Università
Ca'Foscari
Venezia

Master's Degree

in Computer Science

Final Thesis

Modularity Based Community Detection on the GPU

Supervisor

Ch. Prof. Claudio Lucchese

Graduand

Federico Fontolan

Matriculation Number

854230

Academic Year

2019 / 2020

Abstract

Modularity based algorithms for the detection of communities are the de facto standard thanks to the fact that they offer the best compromise between efficiency and result. This is because these algorithms allow analyzing graphs much larger than those that can be analyzed with alternative techniques. Among these, the Louvain algorithm has become extremely popular due to its simplicity, efficiency and precision. In this thesis, we present an overview of community detection techniques and we propose two new parallel implementations of the Louvain algorithm written in CUDA and exploitable by Nvidia GPUs: the first one is based on the sort-reduce paradigm with a pruning approach on the input data; the second one is a new hash-based implementation. Experimental analysis conducted on 13 datasets of different sizes ranging from 15 to 130 million edges shows that the proposed algorithms have different efficiency based on the size of the graph. For this reason, we study also an adaptive solution.

Contents

1 Introduction

2 Nvidia GPUs architecture and CUDA

The scientists, years by years, have to face bigger and bigger problems. Even if Moore's law (that said: "the number of transistors in an integrated circuit double about every two years") determines the increased power of the CPU since the sixties, even the problems size grows and it grows also much more quickly. For example, the web growth since the turn of the millennium raises new challenges that are hard to solve with the standard algorithm, due to the size of the data.

Besides, at the same time, the manufacturers have to face some serious physical limits. The increase in performance was made possible by the reduction in the size of the transistors and the increase in the frequency of the clock cycle. In the first half of the two-thousands, the producers discovered that reducing, even more, the size cause serious problems of heat dissipation and data synchronization. To find a solution to this problems, the manufactures start to produce multi-core CPU: the idea is that if it's impossible to increase further the speed with only one core, they add another processing unit, to ideally halve the execution time.

For these reasons, in recent times the studies of new parallel approaches became fundamental to solve problems that can not be solved classically. As a result of this change and at the same time both the support of floating-point number on the graphics processing units (GPU) and the advent of programmable shaders, it became popular the general-purpose computing on GPU (GPGPU), i.e. the use of a GPU to perform a computation that is commonly handled by the CPU.

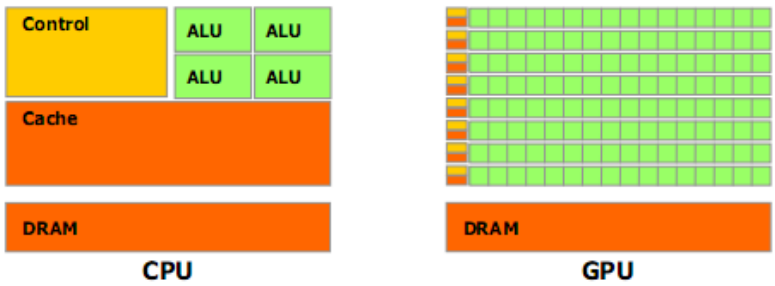


Figure 1: Difference in CPUs and GPUs architecture. This image was reprinted from [12]

The GPU architecture is radically different respect to the CPU. The CPU has sev-

eral ALUs (Arithmetic and Logic Unit), a complex control unit that controls those ALUs, big fast cache memory and dynamic random access memory (DRAM); the GPU has many ALUs, several simple control units, a smaller cache and a DRAM (Figure 1). While the first one is focused on the low-latency, the second one is focused on the high throughput; while the first one is focused on handle various serial complex instruction, the second is focused on handle much parallel simple instruction. In brief, the first one is a versatile processing unit, the second one is highly specialized. Even if in recent time the multi-core CPU performance get closer to the performance of the GPU [14], from the Figure 2 we can see how the performance of the GPU outclasses the performance on the CPU.

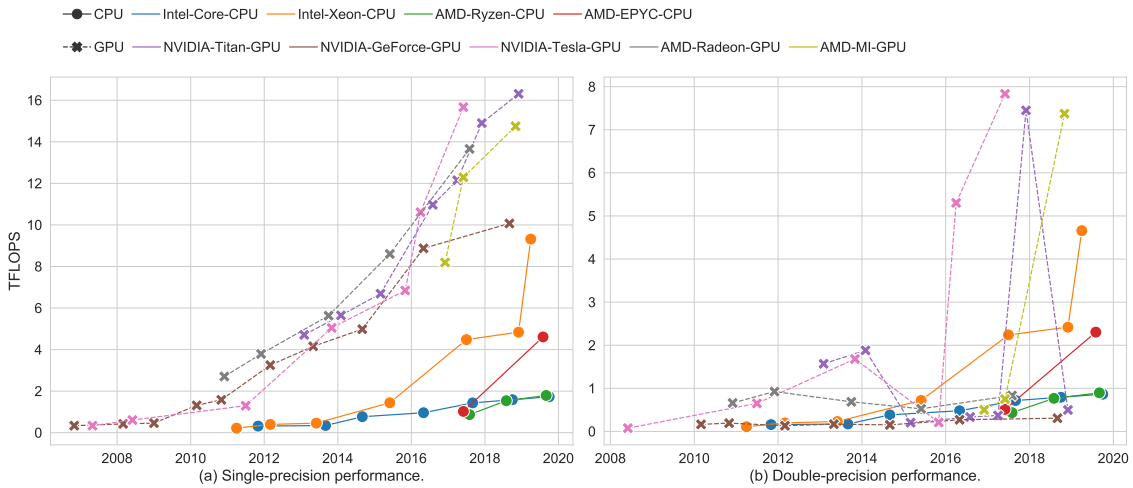


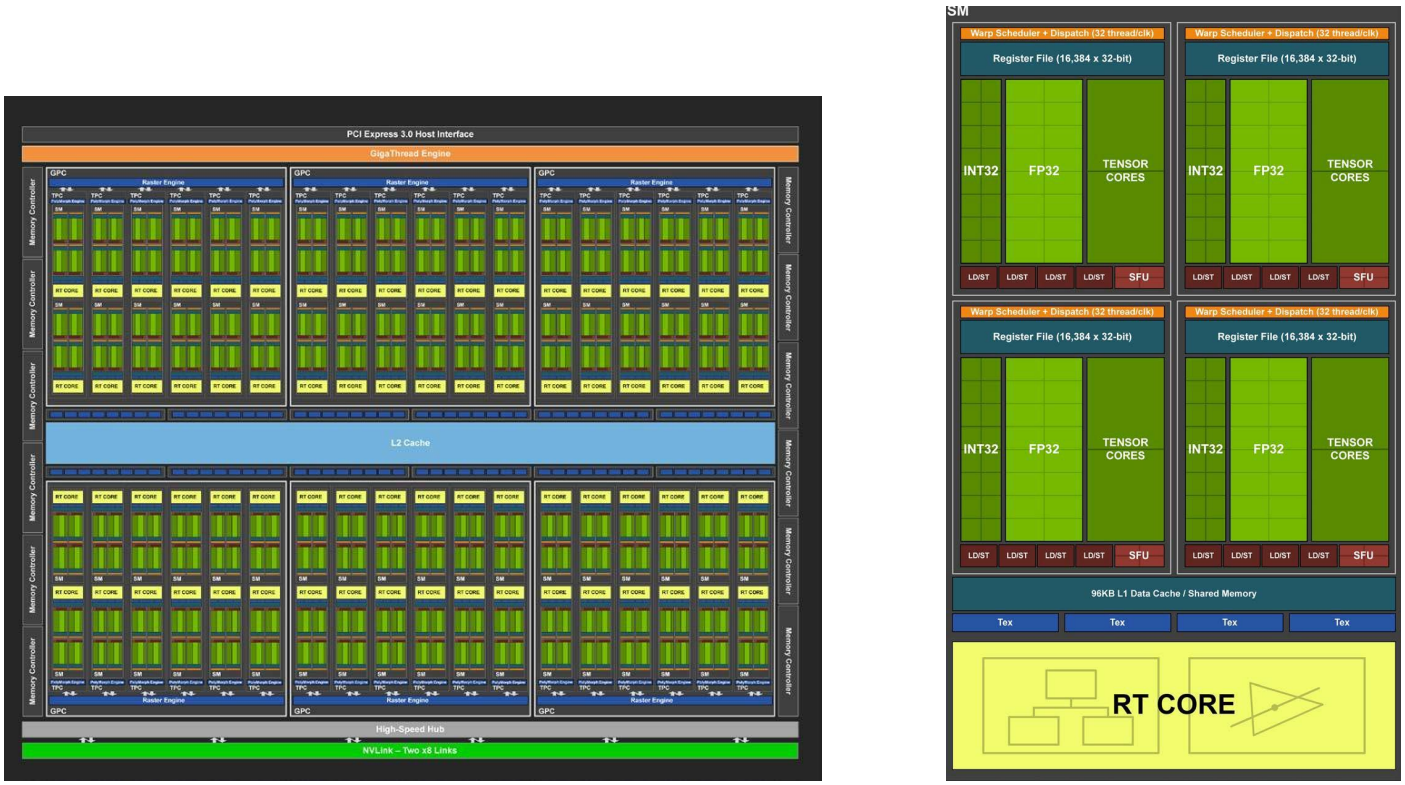
Figure 2: Comparing single-precision and double-precision performance of CPUs and GPUs. The performance are measured in trillion of floating point operations per Second (TFLOPS). This image was reprinted from [14].

For those reasons, the GPU-accelerated applications are the most effective to solve big problems, due to the possibility of reach very high speed-up compared to the classic multi-core applications. To simplify the development of this type of applications, in 2007, Nvidia releases CUDA (Compute Unified Device Architecture), a parallel computing platform and application programming interface (API) model. Nowadays, the CUDA framework is one of the main tools to develop HPC applications, due to its performance and simple API, and for this, we choose to use it in this project. In this chapter, first of all, we present the Nvidia's GPU architecture,

then after we present CUDA basis and Thrust, a library that was used in the project presented in this thesis that simplifies the development. This chapter was based on [13] and [12].

2.1 Nvidia's GPU Architecture

Now we present Nvidia's GPU Architecture to introduce some key concept that is used later in the thesis. This introduction present the last architecture released by Nvidia, Turing. We present the highest performing GPU of the Turing line, The Turing TU102 GPU (Turing machine can be also scaled down to the scaled-down from this one). In Figure (3a) we can see a scheme of this architecture. The cornerstone of each Nvidia's GPU is the concept of Streaming Multiprocessor



(a) (b)

Figure 3: (a): Tuning GPU full architecture; (b) Streaming multiprocessor (SM) in details. Those images was reprinted from [13]

(SM), that are represented in Figure (3b): it contains some cores specialised to solve specific arithmetic operations on specific types of data (like integer, float, double, tensor...). In a Tuning machine, each SM contains 64 FP32 cores, 64 INT32 cores,

eight Tensor cores and two FP64 cores (that aren't present in Figure 3b). In tuning architecture is present also a Ray Tracing cores in each SMs: this core is used in rendering.

The SM is the fundamental unit because, as we see soon, the parallel execution of the code in a CUDA application it's organized in blocks, and each block is executed on a single SM. Moreover, the SM contains also some register (256 KB in Turing), an L1 cache and a shared memory (in Turing 96 KB of L1/shared memory which can be configured for various capacities). The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps: when a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a warp scheduler for execution. A very important notion is that each warp executes one common instruction at a time, so if threads of a warp diverge via a conditional branch, the warp serially executes each branch path taken, ignoring the instruction for the threads that are not on the active path. The registers are private for each thread, but all threads share the SM's shared memory. The SMs are organised in Texture Processing Clusters (TPCs), that in a Turing GPU contains two SMs. In their turn, the TPCs are organized in Graphics Processing Clusters (GPC) that in a TU102 contains six TPCs. Finally, each GPU's contain six GPCs. Shared to all component there is an L2 cache: this is used as global memory, i.e. each thread can have access to it. In the Turing GPU. it is large 6144 KB. Therefore, in summary, a Turing TU102 GPU contains 72 SMs and than 4608 FP32 cores, 4608 INT32 cores, 576 tensor core and 144 FP64 cores.

2.2 CUDA

In November 2006, CUDA (that stands for Compute Unified Device Architecture) was realised by NVIDIA. This general-purpose parallel computing platform aims to give a framework to the developers that allow building applications that transparently scale its parallelism with a low learning curve. To overcome this challenge, CUDA was designed as a C++ language extension: in this way, a programmer that already know the language syntax can start to develop a GPU-accelerated application with a minimal effort. The support of other language was introduced years by

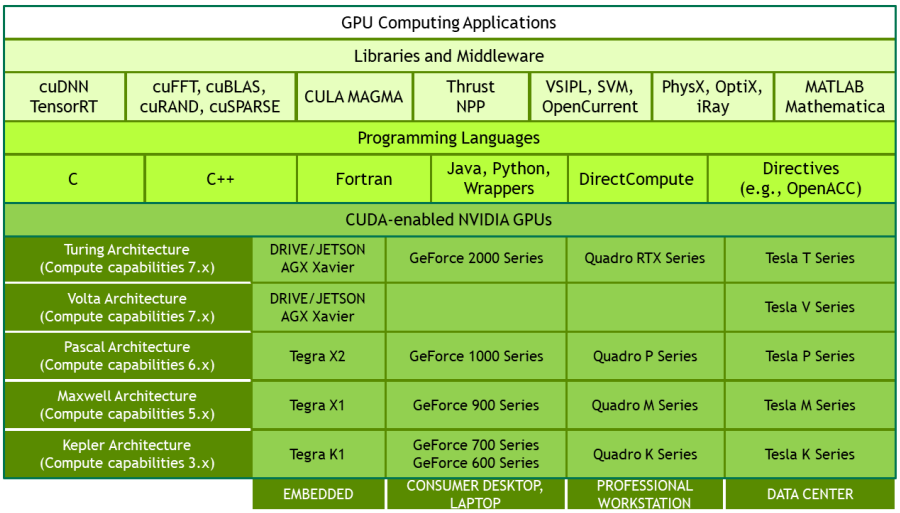
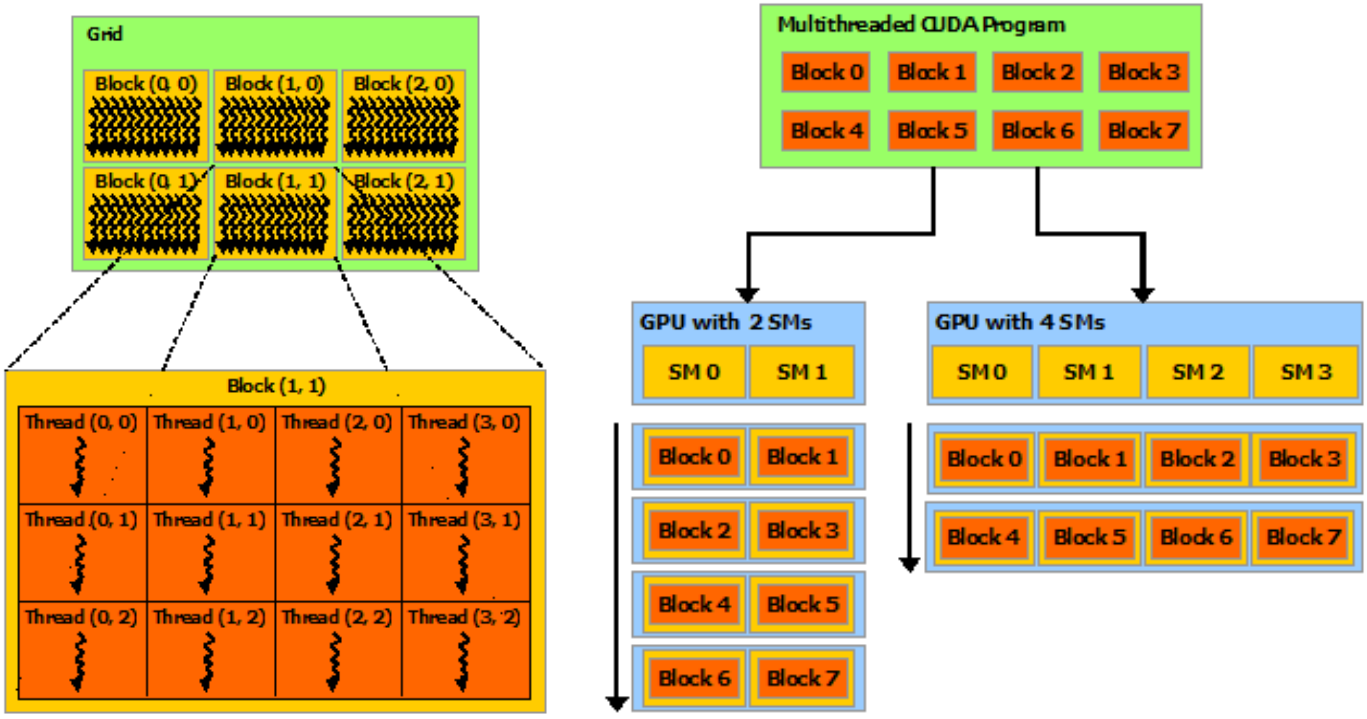


Figure 4: GPU Computing Applications. This image was reprinted from [12].

years, as illustrated in Figure 4. In this chapter we present the C++ extension, that was used to develop the project illustrated in this thesis, even if all extensions share the same concept and programming model.

The first key concept is the **kernel** function. In a CUDA-based application we define as **device** the GPU and as **host** the CPU. The application starts to the host, and when it is needed, it calls a kernel function that executes the function N times in parallel by N different threads. To define a kernel, we have to add `__global__` declaration specifier to the method and the number of thread that have to execute the kernel call. Each thread has a unique ID. To set this number we use an execution configuration syntax: after the method name, we include this setup enclosed in three angle brackets `<<< ... >>>`. The configuration is used to define the number and sizes of the blocks: a block is a group of threads that are organized in a one, two or three dimensional way. To identify the threads referring to the block, they have the three-component vector `threadIdx` that identify its position. In their turn, also the blocks are organized into a one-dimensional, two-dimensional or three-dimensional grid. Similar to the previous one, the vector `blockIdx` identify the block into the grid, and each thread can see each owns. To define the dimensions of the grids and the blocks in the angle brackets, we use two `dim3` (or eventually `int` to define a one dimension grid/blocks). The total number of threads is equal to the number of threads per block times the number of blocks: using the same logic, we can recover



(a) (b)

Figure 5: (a): Grid of Thread Blocks; (b) Automatic Scalability. Those images was reprinted from [12]

the unique ID of the vector from `threadIdx` and `blockIdx`. In Figure 5a is illustrated the grid-blocks schema.

As mentioned above, each block is assigned to a different streaming multiprocessor. On current GPUs, a block has threads limit set to 1024, due to the limited memory resources of the SM. This block scheme is used to implement automatic scalability: indeed, the GPU schedule each block thread on any available SM, in any order. For example, if we have a program that divides the threads into eight blocks, it can be executed from both two GPU's with respectively two and four SMs without any intervention on the scheduling from the developer (Figure 5b). By other hands, the block schema allows also threads collaboration: as illustrated in the Figure 6, thanks to the allocation of each block to the same SM, allows those threads to share a fast per-blocks memory. Besides, all threads share the global device memory, even if they belong to different blocks or kernel (some advanced settings permit to execute two kernels simultaneously if there are enough resources. Those settings are not presented here because there aren't used in this thesis because a large amount of

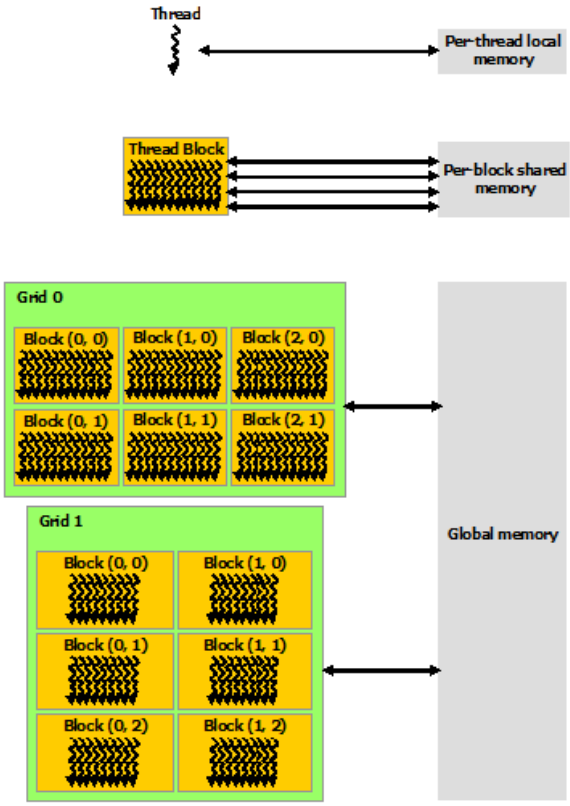


Figure 6: Memory Hierarchy. This image was reprinted from [12].

data doesn't permit to parallelize those kernels. Moreover, for the sake of completeness, they are well described in [12]). The data must be copied to this memory from the host before the kernel execution. Those two type of memory, combined to several primitives that synchronize thread at warp, blocks or device levels, permitting threads collaboration. Those synchronizing function acting as a barrier: all threads in the specific level must wait for the others before any is allowed to proceed. In addition, CUDA exposes some other primitives that allow atomic operations: if multiple threads call one of those methods on a specific memory address, the access to it will be serialized. No information about the order of the operation will be given a priori. In conclusion, we remark that every new hardware architecture could introduce new features that aren't supported by the old GPU. For this reason, CUDA uses the concept of Compute Capability to identify the features supported by the GPU hardware.

2.3 Thrust Library

To conclude this CUDA introduction, we present Thrust, a powerful library of parallel algorithms and data structures that are largely used in this thesis project. This C++ Standard Template-based library is included in the CUDA toolkit and provides a reach collection data-parallel primitives (as transform, sort or reduce) that allows writing a high performing and readable code with minimal effort. This presentation is based on the Thrust section present in the CUDA manual [12].
We start the presentation from the two vector containers, `host_vector` and `device_vector`. As their name says, they are arrays that are dynamically allocated respectively in the host and the device memory. Like the `std::vector`, they are generic containers, their elements are allocated in contiguous storage locations and they can dynamically change the size. Indeed, using the `=` operator, we can copy a `host_vector` in a `device_vector` and vice-versa. Thrust also provides many useful parallel algorithms, implemented for both host and device, like:

- Sort that performs the sorting of a vector. It is also present its "by key" version that sorts a vector of values using another vector as a key;
- Reduce that performs a reduction of a vector. It is also present its "by key" version that, given a vector of values and a vector of keys, performs a reduction of the values for each consecutive group of keys;
- Transform that applies a function to each element of the vector;
- Exclusive and Inclusive Scans that perform a prefix sum, respectively ignoring and considering the corresponding input operand in the partial sum.

In this thesis, we use only the device CUDA-based version of this algorithm. The last useful feature that Thrust provides is the fancy iterators. These iterators are used to improve performance in various situations. The `transform_iterator`, for example, were used to optimize the code performing the transformation during the execution of an algorithm. Another very useful iterator is the `zip_iterator` that takes multiple input sequences and yields a sequence of tuples: in this way we can treat many vectors as a single one and perform more operations simultaneously.

3 Community Detection State of the Art

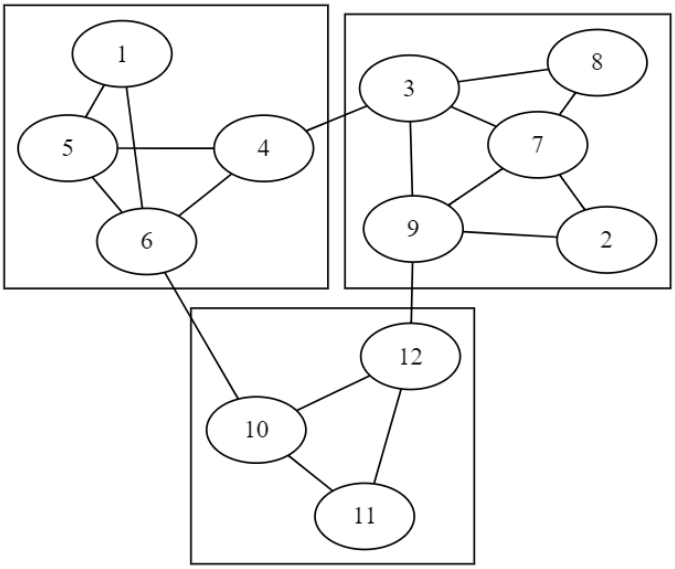


Figure 7: An example of a communities structured graph. In it is well visible three community, enclosed by the rectangles. The image was made with Graphviz.

The problem of community detection raises in many application scenarios from the necessity of finding groups of objects that have a large number of connections to each other. To represent problems where it is fundamental to empathize connection between objects, the graph theory is the main tool. A graph is a mathematical structure composed of nodes (or vertices) that denote the objects and edges (or links) that express some kind of relationship between objects and possibly having a weights that quantifies this relationship. The Graph Theory born in 1736 when Euler used this mathematical abstraction to solve the puzzle of Königsberg's bridges. Since then, this tool was used in several of Mathematics, Social, Biological and Technological application. In recent time, the approach to this studies has been revolutionized to deal with bigger and more complicated challenges, supported by the increasing computing power.

The necessity of finding this high-connected substructure in graph arises from real problems of the previous field: for example, the study of Protein-Protein Interaction (PPI) networks is very important because the interaction between proteins is the basis of all process in the cell.

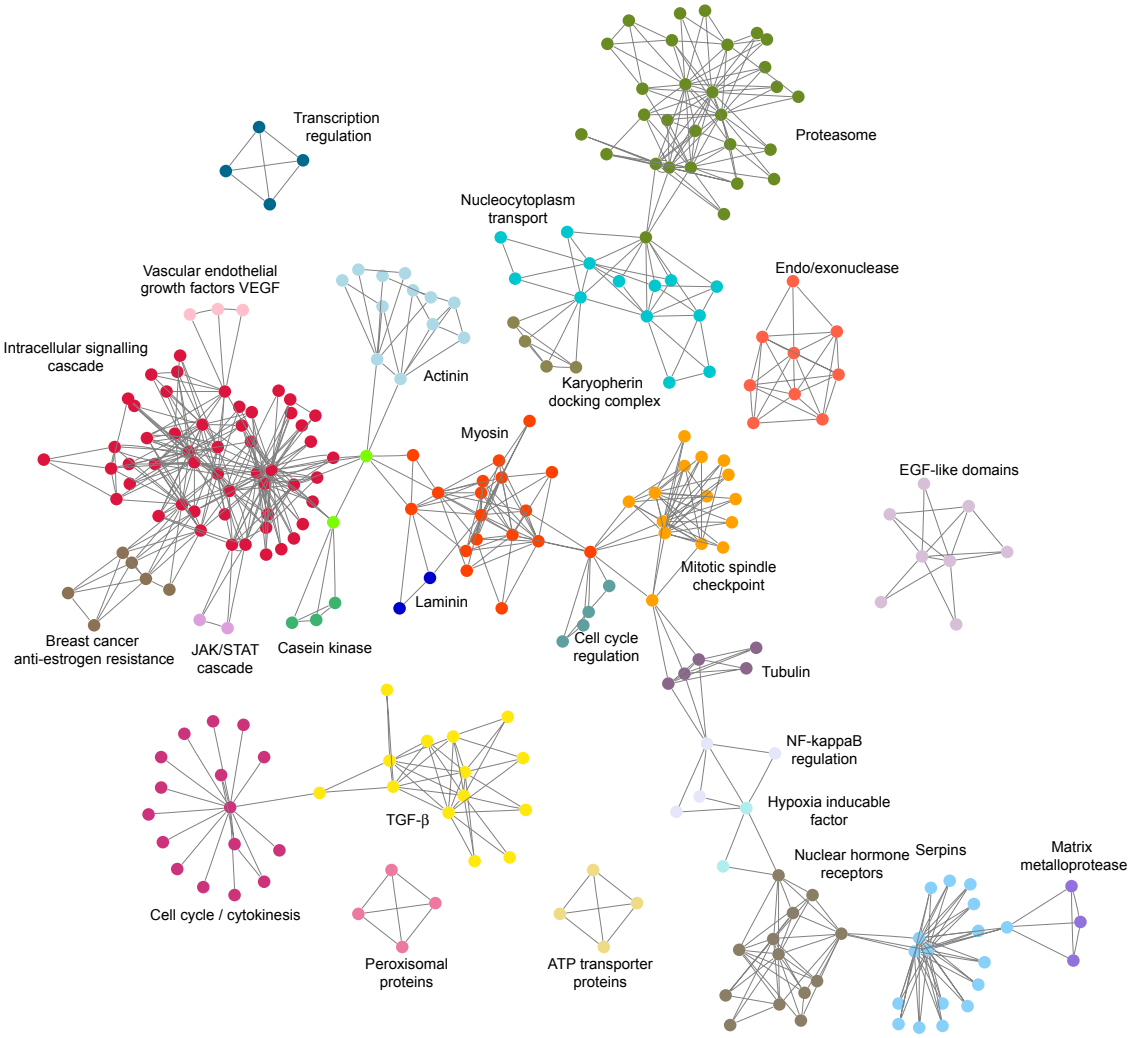


Figure 8: A protein protein iteration network of a rat cancerous cell. This image was reprinted from [8].

A study demonstrated that this type of network shown to be useful for highlighting key proteins involved in metastasis. [8]

Other examples can be found in the field of sociology: a historically well-know scenario is the Zachary's Karate Club. This dataset captures members of a Karate Club for 3 years.[3] An edge between two nodes represents an interaction between two members outside the club. At some point, a conflict between the administrator and a master led to split of the club into two separate groups. The question is if it is possible to infer who compose these two new groups basing on the information that this graph give to us. This small network of 1977 is famous because it has often been

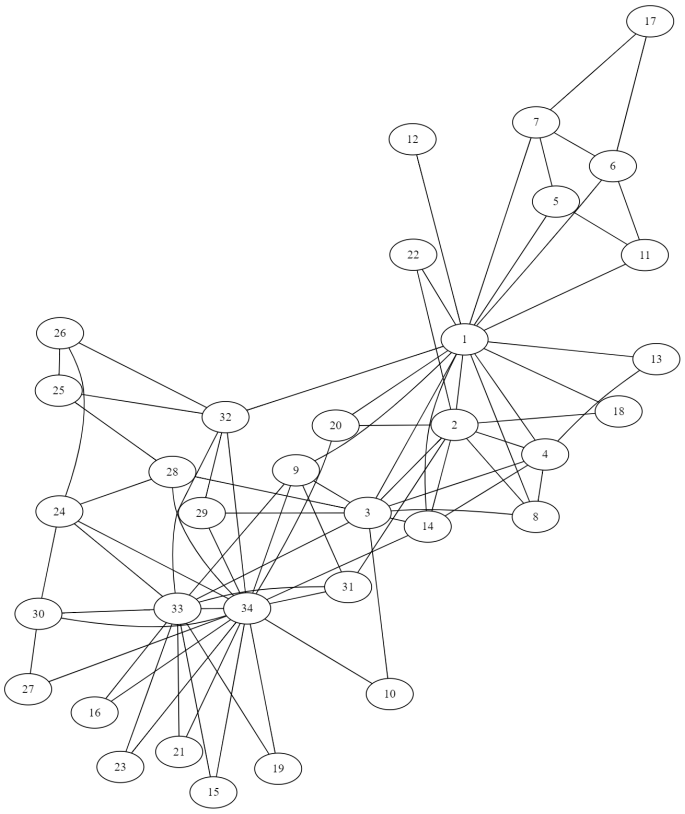


Figure 9: Zacahry's karate club. [3] This image was made with Graphviz.

used as a reference point to test the detection algorithms used to analyze huge social web networks. In general this kind of problem, i.e. clustering people that belong to the same community base on interaction, it's useful not only in sociology but also in marketing: by knowing people with similar interests, it's possible to make better recommendation systems.

There are several of similar scenarios to apply this method in the real-world, all united by the fact that the data are unregular but it's present some well-defined topological structure that in a completely random graph are absent. A random graph is a fully disordered graph, firstly proposed by Erdös and Rényi [1] in 1959: it's a graph where the probability that there is an edge between two nodes it's equal for all pairs of nodes and, for this reason, the degree of the nodes (i.e. the number of edges incident to a node) is homogeneous. In real networks, this is not true, because they are often scale-free (fallow a power-law distribution). An example of this is the study about the citations in scientific papers made by Derek J. de Solla Price in 1965 [2] or the study about World Wide Web growing made by Albert-László

Barabási et al in 1999 [4]. Furthermore, the degree distribution of the nodes is non-homogeneous not only globally but also locally, this due to the observation that there is a high concentration of edges within sets of nodes and a low concentration of edges between this sets. These two concepts are essential to formulate the formal definition of Community and Modularity. In this chapter will be presented some definitions of community and will be given an overview of some methods that are used to identify communities.

3.1 Community Definitions

The informal definition of community is there are many more edges inside the community versus the rest of the graph, but there isn't a unique quantitative definition of community. This kind of freedom is necessary because the concept of community is strictly connected to the problem that will be analyzed: for example, in some cases, it's necessary that community overlap, but in other problems, this is not necessary. There is a unique key constraint that allows talking about community detection: the graph must be sparse. A sparse graph is a graph where the number of nodes has the same magnitude of the number of edges. In the unweighted graph case, if the number of edges is far greater than the number of nodes, the distribution of edges among the nodes is too homogeneous for communities to make sense [11]. In that case, the problem nature is little different: we aren't interested anymore on the edge density between nodes but we have to use some kind of metrics (like similarity or distance) to clustering. In that case, the problem is more similar to data clustering. Despite this, assuming that a community is a subset of similar nodes it's reasonable, for this reasons some techniques (like spectral o hierarchical clustering) belonging to this field are adopted in community detection and will be shortly presented later on this thesis. Following this, Fortunato [11] defines three main classes of community's definitions: *local, global and based on vertex similarity.* Other types of definitions are still possible, but these three offers give a good summary of the problem. Now those classes will be presented to give an overview of the various approach that has been used to define this problem.

3.1.1 Local definitions

Considering that a community has a lot of interactions with the other nodes that are in it and few connections outside, it is fair to think about the communities as autonomous objects. The local definitions are based on this concept. Directly from this concept, we can think at the community as a clique, i.e. a subset whose vertices are all adjacent to each other. This type of definitions it's too strict: even if just one edge is not present, the subset is not a clique, but the subset has a very high concentration of edges. For this reason, the clique definition is often relaxed, using, for example, n-clique, i.e. a subset in which all the vertices are connected by a path of length less than n.

Anyway, this type of definitions ensure that there is a strong cohesion between the nodes in the subset, but not ensure that there isn't a comparable cohesion between the subset and the rest of the graph. For this purpose, other definitions were proposed. Given a graph $G(V, E)$, the relative adjacency matrix A and a subset of nodes C where $C \in V$, we define the internal degree k_v^{int} and the external degree k_v^{ext} for each vertex v that belongs to C as the number of edges that connect the node v with another node that belongs to C and not belongs to C, respectively:

$$k_v^{int} = \sum_{k \in C} A_{vk} \qquad\qquad k_v^{ext} = \sum_{k \notin C} A_{vk} \tag{1}$$

We also define the internal degree k_C^{int} and the external degree k_C^{ext} as the sum of all internal and external degree of nodes that belongs to C.

$$k_C^{int} = \sum_{i,j \in C} A_{ij} \qquad\qquad k_C^{ext} = \sum_{i \in C, j \notin C} A_{ij} \tag{2}$$

A strong community is a subset of nodes such that the internal degree k_n^{int} for each vertex n is greater than its external degree k_n^{ext} . This type of definitions once again very strict, for this reason we define as weak community a subset of nodes where the internal degree of the subset k_C^{int} is greater than its external degree k_C^{ext}. Many other variants of these definitions were presented in the literature.

3.1.2 Global definitions

The previous class quantify the community independently, considering every subset individually. Overturning the point of view, we can define communities in a graph-dependent way, considering them as an essential and discriminant part of it. There are many different interpretations of this approach in the literature, but the most important definitions are focused on this key fact: it's not expected to see a community structure in a random graph. For this reason, we define as *null model* of a graph another graph that have some features in common with the original one but it's generated randomly. This graph is used as a comparison term to identify if it's present a community structure in the graph or not and, if it is present, to quantify how it is pronounced. This approach, which is based the Modularity Optimization, is the main object of this study and is presented in detail in the next chapter.

3.1.3 Based on Vertex Similarity

The last class of definitions assumes that edges in the same community are similar to one another. All the definition used in the classic clustering methods belongs to this class because they calculate a distance (similarity) between object and aren't based on the edge density like the previous definitions. This distance can be calculated in various ways: if it is possible to embed the vertices into a n-dimensional Euclidean space by assigning a position to them, one method consists to calculate the distance between two nodes, considering that similar vertices are expected to be close to each other. To calculate the distance, one could use a norm. Three norms often used in the literature are the following. Given two points $A = (a_1, ..., a_n)$ and $B = (b_1, ..., b_n)$ that belongs to the n-dimensional euclidian space E, we define the norms l_1 (Manhattan distance), l_2 (Euclidian distance) and l_3 (Maximum distance)

as:

$$l_1(a,b) = \sum_{k=1}^{n} |a_k - b_k| \tag{3}$$

$$l_2(a,b) = \sum_{k=1}^{n} \sqrt{(a_k - b_k)^2} \tag{4}$$

$$l_3(a,b) = \max_{k \in [1,2]} |a_k - b_k| \tag{5}$$

Another option is the cosine similarity $\cos(a,b)$, that is very popular in literature:

$$\cos(a,b) = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} (a_i)^2} \sqrt{\sum_{i=1}^{n} (b_i)^2}} \tag{6}$$

If it is not possible to embed the graph in a Euclidean Space, it is possible to infer the distance from the adjacency matrix. If it is not possible to embed the graph in a Euclidean Space, it is possible to infer the distance from the adjacency matrix. One idea is to map the distance in order to assign smaller values at nodes with the same neighbourhood. Given an adjacency matrix A we define the distance between two nodes a and b as:

$$d(a,b) = \sqrt{\sum_{k \neq a,b} (A_{ak} - A_{bk})^2} \tag{7}$$

Many other variants of that definition (but based on the same principle) were presented in the literature, for example considering the overlap between neighbourhood respect to the union.

Other alternative measures consider the number of independent paths between nodes, i.e. path that does not share any common edges, or they are based on random walk on a graph: for example, the average number of steps needed to reach one vertex from another by a random walker.

3.2 Community Detection Algorithms

We now present some techniques used in the field of community detection: AGGIUNGERE METODI. Moreover, the Girvan and Newman algorithm is presented later on: this is because this method firstly introduced the modularity function and

it is presented separately. The goal of this chapter is to give a useful overview in order to get the differences with the Modularity optimization and empathize the motivations that make the Louvain algorithm one of the most used nowadays. For this reason, all the methods that are presented in this thesis find non-overlapping community, as the Louvain methods. For the sake of completeness, we remark that in Fortunato's report [11], that was mainly used to write this chapter, is also present an analysis of those overlapping algorithms.

3.3 Modularity Optimization

Historically, the modularity function Q was introduced as a stop criterion for the Girvan and Newman algorithm in 2002. This is a quality function, i.e. a function that allows distinguishing from a "good" cluster and a "bad" one. The function assigns to a partition a score that is used to compare partitions. This is not a trivial goal, because define if a partition is better than another is an ill-posed question: the answer may depend on the particular concept of community that it is adopted. Nevertheless, this sometimes is necessary, for example in the case of hierarchical clustering, where it's necessary to identify the best partition in the hierarchies. A simple example of this kind of function are the sum of the difference between internal degree k_v^{int} and the external degree k_v^{ext} [3.1.1].
The modularity function became very popular and a lot of methods based on this quality function were created. In this chapter we present the functions and its limits in details and the algorithm in which it was firstly used, some optimization techniques based on modularity.

3.3.1 Function

The function is based on the idea that we did not expect to see a graph structure in a random graph. We define as a *null-model* of a graph another one that it's generated randomly keeping some structural proprieties of the original one. Comparing the graph with its null model, we can quantify how much is well defined the community structure. Therefore, the modularity function is dependent on the choice of the null model. Given an undirected graph $G = (V, E)$, a partition of nodes C and

a function $c(x)$ that assign each nodes x to its community, we define a generic modularity function as :

$$Q = \frac{1}{2|E|} \sum_{i,j \in V} (A_{ij} - P_{ij}) \delta(c(i), c(j)) \tag{8}$$

where A is the adjacency matrix of G, P is the matrix of expected number of edges between nodes in the null model and δ is an filter function: its yields one if $c(i) = c(j)$, zero otherwise.

In principle, the choice of a null model is arbitrary, but we have to consider carefully the graph properties to keep in the null model because they determine if the comparison is fair or not. For instance, it's possible to choose as a model that keeps only the nodes and edges numbers, assuming that an edge is present with the same probability for each pair of nodes (in this case P_{ij} is constant). For this reason, The standard null model of modularity imposes that the expected degree sequence(after averaging over all possible configurations of the model) matches the actual degree sequence of the graph [11]. In this scenario, the probability that two vertices i and j are connected by an edge is equals to the probability to get two stubs (i.e. half-edges) incident to i and j.

This probability p_i of piking a stub from the nodes i is $\frac{k_i}{2|E|}$ where k_i is the degree of nodes i. The probability that two stub joining is $p_i p_j = \frac{k_i k_j}{4|E|^2}$. Therefore, the expected number $P_i j$ of connections between the nodes i and j is:

$$P_{ij} = 2m p_i p_j = \frac{k_i k_j}{2|E|} \tag{9}$$

Replacing P_{ij} from (9) in (8) we obtain:

$$Q = \frac{1}{2|E|} \sum_{i,j \in V} \left(A_{ij} - \frac{k_i k_j}{2|E|} \right) \delta(c(i), c(j)) \tag{10}$$

that is the standard modularity function. This function can be rewritten considering that only the vertex pairs in the same community contribute in the sum:

$$Q = \sum_{c}^{|C|} \left(\frac{l_c}{|E|} - \left(\frac{k_c}{2|E|} \right)^2 \right) \tag{11}$$

where l_c is the sum of edges that connect nodes in c and k_c is the sum of degree of nodes that belongs to c, i.e. total degree.
The modularity function Q it is in range [-1/2, 1] [10], and if we consider the whole graph as a unique community c we obtain $Q = 0$. Opposite, if we consider each nodes as community, $Q < 0$. Then, if a partition has a modularity score < 0, the partition hasn't a modularity structure.

3.3.2 Resolution Limit

There is a well-known limit of the modularity function, identified by Fortunato and Barthélemy [7] in 2006. Considering (9), we can easily compute the expected number of edges P_AB between two clusters c_A and c_B, that are separate cluster in partitions C, as:

$$P_{AB} = k_A k_B / 2m \tag{12}$$

where k_c is the total degree of c. We can compute from (10) the difference ΔQ_{AB} that affecting the modularity when we consider c_A and c_B in a partition where they are two different cluster respect to the partition where they are merged in one cluster c_AB:

$$\Delta Q_{AB} = \frac{l_{AB}}{|E|} - \frac{k_A k_B}{2|E|} \tag{13}$$

where l_AB is the sum of edges that connect nodes that belongs to A to nodes that belongs to B. Now considering the case $l_{AB} = 1$: there is only one edge that connects these two clusters. Therefore we expect that we obtain a greater modularity score keeping these two clusters separate respect to merging them. Instead, from (13) we have that the modularity increase if $\frac{k_A k_B}{2|E|} < 1$. For the sake of simplicity, we assume that $k_A = k_B = k$. We obtain that if $k < \sqrt{2|E|}$, the modularity is greater if we merge the communities. From this it follows that if the communities are sufficiently

small in degree, the expected number is smaller than one: in this case if there is only one edge between the two communities, we obtain a better result merging them. The result of this observation is that the modularity optimization has a resolution limit that prevents it to detect communities that are smaller respect the graph as a whole. This problem has many implications: the real networks graph have a community structure composed by communities very different in size, so some of this community may be wrongly merged. Fortunato identifies as week point the assumption that in the null model each vertex can interact with every other vertex [11]. Some solutions are proposed, as tunable parameters that allow avoiding the problem or also algorithm that eliminate artificial mergers. By the way, in many real cases, the modularity-based algorithms still obtain a very good result and permit to analyze quickly very large graph. For those reasons, the algorithms of this class of algorithm remain the most used, but it's important to remark their limits.

3.4 Girvan and Newman algorithm

Now we present the Girvan and Newman algorithm [5]. This method deserves to be presented because it is the first method that uses the modularity as quality function [6] and in some sense represents a turning point in the history of community detection. This method is a divisive algorithm, i.e. it tries to identify edges that connect two communities and then remove that edge. The goal of the algorithm is to get clusters disconnected from each other. To select which edge we have to remove, we introduce the concept of edge betweenness. The edge betweenness it is a measure that quantifies how an edge is least central for a community. If an edge connected two communities, it should have a greater value compared to an edge that is incident to two nodes that are in the same community.
The algorithm has 2 steps iterated until all edges are removed:

1. computation of the edge betweenness for each edge;
2. removal of the edge with the largest betweenness;

The algorithm construct an entire dendrograms of partitions, and the modularity is used to select the best one.

Girvan and Newman proposed three different definitions of edges betweenness [6]: shortest-path, current-flow and random walk. The first one is the number of shortest paths between all vertices which contributes the edge. The computation of this value for each edge of the graph has a complexity $O(n^2)$ on a sparse graph [6]. The second definitions consider the graph as a resistor network created by placing a unit resistance on every edge of the network. If a voltage difference is applied between any two vertices, each edge carries some amount of current. The current flows in the network are governed by Kirchhoff's equations and the calculations are performed on each edge in the graph. This calculation has a complexity $O(n^3)$ on a sparse graph [6]. The last one is the expected frequency of the passage of a random walker on the edges. The calculation requires the inversion of the adjacency matrix followed by the calculus of the averaging flows for all pairs of nodes. The complexity is $O(n^3)$ on a sparse graph [6]. The first definition is the most used for its speed ($O(n^2) < O(n^3)$ and it is also shown that in practical application this edge betweenness gives better results [6]. The authors also show that the recalculation step is essential to detect correctly communities: this means that we have to recalculate the betweenness every time an edge will be removed, raising the complexity of the algorithm to $O(n^3)$ on a sparse graph. The complexity is the strongest limit of this algorithm, which, however, was the first one to introduce the modularity and has many ideas that were used later on.

3.5 Modularity Optimization Techniques

3.5.1 Greedy Techniques

3.5.2 Extremal Optimization

3.5.3 Simulated Annealing

3.5.4 Spectral optimization

4 Louvain Algorithm

The Louvain algorithm is a greedy modularity optimization techniques created from a team of researcher from Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte and Etienne Lefebvre in the 2008 [9]. The algorithm bears the name of the university to which they belong to, i.e. *Université Catholique de Louvain*. This algorithm and its parallel version is the main subject of this thesis. The algorithm is very popular due to his simplicity, efficiency and overall precision. In this chapter, we present the sequential algorithm in details and some optimization technique presented in the literature. Then we present the parallel version of the algorithm, focusing on the implementations dedicated to the GPU.

4.1 Description

4.2 Parallel Implementations

5 The GPU's Algorithms

5.1 Fast-Sort-Reduce

5.2 Hashmap Version

6 Performance and Analysis

7 Conclusion

References

[1] A.Rényi P.Erdös. "On Random Graphs". In: *Publ. Math. Debrecen 6* (Dec. 1959), pp. 290–297.

[2] Derek J. de Solla Price. "Networks of Scientific Papers". In: *Science* 149.3683 (1965), pp. 510–515. ISSN: 0036-8075. DOI: 10.1126/science.149.3683.510. eprint: https://science.sciencemag.org/content/149/3683/510.full.pdf. URL: https://science.sciencemag.org/content/149/3683/510.

[3] W.W. Zachary. "An information flow model for conflict and fission in small groups". In: *Journal of Anthropological Research* 33 (1977), pp. 452–473.

[4] Albert-László Barabási and Réka Albert. "Emergence of Scaling in Random Networks". In: *Science* 286.5439 (1999), pp. 509–512. ISSN: 0036-8075. DOI: 10.1126/science.286.5439.509. eprint: https://science.sciencemag.org/content/286/5439/509.full.pdf. URL: https://science.sciencemag.org/content/286/5439/509.

[5] M. Girvan and M. E. J. Newman. "Community structure in social and biological networks". In: *Proceedings of the National Academy of Sciences* 99.12 (June 2002), pp. 7821–7826. ISSN: 1091-6490. DOI: 10.1073/pnas.122653799. URL: http://dx.doi.org/10.1073/pnas.122653799.

[6] M. E. J. Newman and M. Girvan. "Finding and evaluating community structure in networks". In: *Physical Review E* 69.2 (Feb. 2004). ISSN: 1550-2376. DOI: 10.1103/physreve.69.026113. URL: http://dx.doi.org/10.1103/PhysRevE.69.026113.

[7] S. Fortunato and M. Barthelemy. "Resolution limit in community detection". In: *Proceedings of the National Academy of Sciences* 104.1 (Dec. 2006), pp. 36–41. ISSN: 1091-6490. DOI: 10.1073/pnas.0605965104. URL: http://dx.doi.org/10.1073/pnas.0605965104.

[8] Pall F. Jonsson et al. "Cluster analysis of networks generated through homology: automatic identification of important protein communities involved in cancer metastasis". In: *BMC Bioinformatics* 7.1 (Jan. 2006), p. 2. ISSN:

1471-2105. DOI: 10.1186/1471-2105-7-2. URL: https://doi.org/10.1186/1471-2105-7-2.

[9] Vincent D Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008), P10008. ISSN: 1742-5468. DOI: 10.1088/1742-5468/2008/10/p10008. URL: http://dx.doi.org/10.1088/1742-5468/2008/10/P10008.

[10] Ulrik Brandes et al. "On Modularity Clustering". In: *IEEE Transactions on Knowledge and Data Engineering* 20.2 (2008), pp. 172–188. DOI: 10.1109/TKDE.2007.190689.

[11] Santo Fortunato. "Community detection in graphs". In: *Physics Reports* 486.3-5 (Feb. 2010), pp. 75–174. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2009.11.002. URL: http://dx.doi.org/10.1016/j.physrep.2009.11.002.

[12] *Cuda Programming Guide*. 2018. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[13] *NVIDIA TURING GPU ARCHITECTURE*. 2018. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[14] Yifan Sun et al. *Summarizing CPU and GPU Design Trends with Product Data*. 2019. arXiv: 1911.11313 [cs.DC].