and the device memory requirement of the accelerated solver to process it in double precision is only $2.9GB$.

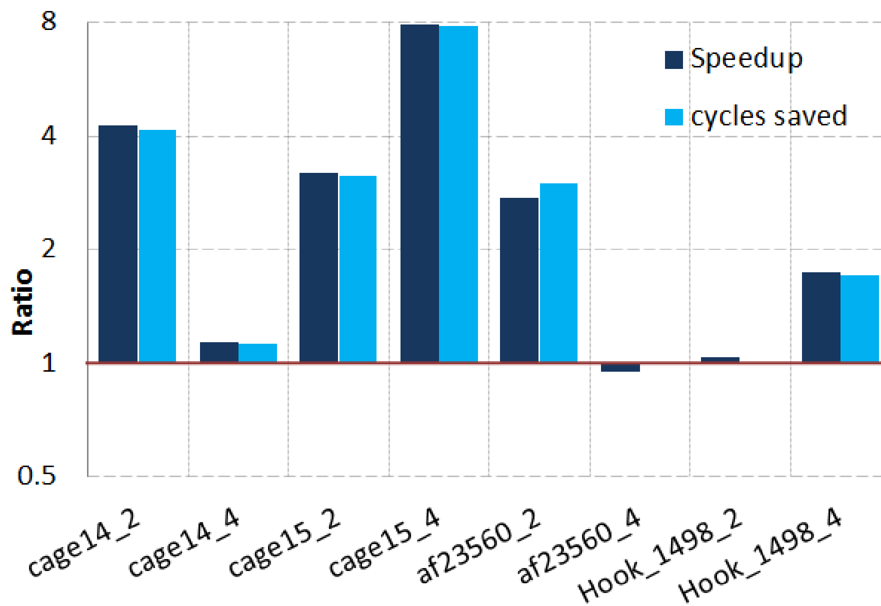


Fig. 3.9 Speedup and cycles saved in MIRAMns vs. IRAM.

Convergence and stability

MIRAMns involves many parameters such as the number of wanted eigenpairs, the size of the largest subspace and the number of subspaces. Different parameters can lead to significant differences in terms of time for the same data set. For example cage15 takes 489 restarts to converge when looking for 2 eigenvalues, but by changing the number of wanted eigenvalues to 8 (without changing any other parameter), the number of restarts rises over 2000. Figure 3.10 shows the stability of the solver at

that scale, and after so many restarts during a long execution ($matrix = cage15$, $k = 8$, $m = 25$).

In Figure 3.10 both methods struggle to converge. However, notice that there are spikes breaking stagnation in MIRAMns while some divergence phases happen in IRAM. Also, notice the final overtaking of MIRAMns while IRAM starts to show higher signs of divergence.

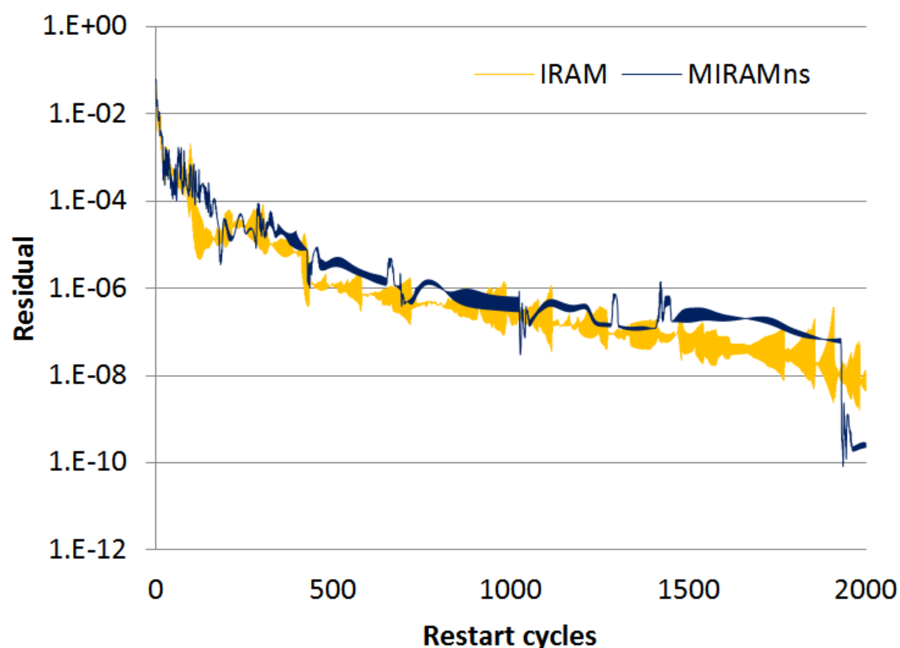


Fig. 3.10 Comparison of the residuals of MIRAMns and IRAM on Cage15.

In Figure 3.11 MIRAMns is 2.7x faster by making active use of different subspace sizes. It shows that MIRAMns starts to use smaller subspaces from the beguning, leading to a better residual. It also avoids bad spikes that happen in IRAM. In the end, MIRAMns solver converges in 9 cycles instead of 30 for IRAM.

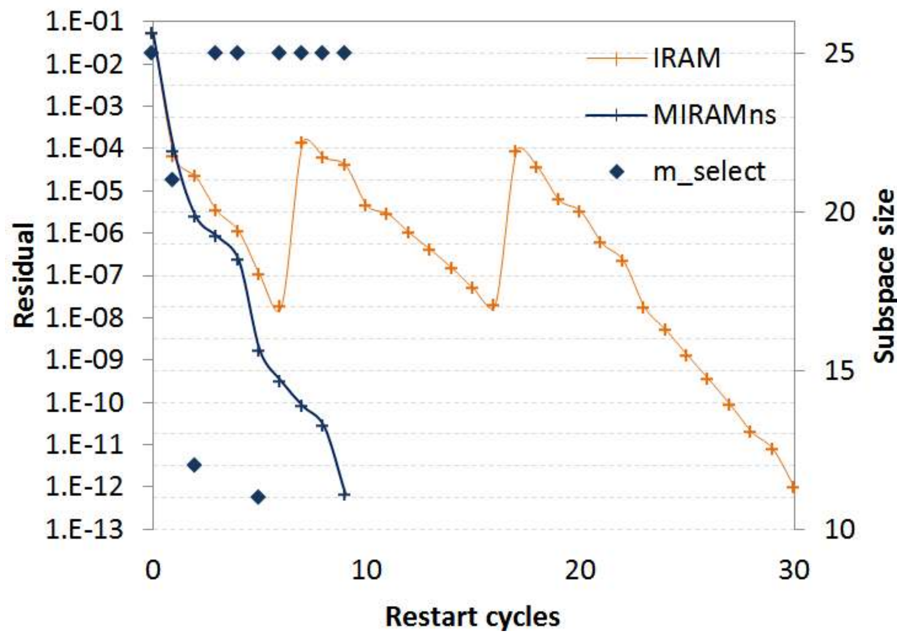


Fig. 3.11 Selected subspace size in MIRAMns vs. IRAM on af23560

Variation of the number of nested subspaces

By only changing the frequency that controls the number of subspaces of interest, it is possible to increase or decrease the number of subspaces between m_{k+C} and m_{max}.

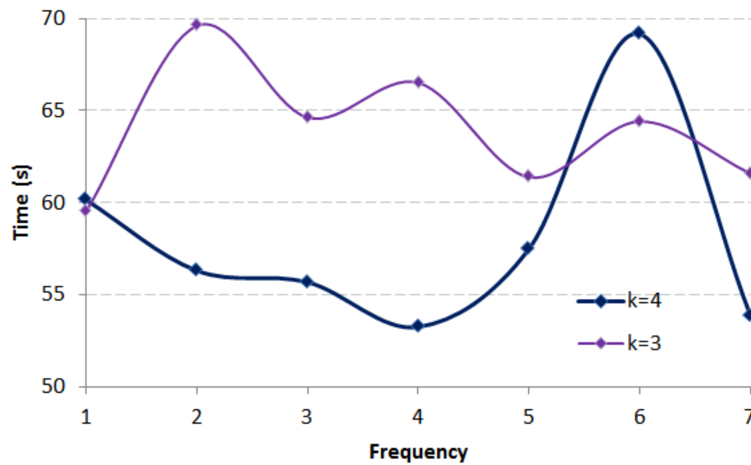


Fig. 3.12 Variation of the subspace frequency in MIRAMns on cage14

Typically, we would expect that the more subspaces we have, the better it should be to find the one with the lowest residual. However, it requires to compute the residual of more subspaces and has a cost in term of time. We also found that a high number of

subspaces does not always lead to a smaller number of restart cycles (and thus better execution time). Figure 3.12 illustrates this phenomenon. Results do not seem to follow a precise rule but we can see that more subspaces (ie. low frequencies) do not lead to better results.

3.5 Conclusion and future works

We presented the first hybrid, accelerated, MIRAMns solver that leverages GPUs. It makes use of the high GPU throughput for large sparse operations without suffering from under-occupancy by solving small coarse problems on the CPU. The hybrid strategy can benefit to other methods with reduction-projection patterns. We also discussed collateral problems we had to deal with during the implementation of this solver such as the load imbalance and the host device memory management.

We showed that the accelerated MIRAMns solver competes against other accelerated eigenvalue methods that are known to be efficient on GPU architectures. The power method is 2x faster on a single GPU (1 Nvidia Tesla K20) than on CPU (Intel's MKL on 32 cores of a dual-socket Xeon Haswell CPU).

The IRAM solver is between 4x and 10x faster in average than the accelerated power method on GPU.

The MIRAMns solver is up to 8x faster than the IRAM solver on GPU.

In addition, our experiments lead to new ideas for improving the auto-tuning in MIRAMns. For instance, the number of subspaces of interest can have an important impact on the performances of the method. Indeed, a high number of subspaces is not always the best strategy and a too low number either. In the future, we believe that MIRAMns can be improved by adapting the number of nested subspaces.

Chapter 4

Spectral modularity clustering

4.1 Introduction

In this chapter we develop a novel parallel approach for computing the modularity clustering often used to identify and analyse communities in social networks.

An illustration of graph clustering is showed in Figures 4.1 and 4.2. As seen in the previous chapter, Figure 3.2 represents frequent co-purchasing of books on Amazon.com. In Figure 4.1 vertices have been given colors pink, yellow, or green to indicate whether they are liberal, neutral, or conservative. These clusters were assigned based on a reading of the descriptions and reviews of the books posted on Amazon. In Figure 4.2 vertices are colored according to their cluster found by our spectral modularity maximization. There is a 84% hit rate compared to Figure 4.1.

In this chapter, we show that modularity can be approximated by looking at the largest eigenvalue and eigenvector of the weighted graph adjacency matrix that has been perturbed by a rank one update. Also, we generalize this formulation to identify multiple clusters at once. We develop a fast parallel implementation for it that takes advantage of the Lanczos eigenvalue solver and k-means algorithm on the GPU. Finally, we highlight the performance and quality of our approach versus existing state-of-the-art techniques.

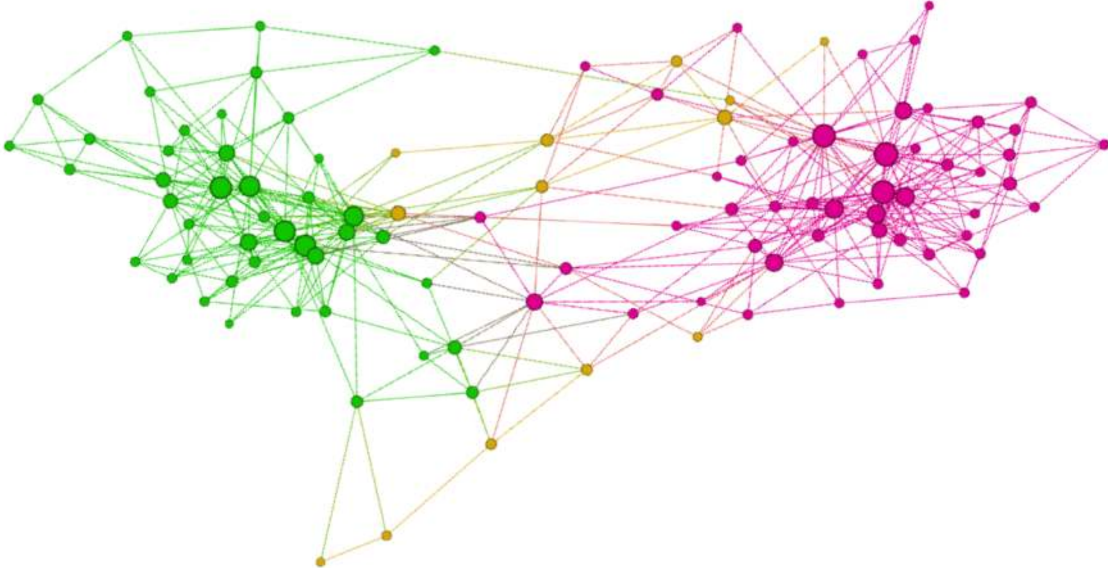


Fig. 4.1 Same graph as in Figure 3.2 where vertices are coloured according to their ground truth cluster.

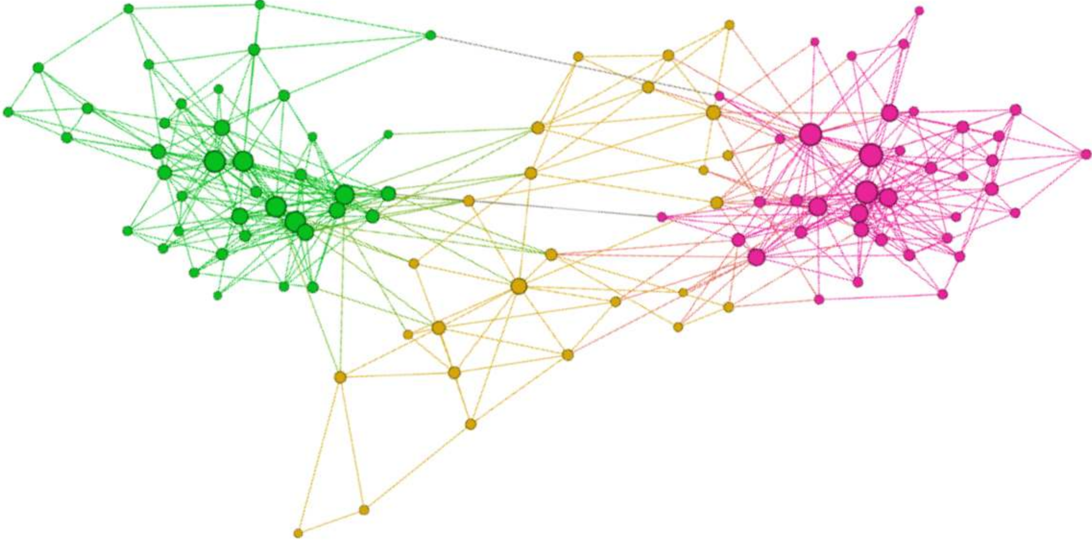


Fig. 4.2 Same graph as in Figure 3.2 where vertices are coloured according to their cluster found by our spectral modularity maximization.

We focus on the mathematical theory behind modularity. We use it to setup an approximate continuous eigenvalue based solution and show how to use the k-means algorithm to transform a continuous solution back into a discrete one, which is our clustering solution. We follow an outline similar to the approach developed for minimum cut partitioning in (Naumov and Moon, 2016).

We show how to generalize the method to work with weighted graphs and to identify a set of fixed multiple clusters at once. Notice that in our approach the number of fixed clusters p is arbitrary. It does not need to be a power of two, i.e. $p = 2^k$, as when repeated bisection is used k times. Moreover, we outline how the proposed approach could be modified to find an adaptive number of clusters.

Also, we analyse the effects of using single and double precision to solve the problem. Moreover, we show that in our approach the clustering information could be derived from the smaller, same or larger number of eigenvectors, with the former case exchanging lower quality for higher performance.

Finally, in our numerical experiments we compare the modularity and minimum balanced cut metrics on the clustering obtained by the random, spectral and modularity approach developed in this chapter. We comment on the quality and performance tradeoffs when they are applied to large social network graphs that often have power law-like distribution of edges per node. We highlight the impressive performance obtained by our novel parallel approach on the GPU. For example, it can find 7 clusters with a modularity score over 0.5 in about 0.8 seconds for `hollywood-2009` network with over a hundred million undirected edges using a single GPU.

The work presented in this chapter has been published as Fender et al., 2017a.

4.1.1 Modularity

Let a graph $G = (V, E)$ be undirected with $w_{i,j} \equiv w_{j,i}$ and consequently matrix A is symmetric. If it is not, we can always work with $\tilde{G}$ induced by $A + A^T$. Also, we assume that we do not include self-edges, diagonal elements, in the definition of the weighted adjacency matrix A.

Let two disjoint set of vertices S and $T \subseteq V$, then let us define a cut $C = (S, T)$ of a graph to be a partition of vertices V into these sets. In graph clustering we are often interested in finding a cut C such that it minimizes or maximizes a particular metric, such as modularity.

In the following discussion, let $|.|$ denote cardinality (number of elements) of a set and d_i denote the degree (number of edges) of the vertex $i \in V$. Also, let us define the volume of a node v_i to be

$$v_i = \sum_{j=1}^{n} a_{i,j} \tag{4.1}$$

and volume of a set of vertices vol(V) to be

$$\mathrm{vol}(V) = \sum_{i=1}^{n} v_i = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{i,j} = 2\omega \tag{4.2}$$

Notice that for unweighted graphs $a_{i,j} = 1$ and therefore $v_i = d_i$ and $2\omega = 2m$. An intuitive way to identify structure in a graph is to assume that similar vertices are connected by more edges in the current graph than if they were randomly connected. The modularity measures the difference between how well vertices are assigned into clusters for the current graph $G = (V,E)$, when compared to a random graph $R = (V,F)$ (Newman, 2006, 2010; Newman and Girvan, 2004).

The reference random graph $R = (V,F)$ is constructed with the same set of vertices, but a different set of edges as the current graph. The set of edges F of the random graph is constructed such that the number of edges $|F| = |E| = m$ and degree d_i of each vertex is the same, but the edges themselves are rewired randomly between vertices in V.

Notice that every broken edge, generates two edge ends that are available for rewiring, as shown on Fig 1. Then, the weighted probability of a particular edge end to be connected with some edge end at node i is $v_i/2\omega$. Therefore, the probability of node i and j to be connected during the rewiring is $(v_i v_j)/2\omega$.

The modularity is the difference between existing edges and the probabilities of edges in random graph across all nodes that belong to a given set of clusters.

Definition 1. *Let graph $G = (V,E)$ and $c(i)$ be an assignment of nodes into clusters. Then, modularity Q can be expressed as*

$$Q = \frac{1}{2\omega} \sum_{i=1}^{n} \sum_{j=1}^{n} \left(a_{i,j} - \frac{v_i v_j}{2\omega} \right) \delta_{c(i),c(j)} \tag{4.3}$$

where

$$\delta_{c(i),c(j)} = \begin{cases} 1 & \text{if } c(i) = c(j) \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

The above definition can be reduced to the special case in (Newman, 2006, 2010; Newman and Girvan, 2004) if we choose to ignore the edge weights during rewiring or work with unweighted graphs, in which case $\frac{v_i v_j}{2\omega} = \frac{d_i d_j}{2m}$. Notice that modularity is bounded.

Lemma 1. *The modularity Q is between*

$$-\frac{1}{2} \leq Q \leq 1 \tag{4.5}$$

Proof. See Lemma 1 in (Brandes et al., 2008). □

Let us now define the modularity matrix, state its properties and show its relationship to modularity metric.

Definition 2. *Let the volume vector be $\boldsymbol{v}^T = [v_1, ..., v_n]$, then the modularity matrix can be written as*

$$B = A - \frac{1}{2\omega} \boldsymbol{v}\boldsymbol{v}^T \tag{4.6}$$

Lemma 2. *The modularity and adjacency matrices have the following properties*

$$B\boldsymbol{e} = \boldsymbol{0} \tag{4.7}$$

$$A\boldsymbol{e} = \boldsymbol{v} \tag{4.8}$$

$$\boldsymbol{v}^T\boldsymbol{e} = \boldsymbol{e}^T A\boldsymbol{e} = 2\omega \tag{4.9}$$

where $\boldsymbol{e} = [1, ..., 1]^T$.

Proof. The latter two follow from (4.1) and (4.2). The former follows from the latter two and $B\mathbf{e} = A\mathbf{e} - (\frac{\mathbf{v}^T\mathbf{e}}{2\omega})\mathbf{v} = 0$. □

Notice that the modularity matrix B is symmetric indefinite. Indeed, according to Lemma 2 it is singular, with an eigenvalue 0 and corresponding eigenvector $\mathbf{e} = [1, ..., 1]^T$.

Let us now assume that we are working with many clusters at once. Then, let us define a tall matrix $U = [u_{i,k}]$, that can be interpreted as a set of vectors $U = [\mathbf{u}_1, ..., \mathbf{u}_p]$ where each vector $\mathbf{u}_k$ corresponds to a cluster S_k for $k = 1, ..., p$, with elements

$$u_{i,k} = \begin{cases} 1 & \text{if } c(i) = k \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

Theorem 1. *Let the matrix $U = [\boldsymbol{u}_1, ..., \boldsymbol{u}_p]$ be defined in (4.10). Then,*

$$\mathcal{Q} = \frac{1}{2\omega} Tr(U^T B U) \tag{4.11}$$

where $Tr(.)$ is the trace (sum of diagonal elements) of a matrix.

Proof. Notice that

$$\begin{aligned} \mathcal{Q} &= \frac{1}{2\omega} \sum_{k=1}^{p} \sum_{\substack{\forall i \text{ s.t.} \\ c(i)=k}} \sum_{\substack{\forall j \text{ s.t.} \\ c(j)=k}} \left(a_{i,j} - \frac{v_i v_j}{2\omega} \right) \\ &= \frac{1}{2\omega} \sum_{k=1}^{p} \left(\mathbf{u}_k^T B \mathbf{u}_k \right) \\ &= \frac{1}{2\omega} \text{Tr}(U^T B U) \end{aligned} \tag{4.12}$$

with elements of U constrained to be in set $\mathcal{C} = \{0, 1\}$. □

Notice that ultimately we are interested in finding the cluster assignment c that achieves the maximum modularity

$$\max_c \mathcal{Q} = \frac{1}{2\omega} \max_{U \in \mathcal{C}} \text{Tr}(U^T B U) \tag{4.13}$$

The exact solution to the modularity maximization problem stated in (4.13) is NP-complete (Brandes et al., 2008). However, we can find an approximation by relaxing the requirement that elements of matrix U take discrete values (Naumov and Moon, 2016; Von Luxburg, 2007).

Notice that $U^T U = D$, where $D = [d_{k,k}]$ is a $p \times p$ diagonal matrix with elements $d_{k,k} = |S_k|$. Then, introducing the auxiliary matrix $\widetilde{U} = U D^{-1/2} \in \mathbb{R}^{n \times p}$, we can start by looking for

$$\max_{\widetilde{U}^T \widetilde{U} = I} \text{Tr}(\widetilde{U}^T B \widetilde{U}) \tag{4.14}$$

Notice that by the Courant-Fischer theorem (Horn and Johnson, 1986) this maximum is achieved by the largest eigenpairs of the modularity matrix. Now, we still need to convert the real values obtained in (4.14) back into the discrete assignment into clusters.

Since we are working in multiple dimensions, it is natural to use the distance between points as a metric of how to group them. In this case, if we interpret each row of the matrix U as a point in a p-dimensional space then it becomes natural to use a clustering algorithm, such as k-means (Arthur and Vassilvitskii, 2007; Lloyd, 1982) to identify the p distinct partitions. We are not aware of a theoretical result guaranteeing that the obtained approximate solution will closely match the optimal discrete solution, but in practice we often do obtain a good approximation.

4.2 Spectral modularity maximization

4.2.1 Algorithm

We are now ready to describe an outline of the modularity clustering technique, shown in Algorithm 11.

Algorithm 11 Modularity Clustering

1: Let $G = (V, E)$ be an input graph and A be its weighted adjacency matrix.
2: Let p be the number of desired clusters.
3: Set the modularity matrix $B = A - \frac{1}{2\omega}\mathbf{v}\mathbf{v}^T$.
4: Find the p largest eigenpairs $BU = U\Sigma$, where $\Sigma = \text{diag}(\lambda_1, ..., \lambda_p)$.
5: Scale the eigenvectors U by row or by column (optional).
6: Run a clustering algorithm, such as k-means, on the points defined by the rows of U.

Notice that the general outline of the modularity clustering closely resembles the spectral partitioning in (Naumov and Moon, 2016). The main difference is that in the former case we use the modularity matrix B and find its largest eigenpairs, while in the latter case we use the Laplacian matrix L and find its smallest eigenpairs. The properties of modularity and Laplacian matrices are also different, requiring a different choice of eigenvalue problem solvers.

4.2.2 Eigenvalue Problem

In order to find the largest eigenpairs we choose to use the restarted Lanczos eigenvalue solver (Bai et al., 2000; Saad, 1992). It is one of the most efficient and fastest eigenvalue solvers for finding the largest eigenvalues of symmetric problems. Its pseudo-code is shown in Algorithm 8.

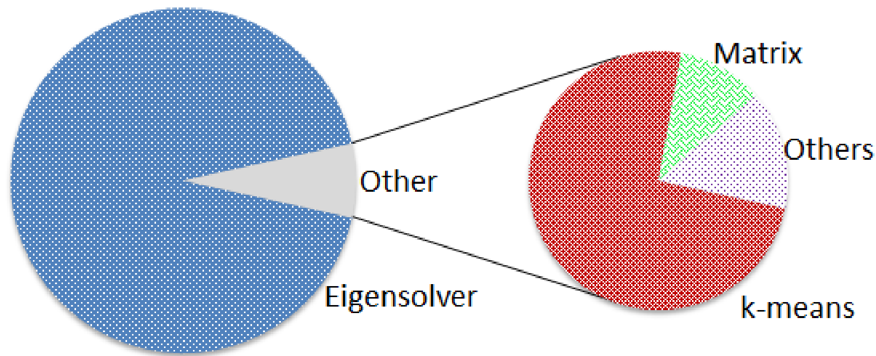


Fig. 4.3 Profiling of the modularity algorithm

In our numerical experiments we have found that the eigenvalue solver is a critical part of this technique because it is the most time consuming part of the computation, as shown in Figure 4.3. Also, the accuracy of the solution of the eigenvalue problem has a significant impact on the quality of the obtained clustering, with insufficient accuracy usually resulting in poor approximation to the original discrete problem. Moreover, a failure in convergence of the eigenvalue solver results in failure of the entire algorithm.

Finally, the most time consuming part of the Lanczos eigenvalue solver is the sparse matrix-vector multiplication (csrmv) with remaining time consumed by BLAS operations (NVIDIA, 2017), as shown in Figure 6.2.

4.2.3 Clustering Problem

Assuming that we have solved the optimization problem (4.14), let us now show how to solve the related discrete problem (4.13) and find the assignment of nodes into clusters.

Let us interpret each row of U as a point $\mathbf{x}_i$ in p-dimensional space, so that

$$U = \begin{pmatrix} u_{11} & \dots & u_{1p} \\ u_{21} & \dots & u_{2p} \\ \vdots & & \vdots \\ u_{n1} & \dots & u_{np} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{pmatrix} \tag{4.15}$$

Then, we will need to find cluster sets S_k for $k = 1, \ldots, p$, each with a centroid (point in the center) $\mathbf{c}_k$, such that

$$\min_{S_k} \sum_{k=1}^{p} \sum_{i \in S_k} \|\mathbf{x}_i - \mathbf{c}_k\|_2^2 \tag{4.16}$$

The exact solution of this problem is NP-complete, but we can find an approximation to its solution using many variations of the k-means clustering algorithm (Arthur and Vassilvitskii, 2007; Lloyd, 1982). The pseudo-code of the k-means Lloyd algorithm is shown for completeness in Algorithm 12.

Algorithm 12 K-means Lloyd Algorithm

1: Let centroids $\mathbf{c}_k$ for $k = 1, \ldots, p$ be an initial guess.
2: **for** $j = 1, \ldots, p$ **do** ▹ Assign points $\mathbf{x}_i$ to clusters S_k
3: Compute distance $d_{ij} = \|\mathbf{x}_i - \mathbf{c}_j\|_2$ for $i = 1, \ldots, n$
4: **end for**
5: Assign points $\mathbf{x}_i$, so that cluster $S_k = \{i : d_{ik} \le d_{ij}\}$ for $j = 1, \ldots, p$
6: **for** $l = 1, 2, \ldots$convergence **do**
7: Compute error $\epsilon_l = \sum_{k=1}^{p} \sum_{i \in S_k} d_{ik}^2$
8: Check convergence $|\epsilon_l - \epsilon_{l-1}|/n <$ tol.
9: **for** $k = 1, \ldots, p$ **do** ▹ Update centroids $\mathbf{c}_k$ of S_k
10: Compute centroid $\mathbf{c}_k = \left(\sum_{i \in S_k} \mathbf{x}_i\right)/|S_k|$
11: **end for**
12: **for** $j = 1, \ldots, p$ **do** ▹ Assign points $\mathbf{x}_i$ to S_k
13: Compute distance $d_{ij} = \|\mathbf{x}_i - \mathbf{c}_j\|_2$ for $i = 1, \ldots, n$
14: **end for**
15: Assign points $\mathbf{x}_i$, so that cluster $S_k = \{i : d_{ik} \le d_{ij}\}$ for $j = 1, \ldots, p$
16: **end for**

Finally, notice that the number of partitions identified by the clustering algorithm does not necessarily need to match the number of computed eigenvectors. In fact it can be chosen adaptively based on the modularity score (Smyth and White, 2005) or

x-means algorithm proposed in (Pelleg et al., 2000). We will revisit this point in the next sections.

4.2.4 Parallelism and Energy Efficiency

Most real networks are very large and for the vast majority of applications the analysis of the structure of the graph is on the critical path of complex data analytics. As a result cluster detection is expected to be done very quickly. Therefore performances and scalability are a primary concern for the overall success of a graph clustering algorithm.

Notice that each step in Algorithm 11 could benefit from parallelism. Also, the parallel variant of each building block can be leveraged on the GPU (commonly referred as device). This hardware platform is relevant for the modularity clustering because the performance of Algorithm 11 is limited by memory bandwidth which is higher on the device than on the CPU (commonly referred as host). Thus, taking advantage of parallelism and accelerators is critical for successful application of Algorithm 11 in practice.

In our implementation all data structures, including the adjacency matrix A, are stored in the device memory. The modularity matrix B is stored implicitly and the result of its action on a vector is computed using sparse matrix-vector multiplication (csrmv) with the adjacency matrix and rank-one update. Also, all vectors and scalars required by Algorithm 8 and Algorithm 12 are stored on the device.

Therefore all the algorithms are also implemented on the device. The Lanczos algorithm is implemented by using the sparse and dense basic linear algebra subroutines in CUSPARSE and CUBLAS libraries (NVIDIA, 2017). The k-means algorithm Algorithm 12 is also accelerated on the GPU using custom kernels. Indeed, the most expensive part in it is an embarrassingly parallel computation of the distances to the centroids, which can be done efficiently on the device.

We can expect that using GPU would also be advantageous from the energy efficiency perspective. While we do not measure the energy consumed by the algorithm directly, in broad terms we can relate it to the difference in Thermal Design Power (TDP). The TDP measures the average power a processor dissipates when operating with all cores active. The real energy usage may be different and may change depending on the hardware generation and time consumed by the algorithm on different

hardware platforms. For instance, in the next section we will perform experiments on Nvidia Titan X (Pascal) GPU and Intel Core i7-3930K CPU with 250 and 130 Watts TDP, respectively. Also, we will show that our algorithm on the GPU outperforms the state-of-the-art implementation on the CPU by $\sim 3\times$ on average. Since the ratio between the speedup and ratio of TDP ($250/130 \sim 2\times$) on these platforms is $3/2 > 1$, we can in general expect to achieve a better power efficiency on the GPU. The GPU power consumption can be estimated more accurately based on statistical approach and performance counters as presented in (Nagasaka et al., 2010).

4.3 Numerical Experiments

4.3.1 Context

Let us now study the performance and quality of the clustering obtained by the proposed modularity algorithm Algorithm 11 on a set of graphs from the DIMACS10, LAW and SNAP graph collections (Bader et al., 2013; Davis and Hu, 2011). In this sample set, we have selected large networks for which the clustering problem is relevant, as shown in Table 4.1.

#	Matrix	$n = \|V\|$	$m = \|E\|$	Application
1.	preferentialA...	100,000	499,985	Artificial
2.	caidaRouterLevel	192,244	609,066	Internet
3.	coAuthorsDBLP	299,067	977,676	Coauthorship
4.	citationCiteseer	268,495	1,156,647	Citation
5.	coPapersDBLP	540,486	15,245,729	Affiliation
6.	coPapersCiteseer	434,102	16,036,720	Affiliation
7.	as-Skitter	1,696,415	22,190,596	Internet
8.	hollywood-2009	1,139,905	113,891,327	Coauthorship

Table 4.1 General information on networks

In the modularity algorithm, we let the stopping criteria for the Lanczos eigenvalue solver in Algorithm 8 be based on the norm of the residual of the largest eigenpair $||\mathbf{r}_1||_2 = ||B\mathbf{u}_1 - \lambda_1 \mathbf{u}_1||_2 \leq 10^{-3}$ and maximum number of iterations 800 (with restart at every 20 iterations), while for the k-means Algorithm 12 we let it be based on the scaled error difference $|\epsilon_l - \epsilon_{l-1}|/n < 10^{-2}$ and maximum number of iterations 20.

Also, all numerical experiments are performed on a workstation with Ubuntu 14.04 operating system, gcc 4.8.4 compiler, CUDA Toolkit 8.0 software and Intel Core i7-3930K CPU 3.2 GHz and Nvidia Titan X (Pascal) GPU hardware. The performance of the algorithms was always measured across multiple runs to ensure consistency.

4.3.2 Clustering and Effects of Precision

First, let the number of cluster into which we would like to partition the graph be fixed. For instance, suppose that we have decided to partition the graph into 7 clusters. We plot the corresponding time, number of iterations and modularity score in Figures 4.4, 4.5 and 4.6.

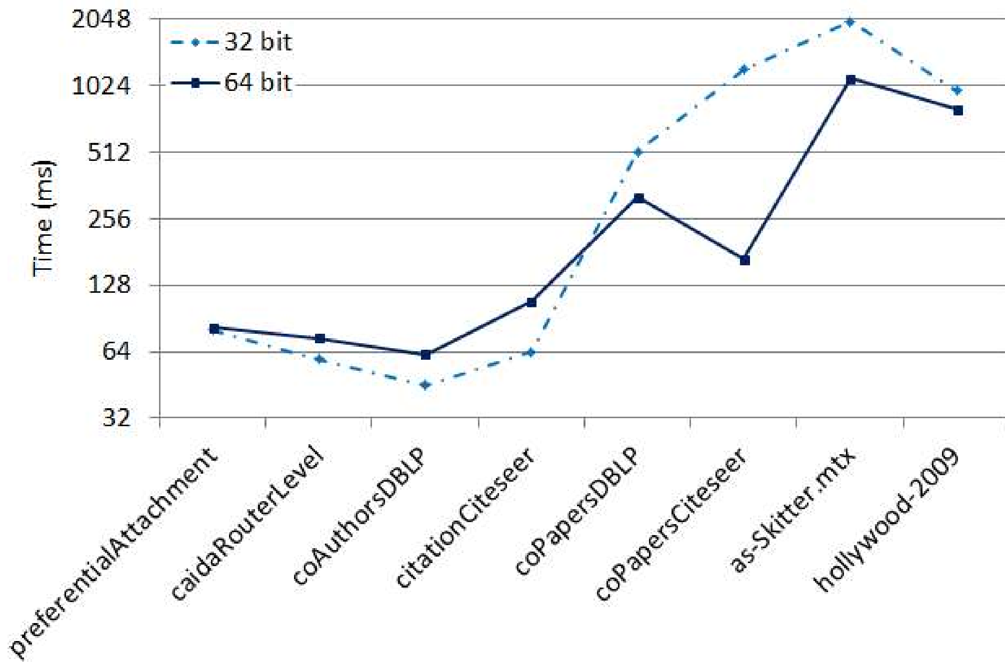


Fig. 4.4 The time achieved for 64 and 32 bit precision, when splitting the graph into 7 clusters

In (Newman, 2003) this type of clustering was achieved by recursive bisection, which has significant drawbacks. In particular, there is no guarantee that the recursive bisection correctly discovers an odd number of clusters. For example, if the network was already split in two cluster there is no reason why the third cluster would be perfectly included inside one of them. In fact, there is a very good chance that

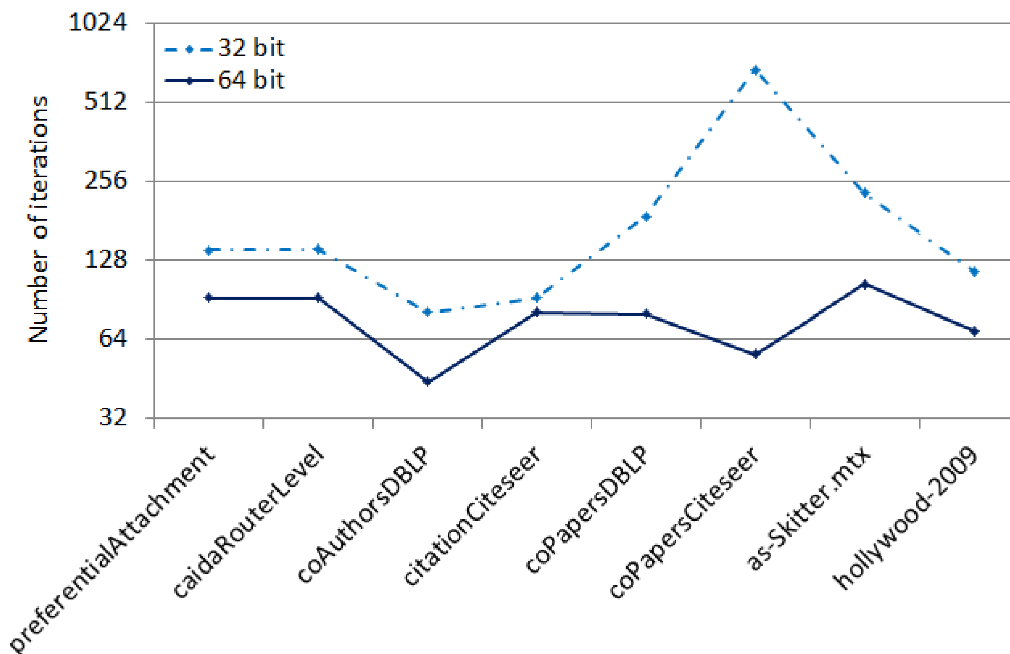


Fig. 4.5 The number of iterations achieved for 64 and 32 bit precision, when splitting the graph into 7 clusters

a clustering in three groups looks completely different and cannot be computed recursively (Chen et al., 2014).

The modularity algorithm is robust and converged to the solution on all networks of interest. Also, the computed modularity score remained in the interval $[-0.5, 1]$ as predicted by the theory (Table 4.2). Moreover, the modularity score computed by random assignment of nodes into clusters was 0 for all the networks as expected. It is an important baseline for comparing attained modularity scores.

The behavior of the algorithm differs when the computation is performed using single (32 bit) and double (64 bit) floating point arithmetic. In particular, the total time to the solution can be significantly better in 64 bit than in 32 bit precision as shown in Figure 4.4.

This may be surprising given that single precision has a significantly higher raw floating point performance and requires less bandwidth to access the data. However, single precision also results in unwanted perturbations during the computation of the Krylov subspace by the Lanczos eigenvalue solver. Those perturbations can impact the number of iterations and the overall quality of the approximation. Their effect depends on the sensitivity of the problem that is measured by its condition

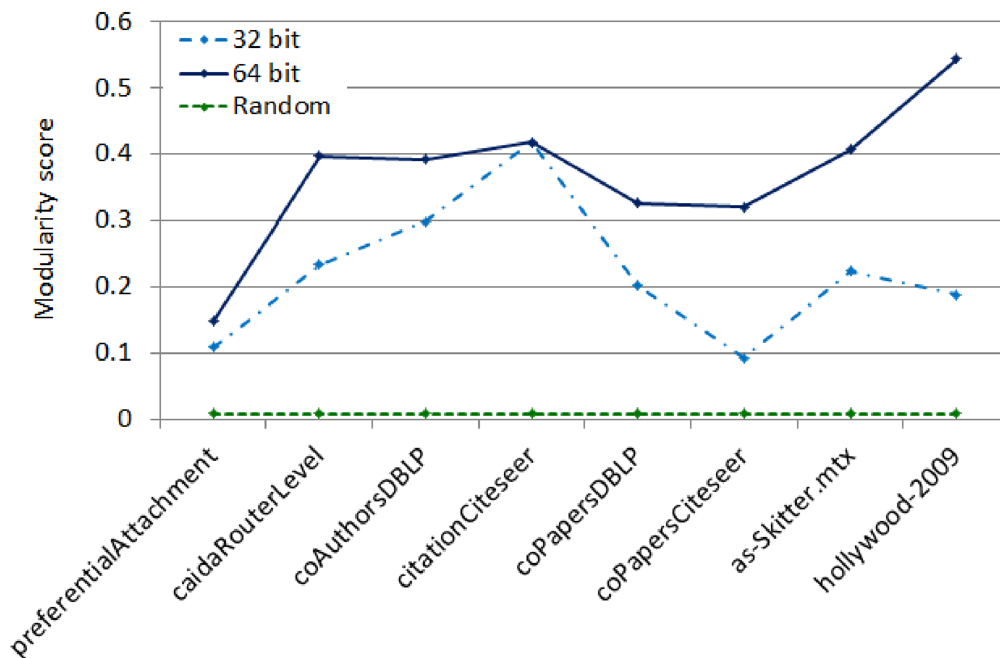


Fig. 4.6 The modularity score achieved for 64 and 32 bit precision, when splitting the graph into 7 clusters

number. We can empirically see these effects when we plot the number of iterations on Figure 4.5. It shows that using 64 bit precision often requires a lower number of iterations for convergence. A particularly stark illustration of this behavior happens on `coPapersCiteseer` network, which takes almost twice the number of iterations in 32 bit precision to converge.

Nevertheless, when the number of eigenvalue solver iterations is not affected by the use of lower precision, as is the case for `citationCiteseer` network, then the computation in 32 bit can outperform 64 bit precision as shown in Figure 4.4.

The choice of arithmetic precision also has direct impact on the quality of the discrete solution. Figure 4.6 clearly shows that the use of 64 bit precision almost always leads to a significant improvement in the quality of the final result. Therefore, we have found that using 64 bit precision is a safer option, unless the modularity matrix is particularly well conditioned.

The detailed results are summarized in Table 4.2.

	64 bits			32 bits		
#	Mod	T (ms)	It	Mod	T (ms)	It
1.	0.147	82	92	0.108	80	140
2.	0.397	74	92	0.233	59	141
3.	0.392	62	44	0.297	45	81
4.	0.417	108	81	0.417	64	92
5.	0.326	318	80	0.201	514	188
6.	0.319	168	56	0.092	1206	681
7.	0.407	1104	104	0.223	2001	230
8.	0.544	796	69	0.187	973	116

Table 4.2 The modularity (Mod), time (T) and # of iterations (It) achieved for 64 and 32 bit precision, when splitting the graph into 7 clusters

4.3.3 Adaptive Clustering

In the previous experiments we have kept the number of eigenpairs and k-means clusters the same as suggested by the theory developed in earlier sections. Next, we investigate what happens when we decouple these parameters. On one hand, we would expect that by selecting more k-means clusters than eigenpairs we would trade lower quality for higher performance. On the other hand, we could interpret selecting fewer k-means clusters than eigenpairs as filtering the noise in the data and perhaps obtaining a better solution.

In our experiments we have indeed found that for most networks it is possible to maximize the modularity by varying the number of k-means clusters independently of the number of eigenpairs as shown in Figure 4.7. In these experiments we have computed 2 to 8 eigenpairs, and afterward we have continued to increase only the number of k-means clusters. Notice that the plot demonstrates how the choice of the number of eigenpairs impacts the modularity score. Based on the formulation of the modularity problem one can expect that the best case scenario would be to compute a number of eigenpairs equal to the number of clusters. This is mostly, but not always, true, with the exceptions often due to loss of orthogonality or low quality approximation of the latest eigenvectors.

We also see that it is possible to compute fewer eigenpairs for an equivalent clustering quality as shown in Figure 4.7. Indeed, modularity values seem to follow a trend set by the number of clusters and the selected number of eigenpairs is secondary. Hence, given a constant number of eigenpairs it is possible to maximize modularity

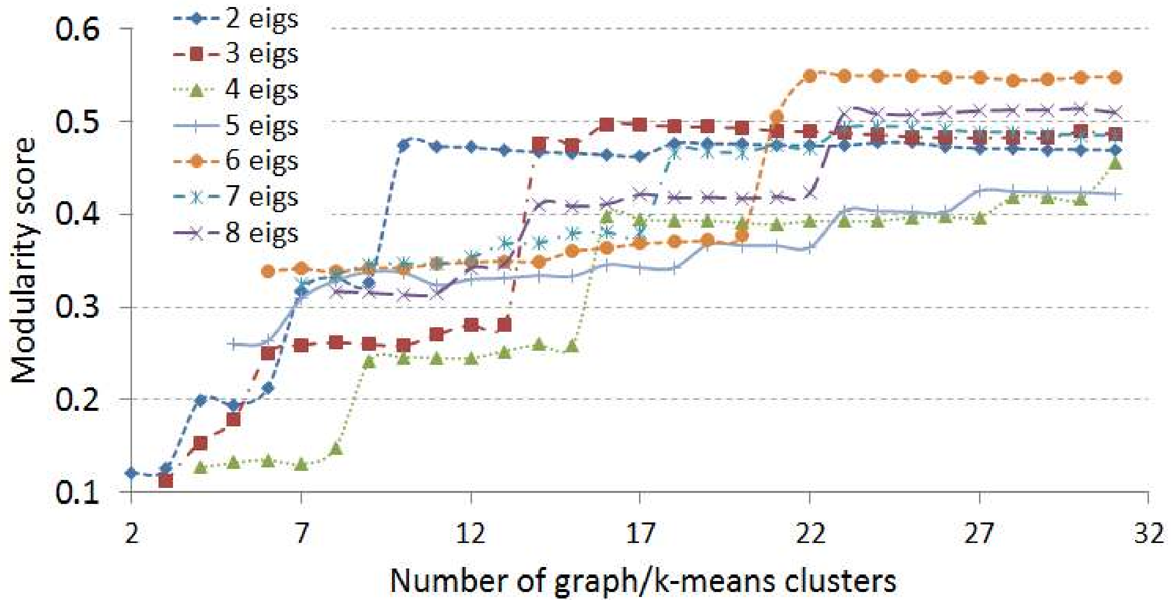


Fig. 4.7 Comparing the impact of varying the number of clusters used for assignment for different number of computed eigenvectors

by only increasing the number of k-means clusters. We can take advantage of this property for networks with many clusters, because computing a very large number of eigenpairs could prohibitively increase memory and time requirements of the eigenvalue solver, which has a direct impact on total time taken by the modularity computation.

The above experiments lead us to propose the following method for discovering an approximation to the natural number of clusters, which has also been proposed for small networks in (Smyth and White, 2005). We propose computing as many eigenpairs as clusters up to 7, and afterward continuing to increase the number of k-means clusters only, while keeping track of modularity score as shown on Figure 4.8. Since the plotted modularity score curve has a bell shape it is straight-forward to detect that its maximum is at 17 clusters on the x-axis. A similar trend can be seen for several other networks in our experiments. Moreover, we also found that it is better to over-estimate rather than under-estimate the number of clusters.

Also, notice on Figure 4.8 that when we increase the number of clusters by 10× from 2 to 20 the time to compute them only increases by about 20% from 95 ms to 120 ms. The plotted time line has a very low slope with respect to the x-axis, because the

number of computed eigenpairs does not increase past 7 in this experiment. Hence, the time growth shown on the figure only reflects additional time spent in the k-means step.

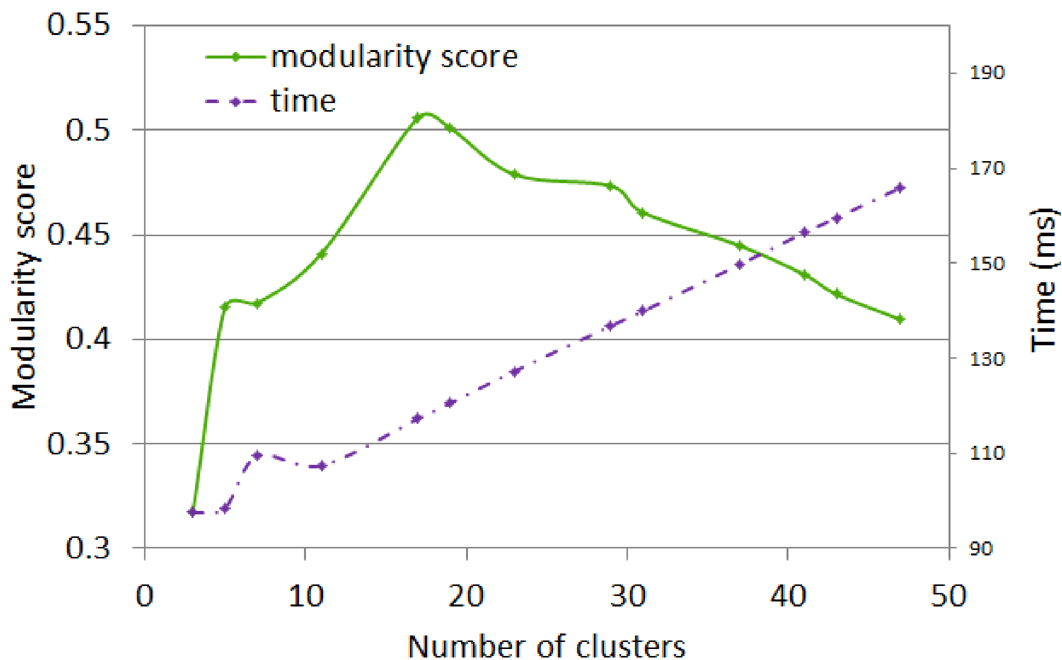


Fig. 4.8 The modularity score achieved when changing the number of clusters for citationCiteseer network in 64 bit precision

#	Clusters	Modularity	Time (ms)	# of iterations
1.	7	0.15	75	92
2.	11	0.40	70	92
3.	7	0.39	63	44
4.	17	0.51	117	81
5.	73	0.54	445	80
6.	53	0.64	248	56
7.	7	0.41	1095	104
8.	11	0.55	821	69

Table 4.3 The modularity (Mod), best number of clusters (Clu) according to modularity, time (T) and number of iterations (It) achieved in 64 bit precision

Using this technique we were able to detect the best clustering for all of the networks in Table 4.1. The resulting number of clusters and the modularity score found by our method are shown in Table 4.3.

In general, our algorithm can very quickly computes modularity for many clusters with only limited memory requirements. For example, we computed 53 clusters in half a second for coPapersCiteseer network with 16 million edges. Also, it takes only 0.8 seconds on a single GPU to find a clustering with a modularity score over 0.5 for hollywood-2009 which has 1,139,905 vertices and 113,891,327 edges.

4.3.4 Related Work

It is important to know how our algorithm compares against previous results. The earlier work on modularity was based on a reformulated modularity metric and an implementation of a greedy algorithm on the GPU (Auer, 2013). Unfortunately, we do not have access to the corresponding code and are forced to make the comparisons with the Tesla C2075 GPU used in it. Since in both algorithms the execution time is limited by memory bandwidth, we estimate a factor of $\sim 3\times$ as baseline performance difference between the Tesla C2075 with 144GB/s and TitanX with 337GB/s bandwidth. We believe this estimate to be fair in practice.

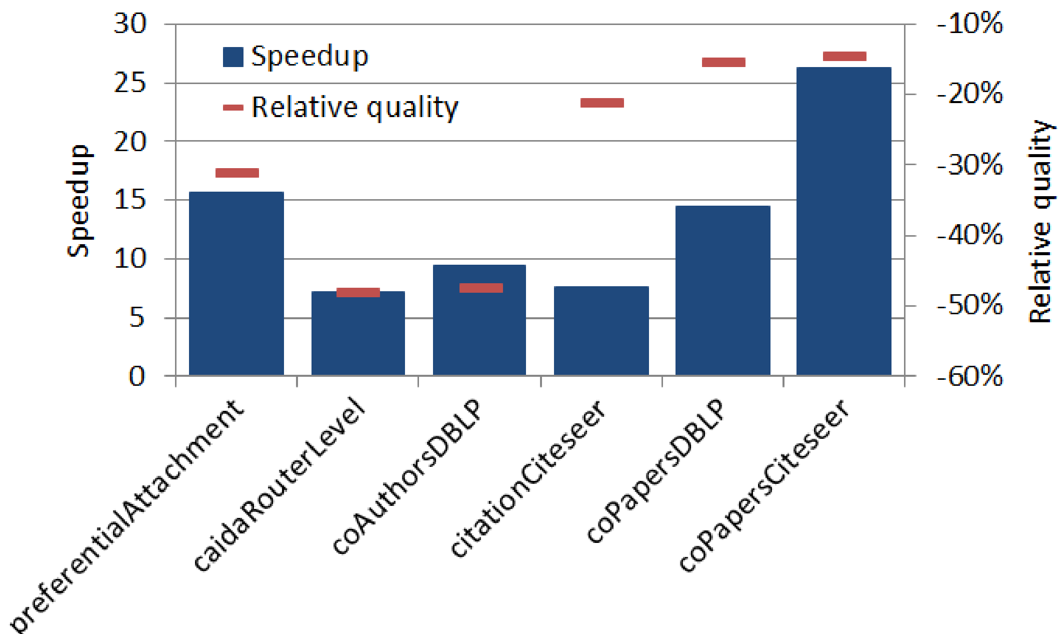


Fig. 4.9 The speedup and relative quality when compared to the reference results for large data sets on GPU in (Auer, 2013)

The performance of our approach versus earlier results is plotted in Figure 4.9. Our implementation achieves speedups of up to 8× compared to previous results, even when compensating for bandwidth difference of up to 3× due to difference in the hardware resources. It also allows us to handle networks with up to hundred million edges in less than a second. However, as a tradeoff in some large cases our approach obtains a lower modularity score for these networks, see Table 4.4.

On the other hand, for small cases the modularity algorithm we propose often attains a better modularity score than the reference results in (Auer, 2013). In fact its score is better in 5 out of 6 cases considered in the study, as shown in Table 4.5. Since we implement different algorithms for computing modularity, it is not completely surprising that their behavior varies on different data sets. Unfortunately, we could not identify any particular trend that would tell us when one algorithm would be better than the other in terms of quality. However, we always outperform the reference approach on large cases.

In particular, on the famous Zachary Karate Club social network, our clustering is 100% exact when comparing against the reference factions in (Zachary, 1977) as shown in Figure 4.10.

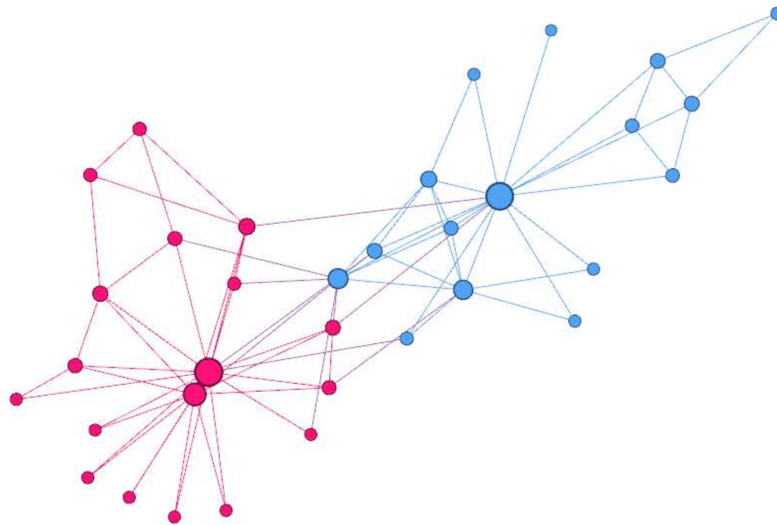


Fig. 4.10 Zachary Karate Club network where vertices are coloured according to their faction.

Matrix	# Clusters	Modularity	Ref Modularity
preferentialAtt...	7	0.147	0.214
caidaRouterLevel	11	0.397	0.768
coAuthorsDBLP	7	0.392	0.748
citationCiteseer	17	0.506	0.643
coPapersDBLP	73	0.540	0.640
coPapersCiteseer	53	0.636	0.746

Table 4.4 The modularity for a given # of clusters when compared to the reference results for large data sets in (Auer, 2013)

Matrix	# Clusters	Modularity	Ref Modularity
karate	3	0.390	0.363
dolphins	5	0.509	0.453
lesmis	11	0.250	0.444
adjnoun	5	0.255	0.247
polbooks	3	0.504	0.437
football	7	0.575	0.412

Table 4.5 The modularity for a given # of clusters when compared to the reference results for small data sets in (Auer, 2013)

Another more recent work on modularity developed a hierarchical algorithm for computing it on the CPU (Lasalle and Karypis, 2015). We have experimented with this algorithm using all CPU cores available on the machine. We computed 7 clusters in 64 bit precision on the data sets from Table 4.1. The performance of our approach versus these results is plotted in Figure 4.11. Notice that on average our algorithm outperforms the hierarchical approach by about 3×, but it does suffer from the same quality tradeoffs as shown in Table 4.6.

4.3.5 Modularity and Spectral Clustering

Finally, we compare the modularity score using the assignment into clusters obtained by two different clustering techniques. The first is obtained by the modularity algorithm in this chapter, while the second is obtained by spectral algorithm, that finds the minimum balanced cut of a graph (Naumov and Moon, 2016). Both techniques solve an eigenvalue problem and cluster the values of the eigenvectors using k-means algorithm. However, in the former case we use the modularity matrix B and find its

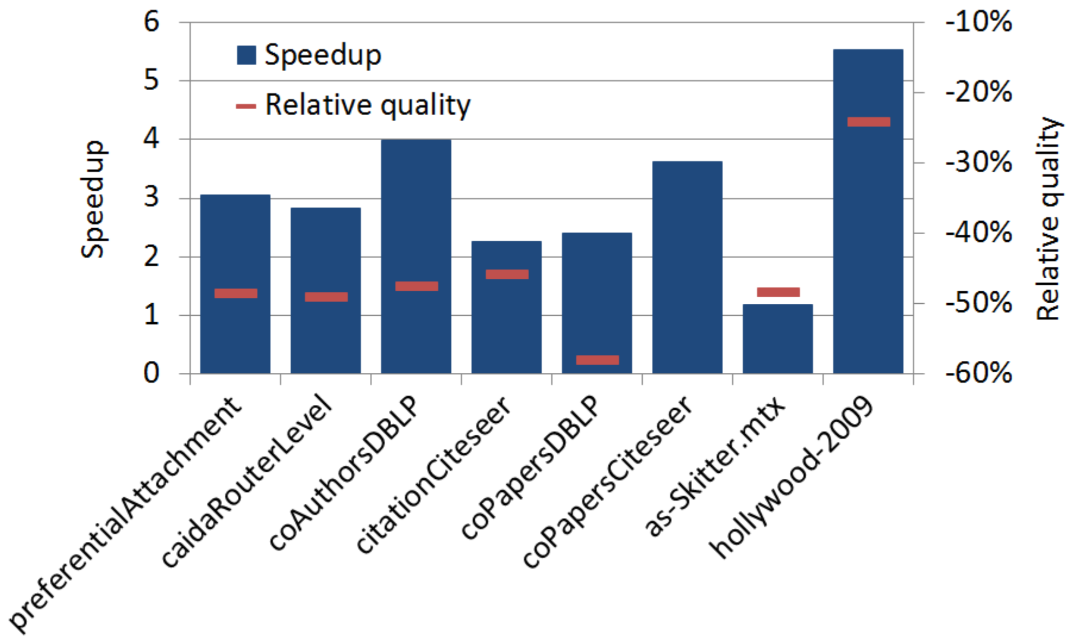


Fig. 4.11 The speedup and relative quality when compared to the reference results for large data sets on CPU in (Lasalle and Karypis, 2015)

	Spectral (GPU)		Hierarchical (CPU)	
#	Mod	T (ms)	Mod	T (ms)
1.	0.147	82.23	0.285314	251.39
2.	0.397	74.01	0.778106	209.885
3.	0.392	62.38	0.745344	249.412
4.	0.417	108.10	0.769948	243.494
5.	0.326	318.42	0.776566	762.448
6.	0.319	168.65	0.810843	612.387
7.	0.407	1104.78	0.78723	1307.34
8.	0.544	796.79	0.716734	4415.08

Table 4.6 The modularity score (Mod) and time (T) obtained by our spectral modularity maximization (GPU) and reference results from (Lasalle and Karypis, 2015) on hierarchical modularity maximization (CPU)

largest eigenpairs, while in the latter case we use the Laplacian matrix L and find its smallest eigenpairs.

Although, we found that modularity maximization has more predictable quality and achieved better scores for many networks, it is interesting to note that there are cases where spectral minimum balanced cut leads to assignments that have better modularity scores as shown in Figure 4.12.

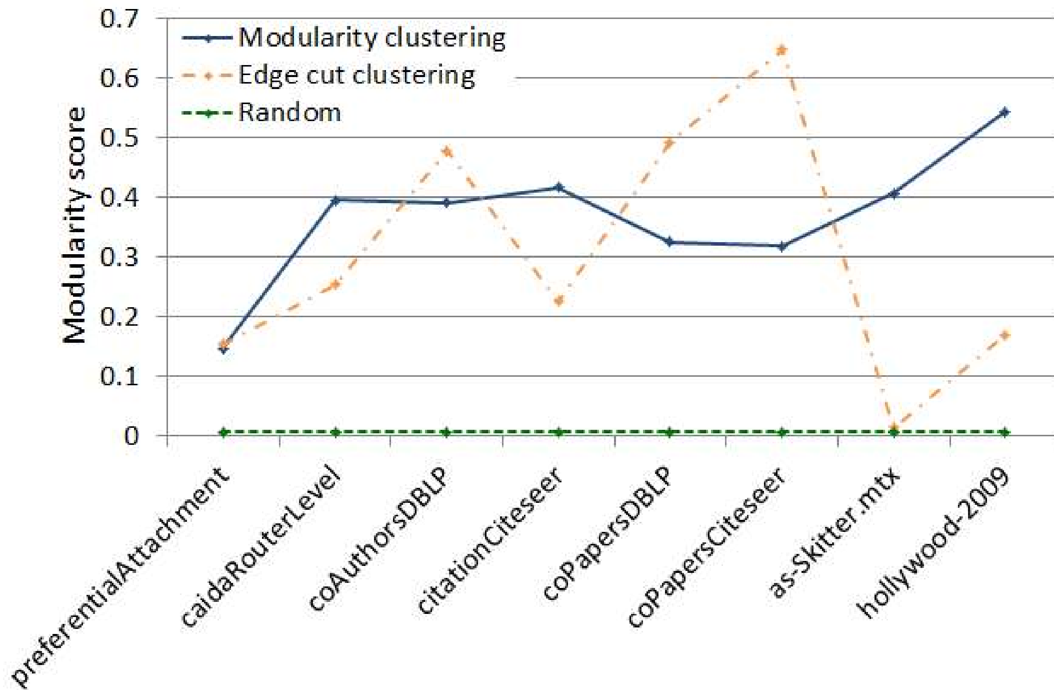


Fig. 4.12 The modularity score obtained on large cases by using assignment to clusters generated by modularity and minimum balanced cut algorithms for 7 clusters in 64 bit precision

	Modularity Maximization		Minimum Balanced Cut	
#	Mod	T (ms)	Mod	T (ms)
1.	0.147	82.23	0.154	143.70
2.	0.397	74.01	0.254	171.37
3.	0.392	62.38	0.478	221.21
4.	0.417	108.10	0.225	289.86
5.	0.326	318.42	0.493	1106.0
6.	0.319	168.65	0.648	798.41
7.	0.407	1104.7	0.015	2961.0
8.	0.544	796.79	0.171	3173.2

Table 4.7 The modularity score (Mod) and time (T) obtained by using the assignment to partitions generated by modularity and minimum balanced cut algorithm for 7 clusters in 64 bit precision

Also, we note that the modularity maximization is always faster than the spectral technique because it requires fewer iterations to solve the eigenvalue problem. Indeed, the smallest eigenvalues of the Laplacian are often more clustered than the largest eigenvalues of the modularity matrix. Since the number of eigensolver iterations often depends on the gap between the eigenvalues, the convergence to eigenvalues of the

modularity matrix is faster. As a result the modularity algorithm attains on average 3× speedup over the spectral scheme as shown in Table 4.7, where we have used the same parameters for both techniques.

4.4 Conclusion and Future Work

In this chapter we have revisited the modularity theory and its generalization to handle weighted graphs as well as to be able to find multiple clusters at once. The latter allows us to avoid using recursive bisection, while still maximizing the modularity score when looking for many clusters.

We have developed a parallel variation of the technique that computes multiple eigenpairs with the Lanczos algorithm and perform a multidimensional k-means clustering on the obtained eigenvectors on the GPU. This implementation has achieved speedups of up to 8× compared to previous results, even when compensating for bandwidth difference of up to 3× due to difference in the hardware resources. It also allowed us to handle networks with up to hundred million edges in less than a second.

Our experiments on real networks showed that modularity benefits from more accurate and stable approximation of the eigenpairs and therefore often requires use of 64 bit precision floating point arithmetic.

Also, we experimented with using l k-means centroids, while computing $k \ll l$ eigenvectors. This has allowed us to compute l clusters for the original graph more quickly at the expense of lower quality. We also highlighted that it is possible to use this for developing a technique to detect and adaptively select the natural number of clusters in the graph.

Chapter 5

Jaccard and PageRank weights in spectral clustering

5.1 Introduction

In this chapter we propose to generalize Jaccard and related measures, often used as similarity coefficients between two sets. We define Jaccard, Dice-Sorensen and Tversky edge weights on a graph and generalize them to account for vertex weights. We develop an efficient parallel algorithm for computing Jaccard edge and PageRank vertex weights. We highlight that the Jaccard weights computation can obtain more than 10× speedup on the GPU versus CPU. Also, we show that finding a minimum balanced cut for modified weights can be related to minimizing the sum of ratios of the intersection and union of nodes on the boundary of clusters. Finally, we show that the novel weights can improve the quality of the graph clustering by about 15% and 80% for multi-level and spectral graph partitioning and clustering schemes, respectively.

Let us now show how these weights help to naturally represent and express the structural information contained in a graph. For instance, the graph representing the Amazon book co-purchasing data set (Bader et al., 2013; Bastian et al., 2009; Davis and Hu, 2011; Newman, 2010) with original weights is shown on Figure 5.1, while the effect of using modified weights is illustrated on Figure 5.2, where thicker connections and larger circles indicate larger edge and vertex weights.

Notice that the graph on Figure 5.2 has apparently distinct clusters, which are easier to visually identify with Jaccard weights.

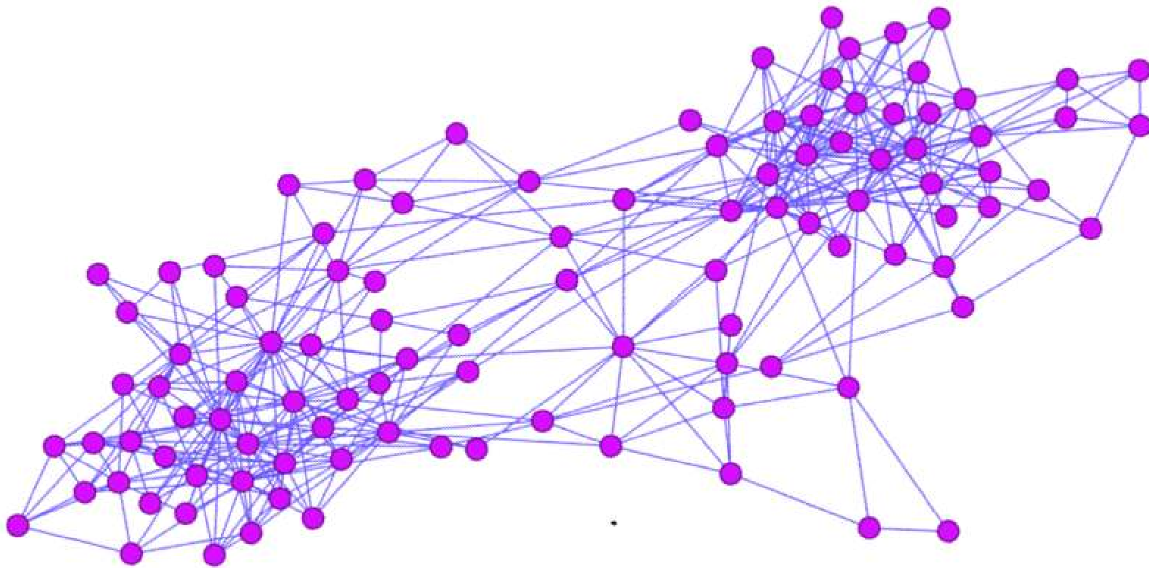


Fig. 5.1 Amazon book co-purchasing original graph

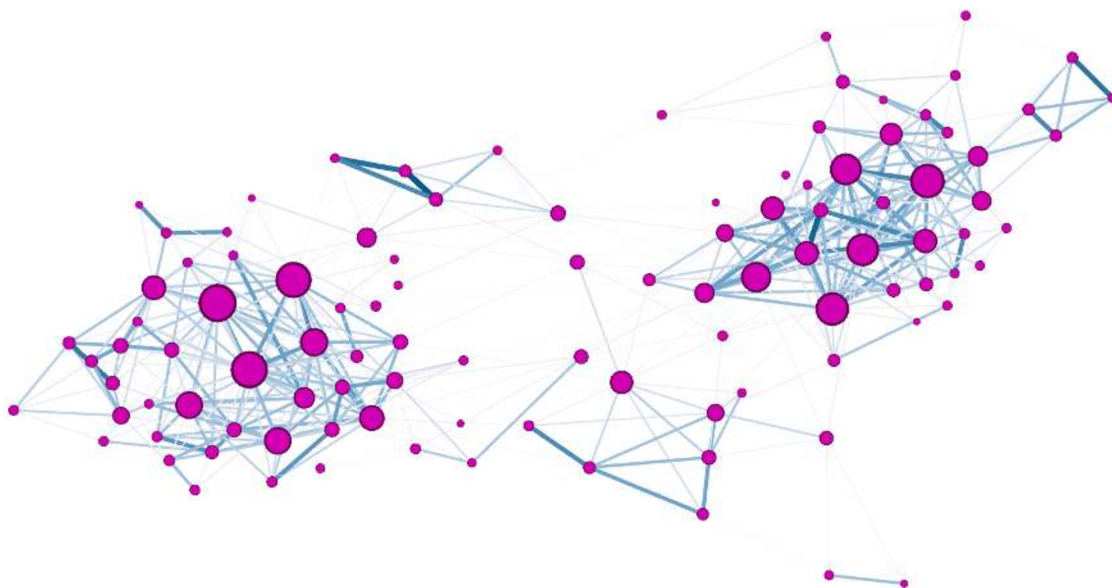


Fig. 5.2 Amazon book co-purchasing graph with Jaccard

Many processes in physical, biological and information systems are represented as graphs. In a variety of applications we would like to find a relationship between

different nodes in a graph and partition it into multiple clusters. For example, graph matching techniques can be used to build an algebraic multigrid hierarchy and graph clustering can be used to identify communities in social networks.

In this chapter we start by reviewing the Jaccard, Dice-Sorensen and Tversky coefficients of similarity between two sets (Dice, 1945; Jaccard, 1902; Sørensen, 1948; Tversky, 1977). Then, we show how to define graph edge weights based on these measures (Santisteban and Tejada Carcamo, 2015). Further, we generalize them to be able to take advantage of the vertex weights and show how to compute these using the PageRank algorithm (Page et al., 1998). We develop an efficient parallel algorithm for computing Jaccard edge and PageRank vertex weights. We highlight that the Jaccard weights computation can obtain more than 10× speedup on the GPU versus CPU. Also, we show that the modified weights, when combined with multi-level partitioning (Karypis and Kumar, 1998) and spectral clustering schemes (Naumov and Moon, 2016; Von Luxburg, 2007), can improve the quality of the minimum balanced cut obtained by these schemes by about 15% and 80%, respectively. Finally, we relate the Jaccard weights to the intersection and union of nodes on the boundary of clusters.

The work presented in this chapter has been published as Fender et al., 2017b.

5.2 Jaccard Weights

5.2.1 Jaccard and Related Coefficients

The Jaccard coefficient is often used as a measure of similarity between sets S_1 and S_2 (Jaccard, 1902; Levandowsky and Winter, 1971). It is defined as

$$\mathcal{J}(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{5.1}$$

where $|.|$ denotes the cardinality of a set. Notice that $J(S_1, S_2) \in [0, 1]$, with minimum 0 and maximum 1 achieved when the sets are disjoint $S_1 \cap S_2 = \{\emptyset\}$ and the same $S_1 \equiv S_2$, respectively. It is closely related to the Tanimoto coefficient for bit sequences (Rogers and Tanimoto, 1960).

Also, Jaccard coefficient is related to the Dice-Sorensen coefficient (Dice, 1945; Sørensen, 1948) often used in ecology and defined as

$$\frac{1}{2}\mathcal{D}(S_1,S_2) = \frac{|S_1 \cap S_2|}{|S_1| + |S_2|} = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2| + |S_1 \cap S_2|} \tag{5.2}$$

and Tversky index (Tversky, 1977) used in psychology and defined as

$$\mathcal{T}_{\alpha,\beta}(S_1,S_2) = \frac{|S_1 \cap S_2|}{\alpha|S_1 - S_2| + \beta|S_2 - S_1| + |S_1 \cap S_2|} \tag{5.3}$$

where $S_1 - S_2$ is a relative complement of set S_2 in S_1 and scalars $\alpha, \beta \geq 0$. Notice that $\mathcal{T}_{\frac{1}{2},\frac{1}{2}}(S_1,S_2) = \mathcal{D}(S_1,S_2)$ and $\mathcal{T}_{1,1}(S_1,S_2) = \mathcal{J}(S_1,S_2)$.

5.2.2 Jaccard and Related Edge Weights

Let a graph $G = (V, E)$ be defined by its vertex V and edge E sets. The vertex set $V = \{v_1, ..., v_n\}$ represents n nodes and edge set $E = \{(i_1, j_1), ..., (i_m, j_m)\}$ represents m edges. Also, we associate a nonnegative weight $w_{ij} \geq 0$ with every edge $(i, j) \in E$.

Let the adjacency matrix $A = [a_{ij}]$ corresponding to a graph $G = (V, E)$ be defined through its elements

$$a_{ij} = \begin{cases} w_{ij} & \in E \\ 0 & \text{otherwise} \end{cases} \tag{5.4}$$

We will assume that the graph is undirected, with $w_{ij} \equiv w_{ji}$, and therefore A is a symmetric matrix.

Let us define a neighbourhood of a node v_i as the set of nodes immediately adjacent to v_i, so that

$$\mathcal{N}(v_i) = \{v_j \mid (i,j) \in E\} \tag{5.5}$$

For example, for the unweighted graph shown on Figure 5.3 the neighbourhood of v_3 is $\mathcal{N}(v_3) = \{v_2, v_4, v_5\}$.

In order to setup Jaccard-based clustering, we propose to define the following intermediate edge weights in the graph. The intersection weight

$$w_{ij}^{(I)} = \sum_{v_k \in \mathcal{N}(v_i) \cap \mathcal{N}(v_j)} v_k \tag{5.6}$$

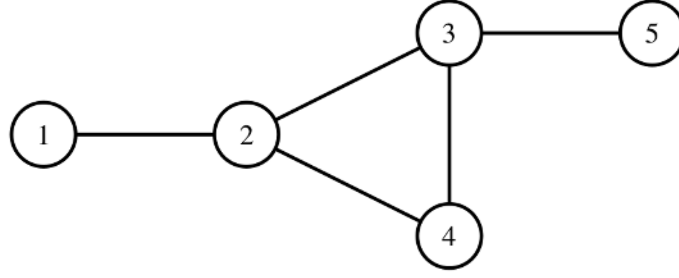


Fig. 5.3 Graph example, G = (V, E)

the sum weight

$$w_{ij}^{(S)} = \sum_{v_k \in \mathcal{N}(v_i)} v_k + \sum_{v_l \in \mathcal{N}(v_j)} v_l \tag{5.7}$$

the complement weight

$$w_{ij}^{(C)} = \sum_{v_k \in \mathcal{N}(v_i)} v_k - w_{ij}^{(I)} \tag{5.8}$$

and the union weight

$$\begin{aligned} w_{ij}^{(U)} &= w_{ji}^{(S)} - w_{ij}^{(I)} && (5.9) \\ &= w_{ij}^{(C)} + w_{ji}^{(C)} + w_{ji}^{(I)} && (5.10) \end{aligned}$$

For instance, for the special case of unweighted graphs, with $v_i = 1$ and $w_{ij} = 1$, we can omit the vertex weight and write these weights as

$$\begin{aligned} w_{ij}^{(I)} &= |\mathcal{N}(i) \cap \mathcal{N}(j)| && (5.11) \\ w_{ij}^{(S)} &= |\mathcal{N}(i)| + |\mathcal{N}(j)| && (5.12) \\ w_{ij}^{(C)} &= |\mathcal{N}(i)| - |\mathcal{N}(i) \cap \mathcal{N}(j)| = |\mathcal{N}(i) - \mathcal{N}(j)| && (5.13) \\ w_{ij}^{(U)} &= |\mathcal{N}(i)| + |\mathcal{N}(j)| - |\mathcal{N}(i) \cap \mathcal{N}(j)| \\ &= |\mathcal{N}(i) - \mathcal{N}(j)| + |\mathcal{N}(j) - \mathcal{N}(i)| + |\mathcal{N}(i) \cap \mathcal{N}(j)| \\ &= |\mathcal{N}(i) \cup \mathcal{N}(j)| && (5.14) \end{aligned}$$

Then, we can define Jaccard weight as

$$w_{ij}^{(\mathcal{J})} = w_{ij}^{(I)} / w_{ij}^{(U)} \tag{5.15}$$

Dice-Sorensen weight as

$$w_{ij}^{(\mathcal{D})} = w_{ij}^{(I)} / w_{ij}^{(S)} \tag{5.16}$$

Tversky weight as

$$w_{ij}^{(\mathcal{T})} = w_{ij}^{(I)} / (\alpha w_{ij}^{(C)} + \beta w_{ij}^{(C)} + w_{ij}^{(I)}) \tag{5.17}$$

For example, for the unweighted graph on Figure 1 the original adjacency matrix can be written as

$$A^{(\mathcal{O})} = \begin{bmatrix} & 1 & & & \\ 1 & & 1 & 1 & \\ & 1 & & 1 & 1 \\ & 1 & 1 & & \\ & & 1 & & \end{bmatrix} \tag{5.18}$$

while based on Jaccard weights it can be written as

$$A^{(\mathcal{J})} = \begin{bmatrix} & 0 & & & \\ 0 & & 1/5 & 1/4 & \\ & 1/5 & & 1/4 & 0 \\ & 1/4 & 1/4 & & \\ & & 0 & & \end{bmatrix} \tag{5.19}$$

Notice that if we simply use the Jaccard weights the new graph might become disconnected. For instance, in our example the intersections of neighborhoods of $\mathcal{N}(1) \cap \mathcal{N}(2)$ and $\mathcal{N}(3) \cap \mathcal{N}(5)$ are empty $\{\emptyset\}$ and consequently nodes 1 and 5 are disconnected from the rest of the graph. While it is possible to work with disconnected graphs, in many scenarios such change in the graph properties is undesirable.

Also, notice that the original weights $w_{ij}^{(\mathcal{O})}$ have arbitrary magnitude, while Jaccard weight $w_{ij}^{(\mathcal{J})} \in [0, 1]$. Therefore, adding these weights might result in non uniform effects on different parts of the graph (with small and large original weights) and make these effects scaling dependent.

In order to address these issues we propose to combine Jaccard and original weights in the following fashion

$$w_{ij}^{(*)} = w_{ij}^{(\mathcal{O})} \left(1 + w_{ij}^{(\mathcal{J})}\right) \tag{5.20}$$

Notice that in this formula the Jaccard weight is used to strengthen edges with large overlapping neighbourhoods.

In the next section we will show how we can efficiently compute Jaccard weights in parallel on the GPU. The Dice-Sorensen and Tversky weights can be computed similarly.

5.3 Implementation

5.3.1 Parallel Algorithm

The graph and its adjacency matrix can be stored in arbitrary data structures. Let us assume that we use the standard CSR format, which simply concatenates all non-zero entries of the matrix in row-major order and records the starting position for the entries of each row as presented in Section 2.1.2. For notation convenience, we use Ap to denote the row pointers array, Ac for the column indices array, and Av for the values array.

Then, the intersection weights in (5.6) can be computed in parallel using Algorithm 13, where the binary search is done according to Algorithm 14. Notice that in Algorithm 13 we perform intersections on sets corresponding to neighbourhoods of nodes i and j. These sets have potentially different number of elements $N_i = e_i - s_i$ and $N_j = e_j - s_j$. In order to obtain better computational complexity we would like to perform the binary search on the largest set. In the above pseudo-code we have implicitly assumed that the smallest set corresponds to node i. In practice, we can always test the set size by looking at whether $N_i < N_j$ and swap indices i and j if needed.

Then, the sum weights in (5.7) can be computed using the parallel Algorithm 15, where the sum operation on line 6 and 10 can be written for general graphs as

$$\mathrm{sum}(s, e, \mathrm{Av}) = \mathrm{Av}[s] + \mathrm{Av}[s+1] + \ldots + \mathrm{Av}[e] \tag{5.21}$$

and for unweighted graphs as

$$\mathrm{sum}(s, e, \mathrm{Av}) = 1 + \ldots + 1 = e - s \tag{5.22}$$

Finally, the union and the corresponding Jaccard weights can be obtained using (5.9) and (5.15), respectively.

Algorithm 13 Intersection Weights

1: Let n and m be the # of nodes and edges in the graph.
2: Let Ap, Ac and Av represent its adjacency matrix $A^{(O)}$.
3: Initialize all weights $w_{ij}^{(I)}$ to 0.
4: **for** $i = 1, \ldots, n$ **do in parallel**
5: Set $s_i = \mathsf{Ap}[i]$ and $e_i = \mathsf{Ap}[i+1]$
6: **for** $k = s_i, \ldots, e_i$ **do in parallel**
7: Set $j = \mathsf{Ac}[k]$
8: Set $s_j = \mathsf{Ap}[j]$ and $e_j = \mathsf{Ap}[j+1]$
9: **for** $z = s_i, \ldots, e_i$ **do in parallel** ▷ Intersection
10: $l = \mathtt{binary_search}(\mathsf{Ac}[z], s_j, e_j - 1, \mathsf{Ac})$
11: **if** $l \geq 0$ **then** ▷ Found element
12: AtomicAdd($w_{ij}^{(I)}$, Av[l]) ▷ Atomic Update
13: **end if**
14: **end for**
15: **end for**
16: **end for**

Algorithm 14 binary_search(i, l, r, x)

1: Let i be the element we would like to find.
2: Let left l and right r be the end points of a set.
3: Let sorted set elements be located in array x.
4: **while** $l \leq r$ **do**
5: $m = (l + r)/2$ ▷ Find middle of the set
6: $j = \mathsf{x}[m]$
7: **if** $j > i$ **then**
8: Set $r = m - 1$ ▷ Move right end point
9: **else if** $j < i$ **then**
10: Set $l = m + 1$ ▷ Move left end point
11: **else**
12: Return m ▷ Done, element found
13: **end if**
14: **end while**
15: Return -1 ▷ Done, element not found

Let us assume a standard theoretical PRAM (CREW) model for analysis (Jaja, 1992). Notice that the sequential complexity of Algorithm 13 is

$$\sum_{i=1}^{n} \sum_{j=1}^{N_i} N_i \log N_j \tag{5.23}$$

Algorithm 15 Sum Weights

1: Let n and m be the # of nodes and edges in the graph.
2: Let Ap, Ac and Av represent its adjacency matrix $A^{(\mathcal{O})}$.
3: **for** $i = 1, ..., n$ **do in parallel**
4: Set $s_i = \text{Ap}[i]$ and $e_i = \text{Ap}[i+1]$
5: Set $N_i = \text{sum}(s_i, e_i, \text{Av})$
6: **for** $k = s_i, ..., e_i$ **do in parallel**
7: Set $j = \text{Ac}[k]$
8: Set $s_j = \text{Ap}[j]$ and $e_j = \text{Ap}[j+1]$
9: Set $N_j = \text{sum}(s_j, e_j, \text{Av})$
10: Set $w_{ij}^{(S)} = N_i + N_j$
11: **end for**
12: **end for**

and, assuming we can store intermediate results of Algorithm 15, is

$$\sum_{i=1}^{n} \log N_i + m \tag{5.24}$$

where $N_i = |\mathcal{N}(v_i)|$ is the number of elements in each row. However, the complexity of both algorithms is

$$\max_i \log N_i \tag{5.25}$$

using $n \max_i N_i^2$ and m processors, respectively, which illustrates the degree of available parallelism.

Also, notice that Algorithm 13 can be interpreted as the sparse matrix-matrix multiplication AA^T, where only elements that are already present in the sparsity pattern of A are left, in other words, we do not allow any fill-in in the result.

The performance of the parallel implementation in CUDA of the algorithm for computing Jaccard weights on the GPU is shown in Figure 5.4. Notice that we compare it with sequential (single thread) as well as openMP (12 threads) implementation of the algorithm on the CPU, with hardware details specified in the numerical experiments section. We often obtain a speedup above 10× on the data sets from Table 5.3. The details of the experiments are provided in Table 5.1.

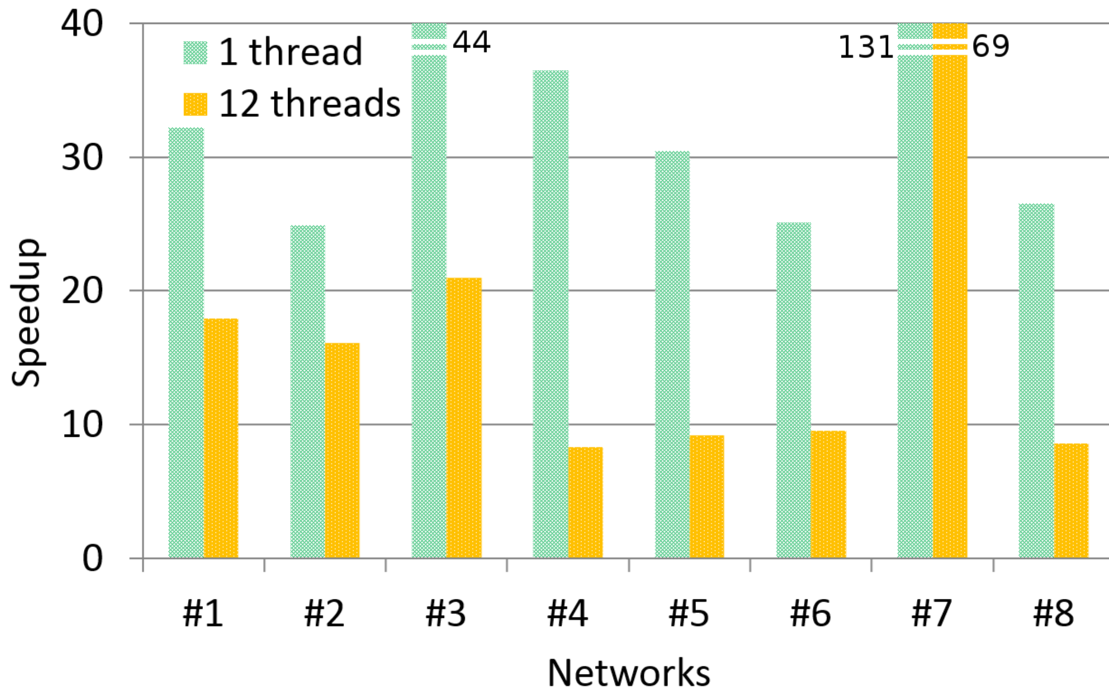


Fig. 5.4 Speedup of the GPU implementation vs. 1 and 12 CPU threads when computing Jaccard Weights

#	CPU (1 thread)	CPU (12 threads)	GPU
1.	155	86	5
2.	193	125	8
3.	172	82	4
4.	340	77	9
5.	9401	2847	308
6.	13514	5130	538
7.	65582	34646	502
8.	337870	109541	12751

Table 5.1 Time(ms) needed to compute Jaccard weights

5.3.2 PageRank and Vertex Weights

The PageRank algorithm measures the relative importance of a vertex compared to other nodes in the graph. Therefore, it is natural to incorporate the vertex weights v_k to measure the importance of neighbourhoods, as shown on Figure 5.2.

The PageRank algorithm (Page et al., 1998), has been a key feature of search and recommendation engines for years (Brezinski et al., 2005; Langville and Meyer, 2006). Recall that the PageRank algorithm is based on a discrete Markov process (Markov

chain), a mathematical system that undergoes transitions from one state to another and where the future states depend only on the current state.

More information about PageRank algorithm is available in Sections 2.2.1 and 3.1 with experimental results in 3.4.

For completeness, the performance of the parallel CUDA implementation of the algorithm for computing PageRank on the data sets from Table 5.3 is shown in Figure 5.5. Notice that we compare NVGRAPH (from Nvidia CUDA Toolkit) with the Intel MKL implementation of Algorithm 3, with hardware details specified in Section 5.5. We often obtain a speedup above 10× on realistic data sets from Table 4.1. The details of the experiments are provided in Table 5.2. In Figure 5.5 and Table 5.2 we always use the same initial guess ($\mathbf{w} = \frac{1}{n}\mathbf{e}$) for CPU and GPU implementations. Also, both reached the desired approximation tolerance ($1e^{-}6$) after the same number of iterations.

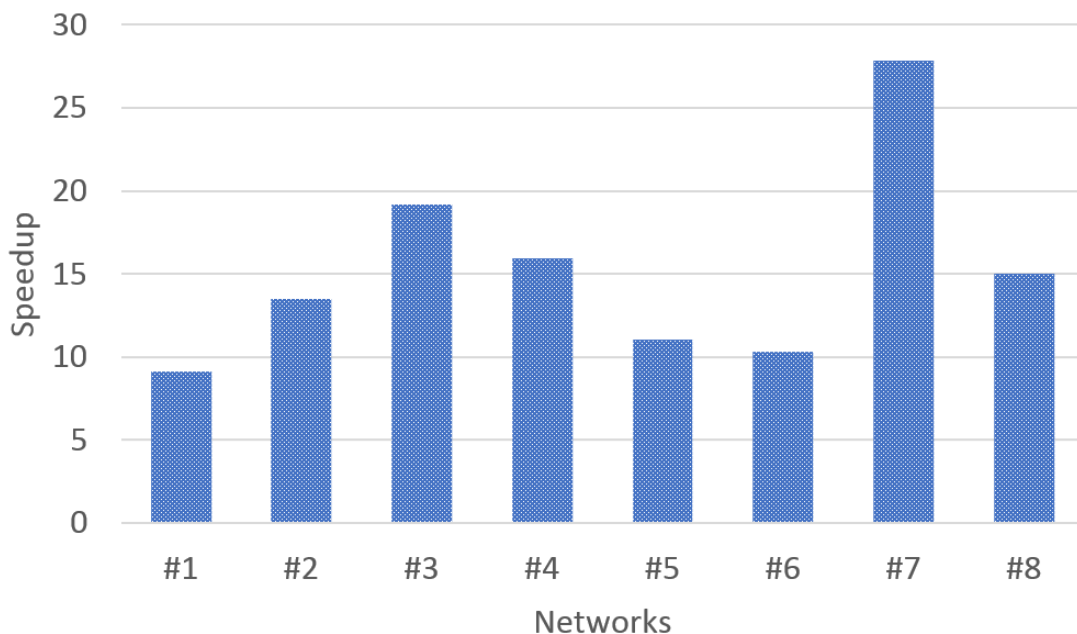


Fig. 5.5 Speedup when computing PageRank

	CPU			GPU		
#	time	it.	$\|\|\mathbf{r}_k\|\|_2/\|\|\mathbf{r}_0\|\|_2$	time	it.	$\|\|\mathbf{r}_k\|\|_2/\|\|\mathbf{r}_0\|\|_2$
1.	61	17	8.3e-06	7	17	8.3e-06
2.	393	76	9.0e-06	29	76	9.0e-06
3.	461	51	9.9e-06	24	51	9.9e-06
4.	470	57	8.9e-06	30	57	8.9e-06
5.	1721	51	2.4e-06	156	51	2.4e-06
6.	1615	53	2.7e-06	157	53	2.7e-06
7.	5228	74	6.6e-06	188	74	6.6e-06
8.	6650	49	1.1e-06	442	49	1.1e-06

Table 5.2 Time(ms) needed to compute PageRank

5.4 Graph Clustering

In graph clustering the vertex set V is often partitioned into p disjoint sets S_k, such that $V = S_1 \cup S_2 ... \cup S_p$ and $S_i \cap S_j = \{\emptyset\}$ for $i \neq j$ (Karypis and Kumar, 1998; Von Luxburg, 2007). More information about spectral graph custering is available in Section 2.2.2. Notice that instead of the original graph $G = (V, E)$ we can use the modified graph $G^{(*)} = (V^{(*)}, E^{(*)})$, with vertex $v_i^{(*)}$ and edge $w_{ij}^{(*)}$ weights computed based on PageRank and Jaccard or related schemes discussed in earlier sections.

5.4.1 Jaccard Spectral Clustering

Notice that we can define the Laplacian as

$$L^{(*)} = D^{(*)} - A^{(*)} \tag{5.26}$$

where $D^{(*)} = \mathrm{diag}(A^{(*)}\mathbf{e})$ is the diagonal matrix.

Then, we would minimize the normalized balanced cut

$$\begin{aligned} \tilde{\eta}(S_1, ..., S_p) &= \min_{S_1,...,S_p} \sum_{k=1}^{p} \frac{\mathrm{vol}(\partial(S_k))}{\mathrm{vol}(S_k)} \\ &= \min_{U^T D^{(*)} U = I} \mathrm{Tr}(U^T L^{(*)} U) \end{aligned} \tag{5.27}$$

where boundary edges

$$\partial S = \{w_{ij}^{(*)} \mid v_i^{(*)} \in S \wedge v_j^{(*)} \notin S\} \tag{5.28}$$

and volume

$$\mathrm{vol}(S) = \sum_{v_i^{(*)} \in S} w_{ij}^{(*)} \tag{5.29}$$

$$\mathrm{vol}(\partial S) = \sum_{(i,j)\in\partial(S)} w_{ij}^{(*)} = \sum_{(i,j)\in\partial(S)} w_{ij}^{(\mathcal{O})}\left(1 + \frac{w_{ij}^{(I)}}{w_{ij}^{(U)}}\right)$$

by finding its smallest eigenpairs and transforming them into assignment of nodes into clusters (Naumov and Moon, 2016). Notice that Jaccard weights correspond to the last term in the above formula, and are related to the sum of ratios of the intersection and union of nodes on the boundary of clusters.

Also, we point out that we choose to use normalized cut spectral formulation because it is invariant under scaling. Notice that based on (5.20) the edge weight $w_{ij}^{(*)} \geq w_{ij}^{(\mathcal{O})}$. Therefore, to avoid artificially higher/lower scores when comparing quality, we need to use a metric that is invariant under edge weight scaling. To illustrate this point suppose that for a given assignment of nodes into clusters the edge weights are multiplied by 2. The clustering has not changed and normalized score stays the same, while ratio cut score increases and therefore is not an appropriate metric for our comparisons.

5.4.2 Tversky Spectral Clustering

So far we have essentially defined Tversky clustering for a special case $\mathcal{T}_{1,1}(S_1, S_2) = \mathcal{J}(S_1, S_2)$. We note that further generalization is possible by introducing

$$A^{(\mathcal{T})} = A^{(I)} \oslash (\alpha L^{(C)} + \beta U^{(C)} + A^{(I)}) \tag{5.30}$$

where $L^{(C)}$ is lower and $U^{(C)}$ is upper triangular part of the matrix $A^{(C)} = [a_{ij}^{(C)}]$ with elements $a_{ij}^{(C)} = w_{ij}^{(C)}$ corresponding to complement weights, $A^{(I)}$ is a matrix with intersection weights and the $\oslash$ operation corresponds to Hadamard (entry-wise) division.

However, we point out that we can only compute Tversky clustering analogously to Jaccard clustering when the scaling parameters $\alpha = \beta$. Notice that if $\alpha \neq \beta$ then the adjacency matrix $A^{(\mathcal{T})}$ and the corresponding Laplacian matrix $L^{(\mathcal{T})}$ are not symmetric.

Therefore, the Courant-Fischer theorem (Horn and Johnson, 1986) is no longer applicable and the minimum of the objective function $\tilde{\eta}$ in (5.27) no longer corresponds to the smallest eigenvalues of the Laplacian.

5.4.3 Profiling

Notice that the computation of Jaccard and PageRank weights is often a small fraction $< 20\%$ of the total computation time, see Figure 5.6. In fact the profiling of the complete spectral clustering pipeline on the GPU shows that most time $> 80\%$ is actually spent in the eigenvalue solver. In our code we rely on the LOBPCG method (Knyazev, 2001), which has been shown to be effective for Laplacian matrices (Naumov and Moon, 2016).

The second most time consuming operation is the computation of PageRank vertex weights. Notice that PageRank also solves an eigenvalue problem, but it finds the largest eigenpairs of the Google matrix and therefore is significantly faster than LOBPCG, which looks for the smallest eigenpairs. We point out that the PageRank computation is optional and can be skipped if needed.

Finally, the computation of Jaccard edge weights is only the third most time consuming operation. Since our implementation supports weighted vertices by design, there is no extra cost for using the vertex weight resulting from PageRank or any other algorithm.

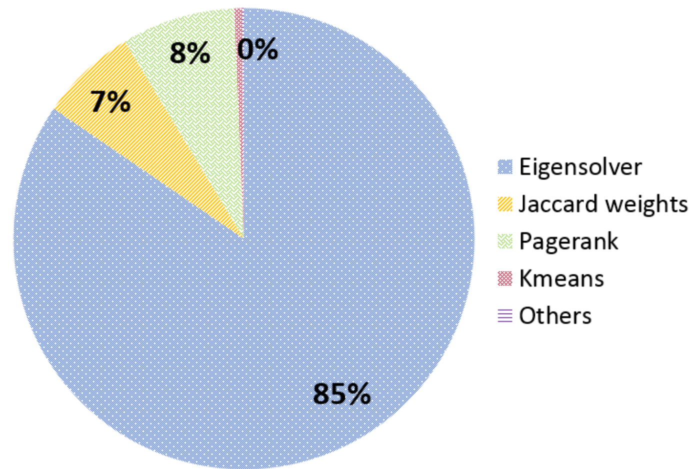


Fig. 5.6 Profile of spectral clustering with PageRank vertex and Jaccard edge weights

5.5 Numerical Experiments

Let us now study the performance and quality of the clustering obtained using Jaccard weights on a sample of graphs from the DIMACS10, LAW and SNAP graph collection (Davis and Hu, 2011), shown in Table 5.3.

#	Matrix	$n = \|V\|$	$m = \|E\|$	Application
0.	smallword	100,000	999,996	Artificial
1.	preferentialA...	100,000	499,985	Artificial
2.	caidaRouterLevel	192,244	609,066	Internet
3.	coAuthorsDBLP	299,067	977,676	Coauthorship
4.	citationCiteseer	268,495	1,156,647	Citation
5.	coPapersDBLP	540,486	15,245,729	Affiliation
6.	coPapersCiteseer	434,102	16,036,720	Affiliation
7.	as-Skitter	1,696,415	22,190,596	Internet
8.	hollywood-2009	1,139,905	113,891,327	Coauthorship

Table 5.3 General information on networks

In our spectral experiments we use the nvGRAPH 9.0 library and let the stopping criteria for the LOBPCG eigenvalue solver be based on the norm of the residual corresponding to the smallest eigenpair $\|\mathbf{r}_1\|_2 = \|L\mathbf{u}_1 - \lambda_1\mathbf{u}_1\|_2 \leq 10^{-4}$ and maximum

of 40 iterations, while for the k-means algorithm we let it be based on the scaled error difference $|\epsilon_l - \epsilon_{l-1}|/n < 10^{-2}$ between consecutive steps and a maximum of 16 iterations (Naumov and Moon, 2016).

In our multi-level experiments we use the METIS 5.1.0 library and choose the default parameters for it (Karypis and Kumar, 1998). Also, we plot the quality improvement as a percentage of the original score based on $100\% \times (\tilde{\eta}^{(modified)} - \tilde{\eta}^{(original)})/\tilde{\eta}^{(original)}$.

All experiments are performed on a workstation with Ubuntu 14.04 operating system, gcc 4.8.4 compiler, Intel MKL 11.0.4, CUDA Toolkit 9.0 software and Intel Core i7-3930K CPU 3.2 GHz and NVIDIA Titan Xp GPU hardware. The performance of the algorithms was always measured across multiple runs to ensure consistency.

5.5.1 Multi-level Schemes (CPU)

Let us first look at the impact of using Jaccard weights in popular multi-level graph partitioning schemes, that are implemented in software packages such as METIS (Karypis and Kumar, 1998). These schemes agglomerate nodes of the graph in order to create a hierarchy, where the fine level represents the original graph and the coarse level represents its reduced form. The partitioning is performed on the coarse level and results are propagated back to the fine level.

In our experiments we compute the modified vertex $v_i^{(*)}$ and edge $w_{ij}^{(*)}$ weights ahead of time and supply them to METIS as one of the parameters. We measure the quality of the partitioning using the cost function $\tilde{\eta}$ in (5.27) and plot it over different numbers of clusters for the same coPaperCitseer network. The obtained improvement in quality when using Jaccard and Jaccard-PageRank versus original weights is shown in Figure 5.7.

Notice that using Jaccard and Jaccard-PageRank weights helped improve METIS partitioning by 18% and 21% on average, respectively. This is a moderate but steady amelioration, taking values within a range of 7% to 25% for Jaccard and 15% to 26% with additional PageRank information.

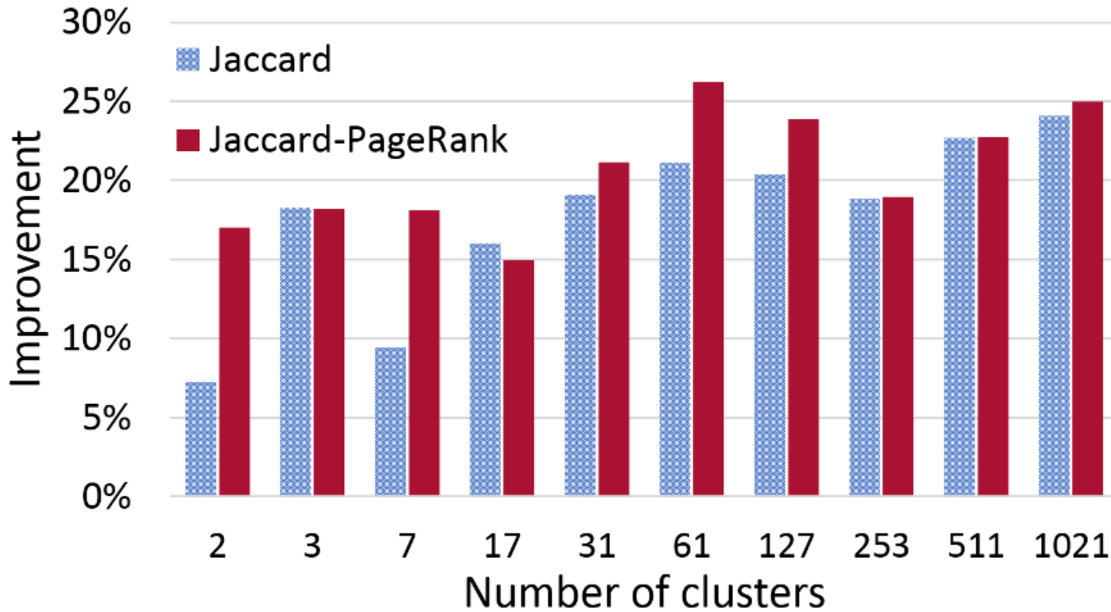


Fig. 5.7 Improvement in the quality of partitioning obtained by METIS, with Jaccard and Jaccard-PageRank for coPapersCitseer graph

5.5.2 Spectral Schemes (GPU)

Let us now look at using Jaccard weights in spectral schemes, that are implemented in the nvGRAPH library. These schemes often use the eigenpairs of the Laplacian matrix and subsequent post-processing by k-means to find the assignment of nodes into clusters.

In our experiments we measure the quality of clustering using the cost function $\tilde{\eta}$ in (5.27) and plot it over different number of cluster for the same coPapersDBLP network. The obtained improvement in quality when using Jaccard and Jaccard-PageRank versus original weights is shown in Figure 5.8. Notice that in spectral clustering it is possible to compute a smaller number of eigenpairs than clusters (Fender et al., 2017a) and in these experiments we have varied them synchronously until 32, after which we have fixed the number of eigenpairs pairs and increased the number of clusters only. The limit of 32 was chosen somewhat arbitrarily based on tradeoffs between computation time, memory usage and quality.

Notice that using Jaccard and Jaccard-PageRank weights helped improve the spectral clustering quality by 49% and 51% on average, respectively. This is a significant

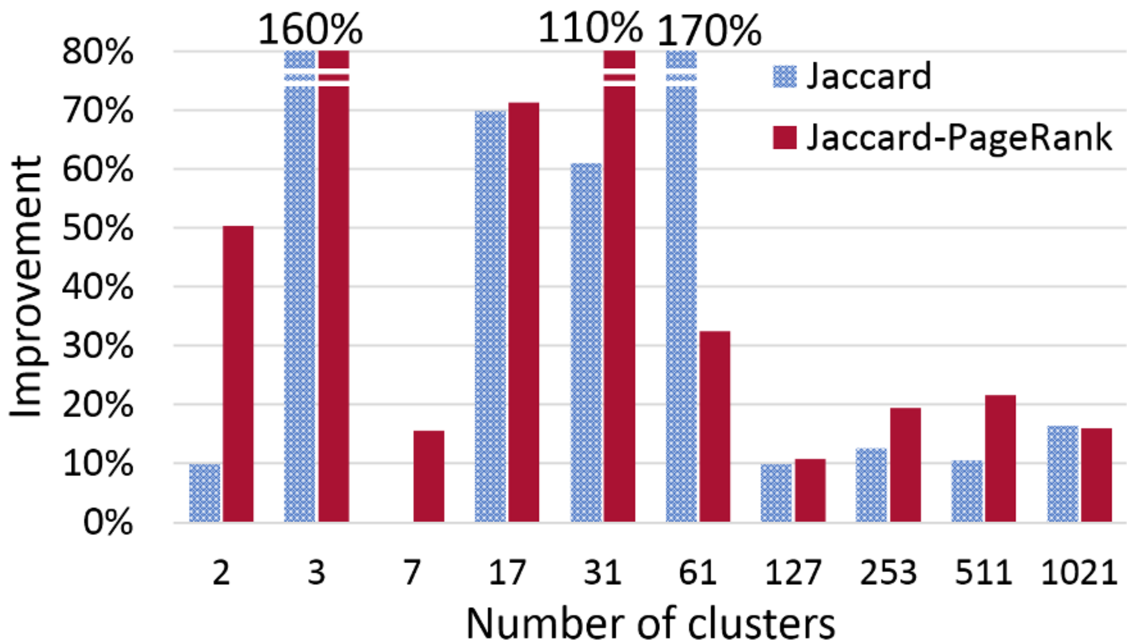


Fig. 5.8 Improvement in the quality of partitioning obtained by nvGRAPH, with Jaccard and Jaccard-PageRank for coPaperDBLP graph

but sometimes irregular amelioration, taking values within a range of −39% to 172% for Jaccard and 11% to 163% with additional PageRank information.

5.5.3 Quality Across Many Samples

Finally, let us compare the impact of using Jaccard and Jaccard-PageRank weights across samples listed in Table 5.3. In this section we fix the number of clusters to be 31, which is a prime number large enough to be relevant for real clustering applications. We maintain the same way to measure quality as described in the previous two sections. The obtained improvement in quality when using Jaccard and Jaccard-PageRank versus original weights is shown in Figure 5.9 and Table 5.4.

Notice that for these graphs the Jaccard weights help to improve the multi-level and spectral clustering quality by about 10% and 70% on average, respectively. When using additional PageRank information this improvement rises to about 15% and 80% on average, respectively. However, the improvements are not always regular, and on occasion might result in lower quality clustering.

	M-L (J)	Spect (J)	M-L (J+P)	Spect (J+P)
smallworld	14.0%	9.9%	14.0%	22.9%
coAuthorsDBLP	14.3%	52.0%	15.1%	33.1%
citationCiteseer	2.1%	-9.0%	4.5%	-20.2%
coPapersDBLP	13.1%	61.0%	11.8%	113.8%
coPapersCiteseer	19.1%	237.7%	21.2%	236.5%

Table 5.4 Improvement in the quality of partitioning obtained by nvGRAPH (Spect) and METIS (M-L), with Jaccard (J) and Jaccard-PageRank (J+P) weights

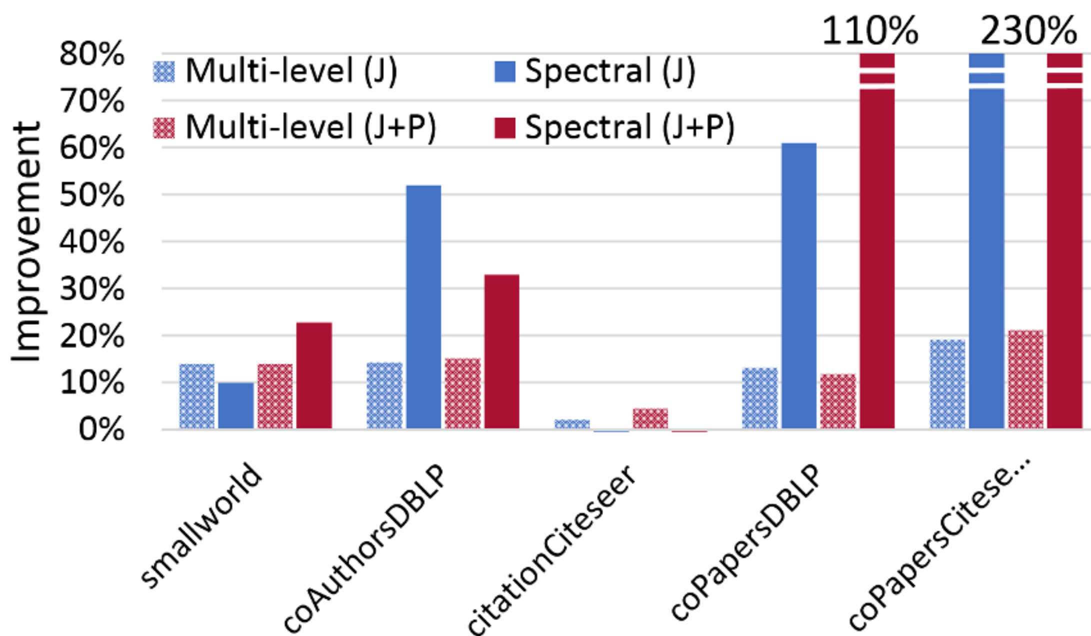


Fig. 5.9 Improvement in the quality of partitioning obtained by nvGRAPH and METIS, with Jaccard and Jaccard-PageRank weights

The spectral clustering has a more intense average amelioration but there is one case that does not benefit from using modified weights. This is consistent with the experiment of Figure 5.8. The multi-level clustering has lower average amelioration, but all cases seem to benefit from using Jaccard and Jaccard-PageRank weights.

Finally, we note that using Jaccard or Jaccard-PageRank weights on `coPapersCiteseer` network leads to an improvement over 230% for the spectral clustering approach. In this case, the high amelioration ratio happens because the spectral clustering method struggles to find a good clustering without weights that represent the local connectivity information.

5.6 Conclusion and Future Work

In this chapter we have extended the Jaccard, Dice-Sorensen and Tversky measures to graphs. Also, we have shown how to incorporate vertex weights into these metrics. In particular, we have shown how to leverage the PageRank algorithm to compute relevant vertex weights.

Also, we have developed the corresponding parallel implementation of Jaccard edge and PageRank vertex weights on the GPU. The Jaccard implementation has attained a speedup of more than 10× on GPU versus a parallel CPU code. Moreover, we have profiled the entire clustering pipeline and shown that computation of modified weights consumes no more than 20% of the total time.

Finally, in our numerical experiments we have shown that clustering and partitioning can benefit from using Jaccard and PageRank weights on real networks. In particular, we have shown that spectral clustering quality can increase by up to 3×, while we also note that the improvements are not uniform across graphs. On the other hand, for multi-level schemes, we have shown smaller but steadier improvement of about 15% on average.

Chapter 6

Multiple implicitly restarted Lanczos with nested subspaces

6.1 Introduction

The numerical technique for solving an eigenvalue problem highly depends on the properties of the graph which is seen as a sparse matrix. In the context of spectral graph analysis it is natural to use a sparse eigenvalue solver which does not modify the sparsity pattern of the input, essentially because of the memory limitations. Sparse iterative methods have been studied and showed promising results on GPU. For example, the power method and Krylov methods such as the Arnoldi method (Golub and Greif, 2006), are good illustrations as seen in Chapter 3. In particular, the implicitly restarted Arnoldi methods (IRAM) (Sorensen, 1997) showed good improvements compared to the power method on GPU for network analysis. Symmetric problems often use the Lanczos method (Matam and Kothapalli, 2011) which can be seen as a special case of the Arnoldi method. The implicitly restarted Lanczos variant (IRL) is known for its stability and good convergence with constant and reasonable memory requirements (Lehoucq, 1995; Sorensen, 1998).

In Krylov methods, the subspace size has an important impact on the performances and is chosen empirically in advance. A better strategy would be to dynamically select the best subspace size at each restart and improve the convergence even more. For the non-symmetric case, this method is called MIRAMns (Shahzadeh Fazeli et al., 2015)

and showed good results on GPU as shown in Chapter 3, (Fender et al., 2016a) and Fender et al., 2016b.

In this chapter we explain how the analysis of networks leads to large and sparse symmetric eigenvalue problem. We propose an adaptation of MIRAMns to IRL. This approach can be seen as an auto tuning technique for choosing the best subspace size in IRL. We present an efficient implementation which combines CPU and GPU strengths to compute the invariant subspace of real scale-free undirected networks, and compare against existing methods. We explain the hybrid acceleration of the implicitly restarted Lanczos method designed for large graphs and clustering problems. We highlight the main challenges for sparse iterative eigenvalue methods and symmetric cases with irregular sparsity pattern. Our solver can compute largest or smallest eigenpairs of large symmetric matrices. Both paths were evaluated upon two real spectral clustering applications which are spectral modularity maximization and edge cut minimization. We present experiments conducted on 8 real networks with up to 113,891,327 edges on 4 different GPU architectures. For instance, in the modularity clustering application presented in Chapter 4, MIRLns solver shows an average reduction of the number of iterations by 41% and an improvement of the quality of the final clustering by 36%. The resulting speedup is up to 2.7× over the regular implicitly restarted Lanczos on Nvidia P100 GPU.

In Section 6.1.1, we present key context elements regarding the spectral graph analysis application, methods related to MIRLns, and the GPU. Section 6.2 describes MIRLns approach and its implementation on GPU. Section 6.3 focuses on experimental results pointing out the efficiency of the approach for large graphs analysis problems on the GPU architecture. Concluding remarks and perspective are presented in Section 6.4.

The work presented in this chapter has been submitted for publication as (Fender et al., 2017c).

6.1.1 Spectral graph analysis and clustering

Let a graph $G = (V, E)$ be defined by its vertex V and edge E sets. The vertex set $V = \{1, ..., n\}$ represents n nodes in a graph, with each node identified by a unique

integer number $i \in V$. The set of edges $E = \{w_{i_1,j_1},...,w_{i_m,j_m}\}$ represents m weighted edges in a graph, with each undirected edge identified by $w_{i,j} \in E$ with $w_{i,j} \geq 0$. In order to address graph analysis in linear algebra the graph is represented by its adjacency matrix. The graph adjacency matrix is defined by

$$a_{i,j} = \begin{cases} w_{i,j} & \in E \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

In this chapter we focus on clustering which can be used to identify and analyse communities in social networks among many other applications. In graph theory, an eigenpair of a graph corresponds to the eigenpair of its adjacency matrix A. In practice for graph clustering application, alterations can be applied to A, depending on the clustering metrics, before solving the eigenvalue problem.

In order to provide relevant examples of improvements offered by our eigensolver on graph applications we selected two different and complementary graph clustering techniques. These techniques are based on the minimum balanced cut (Naumov and Moon, 2016; Von Luxburg, 2007) and modularity maximization (Chen et al., 2014; Newman, 2010).

Recall that the spectral formulation for those clustering problems is presented in Section 2.2.2. In particular, the spectral modularity maximization clustering was developed further in Chapter 4.

6.2 Multiple implicitly restarted Lanczos with nested subspaces

6.2.1 Proposed approach

In IRL, the choice of the subspace size remains empirical but still has an important impact on the overall success of the method. It is known that the eigen-information of interest may not appear when the size of the subspace is too small (Sorensen, 1997). If the subspace size is too large, keeping orthogonality can become an issue. It is also

suggested to avoid setting k in a way that splits clusters of eigenvalues (Sorensen, 1997). This is true for the choice of m as well.

The idea in MIRLns is to improve this point by computing several subspaces of different sizes and select the best one for each restart. This technique is derived from the existing multiple implicitly restarted Arnoldi with nested subspaces (MIRAMns). Further analysis of the nested subspaces approach for the Arnoldi method was done in (Shahzadeh Fazeli et al., 2015).
A Krylov subspace K_{m_i} is considered as better than another subspace K_{m_j} if its residual is smaller : $P(K_{m_i}) < P(K_{m_j})$, with $P(K_{m_i})$ defined in Eq. (6.2).

$$P(K_{m_i}) = max(\rho_{(\lambda_1,w_1)},...,\rho_{(\lambda_k,w_k)}) \tag{6.2}$$

where $(\lambda_1,w_1),...,(\lambda_k,w_k)$ are the extreme eigenpairs of T_{m_i}. The residual $\rho_{(\lambda_i,w_i)}$ is calculated using the Ritz estimate $| \, ||f_i|| e_i^T w_i \, |$ (Sorensen, 1997).

The cost of MIRLns in terms of matrix-vector multiplications is $m_{max} + p \times (nrc - 1)$, with $p = m_{max} - k$ which is the same as IRL with a subspace of size m_{max}. Indeed, in the first cycle the number of matrix-vector multiplications is m_{max} and for each of the restart cycles, the number of matrix-vector multiplication is $m_{max} - k$. The eigenpairs of the tridiagonal matrix $T_{m_{best}}$ can be obtained in $O(m_{best}^2)$ per step with the QR method. For a dense matrix, the space complexity of MIRLns is $n^2 + O(m_{max} \times n)$. In the context of graphs, the complexity is $O(|E| + m_{max} * |V|)$ which is the same as IRL with a subspace of size m_{max}.
In addition, notice that one restart cycle of MIRLns can be less expensive than IRL as it was shown for MIRAMns in (Shahzadeh Fazeli et al., 2015). Indeed, m_{best} is often smaller than m_{max}, as a result all computations involving elements of size m_{best} in MIRLns$(m_1,...,m_{max})$ can be cheaper than the same operation in IRL(m_{max}).

6.2.2 Hybrid acceleration

GPUs have a high parallel throughput and a good power efficiency. Thus, they should be used for the largest and most computationally intensive part as seen in Section 2.3 and 2.3.2 . However, Krylov methods like MIRLns reduce the problem into a small subspace. In general, this subspace is too small to take advantage of the GPU. Fortunately, it is possible to leverage the CPU low latency for those operations and

build a hybrid strategy based on CPU and GPU cooperation.
Initially, the network is assumed to be in the device memory in its CSR representation (Section 2.1.2). Every other data structure of the size of the graph, such as additional vertex or edges information and vectors should be on the device. The reason for that is the assumption that host/device and host bandwidth are considerably smaller than the device bandwidth, so it is primordial to avoid transfers at this scale inside the iterative process.

The parallel overall hybrid (host/device) scheme is described in Figure 6.1. Device operations are performed in parallel on throughput oriented architecture, host steps are processed sequentially at high frequency. Notice that device operations correspond to the most expensive steps of the algorithm (Section 2.3.3) while all host operations correspond to small tasks.

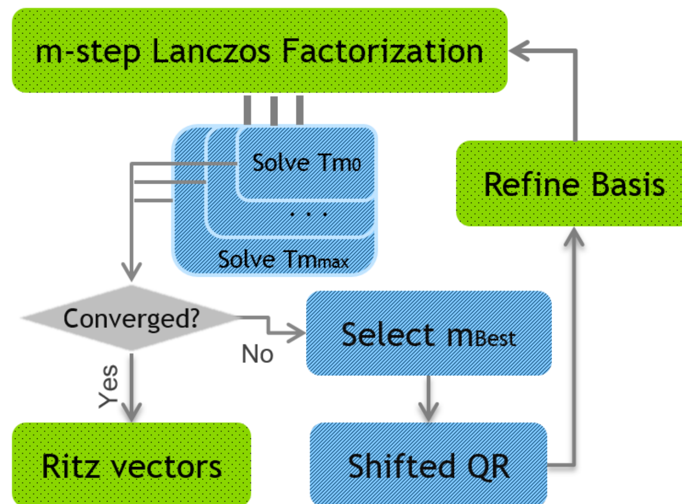


Fig. 6.1 Overview of the accelerated MIRLns solver. Green is associated to the device and blue corresponds to the host

The SpMV and the Gram-Schmidt process are done on the device, thus the Lanczos vectors are directly formed on the device, each column j of V_m corresponds to the result Av_i orthogonalized with the previous vectors. Notice that V_m is a dense matrix and will be stored in column major order by nature. The small matrix T_m is formed at the same time directly on the host, it is important to know that the size of this matrix (m) is around 20 for most graph applications. The smallest subspace size is $k+C$,

where C is a constant to make sure the minimum subspace size is large enough to capture the desired eigenvalues and ensure stability.

For each size, the quality of the newly generated subspace is evaluated. This is done by computing the eigenpairs of T_m on the host for the residual approximation. In the meantime, the next m-step Lanczos factorization can continue on the device or it can wait and potentially exit before reaching the maximum subspace size if the residual is lower than the tolerance.

When the maximum size m_{max} is reached, all residuals are compared and only the best subspace of size m_{best} is kept on the host ($T_{m_{best}}, V_{m_{best}}, f_{m_{best}}$) and other subspaces are discarded. At this point the problem is solved into the subspace and the next step is to update the basis, this is done by using the new matrix T_m^+ and Q_m^+. This part involves V_m which is composed by vectors of the size of the graph and the operation is a tall skinny dense matrix-matrix multiplication. Fortunately, this type of operation is efficiently performed on accelerators (Section 2.3.3), plus the large matrix is already on the device. Only the new subspace information is transferred, which represents a couple of matrices of size m. Finally, a new cycle can begin with exactly the same strategy, completing the m-step Lanczos factorization beginning at the $k + C^{th}$ step.

Algorithm 16 Multiple implicitly restarted Lanczos with nested subspaces

$[T_{m_i} V_{m_i}, f_{m_i}] \leftarrow$ *Lanczos-Factorization*$(A, v_1, 1, m_{max})$
while *Convergence is not reached* **do**
  Compute the eigenpairs of T_{m_i}
  Compute the residuals, stop if converged
  Select the best subspace size
  Set m, T_m, V_m, f_m *accordingly*
  Select $p = m - k$ *shifts* $\mu_1, ..., \mu_p$ *based on unwanted eigenvalues*
  for $j = p, ..., 2, 1$ **do**
    $[Q_j, R_j] \leftarrow$ *QR-Factorization*$(T_m - \mu_j I)$
    $T_m \leftarrow Q_j^H T_m Q_j$
    $Q_m \leftarrow Q_m^H Q_j$
  end for
  $f_k \leftarrow T_m(k+1, k) V_m(1:n, 1:m) Q_m(1:m, j+1) + Q_m(m, k) f_m$
  $V_m(1:n, 1:k) \leftarrow V_m(1:n, 1:m) Q_m(1:m, 1:k)$
  $[T_{m_i}, V_{m_i}, f_{m_i}] \leftarrow$ *Lanczos-Factorization*(A, V_k, k, m_{max})
end while

6.2.3 Profile

The profile of our GPU implementation for modularity maximization shows that 90% of the time is spent in the eigensolver, where the eigensolver time is dominated by the SPMV as shown in Figure 6.2.

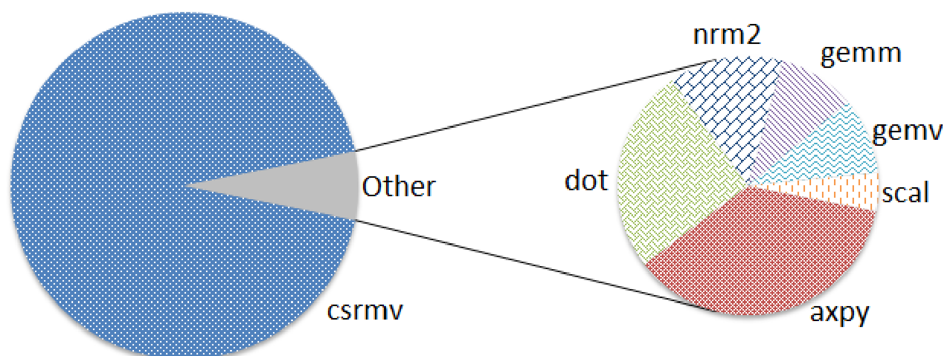


Fig. 6.2 Profiling of the implicitly restarted Lanczos eigensolver

It is possible to estimate the benefits of the GPU acceleration from the performance profile since the solver is composed by basic linear algebra subroutines (BLAS) with known performances. The BLAS are commonly categorized in three levels. The first level performs scalar and vector operations, the second performs matrix-vector operations, and the third performs matrix-matrix operations. Recall that accelerated basic operations are presented in Section 2.3.3.

There is a dozen of level 1 operations per Lanczos iteration. The complexity of those operations depends on the number of vertices, which is low compared to the number of edges. The dot product involves a reduction which scales at a rate of $\kappa * log(\kappa)$ with κ the number of cores and the vector addition scales linearly with the number of cores. Thus, in this application, level 1 operations are relatively cheap to compute because of the relative size of the input and the scalability of the operations in parallel.
The sparse matrix vector multiplication is the most expensive operation in the solver as shown in Figure 6.2. Notice that it is called at each Lanczos iteration. The complexity of this operation depends on the number of edges. There are many specialized SpMV

for GPUs (Bell and Garland, 2008) and the speedup is often between 2× and 3× against comparable parallel CPU implementation.

The matrix-matrix multiplication is known to be expensive sequentially. It appears during the implicit update of the Krylov vectors and at the very end of the method when computing the eigenvectors. In our applications this is not a primary concern because this does not happen in the inner loop. In addition, this is a tall skinny matrix-matrix multiplication where n is large but m and k are very small. This matrix multiplication is efficiently done in parallel on the GPU.

Finally, the host-device transfers are too fast to be visible on the profile because they involve very small matrices. The host part is also invisible as it operates on data sizes that are several orders of magnitude smaller than the device part.

6.3 Experiments

In Chapter 4, we selected a sample of real networks from DIMACS10, LAW and SNAP graph collections (Bader et al., 2013) and (Davis and Hu, 2011) that are relevant for clustering. We propose to reuse the data sets of Table 4.1 in this section. The column V represents the number of vertices (size of the matrix) and E column shows the number of edges of the graph (ie. the number of non zero entries in the matrix).

We let the stopping criteria for the Lanczos eigenvalue solver in Algorithm 16 be based on the maximum residual norm of the eigenpairs $max(\rho_{(\lambda_1,w_1)},...,\rho_{(\lambda_k,w_k)}) \leq 10^{-3}$. Notice that the spectral clustering application does not require very precise eigenvectors as the goal is to compare elements together. However, it still needs an accurate eigensolver to reach this precision. In this section, an iteration refers to one iteration of the m-step Lanczos factorization (Algorithm 7), where one iteration of the m-step Lanczos factorization contains one sparse matrix vector multiplication. A restart cycle refers to one cycle of MIRLns in Algorithm 16. The maximum number of iterations is 500. For consistency, we fixed the parameter corresponding to the number of clusters and the desired eigenvalues (k) to be 7 in all experiments. We choose 7 because it is a non-trivial prime number which is large enough to be relevant for the clustering application. We set the maximum subspace size $m_{max} = 15 + k$ because it is a realistic parameter for IRL.

Experiments cover the three latest Nvidia architectures : Kepler (Tesla K20c and Quadro K6000), Maxwell (Geforce Titan X) and Pascal (Tesla P100 PCIe).

The Tesla P100 experiment was performed on CentOS Linux 7.3 server, driver 375.2, Intel Xeon E5-2698v3 2.30GHz, 256 GB of memory. All other experiments were performed on Ubuntu 14.04 workstation, driver 375.0, Intel i7-3930K 3.20GHz, 8GB of memory.

Our parallel MIRLns solver Algorithm 16 is compared to IRL Algorithm 8 on GPUs within the context of spectral graph clustering. The exact same software, hardware and parameters were selected. We present two spectral clustering techniques that are relevant for evaluating MIRLns solver : modularity maximization and minimum balanced cut. The former involves finding the largest eigenpairs and the later involves finding the smallest ones. The performance improvement is measured by the reduction of number of iterations rather than time because it is more generic and independent of the architecture specifications. Notice that in IRL and MIRLns, the time of one iteration is dominated by one SpMV as shown in Figure 6.2. For completeness, we also present times and speedup results on recent hardware. The quality improvement in the application is estimated by the clustering quality.

In all cases the stopping criteria was met before reaching the maximum number of iterations. We ran MIRLns on graphs as large as `hollywood-2009` with $1,139,905$ vertices and $113,891,327$ edges (Table 4.1). In 32 bit precision, MIRLns reduces the number of iterations by $2.55\times$ and at the same time increases the final modularity quality score by 80 %. Notice that the improvements presented in this sections are directly related to adaptive selection of the subspace size as this is the only part changing between IRL and MIRLns.

6.3.1 Modularity

Figure 6.3 shows the number of iterations achieved for 32-bit (single) and 64-bit (double) precision.

First, notice that at same precision, MIRLns always reduces or matches the number of iterations of IRL. Moreover, MIRLns in 32-bit is able to beat or match the number of iterations of IRL in 64-bit in 6 cases out of 8. This enables the use of 32-bit arithmetic on cases where 64-bit gave better results, allowing additional improvements in terms

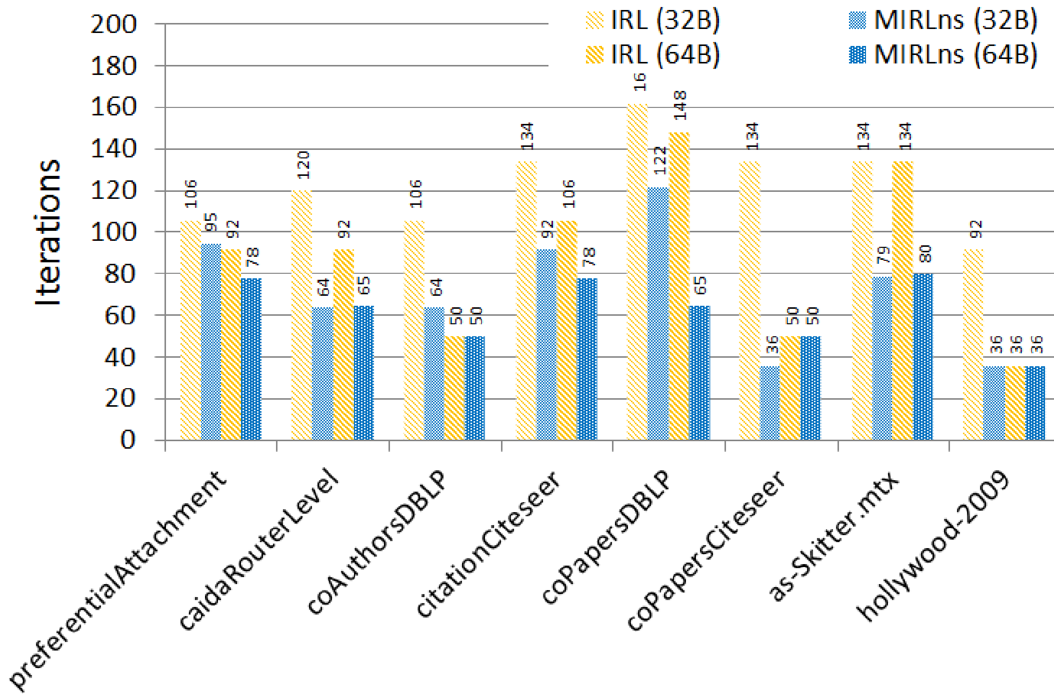


Fig. 6.3 The number of iterations achieved for 64 and 32 bit precision for spectral modularity maximization

of memory requirement and execution time. Indeed, single precision has significantly higher raw floating point performance and requires less bandwidth to access the data.

Second, notice that in both cases 64-bit precision helps to reduce the number of iterations. This is because single precision can results in roundoff errors during the computation of the Krylov subspace. Those perturbations can impact the number of iterations and the overall quality of the approximation. Their effect depends on the sensitivity of the problem that is measured by its condition number. We can empirically see these effects on Figure 6.3, a particularly relevant illustration of this behaviour happens on coPapersCiteseer network, which takes more than twice the number of iterations in 32-bit precision to converge.

Third, notice that there is better consistency between 32-bit and 64-bit results in MIRLns than in IRL. The gap between 32-bit and 64-bit is over 2× in IRL in 3 cases out of 8 while this never happens in MIRLns. Thus, MIRLns is more robust for this application.

	IRL				MIRLns			
	64b		32b		64b		32b	
#	It	Mod	It	Mod	It	Mod	It	Mod
1.	92	0.1343	106	0.1345	78	0.1473	95	0.1379
2.	92	0.4067	120	0.1553	65	0.3945	64	0.4101
3.	50	0.3155	106	0.0550	50	0.2998	64	0.0621
4.	106	0.4702	134	0.4656	78	0.4103	92	0.4539
5.	148	0.3086	162	0.2276	65	0.3335	122	0.2873
6.	50	0.3194	134	0.3013	50	0.2249	36	0.2014
7.	134	0.3305	134	0.2105	80	0.4597	79	0.3117
8.	36	0.5581	92	0.3332	36	0.5587	36	0.5854

Table 6.1 The number of iterations (It) and modularity score (Mod) achieved for 64 and 32-bit precision for spectral modularity maximization (Titan X).

Figure 6.4 shows the modularity score (Newman, 2010) achieved for 64-bit and 32-bit precision. The ideal modularity score is known to be 1 and 0 would approximately corresponds to the score of a random clustering. In 64-bit, MIRLns and IRL lead to similar results in terms of clustering quality. However, in 32-bit MIRLns leads to significant quality improvement in two cases with up to 2.64× improvement on `caidaRouterLevel` and 78% on `hollywood-2009`. Notice that this corresponds to cases where MIRLns converged faster (eg. 2.55× less iterations for `hollywood-2009`). This result makes sense because the dynamic selection of the best subspace allows to reduce unwanted roundoff errors leading to faster convergence and at the same time limits the risk of having poor approximation.

In this experiment MIRLns leads to modularity scores 36% higher in average in 32-bit and preserves the score in 64-bits. This improvement is the result of the better convergence of the eigensolver resulting of the adaptive nested subspace variant.

The Tesla P100 (PCIe) is among the most powerful accelerators at the time we write this article. It has 3584 cores and delivers 9.3 TeraFLOPS (in 32-bit) for 250 W. Hence, we selected this architecture to measure time and speedup. Notice that this experiment was done on the PCIe version of the Tesla P100 but the NVLink alternative is expected to improve the bandwidth by 5× . Thus, NVLink would increase further the speedup of MIRLns over IRL since MIRLns exchange more data with the host.

Figure 6.5 shows the speedup of MIRLns over IRL on Tesla P100. Notice that MIRLns has an average speedup of 58% in 32-bit and 40% in 64-bit. Among 16 cases

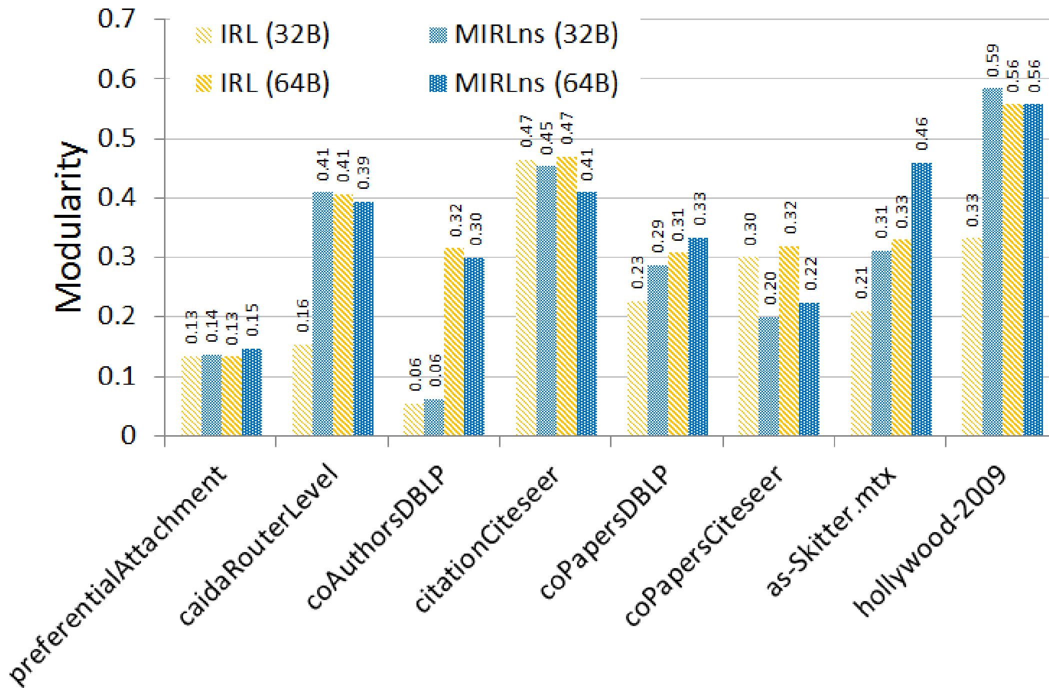


Fig. 6.4 The modularity score achieved for 64 and 32-bit precision for spectral modularity maximization (Titan X).

only 2 slowdowns were reported with the highest one being -15% and the second one -2%. These two slowdowns happened on small data sets of Table 4.1. The intensity of this slowdown and the size of the networks indicate that MIRLns did not sufficiently improve the convergence compared to IRL to cover the cost of the auto tuning of the subspace size. Still, the resulting clustering score was better with MIRLns for these two cases.

Results on Tesla P100 hardware (Figure 6.5) can be different from the results on Titan X (Figure 6.3, Figure 6.4). Indeed, changing the parallel hardware may impact the convergence, for this reason, the number of iterations reported in Table 6.1 does not systematically correspond to the speedup in time of Table 6.2. The details on how the architecture impacts MIRLns are given in Figure 6.8.

Finally, notice that with MIRLns, the spectral modularity clustering is achieved in 0.2 second for `hollywood-2009` which is the largest network of Table 4.1. Another relevant example is `coPapersDBLP` which has over 15 millions edges and where MIRLns is twice faster than IRL which allows to find a clustering in near real time (0.1 second).

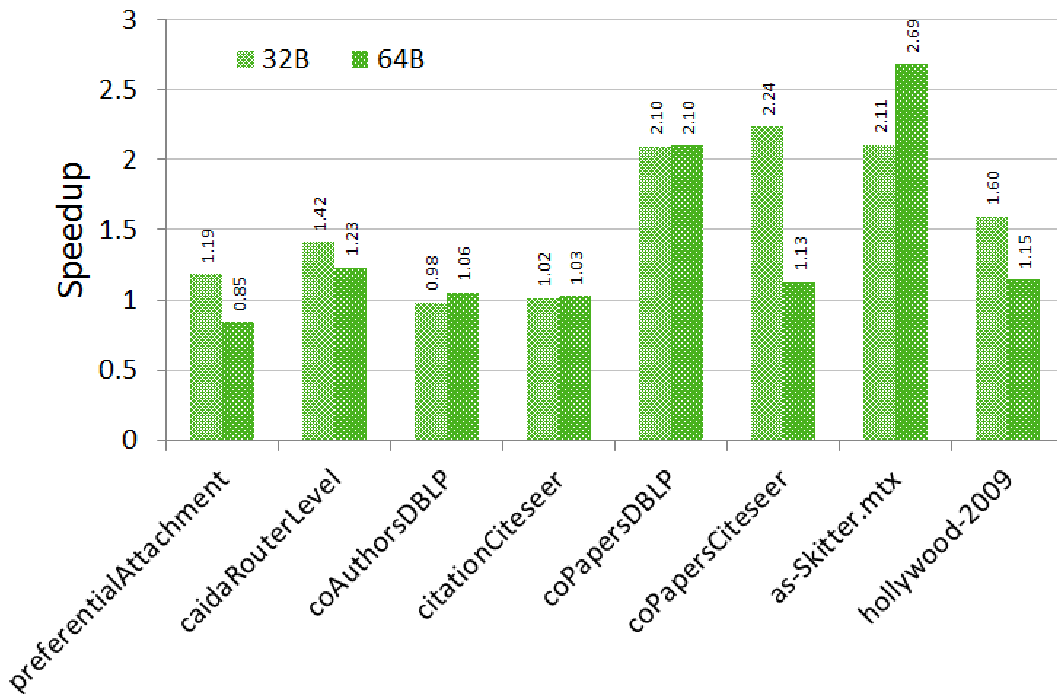


Fig. 6.5 The speedup of MIRLns over IRL on Tesla P100 for 64 and 32-bit precision in the context of spectral modularity maximization.

	IRL				MIRLns					
	32b		64b		32b			64b		
#	T(ms)	Mod	T(ms)	Mod	T(ms)	SU	Mod	T(ms)	SU	Mod
1	107.91	0.12	71.403	0.13	90.977	1.18	0.12	84.492	0.86	0.15
2	78.65	0.28	69.0	0.32	55.458	1.41	0.41	55.932	1.23	0.40
3	37.56	0.05	52.311	0.32	38.143	0.98	0.18	49.312	1.06	0.37
4	71.56	0.42	89.263	0.47	70.459	1.01	0.44	86.589	1.03	0.41
5	327.0	0.11	310.98	0.36	155.67	2.1	0.31	147.94	2.10	0.41
6	144.14	0.32	96.373	0.30	64.345	2.24	0.20	85.364	1.13	0.48
7	1328.5	0.21	1789.5	0.28	630.83	2.1	0.33	665.99	2.69	0.46
8	427.02	0.16	253.82	0.57	267.10	1.6	0.46	221.46	1.15	0.58

Table 6.2 The time (T) in millisecond, modularity score (Mod) and speedup (SU) achieved for 32 and 64-bit precision for spectral modularity maximization on Tesla P100.

6.3.2 Minimum balanced cut

Figure 6.6 shows the number of iterations achieved for 64 and 32-bit precision. MIRLns always reduces or matches the number of iterations of IRL in 64-bit and 75% of the time in 32-bit. Notice that the average number of iterations is higher than for modularity because the smallest eigenvalues of the Laplacian are often highly clustered. Although, in this application, the 64-bit precision does not clearly impact the number of iterations compared to 32-bit for both algorithms because the subspace seems more stable and less impacted by small roundoff errors. Notice that MIRLns in 32-bit is still able to beat or match the number of iterations of IRL in 64-bit in 7 cases out of 8. Finally, in 64-bit MIRLns reduces the number of iterations by 21% while improving quality by 23% over IRL in 64-bit.

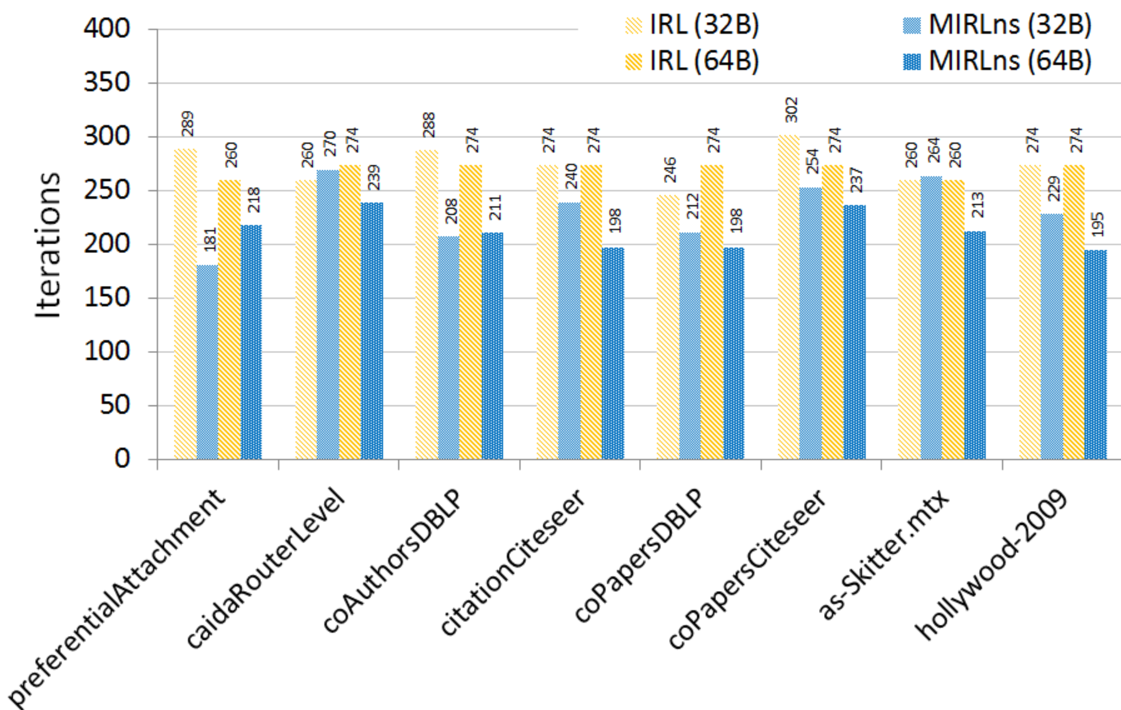


Fig. 6.6 The number of iterations achieved for 64 and 32-bit precision for spectral balanced cut minimization (Titan X).

Figure 6.7 shows the score of the number of edge cuts per vertex achieved in 64-bit and 32-bit precision. The ratio edge cut for a cluster is the ratio of the volume of edges that have an end outside of the cluster divided by the number of vertices inside the

cluster. The total ratio edge cut is the average of all cluster scores. Hence, the smaller this ratio, the better is the clustering according to this metric.

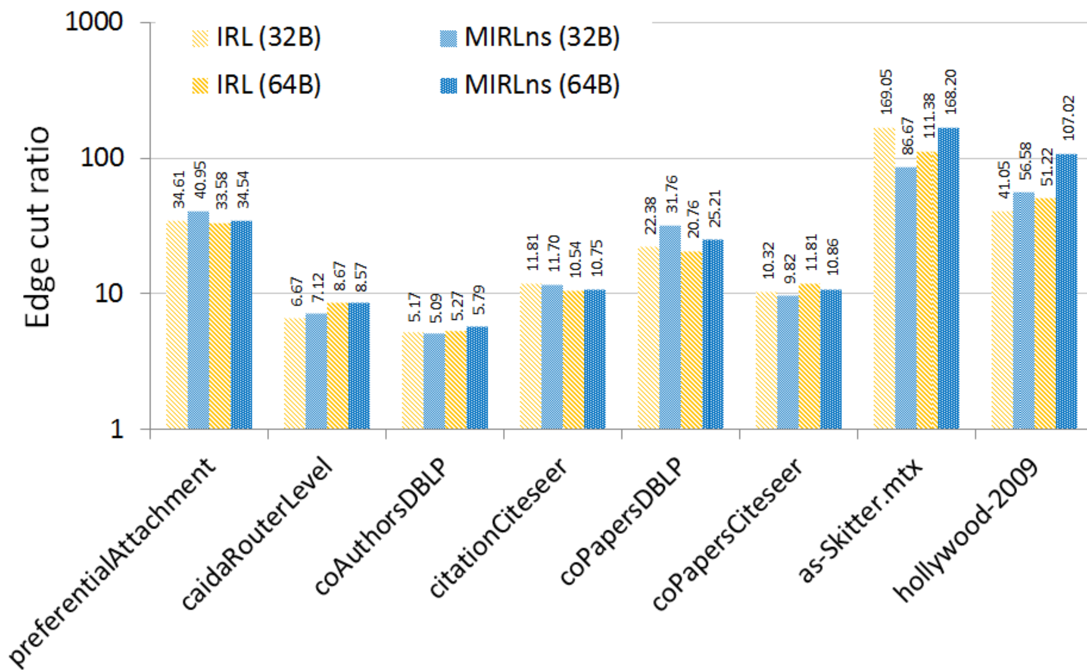


Fig. 6.7 The ratio edge cut score achieved for 64 and 32-bit precision for spectral balanced cut minimization (Titan X).

	IRL				IRLns			
	64b		32b		64b		32b	
#	It	ECR	It	ECR	It	ECR	It	ECR
1.	260	33.581	289	34.605	218	34.540	181	40.951
2.	274	8.6733	260	6.6687	239	8.5707	270	7.1242
3.	274	5.2738	288	5.1697	211	5.7861	208	5.0897
4.	274	10.538	274	11.808	198	10.750	240	11.704
5.	274	20.760	246	22.375	198	25.205	212	31.759
6.	274	11.808	302	10.324	237	10.855	254	9.8163
7.	260	111.37	260	169.04	213	168.20	264	86.665
8.	274	51.219	274	41.050	195	107.01	229	56.580

Table 6.3 The number of iterations (It) and ratio edge cut score (ECR) achieved for 64 and 32-bit precision for balanced cut minimization.

In the first 6 cases, the quality is preserved within the range of small irregular variations. However, on `as-skitter` and `hollywood-2009` there are differences of over 2× (without showing any trend for a particular solver). Since the ratio edge cut is high for both networks it is likely that they do not have a natural division in 7 clusters, which can lead to an unstable clustering approximation. Also, the variations are decoupled from the precision or the eigensolver so we suspect that this comes from the clustering pipeline such as the k-means initialization rather than the eigensolver itself.

6.3.3 Different architectures

In parallel computing, it is common to have reproducibility on the same hardware but possible variations on different architectures. Indeed, operations like reduction can lead to different results in limited arithmetic precision depending on the order of the summations. Typically, optimized parallel codes will take advantage of the architecture by looking at the cache size or the number of cores for heuristics. Considering the number of reduction in Algorithm 7 and their context, this may impact the final result. The following experiment shows how the parallel architecture impacts convergence and time of MIRLns solver. The most recent GPU is the GeForce Titan X based on Maxwell architecture and the second is the Quadro K6000 based on Kepler architecture. The Titan X is expected to have better performances while the Quadro is older but is expected to have a smaller gap between 32-bit and 64-bit performances.

Figure 6.8 shows the number of iterations achieved for 64 and 32-bit precision on those architectures.

In 32-bit, a particularly relevant case is `coPaperDPLB` where rounding approximations are harmless on K6000 but double the number of iterations on Titan X. The opposite happens on `hollywood-2009`. As expected, there are smaller differences between the two architectures when using 64-bit precision.

In all cases, the solver converged and the resulting clustering was significantly better than random assignments. Thus, MIRLns can offer reasonable consistency even when architectures are different.

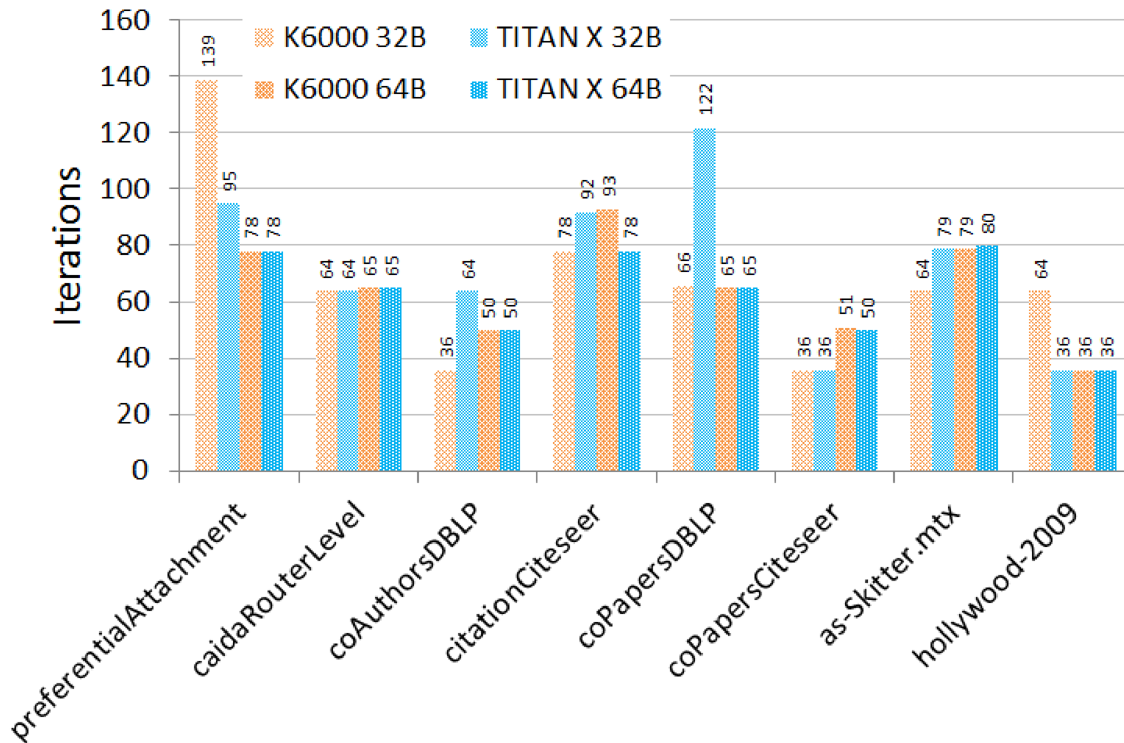


Fig. 6.8 The number of iterations achieved for 64 and 32-bit precision for spectral modularity maximization on k6000 and Titan X hardware.

6.3.4 Variation of the Krylov subspace size

It is not possible to know in advance what subspace size will have the best residual at each restart cycle, this is the reason why MIRLns dynamically detects and takes advantage of the best subspace size. In the next experiment we propose to compare the evolution of the residual of IRL and MIRLns while keeping track of the selected subspace size in MIRLns. The experiment was done in the context of 32-bit arithmetic precision in order to highlight the resilience to roundoff errors in the subspace resulting of repeated rounding.

Figure 6.9 compares IRL(m_{max}) to MIRLns($m_1, ..., m_{max}$) when looking for 7 eigenpairs with $m_{max} = 22$. It illustrates how the Kylov subspace size directly impacts the convergence on `hollywood-2009`, which is our largest data set.

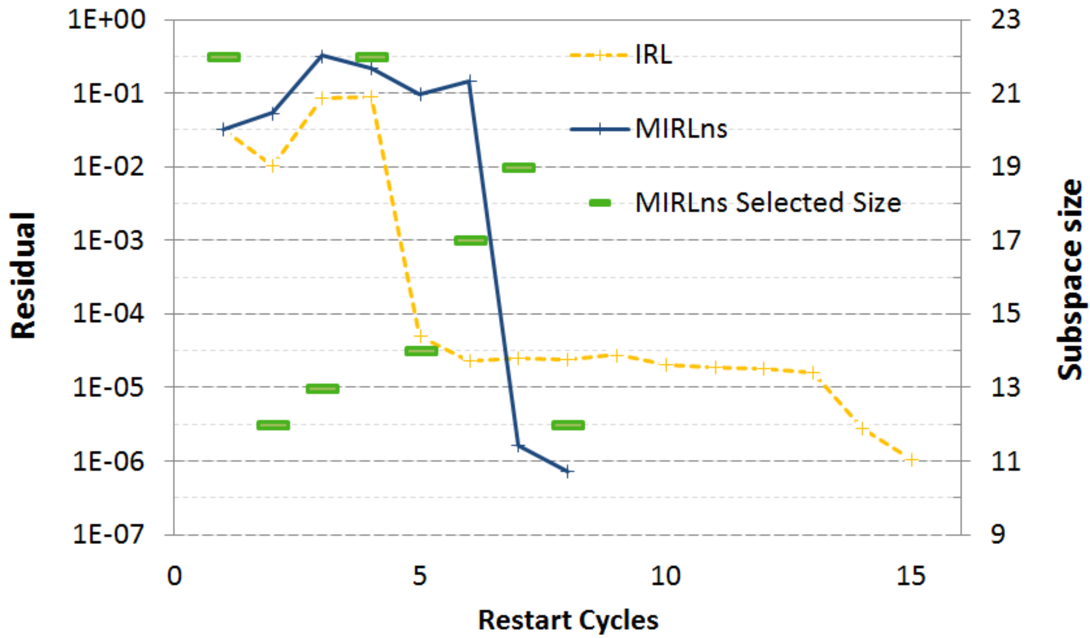


Fig. 6.9 The convergence of IRL and MIRLns on hollywood-2009 with an eye on the selected subspace size $m_{max} = 22$ (Tesla K20c).

In this experiment we let the stopping criteria be based on the maximum residual norm of the eigenpairs $max(\rho_{(\lambda_1,w_1)},...,\rho_{(\lambda_k,w_k)}) \leq 1e^{-6}$ on Tesla k20c. Both methods start with a subspace size of 22 and a residual of 0.032 at the initial restart cycle. In the next restart cycle MIRLns selects the smallest subspace available which appears to have a slightly higher residual after restart than IRL, which continues with a subspace size of 22. Keeping a large subspace pays off for IRL during the first cycles but it starts to stagnate after the 5th cycle. During the same period of time MIRLns continues to switch the subspace size at each cycle till it quickly catches up IRL at the 7th cycle and finally reaches the desired precision at the 8th cycle. Eventually, IRL will also converge at the 15th restart cycle. Roundoff is likely to be the cause of the stagnation in IRL which is often seen in Krylov methods in general. Another hypothesis is that the number of eigenpairs (k) and/or the subspace size (m_{max}) splits a natural clusters of eigenvalues. In the example of Figure 6.9 notice that IRL could reach convergence faster than MIRLns before $1e-4$ but the residual norms do not decrease steadily while those of MIRLns continue to decrease further.

MIRLns appears to suffer less from roundoff issues. It is likely because periodically selecting smaller subspaces that are more precise allows to avoid the impact

of repeated rounding onto the overall method. Moreover, changing the subspace at each restart leads to significant changes in the subspace but preserves the progress of the algorithm which has proved to be a good strategy to break stagnation phases. Finally, it may also help adapting to the natural clusters of eigenvalues. The adaptive subspace size approach remains empirical and could be improved in the future by analyzing the results of intensive experimentation.

6.4 Conclusion

We explained how spectral graph clustering of real networks leads to large symmetric eigenvalue problems. The eigenpairs of interest are a small set among the largest or smallest ones, hence, a sparse iterative method like implicitly restarted Lanczos is suitable for solving this problem.

We proposed a new approach based on the implicitly restarted Lanczos method with subspace size auto-tuning.

We implemented a hybrid, accelerated, solution to leverage GPU throughput to operate on large sparse matrices without suffering from under-occupancy when solving the small, coarse, problem. The details of the hybrid strategy are explained, and can apply to other methods with reduction-projection patterns.

We found that the implicitly restarted Lanczos method benefits from the nested subspaces variant which allows a dynamic selection of the subspace size. The parallel implementation on GPU ran on networks with a million vertices and a hundred million edges. Our experiments showed that the accelerated MIRLns solver improves robustness and achieves better performances than the regular implicitly restarted Lanczos method for different spectral graph clustering applications. For instance, a modularity clustering score of 0.58 was achieved in 0.2 seconds on Tesla P100 GPU for the network representing 113,891,327 collaborations of Hollywood actors. Moreover, our experiments indicated that the adaptive subspace size allowed to reduce floating-point arithmetic precision from double (64-bit) to single (32-bit) without quality loss in the application. In the future, we plan to analyze MIRLns further, investigate the adaptive selection of more eigensolver parameters, and experiment with half (16-bit) floating-point arithmetic precision.

Chapter 7

Conclusions and perspectives

The contributions of this thesis address several interconnected problems in the field of graphs, numerical eigenvalue problems and GPU.

First, we identified and discussed spectral graph analytics problems on GPU. We explained how graphs connect to sparse linear algebra and highlighted problems that can be addressed with an eigenvalue solver. We presented the advantages of the implicitly restarted Arnoldi and Lanczos methods in this context. We studied how the GPU architecture benefits to spectral graph analytics and exposed obstacles.

Second, we supplied the first implementation of the implicitly restarted Arnoldi method with adaptive subspaces on the GPU and experimented it on Pagerank applications. It achieved 2× to 15× speedup over the power method and 2× to 8× speedup over the regular implicitly restarted Arnoldi method. This solver allows near real time ranking of elements in graphs with tens of millions edges. We developed a hybrid solution to leverage GPU throughput to operate on large sparse matrices without suffering from under-occupancy when solving the small, coarse, problem.
In the future, the quality metric of the subspace can be studied further with the goal of improving the subspace selection. In addition, experiments showed that a high number of subspaces is not always the best strategy and a too low number either. Research can be pursued in this direction in order to find heuristics for the optimal number of nested subspaces.

Third, we revisited the modularity maximization on GPU as an eigenvalue problem. We supplied a fast graph clustering implementation which achieves 3× speedup over state-of-the-art agglomerative CPU implementation. Experimentation showed that for a given number of clusters, it is possible to reduce the number of eigenpairs without

losing quality. We also found that a natural number of clusters can be discovered by changing the number of k-means centroids while keeping the number of eigenpairs constant.
As far as we know, the spectral modularity maximization scheme is faster than agglomerative schemes but we could not always reach the same level of clustering quality. In the future, the impact of different graph features should be investigated as well as eigensolver and k-means parameters.

Fourth, we have extended the Jaccard similarity to graphs and showed how to incorporate vertex weights into this metric. We have developed the corresponding parallel implementation of Jaccard edge and PageRank vertex weights on the GPU which achieves 10× speedup over CPU. Our numerical experiments have shown that clustering and partitioning can benefit from using Jaccard and PageRank weights on real networks. In particular, we have shown that spectral clustering quality can increase by up to 3×.
In the future, the impact of other vertex weights than Pagerank should be investigated. Finally, we believe that other applications than clustering could benefit from these weights.

Fifth, we introduced an adaptive strategy for the subspace size in the implicitly restarted Lanczos method. This method generates multiple nested subspaces and selects the best at each restart cycle. We did the first implementation of this method and integrated it into two spectral clustering techniques. Experiments on GPU has shown a reduction of the number of sparse matrix vector multiplications by 41% and increased the modularity clustering quality by 36%, when comparing against the regular implicitly restarted Lanczos method on GPU. Moreover, experiments indicated better stability and showed a 2.7× speedup on newest GPU hardware.
In the future, a deeper analysis of the subspace selection could help to improve the method further. Also, our experiments indicated that the adaptive subspace size allowed to reduce floating-point arithmetic precision from double (64 bits) to single (32 bits) without quality loss in the application. This characteristic can be investigated further by experimenting with half (16 bits) floating-point arithmetic precision.

We have presented how a variety of spectral graph analytics problems are efficiently handled by the GPU and introduced new methods and techniques designed

to solve the difficulties we encountered. Pagerank and spectral clustering parallel implementations are available in the NVGRAPH library which has been distributed in the NVIDIA CUDA Toolkit.

Communications

Peer-reviewed publications

Parallel implicitly restarted Lanczos with nested subspaces and network clustering applications
Alexandre Fender, Nahid Emad, Serge Petiton and Maxim Naumov
Submitted for publication, 2017.

Parallel Jaccard and Related Graph Clustering Techniques
Alexandre Fender, Nahid Emad, Serge Petiton, Joe Eaton and Maxim Naumov
Proceedings of Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA) held at SC'17, accepted for publication, 2017b.

Parallel modularity clustering
Alexandre Fender, Nahid Emad, Serge Petiton and Maxim Naumov
Proceedings of the International Conference on Computational Science (ICCS), MATH-EX workshop, Procedia Computer Science, 108:1793-1802, 2017a.

Leveraging accelerators in the multiple implicitly restarted Arnoldi method with nested subspaces
Alexandre Fender, Nahid Emad, Serge Petiton and Joe Eaton
Proceedings of the IEEE International Conference on Emerging Technologies and Innovative Business Practices for the Transformation of Societies (EmergiTech), 389–394, 2016b.

Accelerated hybrid approach for spectral problems arising in graph analytics
Alexandre Fender, Nahid Emad, Joe Eaton and Serge Petiton
Proceedings of the International Conference on Computational Science (ICCS), Procedia Computer Science, 80:2338–2347, 2016a.

A fine-grained parallel model for the Fast Iterative Method in solving Eikonal equations
Florian Dang, Nahid Emad and Alexandre Fender
Proceedings of the IEEE International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 152-157, 2013

Invited talks

Spectral Graph Analysis with Unite and Conquer Approach
Nahid Emad, Serge Petiton, Alexandre Fender, Joe Eaton and Maxim Naumov
SIAM Conference on Parallel Processing for Scientific Computing, Tokyo, Japan, 2018.

Parallel spectral clustering
Alexandre Fender, Nahid Emad, Serge Petiton and Maxim Naumov
GPU Technology Conference, San Jose, USA, 2017.

Spectral graph partitioning
Maxim Naumov, Tim Moon and Alexandre Fender
SIAM Conference on Computational Science & Engineering, Atlanta, USA, 2017.

Graph analytics on GPU
Alexandre Fender
Teratec Forum, Palaiseau, France, 2016.

Emerging GPGPU applications,
Alexandre Fender, Nahid Emad, Serge Petiton and Joe Eaton
HPCSeminar at CEA (Alternative Energies and Atomic Energy Commission), Bruyères-le-Châtel, France, 2015.

Toward accelerated graph analytics
Alexandre Fender, Nahid Emad, Serge Petiton and Joe Eaton
Grace seminar at Maison de la Simulation, Saclay, France, 2015.

References

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition.

Arnoldi, W. E. (1951). The principle of minimized iteration in the solution of the matrix eigenproblem. *Quart. Appl. Math.*, 9:17–29.

Arthur, D. and Vassilvitskii, S. (2007). K-Means++: the Advantages of Careful Seeding. *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, 8:1027–1025.

Auer, B. O. F. (2013). *GPU Acceleration of Graph Matching, Clustering and Partitioning*. PhD thesis, Utrecht University.

Bader, D. A., Meyerhenke, H., Sanders, P., and Wagner, D. (2013). Graph Partitioning and Graph Clustering. *Contemporary Mathematics*, 588:73–82.

Bai, Z. and Demmel, J. (1989). On a Block Implementation of Hessenberg Multishift Qr Iteration. *International Journal of High Speed Computing*, 01(01):97–112.

Bai, Z., Demmel, J., Dongarra, J., Ruhe, A., and van der Vorst, H. (2000). Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide. *SIAM, Philadelphia, PA*, 11:316.

Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *Science*, 286(5439):11.

Barnard, S. T. and Simon, H. D. (1994). Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117.

Bastian, M., Heymann, S., and Jacomy, M. (2009). Gephi: An Open Source Software for Exploring and Manipulating Networks. *Third International AAAI Conference on Weblogs and Social Media*, pages 361–362.

Batagelj, V. and Brandes, U. (2005). Efficient generation of large random networks. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 71(3):1–5.

Bell, N. and Garland, M. (2008). Efficient Sparse Matrix-Vector Multiplication on CUDA. Technical report, NVIDIA.

References

Berkhin, P. (2005). A Survey on PageRank Computing. *Internet Mathematics*, 2(1):73–120.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10008(10):6.

Brandes, U., Delling, D., Gaertler, M., Gorke, R., Hoefer, M., Nikoloski, Z., and Wagner, D. (2008). On Modularity Clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188.

Brezinski, C., Redivo-Zaglia, M., and Serra-Capizzano, S. (2005). Extrapolation methods for PageRank computations. *Comptes Rendus Mathematique*, 340(5):393–397.

Brody, A. (1997). The Second Eigenvalue of the Leontief Matrix. *Economic Systems Research*, 9(3):253–258.

Bryan, K. and Leise, T. (2006). The $25,000,000,000 Eigenvector: The Linear Algebra behind Google. *SIAM Review*, 48(3):569–581.

Calvetti, D., Reichel, L., and Sorensen, D. C. (1994). An implicitly restarted Lanczos method for large symmetric eigenvalue problems. *Electronic Transactions on Numerical Analysis*, 2(March):1–21.

Chakrabarti, D., Zhan, Y., and Faloutsos, C. (2004). R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. Society for Industrial and Applied Mathematics, Philadelphia, PA.

Chen, M., Kuzmin, K., and Szymanski, B. K. (2014). Community detection via maximization of modularity and its variants. *IEEE Transactions on Computational Social Systems*, 1(1):46–65.

Chung, F. R. K. (1997). *Spectral Graph Theory*. American Mathematical Soc.

Clauset, A., Newman, M. E. J., and Moore, C. (2004). Finding community structure in very large networks. *Cond-Mat/0408187*, 70:066111.

Davis, T. a. and Hu, Y. (2011). The university of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25.

Deo, N., Jain, A., and Medidi, M. (1994). An optimal parallel algorithm for merging using multiselection. *Information Processing Letters*, 50(2):81–87.

Dhillon, I., Guan, Y., and Kulis, B. (2005). A fast kernel-based multilevel algorithm for graph clustering. *Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining - KDD '05*, page 629.

Dice, L. R. (1945). Measures of the Amount of Ecologic Association Between Species. *Ecology*, 26(3):297–302.

Donath, W. E. and Hoffman, a. J. (1973). Lower Bounds for the Partitioning of Graphs.

Dongarra, J. J., Meuer, H. W., Strohmaier, E., and Others (2017). TOP500 supercomputer.

Dubois, J., Calvin, C., and Petiton, S. (2011). Accelerating the explicitly restarted Arnoldi method with GPUs using an autotuned matrix vector product. *SIAM Journal on Scientific Computing*, 33(5):3010–3019.

Emad, N. and Petiton, S. (2016). Unite and conquer approach for high scale numerical computing. *Journal of Computational Science*, pages 1–10.

Emad, N., Petiton, S., and Edjlali, G. (2005). Multiple Explicitly Restarted Arnoldi Method for Solving Large Eigenproblems. *SIAM Journal on Scientific Computing*, 27(1):253–277.

Emad, N., Shahzadeh-Fazeli, S. A., and Dongarra, J. (2006). An asynchronous algorithm on the NetSolve global computing system. *Future Generation Computer Systems*, 22(3):279–290.

Eom, Y. H., Frahm, K. M., Benczúr, A., and Shepelyansky, D. L. (2013). Time evolution of wikipedia network ranking. *European Physical Journal B*, 86(12).

Ermann, L., Chepelianskii, A. D., and Shepelyansky, D. L. (2012). Toward two-dimensional search engines. *Journal of Physics A: Mathematical and Theoretical*, 45(27):275101.

Ermann, L., Frahm, K. M., and Shepelyansky, D. L. (2013). Spectral properties of Google matrix of Wikipedia and other networks. *European Physical Journal B*, 86(5).

Ermann, L., Frahm, K. M., and Shepelyansky, D. L. (2015). Google matrix analysis of directed networks. *Reviews of Modern Physics*, 87(4):1261–1310.

Fender, A. (2014). *Scalable platforms for graph analytics on GPU*. M.sc thesis, Ecole Centrale Paris & University of Versailles.

Fiedler, M. (1973). Algebraic Connectivity of Graphs. *Czechoslovak Mathematical Journal*, 23(2):298–305.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960.

Fortunato, S. (2010). Community detection in graphs. *Physics Reports*, 486(3-5):75–174.

Fu, Z., Personick, M., and Thompson, B. (2014). MapGraph. *Proceedings of Workshop on GRAph Data management Experiences and Systems - GRADES'14*, pages 1–6.

Gantz, J. and Reinsel, D. (2012). THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digi tal Shadows, and Biggest Growth in the Far East. *Idc*, 2007(December 2012):1–16.

References

Girvan, M. and Newman, M. E. J. (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(12):7821–7826.

Golub, G. H. and Greif, C. (2006). An Arnoldi-type algorithm for computing page rank. *BIT Numerical Mathematics*, 46(4):759–771.

Golub, G. H. and Van Der Vorst, H. A. (2000). Eigenvalue computation in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1-2):35–65.

Greathouse, J. L. and Daga, M. (2014). Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2015-Janua(January):769–780.

Green, O. and Bader, D. A. (2016). cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *IEEE High Performance Extreme Computing Conference (HPEC)*, number September, pages 1–6.

Hardouin, L., Cottenceau, B., Lagrange, S., and Corronc, E. L. (2008). Performance analysis of linear systems over semiring with additive inputs. *Proceedings - 9th International Workshop on Discrete Event Systems, WODES' 08*, 3:43–48.

Hilbert, M. and Lopez, P. (2011). The World's Technological Capacity to Store, Communicate, and Compute Information. *Science*, 332(6025):60–65.

Horn, R. a. and Johnson, C. R. (1986). *Matrix Analysis*. Cambridge University Press.

Jaccard, P. (1902). Lois de distribution florale dans la zone alpine. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 38:67–130.

Jaja, J. (1992). *Introduction to Parallel Algorithms*. Addison-Wesley, addison-we edition.

Kanungo;, T. (2000). An efficient k-means clutering algorithm: analysis and implementation. *Proceedings of the 16th ACM symposium on Computational Geometry*, 24(7):881–892.

Karypis, G. and Kumar, V. (1998). A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392.

Kepner, J. (2011). *Graph Algorithms in the Language of Linear Algebra*, volume 67. SIAM.

Knyazev, A. V. (2001). Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. *SIAM Journal on Scientific Computing*, 23(2):517–541.

Krieger, W. (1974). On the uniqueness of the equilibrium state. *Mathematical Systems Theory*, 8(2):97–104.

Kunegis, J. (2015). Handbook of Network Analysis KONECT – the Koblenz Network Collection. *Proceedings of the 22Nd International Conference on World Wide Web Companion*, pages 1–56.

Kwak, H., Lee, C., Park, H., and Moon, S. (2010). What is Twitter, a Social Network or a News Media? *the 19th international conference on World wide web*, pages 591–600.

Lanczos, C. (1950). An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 45(4):255.

Langville, A. and Meyer, C. (2003). Deeper Inside PageRank.

Langville, A. N. and Meyer, C. D. (2006). Updating Markov Chains with an Eye on Google's PageRank. *SIAM Journal on Matrix Analysis and Applications*, 27(4):968–987.

Lasalle, D. and Karypis, G. (2015). Multi-threaded modularity based graph clustering using the multilevel paradigm. *Journal of Parallel and Distributed Computing*, 76:66–80.

Lee, C., Ro, W. W., and Gaudiot, J.-L. (2014). Boosting CUDA Applications with CPU–GPU Hybrid Computing. *International Journal of Parallel Programming*, 42(2):384–404.

Lehoucq, R. B. (1995). *Analysis and Implementation of an Implicitly Restarted Arnoldi Iteration*. PhD thesis, Rice University.

Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., and Ghahramani, Z. (2010). Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11:985–1042.

Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford Large Network Dataset Collection.

Levandowsky, M. and Winter, D. (1971). Distance between Sets. *Nature*, 234(5323):34–35.

Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55.

Liu, Z., Emad, N., Amor, S. B., and Lamure, M. (2013). A parallel IRAM algorithm to compute PageRank for modeling epidemic spread. *Proceedings - Symposium on Computer Architecture and High Performance Computing*, pages 120–127.

Livne, O. E. and Brandt, A. (2012). Lean Algebraic Multigrid (Lamg): Fast Graph Laplacian Linear Solver. *Siam Journal on Scientific Computing*, 34(4):B499–B522.

Lloyd, S. P. (1982). Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137.

Maschhoff, K. J. and Sorensen, D. C. (1996). P_ARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures. In *PARA '96 Proceedings of the Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization*, pages 478–486. Springer, Berlin, Heidelberg.

Matam, K. K. and Kothapalli, K. (2011). GPU accelerated Lanczos algorithm with applications. *Proceedings - 25th IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2011*, pages 71–76.

Mattson, T., Bader, D., Berry, J., Buluc, A., Dongarra, J., Faloutsos, C., Feo, J., Gilbert, J., Gonzalez, J., Hendrickson, B., Kepner, J., Leiserson, C., Lumsdaine, A., Padua, D., Poole, S., Reinhardt, S., Stonebraker, M., Wallach, S., and Yoo, A. (2013). Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–2. IEEE.

Merrill, D. and Garland, M. (2016). Merge-based sparse matrix-vector multiplication (SpMV) using the CSR storage format. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '16*, pages 1–2, New York, New York, USA. ACM Press.

Mohri, M. (2002). Semiring Frameworks and Algorithms for Shortest-Distance Problems. *Journal of Automata, Languages and Combinatorics*, 7(3):321–350.

Moore, G. (1965). Cramming More Components Onto Integrated Circuits. *Electronics*, 38(1):82–85.

Morgan, R. B. (1996). On restarting the Arnoldi method for large nonsymmetric eigenvalue problems. *Mathematics of Computation*, 65(215):1213–1231.

Nagasaka, H., Maruyama, N., Nukada, A., Endo, T., and Matsuoka, S. (2010). Statistical power modeling of GPU kernels using performance counters. *2010 International Conference on Green Computing, Green Comp 2010*, pages 115–122.

Naumov, M. and Moon, T. (2016). Parallel Spectral Graph Partitioning. Technical report, Nvidia.

Newman, M. E. J. (2002). Assortative Mixing in Networks. *Physical Review Letters*, 89(20):208701.

Newman, M. E. J. (2003). The Structure and Function of Complex Networks. *SIAM Review*, 45(2):167–256.

Newman, M. E. J. (2006). Modularity and community structure in networks. *Pnas*, 103(23):8577–8582.

Newman, M. E. J. (2010). *Networks: an introduction*. Oxford University Press.

Newman, M. E. J. and Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 69(2 2):1–16.

Ng, A. Y., Jordan, M. I., and Weiss, Y. (2001). On Spectral Clustering: Analysis and an algorithm. *Advances in Neural Information Processing Systems*, pages 849–856.

Nguyen, D., Lenharth, A., and Pingali, K. (2013). A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, pages 456–471, New York, New York, USA. ACM Press.

NVIDIA (2017). CUDA Toolkit.

Odeh, S., Green, O., Mwassi, Z., Shmueli, O., and Birk, Y. (2012). Merge path - Parallel merging made simple. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012*, pages 1611–1618.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007). A Survey of General Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1998). The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab.

Palmer, C. and Steffan, J. (2000). Generating network topologies that obey power laws. *Globecom '00 - IEEE. Global Telecommunications Conference. Conference Record (Cat. No.00CH37137)*, 1:434–438.

Pelleg, D., Pelleg, D., Moore, A., and Moore, A. (2000). X-means: Extending K-means with efficient estimation of the number of clusters. *Proceedings of the Seventeenth International Conference on Machine Learning table of contents*, pages 727–734.

Petiton, S. G. and Emad, N. (1996). A data parallel scientific computing introduction. *Proceeding of The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, 1132:45–64.

Radke, R. J. (1996). *A Matlab Implementation of the Implicitly Restarted Arnoldi Method for Solving Large-Scale Eigenvalue Problems Master of Arts A Matlab Implementation of the Implicitly Restarted Arnoldi Method for Solving Large-Scale Eigenvalue Problems*. PhD thesis, Rice University.

Rogers, D. J. and Tanimoto, T. T. (1960). A Computer Program for Classifying Plants. *Science*, 132(3434).

Ross, S. M. (2007). *Introduction to Probability Models*. Academic Press.

Saad, Y. (1980). Variations on Arnoldi's method for computing eigenelements of large unsymmetric matrices. *Linear Algebra and Its Applications*, 34(C):269–295.

Saad, Y. (1992). Numerical Methods for Large Eigenvalue Problems. In *Numerical Methods for Large Eigenvalue Problems*, volume 66, pages 1–27. Society for Industrial and Applied Mathematics.

Santisteban, J. and Tejada Carcamo, J. L. (2015). Unilateral Jaccard similarity coefficient. In *CEUR Workshop Proceedings*, volume 1393, pages 23–27.

Schaeffer, S. E. (2007). Graph clustering. *Computer Science Review*, 1(1):27–64.

Seshadhri, C., Pinar, A., and Kolda, T. G. (2011). An in-depth study of stochastic Kronecker graphs. *Proceedings - IEEE International Conference on Data Mining, ICDM*, 67:587–596.

Shahzadeh Fazeli, S. A., Emad, N., and Liu, Z. (2015). A key to choose subspace size in implicitly restarted Arnoldi method. *Numerical Algorithms*, 70(2):407–426.

Smyth, S. and White, S. (2005). A spectral clustering approach to finding communities in graphs. *Proceedings of the 5th SIAM International Conference on Data Mining*, pages 76–84.

Sorensen, D. C. (1997). Implicitly Restarted Arnoldi/Lanczos Methods for Large Scale Eigenvalue Calculations. In Keyes, D. E., , Sameh, A., , and Venkatakrishnan, V., editors, *Parallel Numerical Algorithms*, pages 119–165. Springer Netherlands.

Sorensen, D. C. (1998). Deflation for Implictly Restarted Arnoldi Methods. *SIAM Journal on Matrix Analysis and Applications*.

Sørensen, T. J. (1948). A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons. *Biol. Skr.*, 5.

Stüben, K. (2001). A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1-2):281–309.

Sundaram, N., Satish, N. R., Patwary, M. M. A., Dulloor, S. R., Vadlamudi, S. G., Das, D., and Dubey, P. (2015). GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225.

Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, pages 1–9.

Tremblay, J. C. and Carrington, T. (2007). A refined unsymmetric Lanczos eigensolver for computing accurate eigentriplets of a real unsymmetric matrix. *Electronic Transactions on Numerical Analysis*, 28:95–113.

Tversky, A. (1977). Features of Similarity. In *Readings in Cognitive Science*, pages 290–302. Elsevier.

Vanderstraeten, D. (1999). A stable and efficient parallel block Gram-Schmidt algorithm. *Euro-Par'99 Parallel Processing*, pages 1128–1135.

Von Luxburg, U. (2007). A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416.

Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. (2016). Gunrock. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '16*, pages 1–12, New York, New York, USA. ACM Press.

Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*, 393(June):440–442.

Zachary, W. W. (1977). An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 33(4):452–473.

Appendix A

Resumé en Français

A.1 Motivations

Au cours des 30 dernières années, la capacité de stockage mondiale de l'information a pratiquement doublé tous les 3 ans, ce qui donne lieu à des décennies de données hétérogènes (Hilbert and Lopez, 2011). De plus, la récente transformation des modes de communication et les progrès techniques ont conduit à une explosion du volume des données. Pourtant, il est estimé que seulement 1% de l'information digitale est actuellement analysée (Gantz and Reinsel, 2012).
Simultanément, la fréquence d'horloge des processeurs a cessé d'augmenter provoquant la démocratisation des architectures parallèles et la nécessité d'une nouvelle génération de logiciels (Sutter, 2005).
Depuis lors, le calcul haute performance (HPC) et les techniques parallèles pour résoudre des problèmes complexes de calcul ont atteint des niveaux sans précédent. Les machines les plus puissantes (Dongarra et al., 2017) ont maintenant plus d'un million de cœurs et s'approchent de l'exaFLOPS, ce qui correspond à un milliard de milliards de calculs par seconde.
Nous sommes donc a un tournant de la science où la technologie et les données ouvrent de nouvelles perspectives de transformation de la société à travers l'analyse des données et l'intelligence artificielle.

A.1.1 Analyse de graphe

De nombreuses applications récentes modélisent l'information sous forme de relations entre entités abstraites. Une structure intuitive pour ce modèle est un graphe. Chaque noeud peut représenter une personne, un lieu, une chose et chaque relation représente la façon dont deux nœuds sont associés. Cela permet d'exprimer toutes sortes de données, des réseaux sociaux aux routes, aux réseaux de neurones, à Internet, aux populations ou à toute autre concept défini par des relations (Newman, 2010).
L'analyse de graphe est la science qui consiste a analyser la structure globale ou la nature des composants individuels d'un graphe. Cela permet de trouver des informations clés telles que des communautés, des entités importantes ou des chemins dans le graphe.
Ces problèmes peuvent être décrits en termes d'opérations linéaires sur des tableaux de données (par exemple, des vecteurs et des matrices). La plupart des éléments des systèmes qui en résultent sont en général nuls car chaque élément est connecté à une fraction de l'ensemble des entités du réseau. Ces types de problèmes font partie de l'algèbre linéaire creuse. Les méthodes et les algorithmes creux efficaces équilibrent souvent les contraintes entre stockage, coût de calcul et stabilité (Kepner, 2011). Les problèmes d'analyse structurale avancés font souvent parti des problèmes polynomiaux non déterministes de type NP-difficiles. La classe de complexité théorique NP-difficiles contient des problèmes particulièrement dur à résoudre par nature. En pratique, la définition formelle du problème est souvent réduite à un autre problème plus simple qui peut être résolu en un temps raisonnable.
Cette thèse cible deux sujets majeurs d'analyse de graphe qui sont le classement (ranking) et le groupement (clustering). Le premier problème étant de trouver l'importance de chaque nœud dans un graphe (Page et al., 1998) et le second étant de trouver des sous-ensembles similaires (Schaeffer, 2007).

A.1.2 Problèmes de valeurs propres

En algèbre linéaire, un vecteur propre v d'une transformation linéaire est un vecteur dont la direction ne change pas lorsque cette transformation lui est appliquée. La transformation linéaire peut être représentée comme une matrice carrée A et cette condition peut donc s'écrire $Av = \lambda v$ où λ est un scalaire appelé valeur propre associée

au vecteur propre v.
Beaucoup d'applications dans les domaines de la santé, de l'agriculture, de la publicité, de l'électromagnétique, de l'énergie, du contrôle optimal ou encore la finance conduisent à des problèmes de valeurs propres de très grandes tailles. Chaque problème ayant ses propres spécificités, cela ouvre à de vastes possibilités pour proposer de nouvelles méthodes scientifiques de calcul haute performance.
Les sous-espaces propres des graphes contiennent des informations clés qui peuvent être utiles à plusieurs fins telles que classement et regroupement (Chung, 1997). Les graphes correspondent a de grandes matrices creuses et les méthodes de Krylov sont donc spécialement indiquées car elles permettent de trouver rapidement des approximations précises des valeurs propres en réduisant la taille du problème (Bai et al., 2000). De plus, les méthodes de valeurs propres de Krylov sont reconnues pour leur efficacité à grande échelle (Maschhoff and Sorensen, 1996). Dans ces solveurs, de nombreux paramètres ont un impact direct sur l'efficacité, l'évolutivité, la résilience et la consommation d'énergie. Les meilleurs paramètres diffèrent d'un cas à l'autre et, en général, de nombreux réglages optimaux sont inconnus à l'avance. Pour surmonter ce problème, l'idée d'adapter automatiquement les paramètres au moment de l'exécution a récemment donné des résultats prometteurs (Shahzadeh Fazeli et al., 2015). Dans cette thèse, nous nous concentrons particulièrement sur l'amélioration des méthodes implicites d'Arnoldi et de Lanczos (Sorensen, 1997) pour l'analyse de graphe.

A.1.3 Accélération matérielle

Le processeur graphique (GPU, de l'anglais Graphics Processing Unit) est un accélérateur matériel qui est aujoud'hui l'une des plates-formes de calcul haute performance les plus accessibles au grand publique. Initialement conçus pour exécuter des calculs liés aux graphiques pour des applications telles que les jeux vidéos et la visualisation, les GPU ont évolué en unités de calcul parallèle plus générales et économiques (Owens et al., 2007). Les GPU ont un degré de parallélisme significativement plus élevé que les processeurs multicœurs standard, mais avec des cœurs plus simples. En comparaison avec les CPU, les GPU consacrent plus de transistors aux unités logiques et arithmétiques et moins aux caches et aux structures de contrôle. Les GPU peuvent donc fournir une meilleure efficacité énergétique puisque les cœurs sont dédiés au

calcul et en même temps fournir une performance maximale sur des tâches hautement parallèles.
Pour de nombreuses applications la bande passante de la mémoire est le facteur limitant la performance. Dans les GPU, la mémoire a un rôle particulier car l'architecture est conçue pour accéder et modifier autant de mémoire que possible le plus rapidement possible. Cette propriété profite directement à l'analyse de graphe et à l'algèbre linéaire creuse où les algorithmes sont souvent limités par les accès mémoire. Dans cette thèse nous proposons d'étudier l'analyse spectrale des graphes pour les accélérateurs de calcul et en particulier les GPU.

A.2 Structure de la thèse et contributions

Cette thèse associe l'algèbre linéaire numérique creuse à la science des données grâce à l'analyse spectrale de graphes. Nous étudions deux problèmes de graphes du point de vue spectral qui sont le classement et le groupement. Nous proposons de nouvelles techniques pour résoudre les problèmes de valeurs propres de grande echelle qui apparaissent dans l'analyse de graphe, dans le but d'utiliser efficacement les générations actuelles et futures d'architectures parallèles. Pour chaque solution proposée, nous présentons une implémentation sur GPU et des expérimentations sur de grands ensembles de données. Les résultats montrent une amélioration des performances des solveurs de valeurs propres dans les applications de calcul scientifique et d'analyse de graphes.

Cette thèse repose sur des décennies de progrès scientifique. Le Chapitre 2 est consacré à la présentation et a l'état de l'art du sujet de recherche.

Dans le Chapitre 3, nous proposons d'utiliser les accélérateurs dans la méthode d'Arnoldi implicitement redémarrée avec des sous-espaces imbriqués (MIRAMns). Nous présentons un solveur parallèle rapide pour calculer les valeurs propres dominantes des graphes orientés tels que les chaines de Markov. Nous expliquons la première implémentation de cette méthode sur GPU et les optimisations pour les graphes.

Les expérimentations dans le contexte des applications de Pagerank montrent une convergence plus rapide par rapport à la méthode de la puissance. L'accélération moyenne est de 2× pour le cas simple et de 15× pour les cas compliqués avec des valeurs propres groupées, en comparant avec la méthode de la puissance sur GPU.

Au chapitre 4, nous développons une approche parallèle pour calculer le groupement par modularité qui est souvent utilisé pour identifier et analyser les communautés dans les réseaux sociaux. Nous montrons que la modularité peut être estimée en examinant les valeurs propres dominantes de la matrice d'adjacence modifiée d'un graphe valué. Nous généralisons cette formulation pour identifier plusieurs groupes à la fois et proposons un moyen de détecter le nombre de groupes naturels. Nous développons une implémentation parallèle efficace sur GPU qui se base sur le solveur de valeurs propres de Lanczos et sur l'algorithme des K-moyennes.

Dans le chapitre 5, nous définissons la similarité de Jaccard pour les arrêtes d'un graphe et la généralisons pour tenir compte des poids des sommets, tels que le PageRank. Nous utilisons ces poids pour minimiser la somme des ratios de l'intersection et l'union des voisins des nœuds sur la frontière des partitions. A partir de la construction d'un Laplacien nous montrons comment l'approximation de la coupe minimale peut être formulée comme un problème de valeurs propres. Nous développons une implémentation parallèle rapide sur GPU. Enfin, nous comparons la qualité du groupement obtenu sur de grands ensembles de données.

Au chapitre 6, nous présentons une nouvelle méthode pour résoudre les problèmes de valeurs propres symétriques de grandes tailles, basé sur la méthode de Lanczos implicitement redémarrée couplée à une taille du sous-espace adaptative (MIRLns). Notre implémentation combine les forces du CPU et du GPU pour calculer le sous-espace invariant de grands graphes. Les expérimentations indiquent des améliorations de convergence sur des graphes avec des millions d'entités dans le contexte de problèmes de groupement.

Dans le chapitre 7, nous résumons les principaux résultats obtenus dans cette thèse et présentons nos remarques de conclusion. Enfin, nous proposons des pistes de

réflexions pour les recherches futures.

Les contributions de cette thèse abordent plusieurs problèmes interconnectés dans le domaine des graphes, des problèmes numériques de valeurs propres et des GPU. Cette thèse a introduit un solveur numérique rapide de valeurs propres, des nouvelles implémentations GPU performantes et de nouveaux résultats expérimentaux. Nous avons présenté comment les problèmes d'analyse spectrale de graphes sont résolus efficacement avec les GPU et avons introduit de nouvelles solutions conçues à partir des obstacles que nous avons rencontrés. Les implémentations font partie de la bibliothèque d'analyse de graphes d'NVIDIA (nvGraph) qui est distribuée avec les outils de développement (NVIDIA CUDA Toolkit).