University of Bergen

Department of informatics

Algorithms

Computing Twitter Influence with a GPU

Author: Amund Lindberg
Supervisor: Fredrik Manne

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

August 17, 2020

Abstract

Every day, a seemingly infinite amount of data is generated by social media. Analyzing this data is a complex, but rewarding task that can be solved faster with efficient algorithms and powerful computers. This thesis is concerned with identifying influential users in the Twitter micro-blog network. Designed to identify users that have an important role in spreading (mis)information, our influence measure takes both activity and network relationships into consideration and is based on breadth-first traversal of networks that arise from the sharing of content. The questions we try to answer are how to implement an efficient, parallel solution to our approach in NVIDIA CUDA using a graphical processing unit (GPU), and whether a parallel implementation gives a performance improvement over a sequential one. We give a detailed description of our influence measure in the context of social network analysis and relevant research on Twitter. An introduction to parallel programming and GPUs, as well as an explanation of the basics of CUDA is provided. This is useful as we explore known techniques on parallelizing the graph traversal algorithm Breadth-First Search (BFS), which we adapt and use in our implementation. Our experiments show that a simple implementation can achieve a considerable speedup on Twitter data.

Acknowledgements

Thanks to my supervisor Fredrik Manne for help with coding, writing, and all the great advice.

Thanks to Johannes Langguth and Daniel Schröder at Simula for their help and advice.

Thanks to my friends at UIB for good times, my roommates for the great home-office environment, and my family for all their support.

`Amund Lindberg`
17 August, 2020

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Problem statement and motivation

Social media platforms are more popular than ever and continue to grow. Today, about 3.8 billion people are using them worldwide, almost half the world's population [66]. These often open platforms allow for instantaneous broadcasting of information to a large audience, a privilege that historically has mostly been in the hands of government-regulated media, like TV and radio. In recent years, the spread of online misinformation has become an increasing problem [44, 67]. The rapid growth of social media is an example of technology developing faster than laws and strategies to counteract their misuse. Monitoring social networks is one step along the way, but is an infeasible task for humans. A more realistic solution is to use automated systems.

Social network analysis is the process of studying social structures such as networks of people, groups or organizations and the relationship or interactions between them [55]. Although the field arguably has its roots about 100 years ago [12], it is perhaps more relevant than ever with the internet and in particular social media which provides large amounts of easily accessible data, enabling researchers to find information proving to be of immense value for business, politics, research, etc. A social network can be viewed as a graph whose nodes represent the social entities, and edges the relationship between them. An important part of social network analysis is the concept of *centrality* [5], i.e. the idea that the role of nodes can be understood in the context of the network structure. The objective is often to identify the most important/relevant/influential nodes. Real-life applications include ranking websites in web search engines, recommending friends on social media platforms, targeting influential people to spread rumors of new products, and so forth.

Twitter is a popular social network and micro-blogging service. From its launch in March 2006, it took about one year for Twitter to reach 100,000 users [88]. Since then, it has grown rapidly and become one of the largest social media platforms [59, 62, 68]. In February 2019, Twitter had about 330 million monthly active users, and more than 100 million active daily users internationally [58], publishing 500 million tweets per day [60]. The main features of Twitter revolve around interactions through publishing and sharing short messages called *tweets*. This includes the *Follow* feature, which subscribes to another user's tweets, and the *Retweet* feature which shares a tweet with one's own subscribers. Despite the implications of the size of Twitter combined with its information-sharing model, there are limited regulations on the information being shared. Tweets containing false rumors, false news, or misinformation can spread to a large audience before eventually being removed. For example, a tweet during the outbreak of the Coronavirus in 2020 claiming hand sanitizers are useless against the virus was retweeted almost 100,000 times before getting deleted [65].

While the spread of misinformation is an issue all social media platforms have to deal with, we focus mainly on Twitter in this thesis. Our goal is to explore ways to implement an efficient, parallel algorithm that can identify users who play an important role in spreading tweets. One of several applications of such an algorithm is to evaluate someone's ability to spread misinformation.

Our approach takes advantage of the networks that occur when tracing how tweets spread through retweets. We call these networks *spreading graphs*. By adapting the fundamental graph traversal algorithm *Breadth-First Search* (BFS), we use spreading graphs to identify influential users based on how many times their tweets or retweets reach other users. Given several tweets, we can attempt to see the bigger picture of who is more important over time. The approach is designed by Langguth et al. [67].

The Twitter network is large, and computing the influence of many accounts and tweets is a considerable undertaking. However, given a collection of tweets, we construct several spreading graphs and compute influence on each graph independently. Thus, an important realization is a potential of speeding up the process by doing parallel computations. Graphical processing units (GPUs) are hardware components designed for massively data-parallel tasks and should be well suited for our problem.

The main contribution of this thesis involves the implementation and comparison of parallel algorithms that compute influence on Twitter. We explore a variety of ideas from previous work both on Twitter and on parallelizing BFS in order to make decisions that can improve our solution. We perform experiments with different approaches for parallelizing BFS to see which runs faster on a set of spreading graphs. For development, we use Nvidia CUDA, a parallel computing platform for GPUs, as well as the C and C++ programming languages.

1.2 Thesis overview

The thesis is organized as follows.

In Chapter 2, we give a brief introduction to social network analysis (SNA). SNA provides useful tools to study networks. Then, to understand the dynamics of Twitter and how researchers have studied influential users on Twitter in the past, we explore some of the previous, related work on Twitter. Finally, we give a precise, basic description of our approach of computing influence on Twitter, formulated as a graph problem.

In Chapter 3, we outline common challenges, terminology, architectures, etc, related to parallel programming, together with a brief introduction to GPU's.

In Chapter 4, we introduce the basics of the CUDA platform and GPU architecture.

In Chapter 5, we analyze the process of parallelizing the Breadth-First Search algorithm. We then explore previous work in parallelizing BFS, primarily with CUDA.

In Chapter 6, we explain how we implemented a parallel solution to computing influence in parallel on the GPU.

In Chapter 7, we perform experiments with a variety of implementations and parameters to see which performs better. We then discuss our results.

In Chapter 8, we summarize what we have found, as well as suggest ways to improve our solution in the future.

Chapter 2

Influence on Twitter

There are three main objectives of this chapter. First, while our goal in this thesis is rather technical, i.e. developing an algorithm to process data from Twitter as efficiently as possible, the result is not interesting without a context, nor feasible without the right set of tools. Thus, we begin this chapter with an introduction to Social network analysis (SNA). SNA provides an abstract view of networks through graph theory, as well as other useful tools for studying large and complex network in a structured manner.

Second, because we are mainly concerned with the Twitter network, we provide an overview of the most relevant aspects of Twitter. This includes a basic understanding of how Twitter is used, how information is shared, its network structure, and how researchers have studied influence on Twitter in the past.

Finally, we give a precise description and analysis of our approach to measuring influence on Twitter, and briefly compare it with other methods. Optimizing the computational process with parallel programming is a subject for later chapters.

2.1 Social Network Analysis

SNA is a strategy for studying social structures with origins in sociology [9, 12]. A key aim of SNA is a better understanding of individuals in the context of a network, where a network is defined by a set of individual actors and the relationships between them.

An important application of SNA is "viral-marketing", i.e. targeting certain individuals in a network and rely on their influence to spread rumors of a product by word-of-mouth, rather than traditional marketing [14]. Another interesting example is the evolution in the research of the "Small world problem". Back in the year of 1929, Hungarian author Frigyes Karinthy wrote a short story where he philosophized that the world is getting "smaller" in the sense that all people in the world are connected by only a short chain of acquaintances [1]. This idea is famously known as "Six degrees of separation" and has inspired games, movies, and research, etc.
In 1969, Milgram and Travers [4] did an experiment with 269 volunteers being assigned an arbitrary "target person" to whom they should attempt to deliver a letter by hand, only asking friends they thought would be most likely to know the target

person. For unknown reasons, only 64 of the letters reached their target, but the result was interesting: the median number of intermediate people to reach the target was only five. Of course, such a small experiment cannot be used to draw any conclusions.
In 2001, Watts et al. [10] repeated a similar experiment with approximately 100,000 online-volunteers from around the globe. Out of the 24,163 volunteers that started a message chain, only 384 reached their target. The median length of the completed chains was found to be 4.05. However, as they point out, their result is misleading due to a lack of empirical data.
In 2008, Leskovec and Horvitz [22] released a study of the MSN Messenger network. They had access to far more data than most previous studies of social networks, with a network of 180 million vertices and more than a billion edges. Among other things, they were interested in testing the small world hypothesis. Because large datasets come with great challenges with respect to computational time, they used a random sample of 1000 vertices and approximated the average chain length to be 6.6.
Finally, while this experiment has been performed several times and by others, we end the story at Facebook in 2016. In their dataset of then 1.5 billion users (about the size of the world population in 1929, when Karinthy wrote his story), they approximated the average chain length to be just of 3.57 between Facebook users.

It is interesting to see how data from social media allow us to perform large scale experiments easier than before, but bring challenges such as demanding huge computational resources. The Small-world problem is not intimately related to the goals of this thesis, however, its answers might have implications on how connected people are, and how fast information, misinformation, etc, can spread across the globe. In fact, it has been suggested that the small-world phenomenon could occur in many graphs in nature [7], thus we might expect to find similar patterns in many social networks, including Twitter.

2.1.1 Representing a network

Networks are often viewed as graphs whose vertices represent the network entities and edges the relationship between them. This section provides some definitions and notation that will be useful throughout this thesis.

Formally, a graph G can be defined as a tuple $G = (V, E)$, where V represents the set of vertices/nodes and E the set of edges between vertices. An edge thus consists of two endpoints. The number of vertices in the graph is denoted by $|V|$, and the number of edges by $|E|$. Two vertices u and v are said to be adjacent if they are joined by an edge.

We use the convention that $N(u)$ denotes the neighborhood of u, i.e. the set of all vertices adjacent to u, not including u. We write $deg(u)$ for the *degree* of u, i.e. the number of vertices adjacent to u, which is also equivalent to $|N(u)|$.

Depending on the underlying network, graphs are either directed or undirected. For directed graphs, we let $N_G^-(u)$ and $N_G^+(u)$ respectively denote the in, and out-neighborhood of vertex u, as well as $deg^-(v)$ and $deg^+(v)$ for the in and out-degree, respectively.

A directed acyclic graph (DAG) is a directed graph with no cycles. A DAG has at least one *source*, i.e. a vertex with no edges pointing to it, and at least one *sink*, i.e. a vertex with no outgoing edges.

A *path* is a sequence of edges which joins a sequence of vertices. We say a vertex v is *reachable* from u if there is a path from u to v. Let $dist(u, v)$ denote the *distance* between two vertices u and v, i.e. the minimum number of edges on the path between u and v. There can be more than one such path, and in the case of directed graphs, a path from u to v does not necessarily imply there is a path from v to u, and $dist(u, v)$ is not necessarily equal to $dist(v, u)$. This is because in general, directed edges only define a one-way relationship. In cases where there is no path between two vertices u and v, we set $dist(u, v) = \infty$.

Finally, let the *diameter* of a graph be the greatest distance between any pair of vertices.

2.1.2 Centrality

There are different ways to define *influence* in a network, it depends on the context and the network. One way to measure influence could be to study and compare the attributes of individuals. However, this ignores their relationship to others. For example, consider a network of movie actors. An actor that has acted in many movies and/or made large amounts of money by acting could be viewed as more influential. A politician on the other hand could be viewed as more influential by having more (influential) connections. The important thing to note is that the former focus on individual attributes, while the latter on relationships.

Centrality measures are used to rank relevant vertices based on network relationships and is one of the most studied concepts of SNA [13]. There are several centrality measures, and in order to gain some initial understanding of how influence can be measured in a network, we now present some of the more common [9], all based on definitions from Freeman et al. [5].

One approach of ranking (micro-)bloggers could be to count their number of followers. Similarly, we could say a popular user on Facebook is one that has many friends. In SNA, this can be generalized as *degree centrality*, i.e. the importance of a vertex is based on its degree, defined as:

$$C_D(u) = deg(u)$$

A downside of degree centrality is that it only focuses on the local parts of the network, and disregards the global network structure. Degree centrality can also be defined for directed edges.

Closeness centrality measures the importance of a vertex based on its average distance to other vertices. For example, to give an estimate of how fast information can reach or spread from a vertex compared to others [13]. Closeness is defined as:

$$C_C(u) = \frac{n-1}{\sum\limits_{u \neq v} dist(u, v)}$$

where u and v are vertices, and n is the number of vertices. To rank vertices that are *closer* to other vertices more highly, as opposed to further away, the formula is expressed as the reciprocal of the average shortest path from a vertex u to all other vertices. Note that centrality measures using distances are sensitive to reachability.

Betweeness centrality is based on the idea that important vertices often lie on communication paths between other vertices. If information is more likely to travel through a particular vertex, this vertex has influence by having control of this piece of information [5]. For example, people that serve as links between communities could be considered important. The betweenness centrality for a vertex u can be computed as the sum of the ratio of shortest paths passing through u to the shortest paths between all pairs in a network:

$$C_B(u) = \sum_{s \neq t \neq u} \frac{\delta_{st}(u)}{\delta_{st}}$$

where s, t and u are vertices, the number of shortest paths between s and t is denoted by δ_{st}, and $\delta_{st}(u)$ the number of these paths that pass through u.

2.2 Twitter

Twitter, at least originally, is different from other social media platforms in that it focus on sharing content in a very compressed form to a potentially large audience, rather than focusing primarily on the relationship between people such as, for example, Facebook arguably does. As a result, many have realized that Twitter is well suited for advertisement, spreading political ideas, etc, i.e. an easily accessible platform that allows speaking directly to an audience rather than going through more traditional platforms. The model of Twitter has led to a particular network structure and a way of information sharing that has gained wide interest in research. We begin this section by introducing an overview of the most relevant Twitter features, as well as Twitter-related terminology that we will use. We then explore some of the previous work on Twitter.

2.2.1 Features and terminology

Twitter users, also called *twitterers*, interact through short messages called *tweets*. Tweets can be of maximum 240 characters written in plain text, hyperlinks, etc. This intentionally short format enforces twitterers to stay brief with their message, and readers can easily glance through multiple tweets from a variety of users in a short time. Unlike Facebook where users primarily agree upon connecting to each other, a feature of Twitter is that users can follow each other without necessarily being followed back.

Tweets are published (*tweeted*) by a *publisher*, and each user has a personal timeline/feed displaying a stream of tweets from their *friends/followees*, i.e. the people they have chosen to follow. Some other key features include *Retweet*, which shares someone else's tweet with one's own *followers*, *Reply* is a reply to the publisher, and *Mention* is the name of another user preceded by a "@" symbol inside the tweet body. *Hashtags* are keywords or phrases that help tweets show more easily in the Twitter search [86]. Combined with the fact that tweets by default can be searched for, shared, and read by anyone, including unregistered users, retweets allow tweets to rapidly spread throughout the network and the internet.

2.2.2 Related work

The result of the algorithm we are implementing in this thesis is essentially a ranking of twitterers. There are many studies on Twitter, and measuring influence on Twitter has been done in many ways. We now look at some of the previous, more relevant studies.

When Twitter was still in the early phase of "going viral" and the term "micro-blog" was relatively new, Java et al. [21] studied the topology of the Twitter network and geographical distribution of its users. Some of their results might sound intuitive today, such as Twitter being popular around the globe, and that people seek and share information, news, and experiences from their lives. A few years later, Wu et al. [41] were interested in studying how information flows between predefined classes of users: "elite" users such as celebrities, news media, bloggers, organizations, and ordinary users. They found that elite users are responsible for creating the large majority of the content that spreads on Twitter and that in general, elite users within each class often share each other's content. Also, as one of several studies, they reported low levels of reciprocity on Twitter, i.e. most twitterers do not follow their followers back.

Centrality-based influence

It is often the case that accounts representing famous individuals, news organizations, etc, have many followers on Twitter. These are often considered influential in real life, and thus it seems intuitive that they are also influential on Twitter. While this may be sufficient, a ranking solely based on degree[1] has weaknesses. First, it ignores whether the user ever publishes any content. Second, it assumes all followers read any eventual content. Third, it ignores *who* the followers are, i.e. they all count the same. Finally, some users may gain many followers by simply following many, and will thus be mistaken as influential. One slight improvement is to measure the ratio between followers and followees.

Leavitt et al. [24] had similar arguments, and defined influence on Twitter as "the potential of an action of a user to initiate a further action by another user.", where "action" refers to interactions that move content, i.e. tweets, retweets, replies, etc. They handpicked 12 twitterers they believed to be particularly influential (celebrities, news outlets, and social media analysts) and gathered their user-data and actions within a certain time interval. They found that news outlets have more success in spreading their tweets (retweets), while celebrities create more conversations (replies and mentions).

Cha et al. [30] ranked more than six million active users by degree, the number

[1]In this context, the term *degree* refers to the number of followers on Twitter. We found no studies that used the number of followees to measure influence. If anything, it was mentioned as a way to identify spammers.

of times their tweets had been retweeted, and the number of times they had been mentioned. They found that the top users by each measure were celebrities, news organizations, etc, as they had been expecting. They found a somewhat high correlation between the rankings, which they explain by many tied ranks amongst lower-ranked users. After filtering out the lower-ranked users and focusing only on the top 10% by each measure, they found that people who get retweeted often also get mentioned often, but neither of these measures correlates with the number of followers. They conclude that users with many followers do not necessarily create conversations on Twitter.

In 1998, the founders of Google, Larry Page and Sergey Brin released their paper on PageRank [8], an idea that revolutionized searching on the web and ultimately helped Google (and the internet) become what it is today. Already at that time, there were millions of websites they described as "extremely diverse". They studied the problem of ranking websites, and did so by viewing the web as a large directed graph whose vertices represent websites, and edges the links between them. They argue that a naive ranking of websites, e.g. by degree, leads to certain problems on the web, such as people being able to manipulate the results and that neighboring websites all count the same regardless of their importance. This motivates PageRank, which is based on the idea that the rank of a website depends on the rank of websites that have a link to it. In other words, it's not about the number of links to your website, but who they come from. This also makes the ranking more resilient to manipulation, as fake websites are likely to have a low rank.

If we think of websites as users, and edges as "following status", the PageRank method can be applied to Twitter. Weng et al. [35] prepared a dataset with high reciprocity and found that people who follow each other talk about the same topics. Based on this finding, they proposed an extension of PageRank called "TwitterRank". They argue that a ranking of twitterers should also be based on content, not just the link structure of the Twitter network. As a user's influence could vary between topics, TwitterRank can do a topic-specific ranking, and a total influence value can be computed for each user by aggregating topic-specific influence across all topics. They compared the ranking correlation between TwitterRank and other measures, such as degree and PageRank, and found dissimilar results. They also found that TwitterRank was able to outperform the other measures slightly in recommending friendships.

Kwak et al. [33] found several interesting facts about the Twitter network. As one of several studies, they report a skewed follower-distribution on Twitter, i.e. most users have very few followers. They estimated the average path length between users in terms of the following-relationship. As this is a one-way relationship, they expected longer path lengths than found in other popular social networks. Surprisingly, the average path length was found to be just 4.12. By studying tweets and retweets, they found that most trending topics last a week or shorter and are headline news, that 75 percent of retweets happen in under a day, that more than 95 percent of tweets are retweeted only by immediate followers, but almost regardless of the number of followers a user has will still reach a significant amount of additional users (non-followers). Similar to [30] they ranked users by degree, PageRank, and the number of times they had been retweeted. They found a high correlation between the results of degree and PageRank, but that these again have a low correlation to the number of retweets. The top users ranked by degree and PageRank gave a list consisting of celebrities, politicians, etc, while the top users by retweet were often news sources.

Retweet-based influence

The studies we have presented on Twitter so far ranks users based on either the follower-relationship or statistics related to user-interactions. Zhang et al. [36] adopts ideas from Leavitt et al. [24], but rather than ranking users based on the number of retweets and so forth, they built an "action based"-network that represents information flow. For example, if user A tweets and user B retweets, there is an edge from B to A. They do this in a similar fashion for both retweets and replies. They computed influence in the action-based networks using PageRank where frequencies of actions are converted into probabilities, e.g. the probability that a user j retweets user i is the percentage of all retweets of i's posts done by j. By also adding time parameters to their models, they measured how influence changed over time. Their idea is interesting, but the results are highly speculative.

Bakshy et al. [37] was concerned with predicting the ability of users to spread content on Twitter. They sampled tweets from users who had posted at least one tweet that contained an URL and counted how many followers, followers of followers, etc, that had spread the URL further. By using these numbers and additional user attributes, they were able to find that the best predictors of spreading tweets was a user's number of followers and the average number of retweets by that user's immediate followers in the past. The fact that only the immediate followers were good predictors made sense, as most URLs did not spread very far, if at all. The maximum path length from the origin was found to be only 9. Most tweets were never retweeted. Only a handful of tweets had some-thousands of retweets. Finally, despite using a dataset exclusively of active twitterers, which they claim tend to have more followers in general, they found a skewed follower-distribution on Twitter, i.e. most twitterers have few followers.

Wang et al. [87] was concerned with detecting what they define as *information spreaders* on Twitter, i.e. users who retweet to share information to its own followers. They attempt to answer the question of "Given a user account and its tweets, which of its followers are likely to retweet?". They distinguish this from the problem of finding influential users by pointing out that they are concerned with only a local part of the network, not the global. In that sense, it could be viewed as a local influence measure in a subgraph of Twitter. They found that a user who has retweeted someone before is likely to retweet them again. They argue that using retweet history as a predictor of whether someone will retweet in the future is a poor measure as most people are very inactive on Twitter, and have only retweeted a single time (about 75%). Instead, they used a wide variety of variables such as common followers, common followees, the similarity of their published content, how much they speak with each other through replies and mentions, and profile similarity (e.g. number of followers). They found that similarity of the content (words, URLs, hashtags) in tweets was suited to predict the likelihood of retweeting, but that it depends on how many of the followers who retweet, and that 50 percent are only retweeted by only a single follower. They also ranked users by PageRank and found a low correlation to individuals labeled as information spreaders.

Misinformation on Twitter

Twitter has gained attention for being a medium well suited for spreading misinformation [31]. Chamberlain et al. points out that it's easy and free for everyone to join and share information on Twitter, the short message format of Twitter somewhat justifies sharing statements without providing any evidence, and that real-life events can often be shared on Twitter before the news are verified and spread through more reliable channels. In the meantime, false versions of what happened may have spread far. Misinformation can happen in many different ways, but discussions and research about the subject are often limited to case studies of large events such as natural disasters, epidemics, elections, etc [56].

For example, there have been experiments to see whether an account spreading only false news to a specific target audience could become popular [27]. The content of tweets during epidemics such as the H1N1 virus in 2009 [28, 32] have been analyzed and found to contain misinformation. Rumors such as people turning into zombies or that the disease could spread through the new iPhone were circulating on Twitter during the Ebola outbreak in 2014 [47]. Similar stories occurred during the most recent yellow fever outbreaks in 2015, 2016, and 2017 [54]. During the Coronavirus outbreak in 2020, rumors started spreading online that 5G radio towers were spreading the disease. As a consequence, there were cases of people attacking 5G phone masts, setting them on fire [69]. Social media platforms attempted to limit the rumors by deleting groups or content [63, 64, 70]. There are already researchers studying the effects of the 5G conspiracy theory [61].

Bovet and Makse [57] studied tweets that were posted during the 2016 US presidential election. They analyzed tweets containing URLs and found that 25 percent led to either false or "extremely biased" news. They studied retweet networks and found that most top spreaders of false or biased news were unknown, unverified, and deleted accounts, while tweets containing URLs that linked to traditional news originated from verified accounts.

Rather than focusing on a single event/crisis, Vosoughi et al. [56] recently did a more comprehensive study on comparing how false stories spread in contrast to true ones by analyzing tweets between 2006 and 2017. They sampled a dataset of retweet cascades containing rumors which they categorized as either true or false by checking against several sources. They found that false stories are shared at a much larger scale on Twitter, and suggests that it is because false information is often *new* information, and that humans are more likely to share new information.

2.3 Our approach to measuring influence

The following approach is designed by Langguth et al. [67].

We have seen that measuring influence can be done in different ways, e.g. by comparing user attributes or by the structure of the network. In our approach, we look at tweets. We assume that when a user tweets or retweets, the tweet will reach all their followers. We also assume that when users see multiple friends retweeting the same tweet, it weighs more than just receiving it once. Thus, we define the influence of a user as the number of times their tweets or retweets reach other users. For example, if a user A with n followers publishes a tweet, A reaches n other users. If a user B with m followers is a follower of A and retweets A's tweet, B reaches m users. User A has now reached their own followers *and* the followers of B, and we say that A has reached other users $n + m$ times, i.e. if A and B share common followers, they are counted twice in $n + m$. Notice that this is different from counting how many unique twitterers have been reached.

We formulate the computation of influence as a graph problem, split into two phases. Given a tweet, the publisher, and the list of users who retweeted it and at which time, we first build a spreading graph representing the ways a tweet may have spread through follower-relationships. Then, by using this graph, we can compute how many times the tweet reached other users by accumulating values along its edges. Given a set of selected publishers and their tweets, we can form a picture of which users in their network plays a more important role in spreading their tweets. We continue this section with a more detailed formulation.

2.3.1 The follower graph

Twitter has a large number of active users, and probably millions of registered accounts that are inactive [50]. The entire social network can be viewed as one big graph. Let the *follower graph* be the directed graph $F = (V, E)$ where the vertex set V represent all Twitter user accounts, with a directed edge $(u, v) \in E$ to indicate that user v follows user u on Twitter. In this way, edges point the same way as the flow of information.

Figure 2.1 illustrates what a tiny subgraph of F could look like. Here, user a has at least two followers: b and c. User b has at least four followers: a, c, d and e. User e is followed by f, who has no followers, and so forth.

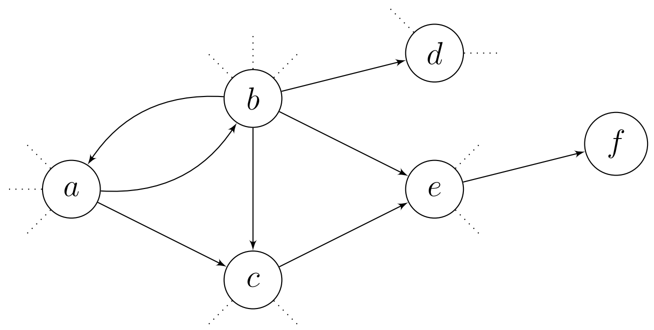


Figure 2.1: Example follower-relationships of six twitterers

2.3.2 The spreading graph

Any public tweet can be found online by searching for it in the Twitter database [82], and can be retweeted by registered users. For example, it is not uncommon for online newspapers to cite tweets by politicians, provided with a link making it easy for anyone to retweet. While tweets may spread this way, we always assume that tweets only spread along the edges of F, i.e. tweets only spread through followers retweeting. Two reasons for this choice are simplicity and the assumption that the followers are those who *usually* retweet their friends.

A retweet appearing on a user's timeline will always come with a link to the original publisher and eventually a list of friends who retweeted. We cannot, however, always give exactly one person credit for spreading a tweet. Again using Figure 2.1 as an example, say user a publishes a tweet and user b, c and e retweets. User e follows both b and c. If we assume that e saw the tweet in its feed, we know it came through b or c, but unless b or c retweeted after e, there is no way of telling whether b or c was the one that first made e aware of the tweet. So we decide that both b and c should be credited for spreading that tweet to e.

For each tweet t there is a publisher account p_t and a list L_t of users who retweeted, where $L_t \cup \{p_t\} \subseteq V$. Let $C_t = L_t \cup \{p_t\}$ be the set of all users assisting in spreading tweet t. All possible ways a tweet could have spread can be viewed as yet another graph. Let the directed graph $S_t = (C_t,\ BE_t)$ be the *spreading graph* of t, where a directed edge $(u, v) \in BE_t$ implies that a user u could have seen v publishing or retweeting t. Because of our assumption that tweets only spread along the edges of the follower graph, the edge set BE_t consists exclusively of edges of F in reversed direction. Also note that for any cycle to occur in S_t, a user would have to retweet oneself. While this is possible in practice, we assume tweets only spread in one direction, and thus S_t is always a DAG with a single sink, i.e the publisher.

2.3.3 Forward phase: Constructing the spreading graph

Now that we have introduced the follower graph and the spreading graph, we show how we construct the spreading graph for a given tweet t. Let $time_t(v)$ be the order of retweets for all $v \in C_t$. Here, we assume no users retweeted t at exactly the same time, and represent time using integers, starting with $time_t(p_t) = 0$. The forward phase, illustrated in Algorithm 1, generates the spreading graph S_t. It works by exploring the follower graph in a breadth-first manner, starting from p_t. For each user that retweeted, we compare the order of retweet with their followers, and add corresponding edges to BE_t, expanding on the path in which tweets may have spread. Note that we restrict our attention to the induced subgraph $F[C_t]$ as users who did not tweet or retweet are not processed. The algorithm is based on *Breadth-First Search*, a fundamental graph-traversal algorithm that is central to this thesis, and come back to in Chapter 5.

Algorithm 1 Constructing the spreading graph

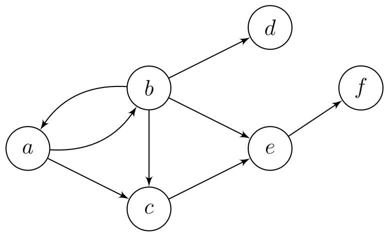
```
 1: function COMPUTE_SPREAD_GRAPH($F[C = L \cup p]$, $time$, $BE$)
 2:     $Q \leftarrow$ FIFO queue with a capacity of $|C|$.
 3:     $Q$.ENQUEUE($p$)
 4:     while $Q \neq \{\}$ do
 5:         $u \leftarrow Q$.DEQUEUE()
 6:         for $v \in N^+(u)$ do
 7:             if $v \in L$ and TIME($u$) < TIME($v$) then
 8:                 $Q$.ENQUEUE($v$)
 9:                 $BE \leftarrow BE \cup \{(v, u)\}$
10:             end if
11:         end for
12:     end while
13: end function
```

We explain Algorithm 1 with an example. Table 2.1 illustrates data associated with two imaginary tweets, tweet 0 and 1, based on users in Figure 2.1. In the table, if a user v tweeted or retweeted t, then $C_t(v)$ is equal to 1, otherwise 0. The *time* column displays the order of retweets, and is undefined for entries that did not tweet or retweet.

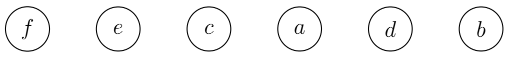
v	$C_0(v)$	$time_0(v)$	$C_1(v)$	$time_1(v)$	$\ldots$
a	1	2	1	4	
b	1	0	1	0	
c	1	3	1	3	
d	1	1	0	-	
e	1	4	1	1	
f	1	5	1	2	

Table 2.1: Example data for two imaginary tweets

We now generate the spreading graph S_0 using Algorithm 1. Initially, we can think of S_0 as a graph with no edges, just the set of users who contributed to the tweet, as shown in Figure 2.2b. The goal is to add edges to represent ways the tweet may have spread. Except for edges to non-involved users, the graph in Figure 2.2a is identical to Figure 2.1 and is shown again here for convenience.



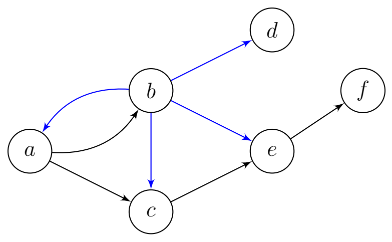
(a) Subgraph $F[C_0]$



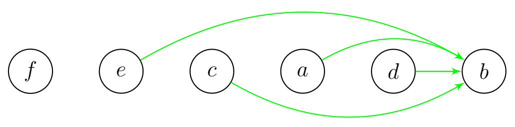
(b) Initial spreading graph S_0

Figure 2.2: Preparing the forward phase

The algorithm uses a queue to decide in which order to visit vertices. The first vertex in the queue is the publisher account, added at line 3. In Table 2.1, we can see that b must be the publisher because $time_0(b) = 0$. Now, with our assumption that tweets only spread along follower-relationships, we know that immediate followers of b are the first to see the tweet. We know that b has four followers, users a, c, d and e. The algorithm inspect each neighbor of b on lines 6-11. Each follower that retweeted at a time after b represents a possible way the tweet can have spread, and are subsequently added to Q at line 8. This is shown by the blue edges in Figure 2.3a. Back edges are then added to S_0 at line 9, as shown by the green edges in Figure 2.3b.



(a) Inspecting neighbors of b

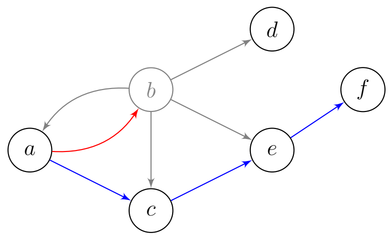


(b) Back edges are added to S_0

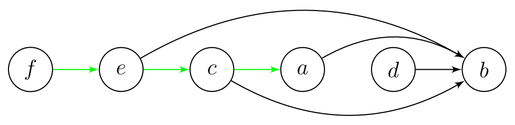
Figure 2.3: Forward phase, first iteration

A back edge was added from each follower of b back to b itself, which makes sense as b was the publisher. Thus, $\{a, c, d, e\}$ were added to the queue.

The algorithm repeats the same process for all vertices currently in the queue, one by one. User a has two followers, b and c. User b has not retweeted (its own tweet), meaning the tweet could not have spread from a to b, as shown by the red edge in Figure 2.4a. User c did however retweet later than a, and e later than c. As only users who retweeted later are added to the queue, a user can be added to the queue at most one time. Vertices and edges that already have been processed are grayed out.



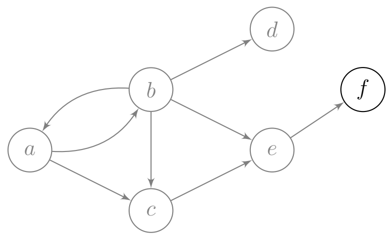
(a) Inspecting neighbors of a, c, d and e



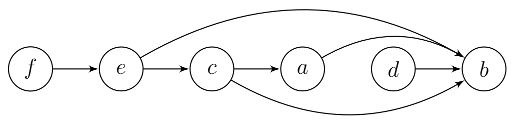
(b) Back edges are added to S_0

Figure 2.4: Forward phase, second iteration

Node f was the last node added to the queue. However, f has no neighbors to add, and the algorithm terminates at line 4.



(a) f has no neighbors



(b) Resulting spreading graph S_0 for tweet 0.

Figure 2.5: Forward phase, third iteration

Starting from just a set of vertices, we built the spreading graph in Figure 2.5b by analyzing time of retweets. In Section 2.3.2 we explained why the spreading graphs are always DAGs with a single sink being the publisher, in this example vertex b. Note that initially, sources in spreading graphs represent users whose followers did not retweet any further, and thus we can have multiple sources, vertex f and d in the example.

This sums up the forward phase. We will now proceed to compute the influence of twitterers, for which we will need the spreading graph.

2.3.4 Backward phase: Computing influence

The backward phase computes the influence of each user. It can be viewed as a continuation of the forward phase. In the backward phase, we start at the sources in the spreading graph, and aggregate influence values along its edges back to the sink, i.e. the publisher.

In short, the backward phase works like this:

1. Associate each vertex with an initial influence value.
2. Then for each source, distribute its influence value evenly across all neighbors in the spreading graph.
3. Remove the current sources. Repeat step 2 for the new sources. Terminate if the source is equal to the sink.

Recall that the influence is defined as the number of times a user reaches other users with their tweets or retweets. The number of times a user reaches other users can vary from tweet to tweet. Thus, we compute influence per tweet, and if given a set of tweets, we can merge the results at the end. Let $I_t(v)$ denote the influence of user v in tweet t, defined as the number of times v reached somebody by tweeting or retweeting t. We know that for any given tweet t, users in C_t reached all their followers in F. Thus, we set the initial influence value of user v to $I_t(v) = n_v$, where n_v is equal to the number of followers of user v in F.

The backward phase, illustrated in Algorithm 2, computes an influence value for each user in C_t by traversing the spreading graph S_t. We know that the sources in spreading graphs represent users whose followers did not retweet any further. For example, in Figure 2.5b, users d and f had no followers that further retweeted the tweet, thus their final influence value is their number of followers. User e on the other hand reached all their followers **and** the followers of user f. The influence of f is passed to e, which in turn evenly distributes its influence to both b and c (as it could have seen the tweet from either of them). We repeat the same process for the rest of the vertices on the path towards the publisher. The influence of the publisher will then always be equal to the sum of followers of all users in C_t.

For the rest of this section, we show this with an example for more detail. However, first notice the similarity of Algorithm 1 and Algorithm 2. In fact, they both are based on Breadth-first search.

Algorithm 2 Computing influence

```
 1: function COMPUTE_INFLUENCE($S = (C, BE)$, $I$, $num\_followers$)
 2:     for $v$ in $C$ do
 3:         $I(v) \leftarrow num\_followers(v)$
 4:     end for
 5:     $Q \leftarrow$ FIFO queue with a capacity of $|C|$.
 6:     for $v \in C$ do
 7:         if $v \in L$ and $deg^-(v) = 0$ then
 8:             $Q$.ENQUEUE($v$)
 9:         end if
10:     end for
11:     while $Q \neq \{\}$ do
12:         $u \leftarrow Q$.DEQUEUE()
13:         for $v \in N^+(v)$ do
14:             $I(v) \leftarrow I(v) + I(u)\ /\ deg^+(u)$
15:             $BE \leftarrow BE \setminus \{(u, v)\}$
16:             if $deg^-(v) = 0$ then
17:                 $Q$.ENQUEUE($v$)
18:             end if
19:         end for
20:     end while
21: end function
```

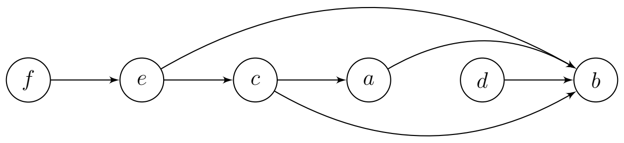


Figure 2.6: Preparing backward phase

For convenience, we again show the spreading graph S_0 computed in the previous section in Figure 2.6. We now perform the backward phase on S_0 to compute the influence of each relevant user. We maintain a table to illustrate how influence changes throughout the course of Algorithm 2. Table 2.2 show the initial influence values for each user, corresponding to its number of follower, computed in lines 2-4.

v	$I_0(v)$
a	n_a
b	n_b
c	n_c
d	n_d
e	n_e
f	n_f

Table 2.2: Initial influence for each user spreading tweet 0

Similar to the forward phase, a queue is used to keep track of which vertices the algorithm should process next. The queue initially consists of sources, being enqueued at line 8.

Now, at line 12, say f is the first vertex added to the queue, we inspect each neighbor of f in S_0 at line 13. The influence of f is evenly distributed amongst the neighbors of f at line 14, meaning they are now credited for reaching f's followers. The same procedure is repeated for user d. In the illustrations, the distribution of influence is shown in blue.

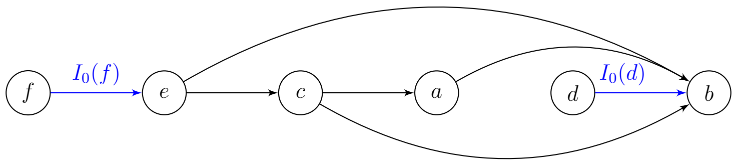


Figure 2.7: Backward phase, first iteration

v	$I_0(v)$
a	n_a
b	$n_b + n_d$
c	n_c
d	n_d
e	$n_e + n_f$
f	n_f

Table 2.3: Updated influence after first iteration

Table 2.3 shows the updated influence values for each user. After distributing influence along an edge, the edge has played its part and is removed at line 15. When a vertex no longer has any edges pointing to it, it will be identified as a source and eventually added to the queue at line 17.

After updating neighbors of f and d, the edges (f, e) and (d, b) are removed. Node b still has more edges pointing to it, so it is not added to the queue, vertex e however is identified as a new source at line 16, and thus added to the queue. The same procedure is then repeated for vertex e, illustrated in Figure 2.8.

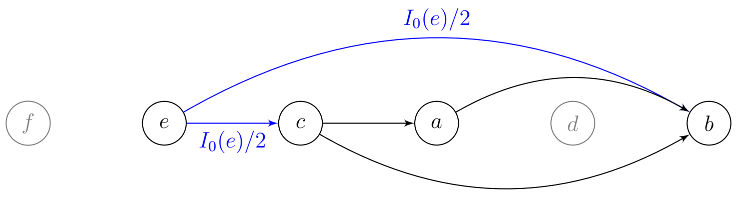


Figure 2.8: Backward phase, second iteration

v	$I_0(v)$
a	n_a
b	$n_b + n_d + \frac{n_e+n_f}{2}$
c	$n_c + \frac{n_e+n_f}{2}$
d	n_d
e	$n_e + n_f$
f	n_f

Table 2.4: Updated influence after second iteration

Table 2.4 show the latest updates to influence. After updating influence and subsequently removing edges (e, c) and (e, b), vertex c is identified as the next source. The same process is thus repeated for c.

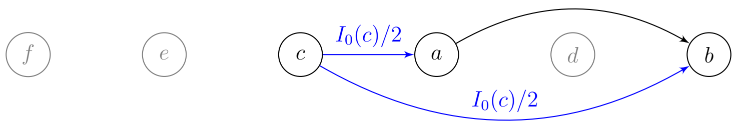


Figure 2.9: Backward phase, third iteration

v	$I_0(v)$
a	$n_a + \frac{n_c+(\frac{n_e+n_f}{2})}{2}$
b	$n_b + n_d + \frac{n_e+n_f}{2} + \frac{n_c+(\frac{n_e+n_f}{2})}{2}$
c	$n_c + \frac{n_e+n_f}{2}$
d	n_d
e	$n_e + n_f$
f	n_f

Table 2.5: Updated influence after second iteration

Finally, after updating the influence of a and b, the edges (c, a) and (c, b) are removed. A final edge remains from node a to the publisher b.

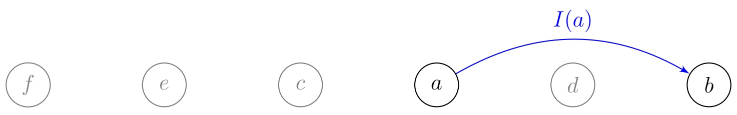


Figure 2.10: Backward phase, fourth iteration

After the influence of b is updated, it is added to the queue. However, a publisher has no outgoing edges in the (any) spreading graph, causing the algorithm to terminate at line 11.

v	$I_0(v)$
a	$n_a + \frac{n_c + (\frac{n_e+n_f}{2})}{2}$
b	$n_b + n_d + \frac{n_e+n_f}{2} + \frac{n_c + (\frac{n_e+n_f}{2})}{2} + n_a + \frac{n_c + (\frac{n_e+n_f}{2})}{2}$
c	$n_c + \frac{n_e+n_f}{2}$
d	n_d
e	$n_e + n_f$
f	n_f

Table 2.6: Final influence of each user in C_0

Table 2.6 shows the final influence values after the algorithm has terminated. If we simplify the expression for b, we have

$$I_0(b) = n_a + n_b + n_c + n_d + n_e + n_f = \sum_{u \in C_0} n_u$$

which makes sense: the publisher reached all its followers and also the followers of all those who retweeted.

When a updates the influence of b, there is a slight inaccuracy occurring because b follows a, which makes user b count as one of the followers reached, when it was in fact its own tweet. Generally, this occurs each time influence is distributed from a user u to a user v, where v also follows u. In practice, this is an inaccurracy we will ignore, both for simplicity and the low reciprocity on Twitter [30, 33, 41].

2.3.5 Combining results for multiple tweets

Our way of measuring influence is perhaps most interesting when we gather several tweets and attempt to see who is most influential over time. We can compute the backward phase for several spreading graphs and get an influence score for the users involved in each independent spreading graph, and then find which users are most influential on average. For example, we could gather several tweets from a specific "high profile"-twitterer and see which of their followers have a more important role in spreading their tweets.

There are several ways to view the final influence score. We let $I_t(v) = 0$ if a user v did not participate in spreading tweet t. Given T tweets (and thus T spreading graphs), some of our options are:

- The final influence for a user v is their total influence from all tweets:

$$I(v) = \sum_{t=0}^{T-1} I_t(v)$$

- The final influence for a user v is their average influence from all tweets:

$$I(v) = \frac{\sum_{t=0}^{T-1} I_t(v)}{T}$$

- The final influence for a user v is an average score of all the times v participated in spreading the tweet:

$$I(v) = \frac{A_v}{T_v}$$

where T_v is the number of tweets where $I_t(v) \neq 0$, and A_v is the sum of all influence in tweets where v took part:

$$A_v = \sum_{v \in t} \sum_{u \in C_t} I_t(u)$$

2.4 Summary and analysis

In this chapter, we introduced social network analysis, previous work on Twitter, and finally our approach of computing influence. Before moving on, we reason about our method of computing influence and compare it to some of the related methods.

First, the main goal of our method is to identify twitterers that have an important role in spreading someone's tweets. Not all approaches are equally attractive for this purpose. For example, if we applied betweenness and closeness centrality to the Twitter follower graph, these measures could only evaluate someone's *potential* of spreading tweets. In other words, by ignoring a user's content, one can only *assume* they are active on Twitter. Additionally, these measures assume that communication happens along the shortest paths, which is not always the case on Twitter (or other networks), and are thus not fit for use on spreading graphs.

Some studies have focused on content, e.g. measuring influence with a user's retweet history. As reported several times [33, 37, 87], retweet histories are unreliable in general as most tweets do not spread. In general, twitterers rarely retweet, and when they do, they only do so a few times. We also saw studies reporting that degree (#followers) is one of the better predictors of whether someone's tweet will spread [37] (again, likely due to most tweets not spreading very far). However, it should be noted that many studies focus on the bigger picture on Twitter, studying large datasets with millions of more or less arbitrary users.

Our model could be suited to target certain individuals and used to identify users within their follower-network that have an important role in spreading their tweets. In this context, the method we use can be considered an improvement over measuring influence by degree, as it could rightly credit someone who does not necessarily have many followers themselves, as long as someone further down their chain of followers of followers who retweeted does. Also, activity plays an important role, those who are more active could get a higher score in the long run.

The last point about the activity is crucial. Computing influence on a large set of tweets is heavy duty for a computer, which is why we are interested in implementing a parallel solution. With this in the back of our minds, we now shift focus. The next few chapters are related to parallel programming and will be important for understanding our implementation in later chapters.

Chapter 3

Parallel Computing

There is a never-ending demand for more computational power. Advances in science, engineering, and business keep pushing the market forward. Recent progress in the development of self-driving cars [80], cryptocurrency mining [78], machine-learning-based chess engines [72] and computer simulations to understand proteins' moving parts [71] are just a few examples of demanding problems largely benefiting from powerful hardware.

From about 1993 to 2003, the primary way of increasing the performance of microprocessors was to increase the clock frequency. The hardware industry could double the performance of single-core CPUs every 18-24 months [17]. By 2002, however, the method of increasing clock frequency began to reach practical limits due to heat [18]. This started a shift towards parallelism. Rather than focusing solely on developing faster single-core processors, manufacturers started putting multiple processors (cores) on a single integrated circuit [40]. In 2005, AMD released their first dual-core CPU for desktops, the AMD Athlon 64 X2 [15]. Intel replied with their own dual-core in 2006 with the Intel Core 2 Duo [75]. Since then, CPUs have steadily become more parallel. Today, CPUs with up to 64 cores are commercially available.

Many computational problems are either too large or too hard for a typical personal computer. Solving more demanding problems with powerful systems is typically referred to as *high-performance computing* (HPC). Such systems are typically built using multiple processors or computers. In recent years, high-performance computing has made large advances by incorporating GPUs. From originally graphics coprocessors of the CPU, mostly used for generating images for video games, GPUs with their highly parallel architecture are now also used for general purpose computing (GPGPU).

This chapter gives a basic introduction to parallel programming and GPUs. The knowledge and terminology from this chapter will be useful to understand the algorithms described in later chapters, which are mainly parallel.

3.1 What is Parallel Programming?

Most traditional computer programs are serial programs, meaning that they consist of a series of instructions that are handled by a processor one by one. Serial programs have always benefited from the increased performance of single-core CPUs. However, running serial programs on modern multi-core architecture does not automatically improve their performance. To solve a problem faster with multiple processors, one has to write a parallel program. Rather than doing one thing at a time in a sequence, a parallel program splits a problem into parts, and the hardware can compute each part simultaneously.

3.1.1 Threads

A processor executing a program must know which instruction to execute next, which memory to use, and so forth. In other words, a program is executed in a *context*. This context is called a *thread of execution*, or simply a *thread*. A serial program only requires one such thread, while parallel programs have multiple threads executing the same program or parts of the same program at the same time. Although a single processor can easily switch between threads and execute them in an interleaved fashion, we will mostly use the term thread to refer to a thread executing a program on a separate core.

3.1.2 Types of parallelism

Parallelism is typically divided into two fundamental types: task parallelism and data parallelism. Task parallelism is a type of parallelism where several independent tasks or functions can be executed in parallel, for example, an operative system scheduling two independent applications to different cores to execute at the same time. Data parallelism, on the other hand, is when the data is distributed between multiple cores or nodes that perform similar types of operations on their allocated data. As we will see, computing influence of Twitter users is an example of a data-parallel problem. Tweets and/or values associated with Twitter users are data that can be distributed to different threads, which all perform the same operation: computing the influence.

3.2 Parallel architectures

There are many different ways of classifying computer architecture. In the context of parallel computing, an often-used classification scheme is Flynn's Taxonomy [2], which classifies computer architectures based on the number of instructions and data streams that are processed simultaneously.

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)

Traditional single-core computers executing a single instruction on a single piece of data at any time are examples of computers falling into the *SISD* category. Computers that have multiple cores that all execute the same instruction on different data at the same time belong to the *SIMD* category. *MIMD* is a type of architecture where multiple processors perform independent instructions on multiple data streams. Modern CPUs and GPUs can often fit into more than one of these categories. We skip the *MISD* architecture as it is rarely used [48].

Another way to classify computer architectures is by their memory organization. Two fundamental architectures are the multi-node systems with distributed memory and the multiprocessors with shared memory. Multi-node systems with distributed memory are sets of loosely or tightly connected computers each having private memory and communicating through a network, and thus able to act as a single system. Multiprocessors with shared memory are processors that share the same address space either physically or virtually.

3.3 Challenges of parallel programming

Developing parallel programs is often considered more difficult than developing serial programs. We now introduce a set of fundamental, often encountered challenges when developing parallel programs that go beyond the choice of parallel architecture. First, it is desirable that the computer makes use of the hardware at its full potential when executing a demanding program. Implementing an optimal algorithm and optimizing memory usage is always important, and with serial programs, we trust the CPU and the compiler to take care of the rest. However, in parallel programs, the programmer has more explicit control of managing the computational resources available.

3.3.1 Load balance

To achieve efficient utilization of the parallel hardware, simply dividing tasks or data between cores may not be sufficient. Suppose, as an example, that we want to compute the integer factorization of N large numbers in parallel, each by brute force. If M threads are scheduled to compute N/M arbitrary numbers each, the unlucky threads eventually being assigned to prime numbers will expectedly take much longer to execute than threads who receive only composite numbers. Thus, we want to make sure to achieve the best *load balance* possible, i.e. the total amount of work is divided as evenly as possible. As our example attempts to display, it is not always easy, perhaps even impossible to achieve perfect load balance.

3.3.2 Non-determinism

Another challenge is that the time it takes for each thread to execute their instructions during a program execution will vary. This can cause non-deterministic behavior, i.e. executing a program over again with the same input gives different outputs. If two or more threads access the same shared data at the same time, a so-called *data-race* is caused. This may lead to unexpected results depending on the operation performed on the shared data. If each thread simply reads a shared value or updates the shared memory location to the same value, it will not affect the correctness of the program.

Suppose however that the result of an operation performed by one thread depends on the result of the other. If one thread's execution of the operation interferes with another thread, the result is non-deterministic. This can lead to bugs that are hard to detect, as higher-level programming languages tend to abstract away delicate details about program execution, e.g. even simple arithmetic operations are divided into several steps at the hardware level (read, modify, write), which can cause non-determinism in a parallel environment. Some hardware and parallel programming APIs provide implementations of such common operations that guarantee no interference between threads, often called *atomic operations*. More generally, a piece of code that must be executed by only one thread at a time is called a *critical section*. Making sure only one thread executes the critical section at a time requires a form of communication called *mutual exclusion*, i.e. the resource is locked for other threads while a thread operates on it. This will enforce sequential execution and will negatively affect the performance of the parallel program and should be avoided as much as possible. Similarly, some situations require threads to *synchronize*, i.e. threads agree to wait at a certain point during execution until all threads have reached that point. This is useful for example when threads share a resource that must be initialized before it is used.

3.4 Graphical Processing Unit (GPU)

GPUs are devices that are made for parallel computing. They were originally designed to render images, often for computer games or simulators. This rendering process, i.e. transforming the data to a final image on the screen consists of different stages in the graphics pipeline. Each stage contains data whose computation is independent of each other and thus can be computed in parallel. The calculations involved in each stage are often fairly straight forward, not involving complex control logic. For this reason, GPUs are made with many cores designed to perform many, simple tasks in parallel. CPUs, on the other hand, have just a handful of cores with a relatively large cache that helps make sure these cores are always busy, and infrastructure for handling unpredictable control flow. In other words, CPUs are optimized for low latency, and GPUs are optimized for high throughput [20]. The architectural difference between GPUs and CPUs is illustrated in Figure 3.1. As mentioned, some of the most modern, high-end CPUs have up to 64 cores, like the AMD 3990X. GPUs however can have thousands of cores, such as the Nvidia V-100 GPU that can have up to 5120 cores. CPUs and GPUs together provide a powerful combination as they excel at different tasks, e.g. in computer games where complex logic related to game mechanics can be taken care of by the CPU, while the GPU is responsible for efficiently rendering graphics.

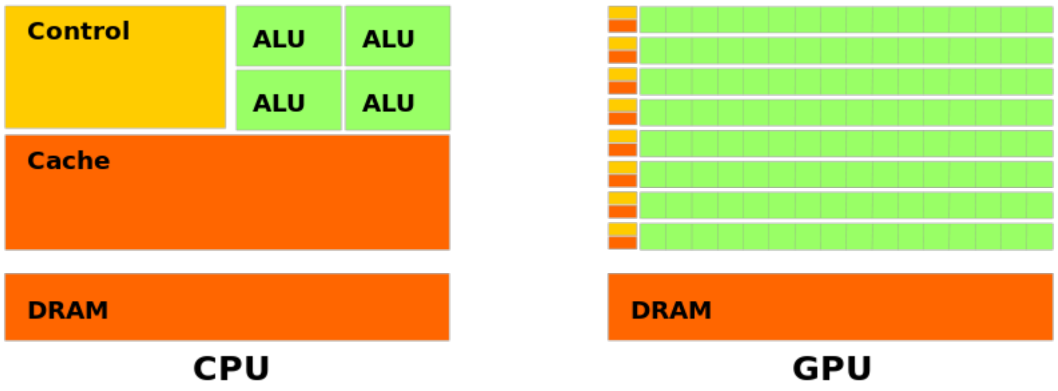


Figure 3.1: CPU vs GPU [48]

Early GPUs were hardwired to compute each stage of the graphics pipeline but has become more programmable over the years. At some point, it became apparent how GPUs could not only be used for graphics but also general-purpose computing, especially being useful for data-parallel tasks where they can outperform CPUs. Thus, GPUs are now standard in HPC and this thesis is also a result of this development. Platforms like OpenCL [74] and CUDA [77] enable general-purpose programming on GPUs.

3.5 Summary

In this chapter, we have seen some basics of writing parallel programs, some main differences between CPUs and GPUs, and so forth. The fact that GPUs are well suited for data-parallel problems is again our motivation to use GPUs in this thesis. Our method of computing influence should be a problem well suited for parallel programming, and GPUs in particular. The challenge lies in making the best use of the hardware possible. In this thesis, we use CUDA for parallel programming on a GPU. CUDA can be thought of as a programmer's interface to the GPU. In the next chapter, we introduce CUDA and the most relevant aspects of GPU architecture from a CUDA point of view.

Chapter 4

Nvidia CUDA

CUDA is a parallel computing platform developed by Nvidia. With CUDA, an Nvidia GPU can be accessed for doing general-purpose computation. In this thesis, we do parallel programming on GPUs, and CUDA is our choice of platform. This chapter provides a description of some basic features of CUDA, an introduction to memory hierarchies, and GPU architecture.

Nvidia released the first version of CUDA in 2006. Both the architecture of GPUs and CUDA have evolved significantly since then. Nvidia categorizes their CUDA-enabled GPUs by their *compute capability*, which can be thought of as a hardware "version", describing available hardware features and/or instructions available on the present GPU [77]. The CUDA Toolkit version refers to the version of the CUDA software platform, which is currently at version 10.2. The CUDA platform is accessible through CUDA-accelerated libraries, APIs, extensions to programming languages such as C, C++, Python, etc [48]. We assume the use of the C language in this chapter.

For illustration purposes, we now introduce two example GPUs. These demonstrate two completely different sides of the scale when it comes to the computation power they provide. The Nvidia GeForce MX-150 is a budget graphics card for laptops, mostly designed for light games and editing. The Nvidia Tesla V100-SXM3 is a high-end data center GPU designed for high-performance computing. We refer to these GPUs as MX-150 and V100.

4.1 CUDA Programming Structure

A heterogeneous system consists of CPUs complemented by GPUs, each with its own memory. As GPUs are managed from the CPU, the CPU is often referred to as the *host* and the GPU as the *device*. A CUDA program is written in C with a small set of extensions, and compiled with the CUDA compiler *nvcc*.

A common pattern in CUDA programs:

1. Copy data from the host to the device.
2. Run code on the device with the data in device memory.

3. Copy data back from the device to the host.

4.1.1 Kernels

The code to be run on the device is in functions called *kernels*. Kernels are invoked from the host with an *execution configuration* defining the *grid* of *thread blocks* that will execute the kernel code in parallel. We will explain grids and thread blocks in the next section. Kernels have similar syntax to C functions, but with a special `__global__` identifier. Kernels have certain restriction, e.g. they can only have the return type `void`. The code within kernels are expressed similarly as sequential functions.

```
__global__ void example_kernel(int *data, ...) {
    //compute something
}
```

Listing 4.1: Kernel syntax

4.1.2 Thread Organization

The grid is the collection of all threads spawned by launching a single kernel. A grid consists of groups of threads called *blocks*. CUDA organizes grids and blocks in three dimensions. This abstraction simplifies the mapping of a wide range of problems onto the underlying architecture. When launching a kernel, the dimension in the x, y, and z direction is specified in the execution configuration. Any dimension that is left uninitialized will have a default value of 1. Threads have access to their coordinates in the grid through the pre-initialized variables *blockIdx* and *threadIdx*, where blockIdx identifies which block the thread belongs to, and threadIdx the thread's index within the block. The grid dimensions are available through the variables *blockDim* and *gridDim*. The dimension in each direction is available through the variables x, y, and z components.

The CUDA kernel call syntax is similar to the C function call syntax. The difference is the execution configuration that is added inside triple angle brackets. In Listing 4.2, we illustrate how we could call the `example_kernel` of Listing 4.1. The *grid* argument is the variable for specifying block dimensions, and *block* to specify thread dimensions.

```
    kernel_example<<<grid, block>>>(argument list);
```

Listing 4.2: Kernel call syntax

Choosing the right execution configuration depends on the problem and can be crucial to performance. The dimension limits depends on the compute capability of the device. For compute capability $\geq$ 3.0, the dimension limit of the grid is 2147483647 x 65535 x 65535 thread blocks, and the number of threads per block is restricted to 1024. Any kernels launched with positive arguments within these limits will compile and run, however the number of active threads on the GPU depends on available resources, which we further discuss in Section 4.3. Figure 4.1 shows the grid for a kernel launched with $grid = (3, 2, 1)$ and $block = (5, 3, 1)$.

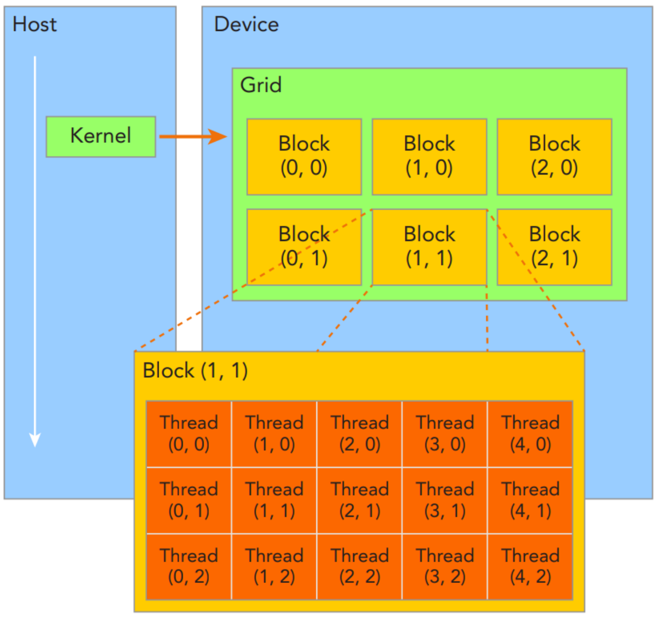


Figure 4.1: A grid of thread blocks [48]

4.1.3 Streams

Although most device work is issued from the host, the GPU and the CPU are separate devices that can operate independently. CUDA makes this possible through asynchronous device operations that return control to the host after initiating tasks on the device [77]. Kernel calls always have this behavior. This frees the host to do other things while the device does its designated work. All functions in the CUDA API, such as memory management functions are classified by CUDA as either synchronous or asynchronous with respect to the host. If a kernel and a memory transfer depending on the kernel execution is launched in a sequence, the memory transfer process will not begin before the kernel is completed.

This can be explained by how CUDA manages device operations in *streams*. A stream is a FIFO queue of device operations. All CUDA operations either implicitly or explicitly run in a stream [48]. By using two or more explicit streams, we can launch multiple concurrent kernels or hide latency caused by kernel execution or memory transferring by overlapping these operations. To allow several streams running concurrently, operations that normally block the host, like memory transfers, have a separate asynchronous version that takes a stream as a parameter. By default, CUDA programs use the implicitly declared stream, also called the NULL stream.

CUDA provides no primitives to synchronize all threads in a grid during a kernel execution. When such communication is required, a technique is to take advantage of the implicit grid-synchronization that happens between kernel launches. For example, problems that run in iterations and require all threads to synchronize in between, the iterations can be managed from the host.

In cases where synchronization between the host thread and device is required, CUDA provides the `cudaDeviceSynchronize` method. This blocks the host thread until the device has finished all issued CUDA calls. This is useful in situations where the host depends on the kernel being finished before proceeding, e.g. for debugging, or when several concurrent kernels are launched from multiple streams and one must wait for all streams to finish before proceeding to use their results.

4.2 CUDA Memory Model

The speed at which data can be transferred between processors and memory can be a bottleneck for many applications. If we keep making faster processors, we also require memory transfers to keep up with the pace, otherwise, the processors have to wait for data. From a hardware perspective, there are several approaches to speed up the memory transfer process, including transferring more memory at the same time with a wider bus and to use high-performance memory [45]. In many cases, we require both high-performance and high capacity memory. Producing a single type of memory that has both properties has proven to be expensive and difficult from an engineering perspective [45]. Because of this, memory models are based on applications following the principle of *locality*, i.e. we can take advantage of the fact that we rarely/never make use of all the memory in a computer at the same time. If we also make sure/assume that data elements that are related to each other are stored close together in memory, we can organize memory in a hierarchy based on its capacity and performance. Roughly speaking, it can be thought of in this way:

Cheap, high capacity memory such as hard drives are at the bottom of the hierarchy. When certain data are required by an executing application, it is first loaded into the slightly faster, but lower capacity memory DRAM. Further up the hierarchy, we have different levels of cache, which are high-performance, low capacity memory. Data that is thought to be related to the data which is currently being processed at a certain time will reside here, and when requested by the processor for computation, it is loaded into the registers, which is the fastest, but smallest and most expensive type of memory. A memory hierarchy is illustrated in Figure 4.2.

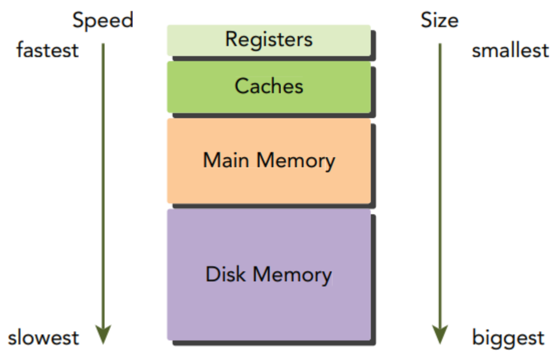


Figure 4.2: Memory hierarchy [48]

CPUs and GPUs share similar principles in memory hierarchy design. However, when developing serial programs for the CPU the programmer usually has limited control of how memory moves through the hierarchy. As we will see, with CUDA, the programmer is exposed to controlling memory movement more explicitly, i.e using more types of programmable memory. We now briefly introduce the memory types on a GPU from a CUDA point of view, also illustrated in Figure 4.3.

When a CUDA program is launched, the memory that is to be used by the device is both allocated, copied to and from, and released by the host. There are a few options as to which memory to use. *Global memory* is a relatively low-performance, high-capacity type of shared memory available for all threads in any kernel during the whole lifespan of the application. This can be considered the main memory of the device, and somewhat of an equivalent to the main memory used by a CPU (RAM). Pointers to global memory are passed as parameters to the kernels, or allocated statically. The capacity of global memory can differ greatly between devices. For example, the MX-150 has only 2 GB of global memory, while the V-100 has 32 GB.

Another option is the *constant memory*, which is a type of memory that resides in global memory but is cached in a dedicated read-only cache. Constant memory must be statically declared from the host and can be seen by all threads in all kernels in the same compilation unit. Constant memory is limited to 64KB for all compute capabilities. *Texture memory* is another read-only type of memory that is optimized for 2D spatial locality.

Another key type of memory in CUDA is *shared memory*. Shared memory is declared within a kernel and is available to all threads within a thread block for its lifespan. Shared memory can either be declared statically or dynamically by passing a third parameter containing the requested amount of bytes to the kernel execution configuration. Similarly to CPUs, GPUs can have multiple levels of non-programmable cache. There are typically two levels, L1 and L2. Shared memory shares the same hardware as the L1 cache in the GPU and can be thought of as a programmable L1 cache, and as a platform for fast communication between threads. Only a limited amount of shared memory is available for each thread block and if used wisely can drastically improve the performance of an application.

The fastest memory on the GPU are registers. Variables declared within kernels are generally stored in registers, and are private to each thread. These variables share the lifetime with the kernel. Only a limited amount of registers are available. If a thread uses too many registers, the excess registers will spill over to *local memory*, which is just a part of global memory and thus not desirable.

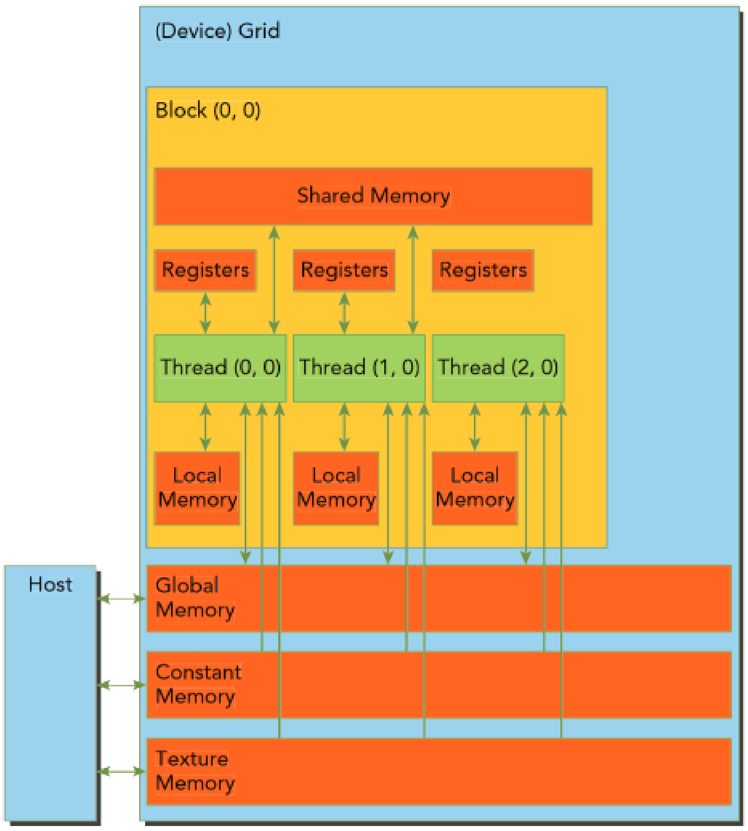


Figure 4.3: Organization of CUDA memory types [48]

4.3 CUDA Execution Model

Being able to understand how the GPU works at lower levels is essential for maximizing performance. This section provides an abstract view of how CUDA executes a program from a hardware perspective.

Streaming Multiprocessors (SM) are the core building blocks in a GPU. Generally, the more SMs in a GPU, the more parallelism from a hardware's perspective. We mentioned in Section 3.4 that a GPU can have thousands of cores, which we also refer to as CUDA cores. SMs contains key components like CUDA cores, shared memory, L1 Cache, registers, and schedulers, and each SM can support many concurrent threads. Now we will see another vast difference between the cards we introduced earlier. For example, the MX-150 has 3 SMs with 128 cores in each and can support up to a total of 6144 threads. The V-100 on the other hand has 80 SMs with 64 cores each and can support 163840 concurrent threads.

When a kernel is launched, its associated thread blocks are scheduled to execute on available SMs, where they reside for their lifespan. Each SM can execute multiple concurrent thread blocks. The amount of resident thread blocks per SM is limited by several factors, but perhaps most importantly, it depends on the amount of free resources in the form of shared memory and registers. Each SM has a limited amount of shared memory that is distributed among the blocks. A kernel will fail to execute unless it has enough resources for at least one thread block.

4.3.1 SIMT

Within each SM, thread blocks are partitioned into an even smaller group of 32 threads called *warps*. These are the smallest schedulable units on the device. SMs have one or more warp-schedulers that manage and execute a limited number of concurrent warps. The schedulers do their best to keep the GPU busy by selecting eligible warps to execute on the CUDA cores for every clock cycle. All threads in a warp start at the same program counter, and all threads in a warp execute the same instruction in each cycle. This is similar to the SIMD architecture discussed in Section 3.2. However, the fact that instructions are executed in the context of threads is an extension to SIMD, which Nvidia calls SIMT (Single Instruction, Multiple Threads). Threads within a warp are allowed individual behavior during execution by each keeping their own state. This freedom does come at a price, however. If threads within a warp disagree on the execution path, diverging threads will be disabled for that cycle. *Warp divergence* enforces sequential execution as threads taking a different path will have to be executed later to finish the program and thus has a negative effect on performance. In some cases, warp divergence can be avoided by awareness by the programmer or optimizations done by the compiler. We will sometimes refer to a thread within a warp as a *lane*.

The warp size of 32 is constant across all devices and is an important number in CUDA. Kernels launched with a thread block size not being a multiple of 32 will still cause a full warp to be scheduled for the remaining threads. However, excessive threads are disabled, but still occupy important resources such as registers.

4.4 Block-level synchronization

For synchronization at the block level, CUDA provides the `syncthreads` function. This creates a barrier that threads within a block cannot cross until all threads within that block have reached it. This guarantees that memory writes to shared memory will be visible to all threads in a block after the barrier.

Listing 4.3 shows an example. At the beginning of the kernel launch, the value of the shared memory variable *counter* is undefined. Without synchronization, threads within the block could continue to execute the kernel before thread 0 initialized the value and thus use the undefined value in its computation, which would likely result in a program error.

```
__global__ void kernel() {
     //counter is shared by all threads in a thread block
    __shared__ int counter;

    if (threadIdx.x == 0) {
        counter = 0;
    }
    __syncthreads();
    //all threads now agree on the value of counter
}
```

Listing 4.3: A common synchronization pattern

One must be careful when using synchronizing threads within conditional code as `syncthreads` expects all threads within a block to execute it. Not doing so could lead to undefined behavior. Another thing to note is that synchronizing should only be done when necessary as it could reduce performance by making threads wait.

4.5 Atomic operations

We briefly discussed atomic operations in Section 3.3. The CUDA API provides atomic functions on 32-bits or 64-bits of global or shared memory for common operations such as add, subtract, compare-and-swap (CAS), etc. It guarantees no interference between threads by effectively locking a memory address to other threads while a thread operates on it. Figure 4.4 show the syntax of the `atomicCAS` operation. It works by comparing the value at *address* with *compare*, and if they are equal, it replaces the value at *address* with *val*. Either way, it returns the old value stored at *address*. Any atomic operation can be implemented based on atomicCAS [77].

```
int atomicCAS(int *address, int compare, int val);
```

Listing 4.4: Atomic compare-and-swap

Atomic operations must be used carefully as they may reduce the performance of a program significantly. There are several reasons for this [48]:

- Atomic operations guarantee that the change is visible to all other threads immediately. This means that read and write operations go all the way to global or shared memory with no caching allowed.
- Conflicting atomic accesses to a shared location might require one or more tries by conflicting threads.
- If threads within the same warp perform an atomic operation to the same memory location, only a single thread can succeed, and all others must retry. In the worst case, an atomic operation must be performed in sequence by thread of an entire warp.

4.6 Memory access patterns

As most application data used by a GPU starts in global memory, it is crucial for performance that the global memory bandwidth is maximized. Warps are again of importance, as they do not only execute the same instructions but also issue memory operations together. When threads in a warp encounter load or store memory instructions, each thread provides an address that is grouped and presented to the device as a memory transaction. These addresses collectively create patterns which we should be aware of.

All global memory accesses go through the L2 cache, which is capable of performing 32-byte memory transactions. The L1 cache can perform 128-byte transactions, but may not be available depending on the device and whether L1 is enabled. Either way, the cache maps to aligned segments of global memory, always starting at multiples of the cache line size. Suppose that all threads in a warp read a 4-byte value from global memory. If the first address is a multiple of the cache line, it is said to be an *aligned memory access.* If the addresses collectively refer to a contiguous chunk of memory, the device is capable of a *coalesced memory access.* If addresses are both aligned and coalesced, all threads within the warp can be serviced by a single transaction. This illustrates the optimal case. In the worst case, addresses are neither aligned nor coalesced, and one transaction is needed for each thread in a warp, for a total of 32 transactions.

4.7 Summary

In this chapter, we have introduced the basics of the CUDA platform. We can start to glimpse ways of parallelizing our approach of computing influence from Chapter 2. Recall that the method we described is fundamentally about processing a collection of spreading graphs, and computing influence on each of them independently. Distributing spreading graphs between groups of threads is the first step. However, managing how groups of threads should cooperate to traverse graphs as efficiently as possible is a far greater challenge. That is what we will begin to explore in the next chapter, which introduces the basics of the Breadth-First Search algorithm, and ways it can be parallelized.

Chapter 5

Breadth-First Search in Parallel

In Chapter 2, we explained how our approach of computing the influence of Twitter users is divided into two phases, the forward and the backward phase. Both algorithms, while different, are based on the *Breadth-First Search* (BFS) algorithm. In this chapter we take a closer look at BFS, how it works, what it can be used for, and introduce some terminology that will be helpful. Then, in Section 5.4, we explore some of the known techniques for parallelizing BFS based on previous work. In Section 5.4.1, we discuss which techniques we have chosen to adapt for this thesis, and why. We then explain these techniques in further detail with a description of how each can be implemented.

5.1 Breadth-First Search

BFS is a fundamental graph traversal algorithm. It is useful for solving problems such as finding connected components, two-coloring of graphs, and shortest paths [23]. It is important in social network analysis, e.g. for computing centrality measures based on shortest paths, such as betweenness and closeness centrality [11].

Given a graph $G = (V, E)$ and an origin $s \in V$, BFS explores vertices and edges of G that are reachable from s. The name *breadth-first* refers to the order in which vertices are discovered: The algorithm visits vertices in *levels*, i.e. before exploring deeper into the graph, the algorithm visits all vertices on the current level before proceeding to the next. The level is equivalent to the distance to each vertex from the origin. In cases where an entire graph is reachable from s, the algorithm visits each vertex exactly once, and in the worst case, it must try all edges to discover all vertices. Thus, the running time of the algorithm is bounded by $O(|V| + |E|)$. Algorithm 3 shows a sequential implementation of BFS.

Algorithm 3 Breadth-First Search

```
1: function SEQUENTIAL_BFS(Graph $G$, origin $s$)
2:     $dist \leftarrow$ Size $|V|$ integer array
3:     for all $u \in V$ do
4:         $dist[u] \leftarrow \infty$
5:     end for
6:     $dist[s] \leftarrow 0$
7:     $Q \leftarrow$ FIFO queue with capacity $|V|$
8:     $Q$.ENQUEUE($s$)
9:     while $Q \neq \{\}$ do
10:        $u \leftarrow Q$.DEQUEUE()
11:        for all $v \in N(u)$ do
12:            if $dist[v] = \infty$ then
13:                $dist[v] \leftarrow dist[u] + 1$
14:                $Q$.ENQUEUE($v$)
15:            end if
16:        end for
17:    end while
18: end function
```

The algorithm starts by exploring the vertices adjacent to the origin. As non-visited vertices are detected, they are added to the back of the queue Q, which keeps track of which vertex to visit next. This process is repeated for each vertex in the queue until no new vertices are discovered. Each vertex is visited exactly once. In this implementation, the *dist* array serves two purposes: recording the level of each vertex, and to keep track of visited vertices. Each vertex is flagged as non-visited in lines 3-5. Once a non-visited vertex is discovered at line 12, it is added to the back of the queue and effectively marked as visited by updating the *dist* value.

5.1.1 Towards a parallel Breadth-First Search

To design a parallel BFS based on Algorithm 3, we look for tasks within the algorithm that can be divided into sub-tasks to be executed by multiple threads more or less independently. The level organization used by BFS proves to be crucial in this regard. To illustrate this point, we first give an example of a sequential BFS based on Figure 5.1a:

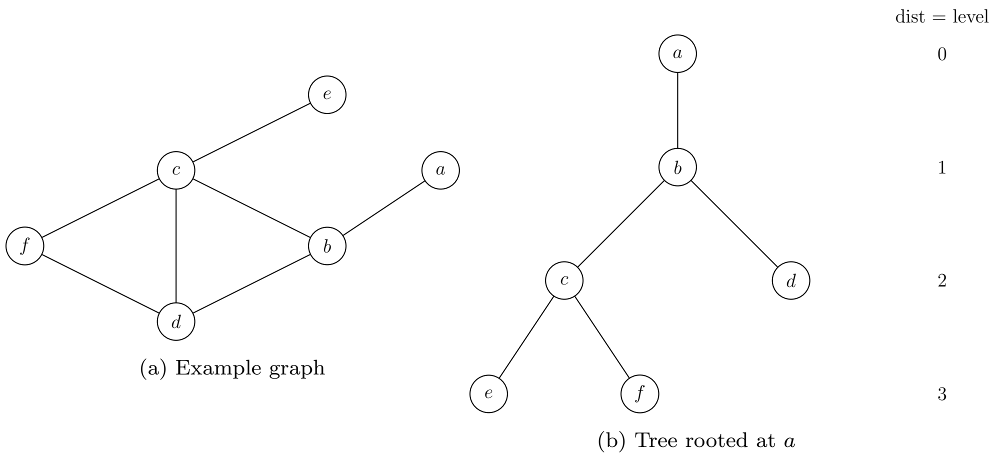


Figure 5.1: BFS example

Let a be the origin, then b is visited before any others as it is the only vertex with a distance of 1 from a. When b is processed, the algorithm again looks for non-visited vertices and finds both c and d, now with a distance of 2 from a. The same procedure is repeated for c and d to finally discover e and f having a distance of 3. The algorithm terminates after processing e and f as no further vertices are discovered.

If we only keep the edges in the graph that were used to discover new vertices, the graph in Figure 5.1b remains, which is a tree. BFS always implicitly produces a tree, sometimes called a BFS tree, whose vertices are those visited by BFS, and edges represent how vertices were discovered. The root of the tree is the origin. The vertices at each level k of the BFS-tree compose a *frontier*. *Frontier propagation* inspects the neighbors of each frontier vertex, adding undiscovered vertices to the next frontier. When necessary, we will use *vertex-frontier* to refer to the vertices, and *edge-frontier* to refer to the collection of all edges from the vertex-frontier.

Now to a most important point of the BFS tree. All vertices on a particular level must be processed before moving to the next level. However, the order in which vertices are processed at a particular level does not matter for the correctness of BFS, nor does the order in which the edges of each frontier vertex are inspected. This is good news as it implies that the frontier propagation step can be done in parallel.

With minor adjustments to Algorithm 3, we can emphasize how BFS operates in levels. Instead of one, two queues can be used, Q_k and Q_{k+1}, respectively containing the frontiers of level k and level $k + 1$. This is illustrated in Algorithm 4.

Algorithm 4 Breadth First Search

```
function SEQUENTIAL_BFS(graph $G$, origin $s$)
    $dist \leftarrow$ Same as Algorithm 3
    $Q_k, Q_{k+1} \leftarrow \{\}$
    $Q_k \leftarrow Q_k \cup \{s\}$
    while $Q_k \neq \{\}$ do
        for all $u \in Q_k$ do
            for all $v \in N(u)$ do
                if $dist[v] = \infty$ then
                    $dist[v] \leftarrow dist[u] + 1$
                    $Q_{k+1} \leftarrow Q_{k+1} \cup \{v\}$
                end if
            end for
        end for
        $Q_k \leftarrow Q_{k+1}$
        $Q_{k+1} \leftarrow \{\}$
    end while
end function
```

The frontier propagation is done in the nested for loops in lines 6-13. Each vertex in Q_k can be processed in parallel, and every edge from each vertex can be processed in parallel.

While the idea of BFS is fairly straight forward, it is by no means an easy algorithm to parallelize as it essentially involves all challenges we described in Section 3.3. In Section 5.4 we look at previous work on parallelizing BFS, focusing primarily on CUDA based implementations. In preparation, we first briefly analyze candidate data structures to represent graphs on GPUs in the next section, as well as showing a first attempt to implement a parallel BFS with CUDA in Section 5.3.

5.2 GPU Graph representation

To choose a suitable data structure to represent the graph in GPU memory we analyze the relevant operations for BFS and memory usage. As we are simply traversing the graph, choosing a structure supporting fast operations for adding or removing vertices or edges is not important. The graph is static throughout the search. What we want, however, is to be able to quickly lookup a vertex' neighbors, and to use as little memory as possible for quick transfer between host and device. It is also desirable that memory is laid out consecutively to support memory coalesced loads.

Using an adjacency matrix is a common way to represent a graph, but it uses $O(|V|^2)$ space, which is wasteful for graphs that are not exceedingly dense. It also takes $O(|V|)$ time to look up each neighbor for a node. Another common way to represent a graph is by adjacency lists. Each vertex is associated with a pointer to a linked list of edges. For sparse graphs, adjacency lists might be better suited than adjacency matrices for BFS thanks to neighbor lookup taking $O(deg(v))$ time for a vertex v, but linked lists with pointers introduce memory overhead and are also not suited for GPUs as memory can be scattered throughout different cache blocks [49]. Similar ideas involving dynamic arrays add unnecessary complexity to the program, and there are better options for our needs.

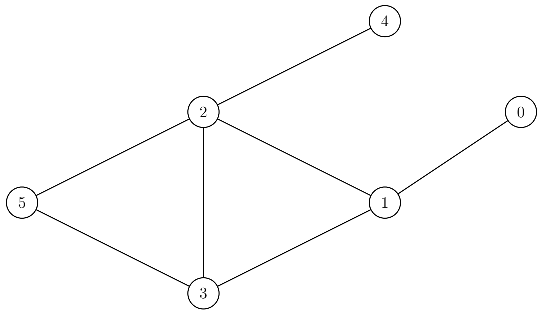


Figure 5.2: Example graph for CSR and COO

Another interesting format is *Compressed Sparse Row* (CSR). In CSR the graph is efficiently stored using two integer arrays V_a and E_a. The sum of neighbors for each vertex is accumulated and stored as offsets in V_a, marking where each vertex should begin to look for the second endpoints of their edges in E_a. The endpoints in E_a are stored in sorted order, such that endpoints of vertex 0 is stored in positions $[0 \mathinner{..} V_a[1]-1]$, endpoints of the vertex 1 in positions $[V_a[1] \mathinner{..} V_a[2]-1]$, etc. Thus, for each vertex v: $deg^+(v) = V_a[v+1] - V_a[v]$. To keep this consistent for every vertex, a last offset is added to V_a, implying V_a has size $|V|+1$. As E_a stores one endpoint of every edge, its size is $|E|$, and $2|E|$ in the case of undirected graphs. Neighbor lookup takes $O(deg(v))$ time in the worst case. Figure 5.3 show an example of how the graph in Figure 5.2 could be stored in memory using CSR.

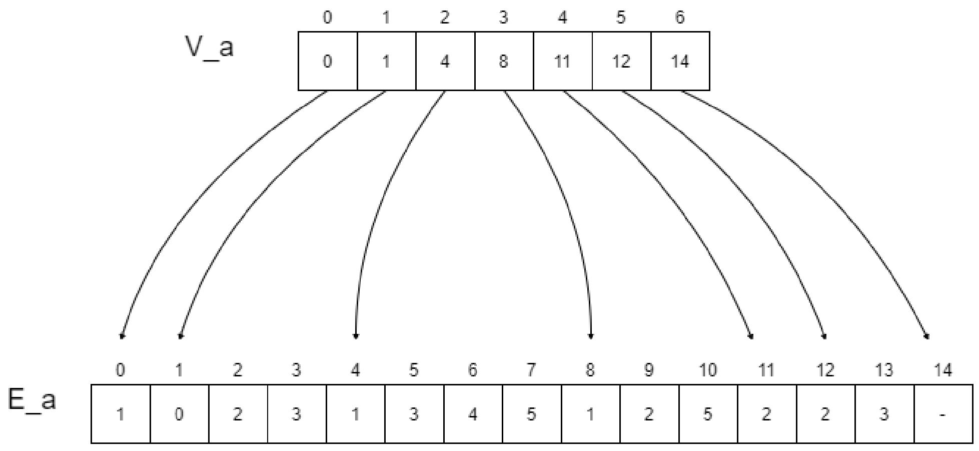
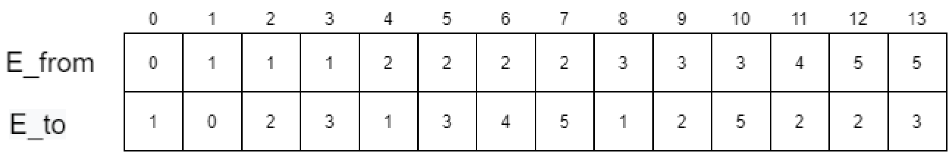


Figure 5.3: A CSR representation of the graph in Figure 5.2

Based on our list of criteria for a suitable graph representation for the GPU, i.e. fast neighbor lookup and a compact, predictable memory representation, CSR is a good candidate. Finally, we also mention that endpoints of all edges of the graph can be stored in two arrays E_{from} and E_{to}. This format is sometimes referred to as Coordinate (COO) [48]. This can be a useful way to balance the workload in cases where each thread should operate on an exactly equal amount of edges of the graph. Figure 5.4 shows how the graph in Figure 5.2 could be stored in COO format.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13
E_from	0	1	1	1	2	2	2	2	3	3	3	4	5	5
E_to	1	0	2	3	1	3	4	5	1	2	5	2	2	3

Figure 5.4: A COO representation of the graph in Figure 5.2

5.3 Introducing parallel BFS with CUDA

We now describe a first attempt of implementing a parallel BFS with CUDA. Afterward, we outline some of its issues. This method is based on threads communicating through a shared queue using atomic operations. We will refer to this method as `AtomicQueue`. In Chapter 4, we saw that the host (CPU) is responsible for managing the device (GPU). In terms of BFS, this implies that initialization steps are performed on the host, and the GPU will compute the frontier propagation step of each BFS-level in parallel. This can be thought of as the host delegating the "hard" work to the device.

Host code

Algorithm 5 describes the host side. The initialization phase in lines 4-11 is similar to sequential BFS. Device memory is generally not visible from the host (and vice-versa), and thus managed through device operations from the host:

- `devicememset(dst, val)` assign values in device memory.
- `devicememcopy_h2d/d2h(dst, src)` copies memory between host and device pointers.

When describing host code, variables with the $_d$ subscript refers to device memory. We assume that the variable keeping track of BFS-level, *level*, is visible and modifiable only by the host, but also visible on the device.

The data structures for queues and storing the distance from the origin can be recognized from sequential BFS. The current and next frontier is stored in Q_d and Q'_d. At line 10, the origin is copied to $Q_d[0]$, which defines the first frontier. The *back* variables refer to the size of a queue. The host launches the `bfs_kernel` until no further frontiers are discovered. By counting the number of iterations, the host keeps track of the current BFS-level.

Algorithm 5 Host pseudo-code for parallel BFS

```
 1: function BFS_HOST_SIDE(graph $G = (V, E)$, origin $s$)
 2:     $G_d \leftarrow$ Graph allocated in device memory
 3:     DEVICEMEMCOPY_H2D($G_d$, $G$)
 4:     $dist_d \leftarrow$ Array of integers of size $|V|$ allocated in device memory
 5:     for all $u \in V$ do
 6:         DEVICEMEMSET($dist_d[u]$, $\infty$)
 7:     end for
 8:     DEVICEMEMSET($dist_d[s]$, 0)
 9:     $Q_d, Q'_d \leftarrow$ Queues with capacity $|V|$ allocated in device memory
10:     DEVICEMEMSET($Q_d[0]$, $s$)
11:     $level \leftarrow 0$
12:     $back \leftarrow 1$
13:     DEVICEMEMCOPY_H2D($Q_d.back$, $back$)
14:     DEVICEMEMSET($Q'_d.back$, 0)
15:     while $back \neq 0$ do
16:         BFS_KERNEL($G_d$, $dist_d$, $Q_d$, $Q'_d$, $level$)
17:         DEVICEMEMCOPY_D2H($Q_d.back$, $Q'_d.back$)
18:         DEVICEMEMCOPY_D2H($back$, $Q_d.back$)
19:         $level \leftarrow level + 1$
20:         $Q_d \leftarrow Q'_d$
21:     end while
22: end function
```

We described the asynchronous behavior between the host and the device in Chapter 4. Here, we assume that the device operation at line 17 blocks the host after the kernel is launched. If this was not the case, the host could proceed to go back and execute line 15 before the kernel had time to discover a new frontier. We always assume kernel calls and device operations belong to a single stream of device work and are thus synchronous with respect to each other.

Device code

Algorithm 6 shows the implementation of the BFS kernel that accompanies the host side. Here we assume that the host launches a grid with at least one thread per vertex on the frontier. Threads that have an ID below the current frontier size grab a vertex from the queue in parallel, and subsequently iterate over all its neighbors. Threads add non-visited vertices to the next frontier as they are discovered, as well as updating their distance value. As in the sequential implementation, the distance value of a vertex is used to determine whether it has been visited before.

Algorithm 6 GPU pseudo-code, AtomicQueue BFS

```
1: function BFS_KERNEL($G = (V, E)$, $dist$, $Q$, $Q'$, $level$)
2:     $tid \leftarrow$ unique thread id
3:     if $tid < Q.back$ then
4:         $u \leftarrow Q[tid]$
5:         for $j \leftarrow V[u]$ to $V[u+1]$ do
6:             $v \leftarrow E[j]$
7:             if ATOMICCAS($dist[v]$, $\infty$, $level + 1$) $= \infty$ then
8:                 $old \leftarrow$ ATOMICADD($Q'.back$, 1)
9:                 $Q'[old] \leftarrow v$
10:            end if
11:        end for
12:    end if
13: end function
```

Suppose two or more vertices on the frontier share the same neighbor on the next frontier. In this case, the algorithm makes sure only a single thread can update the distance value of this particular vertex by performing the atomic compare-and-swap operation at line 7. This consequently makes sure that once a non-visited vertex has been found, only a single thread will enter the if block with this particular vertex and add it to the end of the next frontier queue at line 9. As several threads may find new vertices to insert to the next-frontier queue at the same time, we restrict access to the variable maintaining the end of the queue to only one thread at a time with the `atomicAdd` operation.

The BFS kernel terminates once all threads are finished iterating over their neighbors. The content of Q_d and Q'_d stays in device memory after the kernel terminates. Back on the host side, before launching a new kernel, the pointers of Q_d and Q'_d are swapped, effectively making Q'_d the current frontier by assigning Q_d to Q'_d.

This algorithm is work-efficient, but achieves a low degree of parallelism and is likely to run slowly on many graphs. The following problems are important to address for a faster, parallel BFS:

- The atomic operations performed at lines 7 and 8 are necessary for this implementation, but may drastically reduce performance and force sequential execution of threads.
- There is no effort to balance the workload between threads. For example, one thread may have to iterate over just a few neighbors, while another could iterate over millions.
- There is a large, unused potential to explore the edge-frontier in parallel.
- A difficult problem are non-coalesced memory accesses at lines 5, 6, and 7, potentially causing poor use of memory bandwidth.

5.4 Related work

We ended the previous section by listing some problems with parallelizing BFS. It should also be mentioned that the irregular nature of graphs makes it difficult to design implementations that suits all graphs. In situations where one can make assumptions about a graph's size and structure, there might be room to take "shortcuts" in order to make better use of the hardware. There are many studies on optimizing the performance of graph traversal algorithms, with or without parallel hardware. Developing faster BFS can have serious implications for many applications, but is considered a challenging algorithm to parallelize efficiently. In terms of this thesis, a faster, parallel BFS could imply processing larger amounts of Twitter data more rapidly. In this section, we study some of the previous work on parallelizing BFS.

Other architectures

To get a broader perspective, we briefly review some of the work that has been done on other architectures.

Agarwal et al. [29] implemented a parallel BFS for multi-core CPUs. Their implementation is based on using a shared queue, managed with atomic operations. To make their approach scale for multiple CPUs residing on different sockets, they used socket-local queues to reduce the communication overhead. The content of local queues could then be added to the global queue in batch. The communication involving the global queue was implemented with a *lock-free queue*.

Some graphs are too large to fit into the memory of a single computer. In this case, a graph can be divided between several computational nodes with distributed memory. Yoo et al. [16] worked with graphs that had billions of vertices and edges. The graphs were split between processors, such that each processor was being responsible for a portion of the graph. Because each processor has its own memory, communication was done through *message passing*.

A work-inefficient parallel BFS

The first paper that presented an implementation of BFS using CUDA was by Harish and Narayanan [19] and dates back to 2007, shortly after the release of CUDA. Rather than using a queue, the current frontier can be determined by checking the status of every vertex in the graph. This must be done at every BFS-level, for example by checking each vertex' distance from the origin. Their implementation launched one thread per vertex in the graph, and threads with a frontier-vertex would iterate through all its neighbors in the search of undiscovered vertices. Atomic operations are avoided as none of the data-races that occur lead to program-incorrectness. However, as a trade-off, many threads in each iteration will potentially not be assigned any work, which is a waste of computational resources. We will refer to this idea as `VertexArray`.

Another implementation was later given by Deng et al. [26] based on matrix multiplication, but this algorithm is also work-inefficient as it checks every edge of the graph at each level.

Hierarchical Queue

Now again consider the congestion due to several threads attempting to update the back of a queue simultaneously (e.g. in Algorithm 6 at line 9). Luo et al. [34] addressed this issue by using a hierarchical queue structure. Frontier vertices can be divided between groups of threads, e.g blocks or warps. When vertices on the next frontier are discovered, they are first added to local queues. Once a part of the next frontier is determined locally, a single atomic operation can be performed by one thread in the group to compute an offset in the above-level queue, shared by the group. Using the offset, the content of local queues can be written to the above-level queue in parallel.
Luo et al. implemented a 3-level hierarchy with global, block, and warp-level queues. They used a now outdated GTX 280 card with only 8 CUDA cores per SM, meaning warps were scheduled in four 8-thread groups in the hardware. Atomic operations would still have to be used at the warp-level queue, but the collision of threads within a warp performing an atomic operation on the same memory address was avoided. As the size of the memory used in the lower-level queues is limited and also pre-allocated, they had to convert graphs into near-regular graphs by splitting vertices with large degrees.

Improving load balance

Neither implementation so far has dealt with the load imbalance between threads during the search. That is, some threads are responsible for vertices that have more neighbors than others. Hong et al. [39] presents ideas that can be used to improve load balance for BFS. One idea is the *virtual warp-centric programming method*: Rather than assigning a different task to different threads, a chunk of tasks are allocated to each warp. The kernel then alternates between a SISD and a SIMD phase. During the SISD phase, a single stream of instructions is performed by each warp, i.e. threads within the warp grab the same data. During the SIMD phase, threads within the warp operate on different data to perform their shared task. In terms of BFS, this method can be used to map vertices to virtual warps such that each virtual warp can process the vertex' edges in parallel.

This method has some drawbacks. In situations where tasks are divided such that each warp is assigned less data than the warp width, some CUDA cores will be idle. The performance gain is also limited by the work that has to be performed during the SISD phase (Amdahls Law). One remedy is to split the warp into *virtual warps*, i.e. divide a warp into even smaller groups according to the size of the underlying problem.

Deferring outliers is another technique proposed by Hong et al. to improve load balance. The idea is that large tasks (outliers) can be deferred and executed in parallel at a later time, e.g. by storing the task in a queue. In BFS, vertices with a high degree represent a large task, i.e. many edges must be processed, possibly stalling other threads. To improve load balance, vertices with a similar degrees can then be processed at the same time. Vertices with a particularly large degree can be deferred in a queue to be processed by a larger group of threads at a later time. Vertices with a degree smaller than a certain threshold are processed immediately.

Prefix sum

Merrill et al. [43] proposed an approach that involves using prefix sum operations to build the queue, thus avoiding atomic operations.

Bottom-up BFS

Although not necessarily parallel, an idea called *bottom-up* BFS was recently proposed by Beamer et al. [42]. The traditional (top-down) BFS algorithm can process many edges before finding next-frontier vertices. However, only a single edge is sufficient to discover a next-frontier vertex. Many edges that are being processed can point to vertices that have already been visited. In the worst case, all edges on the edge frontier could be inspected before finding the single edge that points to a single undiscovered vertex. To emphasize, there can be a total of $n * (n - 1)$ edges, but the final BFS-tree only consist of $n - 1$ edges. The idea of bottom-up BFS is to iterate over all vertices in the graph in search of non-visited vertices. When they are found, the algorithm checks if their neighbors are on the current frontier. If they are, the current non-visited vertex belongs to the next frontier. In other words, instead of searching through all neighbors of frontier vertices, non-visited vertices attempts to find any parent among its neighbors. However, this approach is not always beneficial if we consider small frontiers with few edges. For this reason, smart implementations can use a hybrid approach, dynamically switching between top-down and bottom-up when needed.

Liu and Huang [51] implemented a parallel BFS that combines or expands on the aforementioned ideas. These include dynamically switching between top-down and bottom-up BFS depending on properties of the frontier vertices, using prefix sum operations to compute queue offsets to avoid atomic operations and to improve load balance by categorizing frontiers based on vertex degrees. For example, a single thread can be responsible for computing frontier vertices with a low degree, warps can be used for vertices with larger degrees, and so forth.

Other ideas

Zhang et al. [52] presented an approach that uses dynamic parallelism, i.e. a CUDA feature that allows a parent kernel to launch child kernels. A parent grid can handle the vertex-frontier, and a child grid can inspect the edge-frontier. Although their implementation did not perform well initially, they argue that it simplifies the code, which in turn makes it easier to apply optimization techniques (like the ones we have discussed).

Hong et al. [38] proposed a hybrid BFS that use both CPUs and GPUs. They argue that real-world graphs often have a small diameter due to the small-world phenomenon. Thus, inspired by Harish and Narayanan, they claimed that checking the frontier status of every vertex in the graph can be a good alternative to using queues. However, this is only if the frontiers are expected to be large. Their idea was to dynamically switch between a queue-based approach and one that checks all vertices for their frontier status based on the properties of the frontier. The queue step could be done on the CPU, and once frontiers get large, work is handed to the GPU.

In some graphs, searching for a path from an origin to a destination can be done more efficiently by performing two searches simultaneously: One starting at the origin, and one starting at the destination. Once the paths meet, the search is over. This method was first proposed by Pohl et al. [3] and is sometimes called a bi-directional search.

5.4.1 Summary

We have seen several ideas of optimizing a parallel BFS that we could use to speed up the process of computing influence on Twitter. Recall from Chapter 2 that spreading graphs are always DAGs, and that we start traversing DAGs from one or more sources vertices, all the way to a single sink. Every edge processed by the algorithm plays an equally important role in distributing influence along the paths to the sink. Thus, methods that attempt to speed up graph traversal by skipping edges, such as bottom-up BFS or bi-directional BFS, do not seem as attractive at first glance.

We have reasons to believe that spreading graphs are generally small and have short diameters [37]. The structure of spreading graphs are thus often in sharp contrast to graphs many researchers are concerned with when optimizing BFS, which are often larger graphs. Our problem is rather one of several, but “small” DAGs, where influence is computed independently for each of them. Note that spreading graphs *can* be relatively large, and we do not exclude this possibility. However, for small spreading graphs, naive techniques *could* potentially be sufficient for good performance.

Table 5.1 show the names of the ideas we have chosen to adapt to compute influence. In Section 5.5, we will take a closer look at each of these ideas and see how they can be implemented.

Idea	Description
`AtomicQueue`	From Section 5.3
`VertexArray`	Check all vertices in each iteration
`EdgeArray`	Check all edges in each iteration, variation of VertexArray
`Virtual Warp`	(Virtual) Warp-centric method
`Prefix Sum`	Construct queue with prefix sum

Table 5.1: Ideas for parallel BFS

5.5 Implementing a parallel BFS with CUDA

In the algorithms presented below, we assume a host-side similar to Algorithm 5. We already covered AtomicQueue in Section 5.3. Again, our goal is to traverse spreading graphs as fast as possible and compute influence along the way. Because adapting the ideas in Table 5.1 to compute influence mostly requires a similar approach, we focus on the parallel graph traversal here and explain other important implementation details in the next chapter.

5.5.1 VertexArray

Algorithm 7 shows one possible way to implement the ideas of Harish and Narayanan [19], i.e. checking the frontier status of every vertex in each iteration. At each level, the host launches a kernel with at least one thread per vertex in the graph. A vertex is on the current frontier if its distance value is equal to the current level.

Threads that identify their vertex as part of the current frontier iterate through all its neighbors and updates the distance value of non-visited vertices, which implicitly adds them to the next frontier. As long as at least one vertex is updated, there will be another level to check.

Algorithm 7 GPU pseudo-code, VertexArray

```
 1: function BFS_KERNEL($G = (V, E)$, $dist$, $level$, $done$)
 2:     $tid \leftarrow$ get unique thread id
 3:     if $tid < |V|$ and $dist[tid] = level$ then
 4:         for all $j \leftarrow V[tid]$ to $V[tid + 1]$ do
 5:             $v \leftarrow E[j]$
 6:             if $dist[v] = \infty$ then
 7:                 $dist[v] \leftarrow level + 1$
 8:                 $done \leftarrow false$
 9:             end if
10:         end for
11:     end if
12: end function
```

At level 0, only the origin has a *dist* value equal to 0, and thus defines the first frontier. Next-frontier vertices are discovered by comparing the *dist* value to the ∞ flag at line 6. When a new vertex is discovered, it is assigned the distance value of $level + 1$. Thus, it will be recognized as a frontier-vertex in the next iteration. As long as at least one new vertex is discovered, there is another BFS-level to explore. Notice that unlike Algorithm 6, there are no atomic operations used for the work performed at lines 6-9. This is regardless of multiple threads potentially accessing and updating the same memory location simultaneously. This is because of threads updating to the exact same value and thus not affecting the program correctness.

Due to the massive overhead caused by launching threads for every single vertex in the graph at each level, this algorithm is not expected to perform particularly well for many types of graphs. In the worst case, the number of levels is equal to the diameter of the graph. Let L be the number of levels, then checking every vertex at each level has a time complexity of $O(|V| * L)$. Any given edge reachable from the origin is processed twice and thus takes $O(|E|)$ time overall. This implies a total of $O(|V|^2)$ time in total when the diameter is large, a far less work-efficient algorithm than sequential BFS. However, for dense graphs where L is small, this algorithm might pay off.

5.5.2 EdgeArray

A similar approach to VertexArray is to check every edge in each iteration. This implementation is based on Springer et al. [53]. The graph is stored as two arrays E_{from}, E_{to}, which corresponds to the COO format described in Section 5.2. One advantage of this approach is that it achieves a better load balance by all threads performing the same amount of work.

Algorithm 8 Check every edge in each iteration

```
function BFS_KERNEL($E_{from}$, $E_{to}$, $dist$, $level$, $done$)
    $tid \leftarrow$ unique thread id
    $u, v \leftarrow E_{from}[tid], E_{to}[tid]$
    if $tid < |E|$ and $dist[u] = level$ then
        $dist[v] \leftarrow level + 1$
        $done \leftarrow false$
    end if
end function
```

Similar to Algorithm 7, this algorithm is expected to be slow on sparse graphs with a large diameter.

5.5.3 Virtual Warp

We now demonstrate how the parallelism in the edge-frontier can be exploited based on an idea from [39], again by using AtomicQueue as the base algorithm. Here, vertices on the frontier are mapped to warps rather than threads, and edges of each vertex are mapped to the threads within a warp. If we expect an average degree per vertex that is below the warp width, we can divide warps into virtual warps. We use the term *lane* to refer to a thread within a warp.

In Algorithm 9, the SISD-phase consists of lines 2-6. Each lane performs the same instruction to set up for using the same data. The virtual warp size *warp_sz* should be a divisor 32, i.e. the warp size in hardware. The unique *warp_id* is simply the unique thread id *tid* divided by *warp_sz* using integer division. Here we assume the number of warps is equal to the frontier size. In the SIMD-phase in lines 7-14, lanes grab unique edges in parallel.

Algorithm 9 GPU pseudo-code, VirtualWarp

```
1: function BFS_KERNEL($G = (V, E)$, $dist$, $Q$, $Q'$, $level$)
2:     $warp\_sz \leftarrow$ virtual warp size
3:     $warp\_id \leftarrow$ global warp id
4:     $lane \leftarrow$ thread index within the warp
5:     if $warp\_id < Q.back$ then
6:         $u \leftarrow Q[warp\_id]$
7:         for all $j \leftarrow V[u] + lane$ to $V[u+1]$ do
8:             $v \leftarrow E[j]$
9:             if ATOMICCAS($dist[v]$, $\infty$, $level$) $= \infty$ then
10:                $old \leftarrow$ ATOMICADD($Q'.back$, 1)
11:                $Q'[old] \leftarrow v$
12:            end if
13:            $j \leftarrow j + warp\_sz$
14:        end for
15:    end if
16: end function
```

We give an example of Algorithm 9 based on the graph in Figure 5.5. Suppose we already started executing BFS with a as the origin, and have now reached the second BFS-level in the graph, i.e. the current frontier consists of vertices $Q = \{b,\ c,\ d\}$. The next step is to expand the edge-frontier and identify the next frontier. Some vertices in the graph have enough neighbors that it could be beneficial to have a small group of threads assigned to each vertex, inspecting their neighbors in parallel.

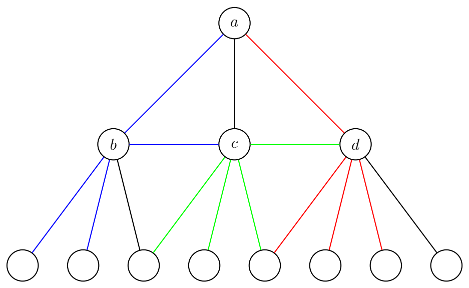


Figure 5.5: Processing neighbors in parallel with virtual warps

We set $warp\_sz = 4$, assigning four threads to each vertex. Each group receives a unique $warp\_id$, and grabs a vertex from the queue. For example, threads with $warp\_id = 0$ grabs vertex b, threads with $warp\_id = 1$ grabs vertex c and threads with $warp\_id = 2$ grabs vertex d. This sums up the SISD phase. The SIMD phase consists of inspecting each neighbor in parallel. This is shown by colored edges in Figure 5.5. Some vertices have more than four neighbors, and thus some threads will have to process more than one edge.

5.5.4 Prefix Sum

Prefix sum or *scan* is an important operation for many applications and parallel algorithms [6]. Suppose we are given a sequence of n numbers:

$$a_0,\ a_1,\ a_2,\ ...,\ a_{n-1}$$

Then prefix sum generates a new sequence:

$$a_0,\ a_0 + a_1,\ (a_0 + a_1) + a_2,\ ...,\ \sum_{i=0}^{n-2} a_i + a_{n-1}$$

Each step along the new sequence is the sum of the current number and all previous numbers. Scans can be generalized for a sequence of any ordered elements, and can be used with any other binary associative operator (such as multiplication (*)). An *exclusive scan* does not include the current element i.

In the context of BFS, prefix sum can be used to compute queue offsets, i.e. where each frontier vertex should be placed in the queue.

We give an example based on the graph in Figure 5.6, where we assume BFS is executed sequentially for the sake of simplicity. Suppose again that a was the origin and we are currently on the second level with $Q = \{b,\ c,\ d\}$ as the frontier, where b is first in the queue.

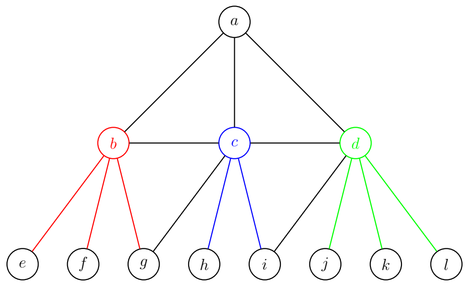


Figure 5.6: generate queue with prefix sum

In the search for the next frontier, vertex b would find three vertices: e, f and g, vertex c would find two vertices: h and i, and d would find the three vertices j, k and l. That is, as a sequence, they respectively found: 3, 2, 3 vertices for the next frontier. The result of performing an exclusive scan on this sequence is 0, 3, 5. This new sequence provides positions of where next-frontier vertices can be inserted in the queue. The neighbors of b are written in positions 0 to 2, neighbors of c at 3 and 4, and neighbors of d in positions 5 to 7. Writing next frontier elements to the queue can now be performed in parallel, for example with one thread writing the neighbors of b, another thread for neighbors of c, and so forth.

5.6 Summary

In this chapter, we have seen how graphs are traversed in a breadth-first manner, and how a BFS can be implemented both sequentially and in parallel. Looking ahead, we have used ideas from Section 5.5 when implementing algorithms to compute influence. It will be more clear how we mixed these ideas together in Chapter 6 and 7.

Chapter 6

Computing Influence in Parallel

In this chapter, we report on our parallel implementation of computing influence. In general, our work includes adapting ideas outlined in Table 5.4.1 to compute influence and to create a framework for experiments. Because of the data we had available for this project, we only parallelized the **backward phase** introduced in Chapter 2.

6.1 Introducing the framework

There are several steps taken in the process of computing influence on the GPU. We cover some of these steps in further detail later in this chapter:

1. **Parsing and data preparation** Program execution starts on the host. The initialization phase includes parsing Twitter-data and preparing all relevant data structures on the host.

2. **Graph analysis** Analyzing the graphs can give us several advantages. These include being able to use the limited resources of the GPU more freely by categorizing graphs based on their structure.

3. **Data transfer** Data are copied to the device. After this step has been performed, the device is ready to execute.

4. **Compute influence sequentially** We also compute influence sequentially. This corresponds to executing Algorithm 2 on each spreading graph. The sequential version comes in handy if we want to verify the device's results. It also serves as the basis of comparison for all parallel implementations. One could argue that if we can not outperform a sequential version, it is not worth to implement these algorithms on the GPU. We come back to this point in Chapter 7, where we measure the speedup in going from sequential to parallel. Our sequential implementation was implemented in C.

5. **Compute influence in parallel** In this step, we execute device code to compute influence in parallel. We used two software libraries for performing certain

operations in parallel, such as prefix sum. *Thrust* [76] is a library included in the CUDA toolkit, and *CUB* [79] is a parallel CUDA library developed by Nvidia Research.

6. **Verifying results** This step includes comparing the influence values computed by the host and the device. It is expected that results vary slightly due to floating-point errors. In cases of program bugs, `cuda-memcheck` and `cuda-gdb` is useful for debugging.

7. **Final influence values** An important step in how we measure influence is to merge the local influence results of each spreading graph into a final score for each user. However, because we focused mainly on parallelizing graph traversal, we did not emphasize this part of the process in our results.

6.2 Parsing and data preparation

To perform computations, our first task is to read our data into meaningful data structures. How we organize data in memory can be crucial for performance, and thus an important part of our implementation. To maximize the use of memory bandwidth when threads perform memory transactions on the GPU, we laid out data as "flat" as possible, i.e. in one-dimensional arrays rather than using complex structures, storing related data together. The datasets we use consists primarily of T spreading graphs, with a total of N nodes and M edges, including U unique twitterers combined.

The datasets have the following format [81]:

- A list of N *graph indicators*: $i \to t$,
  where $\to$ denotes a mapping/function,
  $i \in [0 \,..\, N-1]$ is a node ID,
  and $t \in [0 \,..\, T-1]$ is a tweet ID [1].

- A list of M edges: (i, j),
  where i and j are node IDs.

- A list of N *node labels*: $i \to (r\_id,\ \#f)$,
  where i is a node ID, r_id is a unique (real) Twitter user ID, and
  $\#f$ is their number of followers on Twitter.

- A list of T *graph labels*: $t \to p_t$,
  where t is a tweet ID,
  and p_t is the username of the publisher of tweet t.

Additional to the functions described, we allocate data structures for the in/outdegrees and influence values associated with each node, as well as queues, all implemented as size N arrays. We compute degrees by counting how many times nodes i

[1]We will use the term tweet here to refer to a spreading graph and its associated values

and j appear as a first and second endpoint. Conveniently, the graphs are already stored in COO format. To convert the graphs to CSR format, we allocate an $N+1$ array of integers and perform an exclusive scan of outdegrees.

For threads to be able to recognize which tweet they are working on, we also needed a way to convert the CUDA built-in variables to node IDs. To do this, we perform an exclusive scan over the number of vertices of each graph and store the result in an integer array of size $T+1$. A thread can then convert its local index to a node ID by adding the tweet offset. The last element allows us to use this data structure as a way to access the size of each graph. We keep a similar data structure for edges, used to both access the memory segment containing edges of a particular graph, and the number of edges in each graph.

To compute a final influence value for each user, we gather all local influence results from different graphs into a final score for each user in a size U array of floating-point values. We used similar data structures on the host and device.

6.3 Compute influence in parallel

Spreading graphs have different structures, and there are no easy way to divide them perfectly between computational resources while maintaining efficiency. We used principles from Chapter 3 and 4 during development. This includes striving to distribute data as evenly as possible between threads, avoiding expensive communication by making the host and device work independently whenever we can, and try to put the more limited resources on the GPU to good use. The last point brings us back to the principle of locality. We make sure to use the limited, faster memory (registers and shared memory) on data that is accessed more often. It is also worth keeping in mind that overusing these resources can limit the number of active threads in a streaming multiprocessor.

For the rest of this section, we describe some of these points in more detail and outline the most crucial steps we took in adapting the parallel algorithms in Table 5.1 to compute influence with our approach from Chapter 2.

6.3.1 Distributing work between threads

How work is distributed between threads is crucial for performance. We have seen that the potential for parallel execution in BFS lies within the frontier propagation step. Now that we are dealing with several graphs at the same time, we can think of parallelism in layers: at graph, vertex, and edge layer.

The EdgeArray algorithm that uses the COO format is slightly different than the rest. Here we simply made sure that each thread takes care of more or less the same amount of edges. However, as we introduce much more work than needed in every iteration, this is an interesting tradeoff.

All other implementations use the CSR format. In this case, we generally allocate a group of threads to each graph. A group can range from being a subset of a thread block to an entire grid. However, we usually relied on a block of threads to compute influence on each graph. A thread block consists of 1 to 1024 threads, which could be a reasonable amount for many spreading graphs, and block-synchronization allows us to avoid communicating with the host between each iteration. Furthermore, thread blocks can be divided into subgroups, where each subgroup is responsible for one or more vertices on the frontier, and threads of each subgroup for inspecting outgoing edges in parallel. The idea of arranging threads this way is inspired by the warp-centric method described in Chapter 5. We will see which effect this kind of arrangement has on performance in Chapter 7.

6.3.2 Managing queues

Most BFS implementations that we have seen are based on queues. Although we have used two queues for demonstrating BFS, one for the current frontier, and one for the next, we decided to use only a single queue in practice. A single queue uses less memory, but arguably requires more logic in a parallel environment. We think of the front of the queue as the *head*, and the back as the *tail*. Because *tail* might change during the frontier propagation step, we introduce a third variable $count = tail - head$. All threads must agree on the value of *count* at all times. This is achieved by synchronization of threads. In cases where *count* is a shared variable, it is sufficient that a single thread computes its value.

The primary way we stored queues was to use an array of size N in global memory. A group of threads can keep local queue variables (e.g. in shared memory) and work exclusively on the segment of the queue that is reserved for each graph.

6.3.3 Initial frontier

The initial part of the backward phase is always to find the first frontier. Our only apparent option is to check every node in a graph and decide whether each one is a source. For implementations that require sources to be added to a queue, we have some options. One solution is to use atomic operations. However, the larger the initial frontier is, the more threads will attempt to write to the tail of Q at the same time. This introduces a potential bottleneck.

Adding the first frontier to a queue is a case of a more general operation sometimes called *filtering* or *stream compaction* [25, 46]. Filtering can be thought of as the task of extracting all elements in a set that satisfies a predicate to a compact format. Parallel implementations of filtering are often based on prefix sum [25]. Filtering is an operation that is common enough that it may be optimized for free by the CUDA compiler [46]. It is also a function commonly found in (parallel) software libraries, sometimes called `copy_if`.

We have done experiments with both solutions, using Thrust and CUB's `copy_if` function or CUB's `ExclusiveScan`, and atomic operations.

6.3.4 Managing sources, edge-removal and influence

We now repeat a few crucial steps of the backward phase which require some additional thought in a parallel environment. At any point during the execution of the backward phase, the frontier consists exclusively of sources. Each source's influence value is distributed along edges to its neighbors. Once an edge has served this purpose, it is removed. When a node no longer has any edges pointing to it, its graph-local influence value is final, and it has become a source. We also know that all spreading graphs only have a single sink (the original publisher of the tweet).

Thus, we made the practical decision that no algorithms should ever add the sink to the frontier. This does not change the result, and also avoids division by zero when distributing influence. We did this by saying that a node u is a source if $deg^{-}(u) = 0$ and $deg^{+}(u) > 0$.

Removal of edges is done by decrementing a node's indegree. Sources are discovered exactly once: during the initial phase of execution, or when their indegree has just become zero. Decrementing indegrees and updating influence are operations that happen frequently. It is often the case that multiple sources share a common neighbor and attempt to perform these operations on the same memory address simultaneously. We generally used atomic operations to coordinate threads in this regard.

6.3.5 Categories of spreading graphs

When we parse our datasets, we know how many graphs there are and their sizes. However, if we design programs that are supposed to handle the entire dataset in a single run, our best bet is to guess a set of parameters that fits *most* spreading graphs. A drawback is that too small groups of threads might underutilize the hardware in cases of large graphs, and too large groups can be excessive for small graphs. Additionally, by designing kernels that consume all graphs in the dataset in one run, it is harder to perform experiments that include manually caching data in the faster GPU memory without worrying about running out of resources.

One remedy is to sort graphs into categories based on their features and design category-specific kernels. This also allows us to filter out all graphs that have no impact on the result, such as graphs with only one vertex (a tweet that no one retweeted). We implemented four categories of graphs based on their number of vertices: *Empty, Small, Medium, Large.* Table 6.1 show the exact rules that we used for each category, where n is the number of vertices.

Category	Rule
Empty	$n \leq 1$
Small	$1 < n \leq 32$
Medium	$32 < n \leq 1024$
Large	$n > 1024$

Table 6.1: Category rules

We perform the categorization on the host during the initial phase of our program. We first associate each graph with a category, then count the number of graphs in each category, and finally perform an exclusive scan on these counts to mark where the tweet IDs of each category reside. In this way, we can distribute the tweet IDs belonging to each category to their designated kernel from the host, and fewer changes must be made in each kernel if we adjust the categories.

The bounds we chose are the same as the CUDA warp size and within range of typical block sizes. There are practical reasons for this. For example, a single thread or a warp could be sufficient to compute the influence on small graphs, and a block for medium (or large) sized graphs. It also simplifies implementation, especially in cases where we experimented with the CUB library which has functions designed for using warps and blocks (e.g. `WarpScan` and `BlockScan`).

6.4 Summary

We have outlined the most important parts of our implementation. In practice, we tested a variety of promising ideas, some harder to implement than others. In the next chapter, we present our results.

Chapter 7

Experiments and Results

In this chapter, we present our experiments. We compared the performance of our sequential and parallel algorithms on two datasets with different characteristics. The parallel algorithms include a combination of ideas listed in Table 5.1. Our main objective was to see whether a parallel implementation could outperform a sequential one and if so, by how much.

7.1 Methodology

7.1.1 Hardware

For experiments, we used an Intel Core i5-6600K, 3.50 GHz CPU, and an Nvidia GeForce RTX 2070 Super GPU. This GPU has 2560 CUDA cores divided between 40 Streaming Multiprocessors, 8 gigabytes of memory, and Compute Capability 7.5.

7.1.2 Datasets

The two datasets that we used include a set of 3140 spreading graphs from Twitter which we call *Seibert*, and a larger, randomly generated dataset, which we call *Random Spreading Graphs (RSG)*. We show some of the dataset characteristics in Table 7.1. The variables n and m respectively refer to the number of vertices and edges in spreading graphs.

Dataset	Seibert	RSG
T	3140	15.5K
N	48K	2.9M
M	46K	450M
U	14K	29.7K
$avg(n)$	16	187
$median(n)$	12	74
$max(n)$	49	29.7K
$avg(m)$	15	29.2K
$median(m)$	10	262
$max(m)$	120	39.8M

Table 7.1: Dataset characteristics

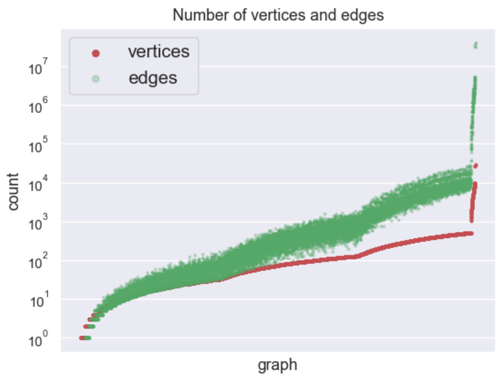
After removing 761 tweets with no retweets (graphs with one vertex) from the Seibert dataset, only a few, relatively small spreading graphs remained. About half of the remaining graphs contained cycles. All the algorithms we implemented assume that input graphs are DAGs. The cycles led to fewer vertices being explored in total.

We were interested in measuring algorithm runtimes on a larger variety of graphs. One option was to duplicate the Seibert dataset. However, because its graphs were small and the issue of cycles, we decided to generate a dataset using a random number generator instead. Our method of generating graphs is explained in Section 7.1.2 and is similar to the model of E. N. Gilbert [73] where each edge has a certain probability of being included, independent of other edges.

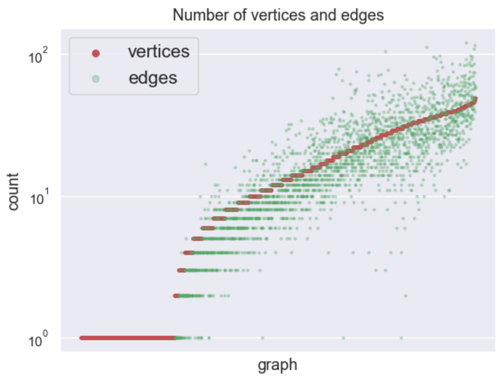
Figure 7.1 show numbers related to the structure of graphs in both datasets. In Figure 7.1a and 7.1b, we plotted each pair (n, m), i.e. the size of the vertex and edge set of each graph. In Figure 7.1c and 7.1d we plotted the average and median outdegree of vertices in each graph.

We see that some graphs in RSG are large compared to Seibert. As mentioned before, we expect that *most* spreading graphs will be small. Tweets with many retweets do occur [83, 84, 85], but spreading graphs are again limited to represent spread via follower-relationships, and it is uncertain how large they can be. We chose to generate a few larger graphs for experiments.

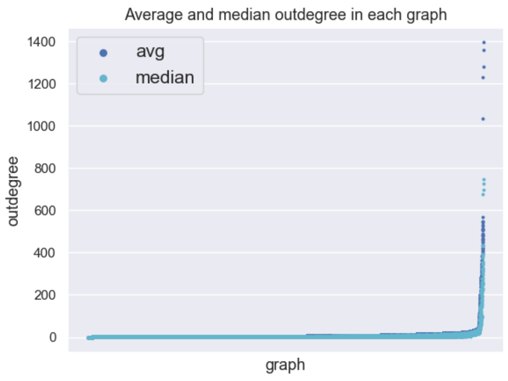
We were expecting to receive a larger real-world dataset, but because of the difficulties of extracting data from Twitter, these did not arrive in time for this project.



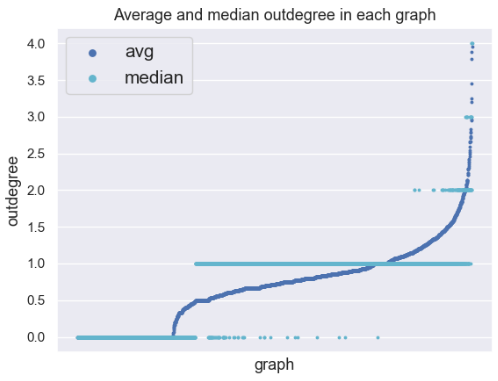
(a) Number of vertices and edges of graphs in RSG



(b) Number of vertices and edges of graphs in Seibert



(c) Outdegrees of graphs in RSG



(d) Outdegrees of graphs in Seibert

Figure 7.1: Dataset characteristics

Random DAG generator

We now describe how the graphs in the RSG dataset were generated. Our aim was that the DAGs should mimic real spreading graphs, i.e. have varying size, with most of them being relatively small, have multiple sources, and a single sink.

We generated the graphs in batches with different probability parameters for graph sizes and generating edges. Algorithm 10 generates a set of N random graphs.

Algorithm 10 Random DAG generator

```
function GENERATE_N_DAGS($N$, $lo$, $hi$, $percent$)
    $A \leftarrow$ Empty set of DAGs
    for $i = 0$ to $N - 1$ do
        $n \leftarrow$ generate random number between $lo$ and $hi$
        $G = (V, E) \leftarrow$ graph with $n$ vertices
        for $u = 0$ to $V - 2$ do
            if $u$ is an outlier (5% chance) then
                Write an edge from $u$ to each $v \in [u + 1 \,..\, n - 1]$
            else
                for $v = u + 1$ to $n - 1$ do
                    Generate an edge $(u, v)$ with $percent$ probability.
                end for
            end if
        end for
        for $u = 0$ to $n - 2$ do
            if $deg^+(u) = 0$ then
                Add an edge from $u$ to the sink.
            end if
        end for
        $A \leftarrow A \cup G$
    end for
    return $A$
end function
```

The number of vertices n is a random number between lo and hi. The *percent* parameter is the probability of generating an edge, which we conservatively kept between 2% and 17% to keep the dataset small enough to fit on the GPU. To add some variation to edges, we added a 5% chance of being an outlier. In this case, an outlier is a node with an edge to all remaining nodes with a higher node ID in the same DAG. To make sure there is only a single sink, we find all additional sinks and connect them to the true sink $(n - 1)$ on lines 15-19.

7.1.3 Execution configuration

In Section 6.3.1, we gave an idea of how we organize threads to compute influence. While the exact number of threads specified in the execution configuration depends on the input and the algorithm, there are some patterns that we follow in general.

All kernels are launched with a grid of b one-dimensional thread blocks each of size

k, where k is a multiple of 32, and $32 \leq k \leq 1024$. This spawns a total $b * k$ threads. Data is distributed with the help of the CUDA built-in variables.

A typical configuration we used is one block per graph, where a block can be divided into groups of threads. Given T tweets, we can launch T blocks of size 256. Each graph can be mapped to a block using the *blockIdx.x* variable. Threads within each block can be split into groups of size g such that each frontier-vertex is handled by g threads, inspecting their neighbors in parallel.

7.1.4 Method

To test the performance of our algorithms, we executed each algorithm several times (20 or more runs) and computed their average runtimes in milliseconds. We started the timer at the beginning of the backward phase and ended after the sink received its final influence update. The time it took to allocate data structures, transferring data, merging influence to a final result, etc, is not included. This is because we focused mostly only on optimizing steps related to the execution time of the backward phase.

Some results are presented as the speedup ratio between parallel and sequential execution time. Generally, the speedup of two algorithms A and B is measured by dividing the resulting runtime of A with the resulting runtime of B. For example, if it takes 1000 milliseconds to run the sequential algorithm, and only 100 milliseconds to run a parallel algorithm, the parallel algorithm runs 10 times faster (a speedup of 10). The execution time of host programs was measured with the `steady_clock` in the C++ chrono library, and the execution time of device code was measured with *CUDA events*. All programs were compiled with the CUDA compiler nvcc, using flags *-arch=sm_75* and *-O3*

7.2 Algorithm runtime comparison

We begin with experiments performed on both datasets. In general, we are interested to see whether our parallel implementation outperforms the sequential one, and which one performs better. We abbreviate the following algorithm names in figures for convenience: AtomicQueue (AQ), VertexArray (VA), EdgeArray (EA).

7.2.1 Result of sequential version

Our first results are the runtimes of the sequential version on both datasets. We ran the algorithm 100 times on each and measured the average runtime. These are the results we used in order to measure the speedup of the parallel algorithms.

Sequential runtime	
Dataset	Miliseconds
Seibert	1.03
RSG	933.53

Table 7.2: Average sequential execution time

The time to compute the Seibert dataset is low. Because of the cycles, only 62.5% of vertices were visited in total if we include the sinks. In comparison, 100% of vertices in RSG were visited.

7.2.2 Performance of EdgeArray

Although this approach has more of a brute-force nature than some of the other algorithms, we were curious to see its performance. After all, it should achieve good load balance and is relatively easy to implement. The implementation used in the following experiment synchronizes with the host between each time all M edges have been inspected.

We did 16 experiments, each doubling the number of edges per thread. Experiment i was performed with 2^i edges per thread ($\approx M/2^i$ threads launched in each iteration), where $i \in [0 \mathinner{..} 15]$. For each experiment, we did a sub-experiment by varying block sizes. It is hard to reason about which block sizes to use, and we wanted to see whether it plays a role.

We begin with the results on RSG, shown in Figure 7.2. The number of edges here is large, and the number of iterations plays a crucial role. The number of iterations corresponds to the maximum number of BFS-levels in any graph. To emphasize, it may happen that M edges are inspected to update the influence of a single node.

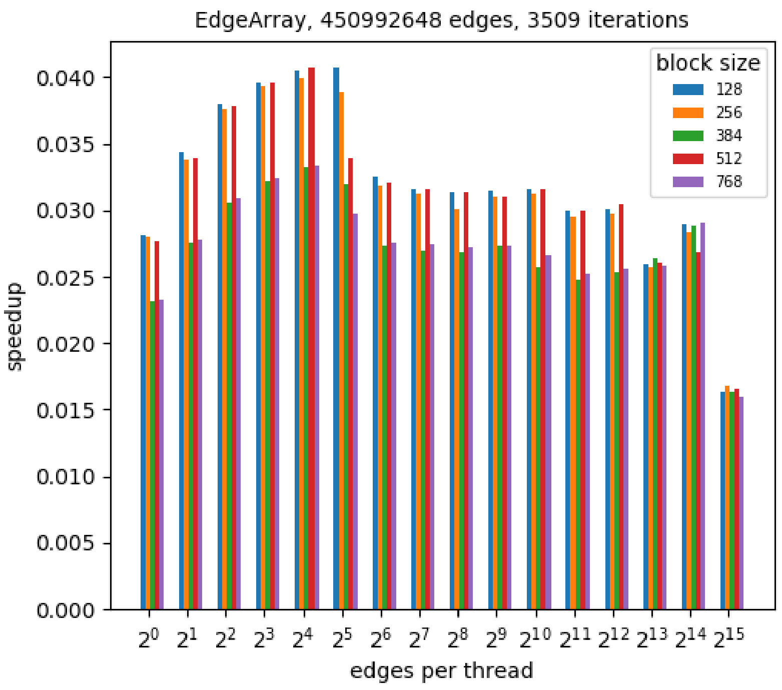


Figure 7.2: Results of EA on RSG

We see that the algorithm ran substantially slower than the sequential version in all cases. The performance is slightly better with around 16 to 32 edges per thread, which corresponds to about 14 to 28 million threads being launched. We can also see that block sizes that are powers of two performed better in general.

The results of running the same algorithm on the Seibert tells a different story, and are presented in Figure 7.3.

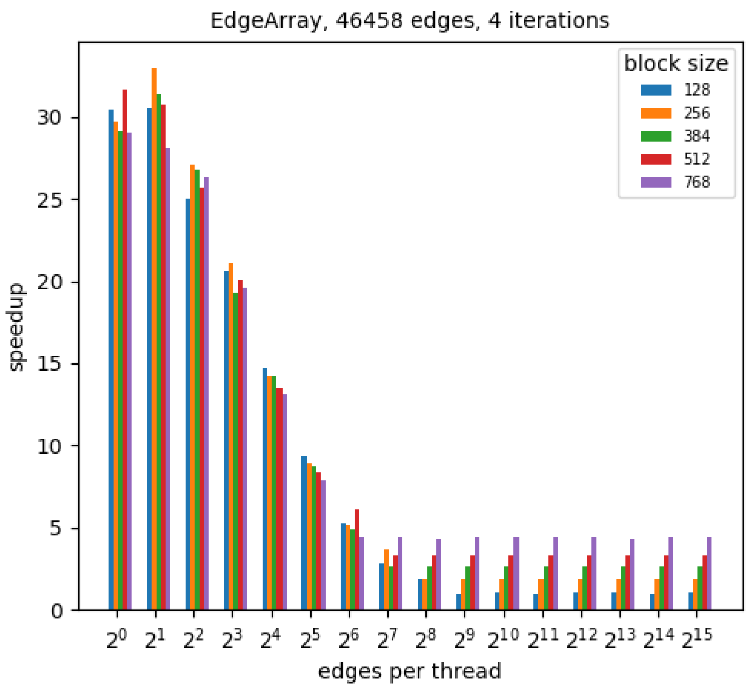


Figure 7.3: Results of EA on Seibert

The algorithm ran faster than the sequential one on Seibert in all experiments. Here, the number of edges and iterations are way less than on RSG. The best result is a speedup of 32.9 when using two edges per thread. We can see that using fewer edges per thread gave better results, and that block size did not play a major role. The results are similar when using between 2^7 and 2^{15} edges per thread. These experiments mostly consist of a subset of threads within a single block doing all the work.

7.2.3 Effect of distributing work

When computing influence, we are dealing with several graphs, where each graph has the potential of being processed in parallel. In this regard, we were interested in answering the following questions:

1. What is the effect of going from a single thread doing all the work (sequential), to using one thread per graph?
2. Rather than just one thread per graph, what is the effect of using a block per graph to process vertices in parallel?
3. What is the effect of dividing the thread block into groups, processing neighbors of each vertex in parallel?

Traversing the neighbors of a vertex is a crucial part of our algorithm, and the CSR graph format makes this easier than the COO format. Using the AtomicQueue and VertexArray algorithms, we performed experiments on both datasets. We used blocks of size 256, which in case of adding edge-parallelism are split into 8 groups of 32 threads. Recall that VertexArray replaces a queue by exhaustively checking every vertex in each BFS-iteration, and does so to avoid threads racing for access to the end of the queue. Thus, it seemed counter-intuitive to run this algorithm with a single thread per graph, so we skipped it. Figure 7.4 show the result of the experiment on the RSG dataset.

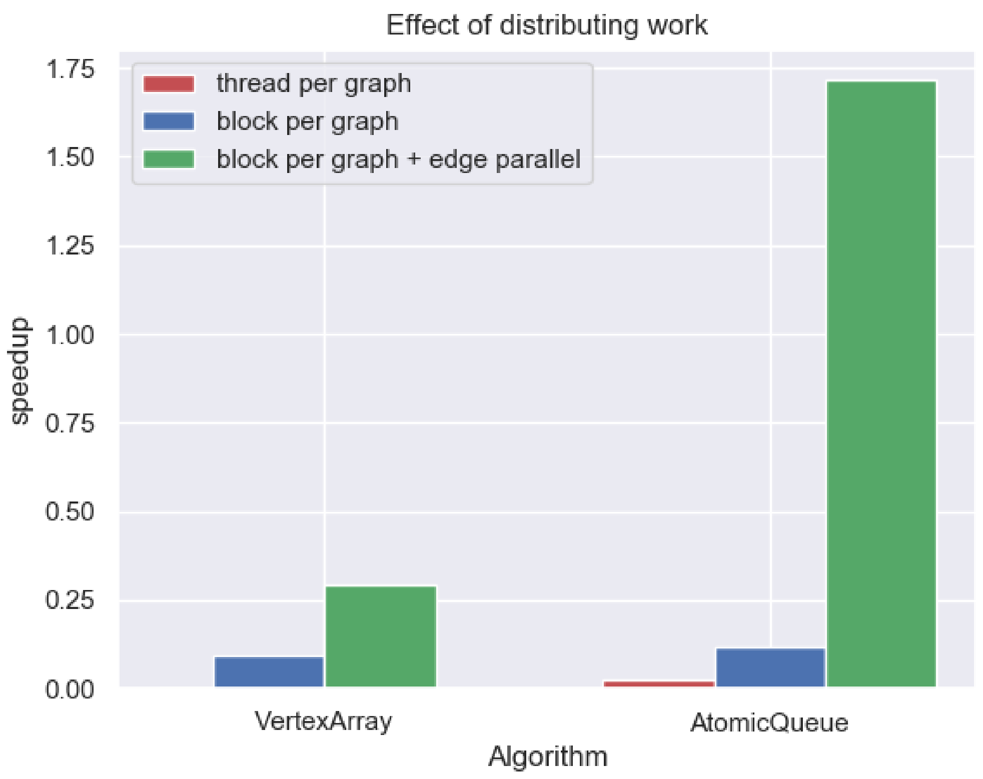


Figure 7.4: Results of AQ and VA on RSG

We see that only the edge-parallel AtomicQueue was able to outperform the sequential version, with a speedup of 1.7. In other cases, the CPU was much faster.

Again the results on Seibert are more promising, shown in Figure 7.5.

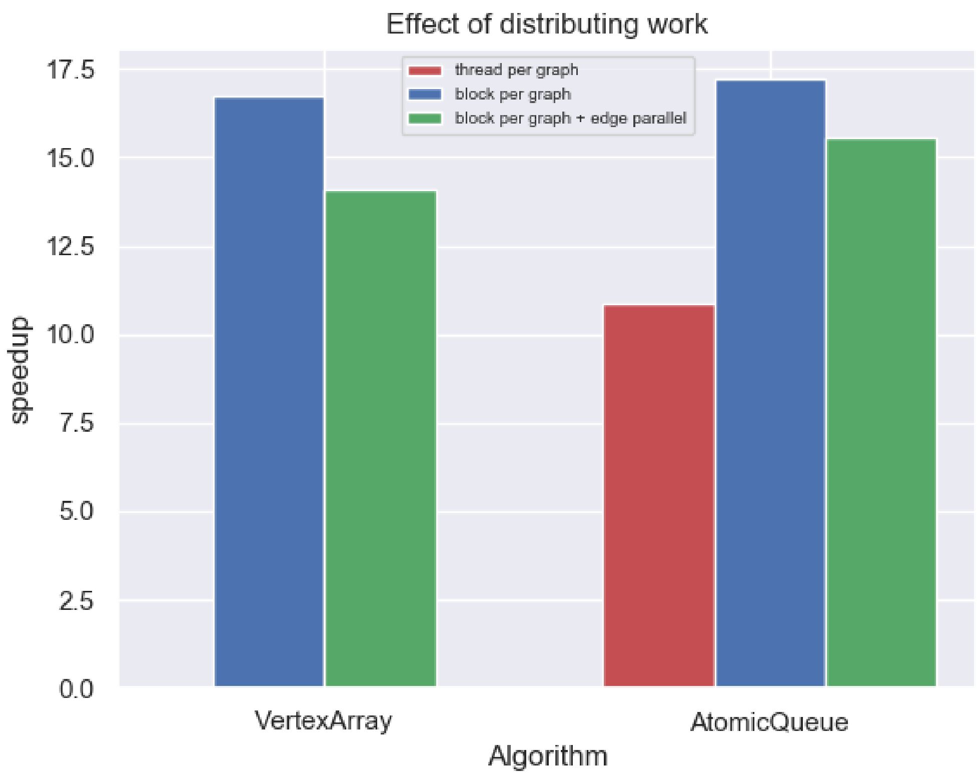
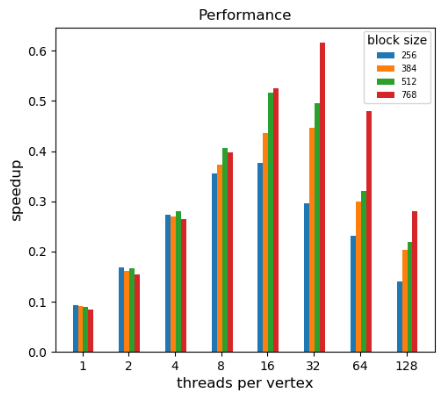


Figure 7.5: Results of AQ and VA on Seibert

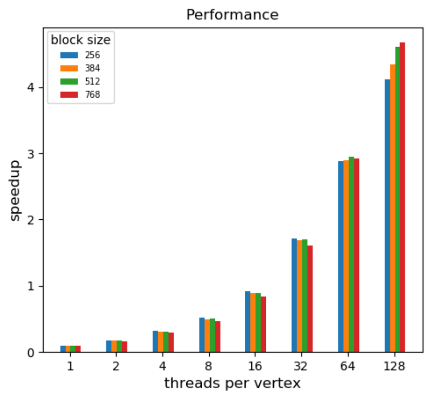
AtomicQueue executed about 10 times faster than sequential only using one thread per graph. There is also a notable leap when using a block per graph. However, we do not see the same effect of processing edges in parallel as we did on RSG. Both algorithms achieved a speedup greater than 15. Both algorithms performed better than EdgeArray on RSG, but worse on Seibert.

7.2.4 Adjusting threads per vertex

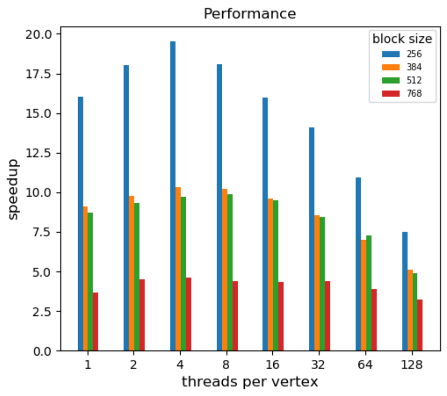
In the previous experiment, edges were processed in parallel using 32 threads per vertex. We wanted to see how changing this parameter affects performance on both algorithms. The results are displayed in Figure 7.6. Recall that blocks are simply divided into equally sized groups, e.g. using a block size of 768 implies inspecting neighbors of $768/128 = 6$ vertices in parallel, etc.



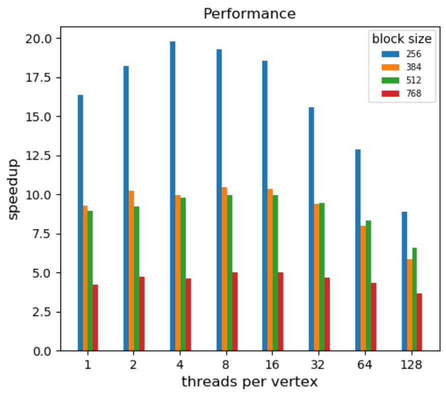
(a) Results of VA on RSG



(b) Results of AQ on RSG



(c) Results of VA on Seibert



(d) Results of AQ on Seibert

Figure 7.6: Varying number of threads per vertex

It is clear from these experiments that varying the number of threads used to process edges in parallel affects performance. Starting with results on RSG from Figure 7.6a and 7.6b, we see that VertexArray performed worse than sequential in all cases. The performance of AtomicQueue, on the other hand, was not only better than sequential in all cases but got better each time more threads were added to assist in processing a vertex' neighbors. The best result was when using 128 threads per vertex and a block size of 512, which resulted in a speedup of 4.6.

In Figure 7.6c and 7.6d, we see that the results was similar for both algorithms on Seibert. The performance decreased when adding more than 32 threads per vertex, but increased when adjusting the number of threads to around 4 per vertex. Interestingly, there seems to be a relationship between smaller block sizes and performance, where blocks of size 256 performed notably better.

Block per vertex

Because of the trend that we see in Figure 7.6b, we performed another experiment with AtomicQueue on the RSG dataset where we increased the number of threads per vertex even further. However, we changed the strategy slightly and used full thread blocks to cooperate on one frontier vertex at a time, processing even more neighbors in parallel. The results are shown in Figure 7.7.

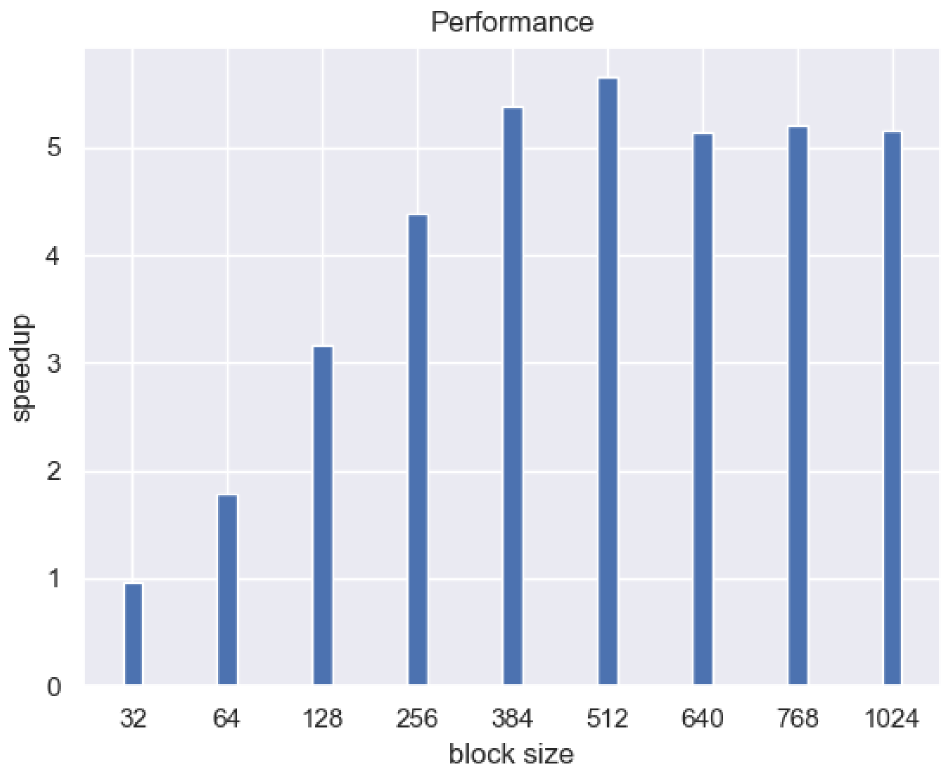


Figure 7.7: Results of AQ on RSG, block per vertex

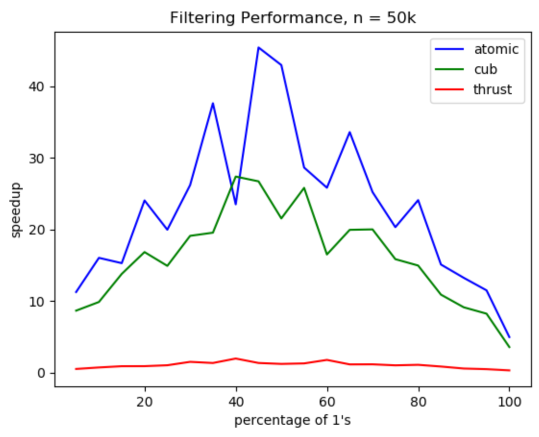
We see that an even greater speedup is achieved by inspecting more neighbors in parallel. The performance increases until it peaks when inspecting 512 neighbors in parallel with a speedup of 5.6.

7.2.5 Stream compaction of initial frontier

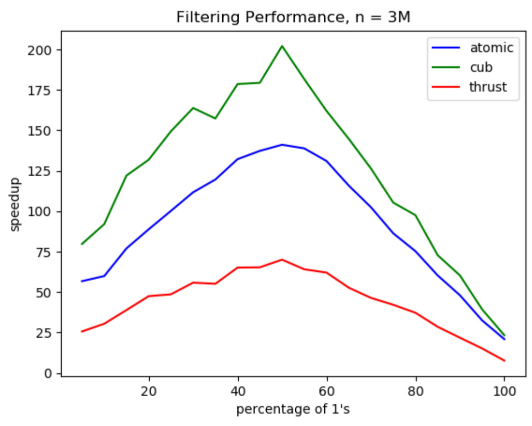
We discussed stream compaction and its relevance in Section 6.3.3. We have explained that our typical way of distributing work is by having groups of threads working at one graph at a time, which implies finding the initial frontier of a particular graph as a local step. However, there is a potential for analyzing the in/outdegrees of all N nodes and identifying all initial source nodes as a separate step. Thus, we were interested in comparing the performance of a naive filtering implementation using atomic operations to Thrust and CUB's parallel `copy_if` function.

We performed four main experiments with a randomly generated array of n boolean variables, where each element has a $[5, 10, 15, .. 100]$ percent chance of being *true*. The filtering consists of traversing the array, compacting each true element into a second array. Figure 7.8a and 7.8b has a number of elements roughly corresponding to the number of nodes in Seibert and RSG, respectively. The effect of adding more elements is illustrated in Figure 7.8c and 7.8d. We compared the performances against a sequential implementation.

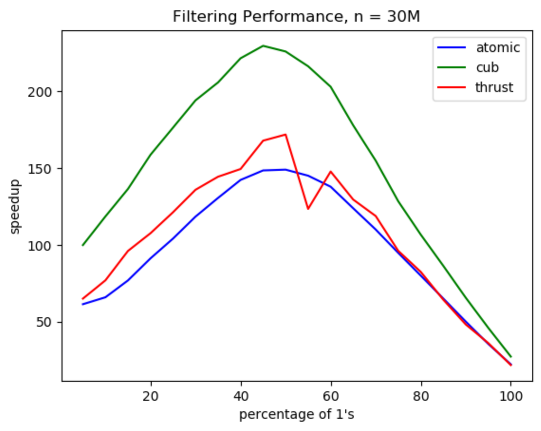
The curves follow roughly the same pattern for all input sizes. That is, less speedup is achieved when there are either few or many true elements. The results are generally better when there are between 40-60% true elements in the array. We can see that



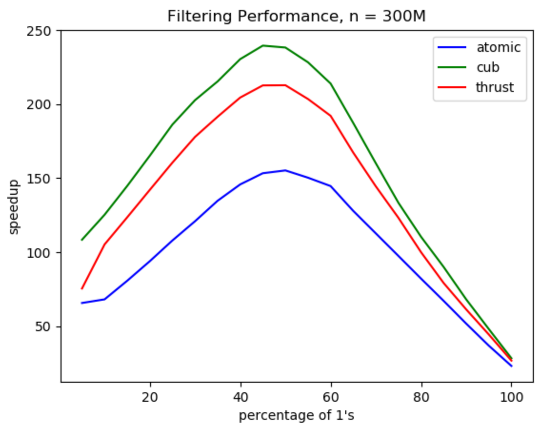
(a) Filtering 50,000 elements



(b) Filtering 3 million elements



(c) Filtering 30 million elements



(d) Filtering 300 million elements

Figure 7.8: Filtering speedup

the atomic version outperforms both Thrust and CUB when the input size is small and that the roles are reversed for larger input sizes.

We also measured the time it took to filter the initial frontier of Seibert and RSG. We decided to not include Thrust further as it was outperformed by CUB in the previous experiment. For this experiment, we present the result as absolute runtimes in milliseconds. To emphasize, in Figure 7.9, shorter bars imply better performance.

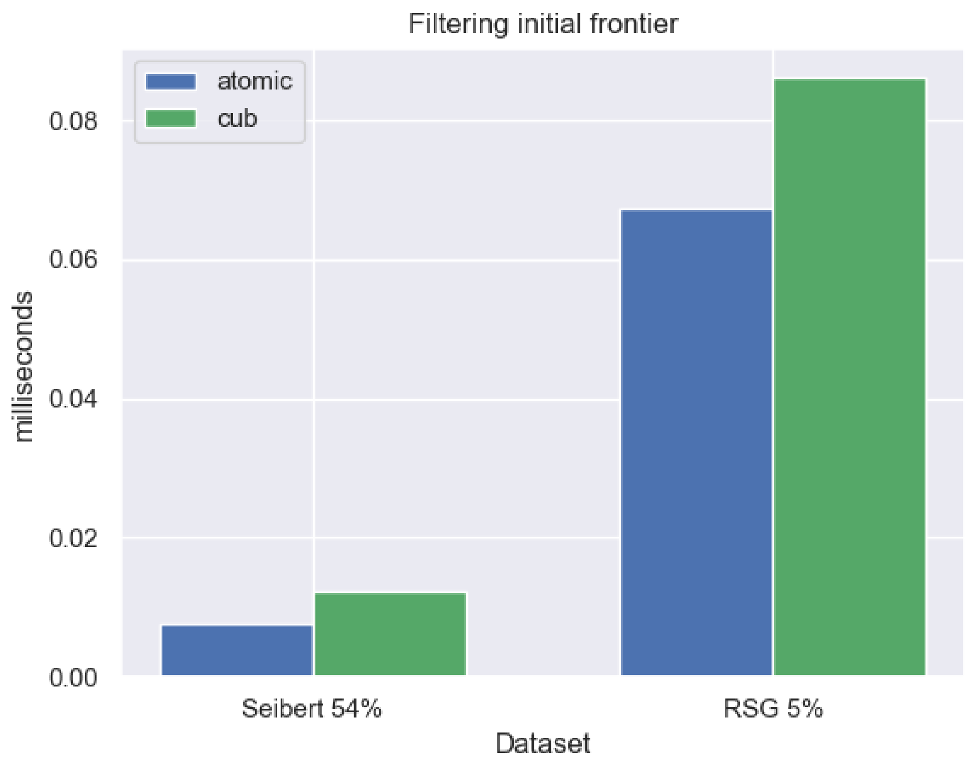


Figure 7.9: Filtering on RSG and Seibert

In the Seibert dataset, 26200 out of 48575 nodes were added to the initial frontier. This implies that only around 1000 vertices were visited after the initial frontier of Seibert. In the RSG dataset, 152745 out of 2894554 nodes are added. We can see that the atomic implementation outperformed CUB both for Seibert and RSG.

7.2.6 Experiments using categories

We explained the motivation behind categorizing graphs in Section 6.3.5, and how it was implemented. In practice, we have seen a limited speedup on the RSG dataset where a single "one size fit all" kernel has been used for all graphs. The best result was a speedup of 5.6 with AtomicQueue when using a block per graph, processing neighbors of frontier-vertices one at a time. In Table 7.3 we see the resulting number of graphs that were put in each category for both datasets, where n is the number of vertices in a graph.

Category	Rule	Seibert	RSG
Empty	$n \leq 1$	764	170
Small	$1 < n \leq 32$	1870	5250
Medium	$32 < n \leq 1024$	506	9845
Large	$n > 1024$	0	170

Table 7.3: Number of graphs in each category

Because of the limitations of the Seibert dataset, we did not include it in further experiments.

We first measured the time it took to compute graphs in each category with the sequential version and the version of AtomicQueue that performed best on RSG previously. The resulting runtimes are presented in Table 7.4 for reference. We can see that most of the time was spent computing the 170 Large graphs.

Runtime in milliseconds		
Category	Sequential	AtomicQueue
Small	2.05	0.99
Medium	90.91	22.01
Large	812.24	123.47

Table 7.4: Runtimes of graphs in RSG

We now turn our attention to experiments done with kernels that aim to optimize the computation on graphs within each category. From previous experiments, we have seen that distributing threads and altering the execution configuration can have an impact on performance. Thus, the first step of optimization is to tune these parameters for each category, as listed below:

- **Small** A block of 256 threads are split into 8 groups of 32 threads, allocating a warp per graph. For implementations using CSR, each warp is again split 8 groups of 4 threads used to process edges in parallel.
- **Medium** For implementation using CSR, a block of 256 threads are split into 8 groups of 32 threads, allocating a block per graph. Each group of 32 threads is used to process edges in parallel. For EdgeArray, a block of 1024 threads were used per graph.
- **Large** One block per graph. We keep the AQ kernel that performed best in the past, i.e. a block of 512 threads per graph, processing neighbors of one frontier-vertex at a time. We keep the result from Table 7.4.

The other optimizations include using the more limited, fast GPU memory. We hypothesized that by caching certain data in faster memory we could increase performance. The following list explains each optimization:

- *original* refers to the version of AQ that did best on RSG previously, shown for comparison
- *AQ, EA, VA* refers to baseline versions of these algorithms that are tuned for each category in terms of distributing threads and execution configurations.

- *queue* refers to maintaining block-wide or warp-wide queue in shared memory.
- *influence* refers to manually caching the influence of each vertex in a graph using shared memory.
- *indegree* refers to manually caching the indegree of each vertex in a graph using shared memory.
- *filtering* refers to using `ExclusiveScan` of CUB to filter the initial frontier rather than atomics.
- *Reg* refers to a kernel for Small graphs that uses one thread per graph, by maintaining a queue of 32 integers in registers. There is a maximum of 255 registers per thread in all CUDA enabled cards, making this idea only suited for smaller graphs.

Figure 7.10 and 7.11 show the resulting runtimes of kernels for the Small and Medium category.

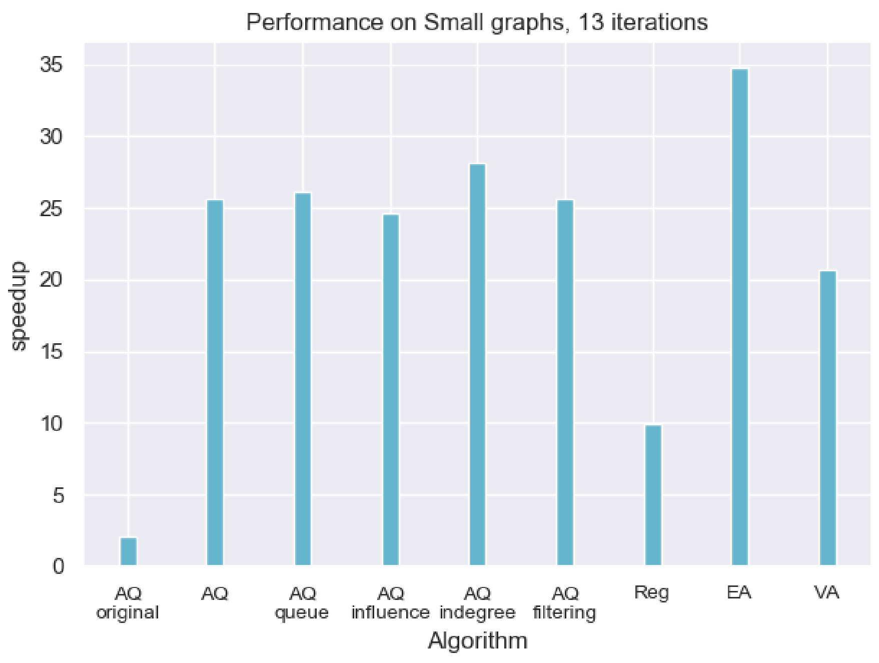


Figure 7.10: Results on small graphs in RSG

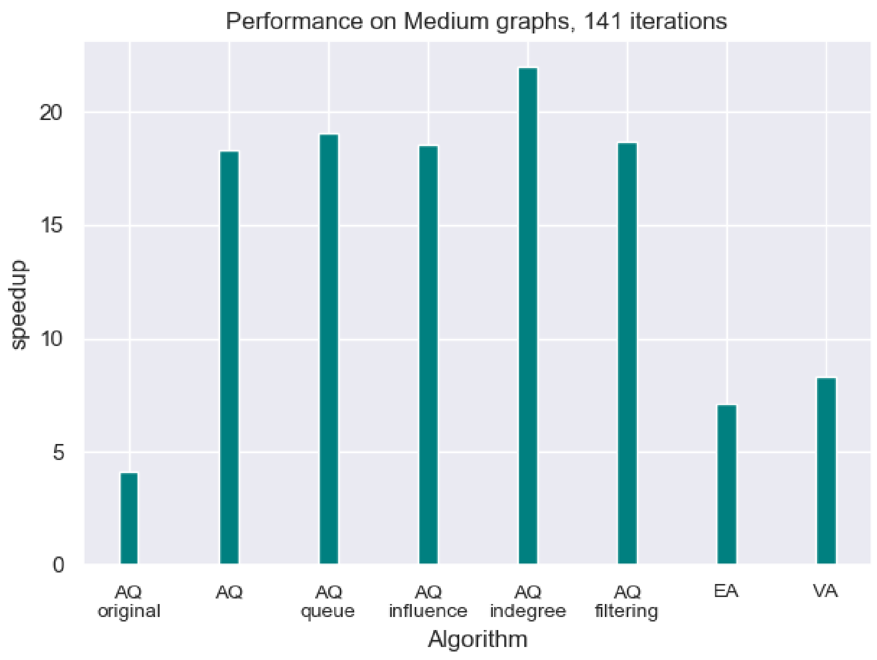


Figure 7.11: Results on medium graphs in RSG

We can see that the original AQ was outperformed in all cases, both for Small and Medium graphs. EdgeArray performed best on graphs in the Small category and AtomicQueue performed best in the Medium category. The effect of storing data in shared memory was limited. The effect was best when storing the indegree of each node. Recall that this value is accessed frequently and decreased with atomic operations. Although not included here, we applied the same caching technique of indegree and influence for both EA and VA, but with little to no effect.

7.3 Discussion

Our goal was to implement a parallel solution to computing influence and tweaking it to see what runs faster. We managed to solve the problem in parallel, and our experiments show that naive implementations work well on small graphs, while work-efficient algorithms perform better on larger graphs. We achieved a decent speedup on the Twitter dataset. We also showed that it can be beneficial to divide graphs into categories.

We achieved a limited speedup on the RSG dataset. The only parallel algorithm to outperform the sequential one was AtomicQueue, and we saw an increase in performance when processing more neighbors in parallel. This makes sense as some of the largest graphs have almost 30,000 nodes and around 40 million edges, which correspond to more than 1,000 edges per node on average. Thus, it was not surprising to see that most of the execution time was spent processing the large graphs. Nor was it surprising that the other algorithms performed poorly on these types of graphs, as both Edge/VertexArray are essentially based on a trade-off between additional work and sequential access to the back of a queue. The work-efficiency of AtomicQueue tips the balance in its favor when graphs are causing many iterations where little work is done per level.

The results on Seibert/Small/Medium graphs are perhaps more interesting and was our main focus. There are several reasons for this which we have mentioned before, one being that it is easier to implement optimizations on graphs with limited sizes. While the performance on the Seibert dataset was good for most of our algorithms, we mentioned that only smaller portions of each of its graphs were processed. The good news is that the algorithms that performed well on the Seibert dataset also performed well on Small graphs in RSG, which required more work. EdgeArray performed the best in this regard, likely due to the combination of few edges, few iterations, and a good load balance. VertexArray on the other hand never performed best. It performed similarly to AtomicQueue on Seibert, but fell slightly behind on Small/Medium graphs. We expected it to perform well when EdgeArray performed well (short diameter), and better when the number of edges implies a large enough overhead.

Our other ideas included storing the queue, influence, and indegree in shared memory, as well as doing local filtering of the initial frontier with CUB. We can see that this had limited effect on this dataset. We perform similar number of operations on indegree and influence, but unlike indegree, the updated influence values must be written back to global memory after the group of threads has finished traversing the graph, which could have an effect. There is also limited effect by storing the queue in shared memory, likely due to each address in a queue only being accessed twice, i.e. when enqueueing and dequeueing. The effect of filtering is again dependent on the graph structure, but did not have an effect here. Finally, the *Reg* kernel used a single thread per graph, storing the queue in registers. There are several reasons why this not perform as well as others, one being low parallelism, and another that we used consecutive threads on consecutive graphs. This implies a warp divergence stalling other lanes each time the graph structure differs. An idea could be to use such a kernel on even smaller graphs, e.g. on two nodes connected by a single edge.

We also tried to decide when it would make sense to switch to using a prefix sum to constructing the initial frontier of the entire dataset. First of all, and importantly, we saw a large parallel speedup that only grew with the number of elements of filtering in general. Somewhat surprisingly, the atomic implementation followed the same curve as using CUB and Thrust. We expected that the atomic version would achieve a greater speedup with less true elements (i.e. less atomic operations needed), and decrease linearly as the number of true elements grew. We mentioned briefly that filtering may be optimized by the compiler, however, we did not investigate this further. Nonetheless, it is hard to make a conclusion on whether filtering is worthwhile based on our limited experiments and data. It is, again, dependent on the number of nodes in the set and the number of initial sources.

Chapter 8

Conclusion and Future work

Our goal in this thesis have been to implement a fast, parallel solution to computing influence on Twitter using a GPU. We investigated previous work done both on Twitter and on parallelizing BFS in order to make better decisions along the way.

We achieved our goal of implementing a parallel solution. We performed experiments on different data in order to see which algorithm performed better under different circumstances and saw that a naive implementation can work well for small graphs, but fall behind on larger inputs. We managed to achieve a decent speedup on Twitter data.

8.0.1 Future work

There are many things we could try with more time, data and experience.

First of all, we are certain that there is a potential to further speed up the process of computing influence with parallel programming. For example, we saw many interesting ideas on ways to parallelize Breadth-First Search in Section 5.4 that we were not able to implement thoroughly. These include using a hierarchical queue structure, a linear non-atomic version using prefix sums, or dynamically switching between algorithms based on frontier properties. Additionally, there is surely more potential using categories to avoid bottlenecks, for example by categorizing based on the number of edges, number of initial source nodes, degrees, etc. In other words, any situation where one can make assumptions about the input should be favorable.

We could also do a more in-depth analysis of our kernels. CUDA provides profiling tools such as NSight/nvprof, providing low level hardware metrics from execution, making it easier to reason about performance. Because we focused on parallelizing the graph traversal, we also did not put any effort into investigating the consequences of our influence measure. For example, it would be interesting to see how the results correlate with other influence measures, e.g. a ranking by degree or PageRank.

Finally, we were surprised by how relevant the prefix sum operation seems to be. It is unclear when it is worthwhile/possible to replace atomic operations with prefix sums. For example, one pattern that we recognized in the Seibert dataset was the case where a twitterer had been retweeted by many of their followers, which is expected. However, this could lead to bottlenecks during parallel execution when using atomic operations.

The graph in Figure 8.1 illustrates the situation. At some point during execution, all nodes $a,\ b, c, ..., i$ would be on the frontier, except for e. Using atomics, only one thread can update the influence and decrement the indegree of e at the time. With larger graphs, this could be a bottleneck. It would be interesting to replace atomic operations with an efficient implementation using prefix sums.

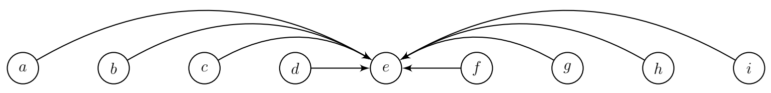


Figure 8.1: Node e was retweeted by many followers

Bibliography

[1] Karinthy Frigyyes. *Chains*. 1929.

[2] Michael Flynn. “Very High-Speed Computing Systems”. In: *Proceedings of the IEEE* 54 (Jan. 1967), pp. 1901–1909. DOI: 10.1109/PROC.1966.5273.

[3] Ira Pohl. “Bi-directional and heuristic search in path problems”. In: 1969.

[4] Jeffrey Travers and Stanley Milgram. *An Experimental Study of the Small World Problem*. 1969. URL: https://snap.stanford.edu/class/cs224w-readings/travers69smallworld.pdf.

[5] Linton C. Freeman. “Centrality in social networks conceptual clarification”. In: *Social Networks* 1.3 (1978), pp. 215–239. ISSN: 0378-8733. DOI: https://doi.org/10.1016/0378-8733(78)90021-7. URL: http://www.sciencedirect.com/science/article/pii/0378873378900217.

[6] Guy E. Blelloch. *Prefix Sums and Their Applications*. 1990.

[7] D.J. Watts and S.H. Strogatz. “Collective dynamics of ’small-world’ networks”. In: *Nature* 393 (1998), pp. 440–442.

[8] Lawrence Page et al. “The PageRank Citation Ranking: Bringing Order to the Web.” In: 1999-66 (1999). Previous number = SIDL-WP-1999-0120. URL: http://ilpubs.stanford.edu:8090/422/.

[9] Evelien Otte and Ronald Rousseau. “Social Network Analysis: A Powerful Strategy, also for the Information Sciences”. In: *Journal of Information Science* 28 (Dec. 2002), pp. 441–453. DOI: 10.1177/016555150202800601.

[10] Peter Dodds, Roby Muhamad, and Duncan Watts. “An Experimental Study of Search in Global Social Networks”. In: *Science (New York, N.Y.)* 301 (Sept. 2003), pp. 827–9. DOI: 10.1126/science.1081058.

[11] Ulrik Brandes. “A Faster Algorithm for Betweenness Centrality”. In: *The Journal of Mathematical Sociology* 25 (Mar. 2004). DOI: 10.1080/0022250X.2001.9990249.

[12] Linton Freeman. “The Development of Social Network Analysis”. In: (Jan. 2004).

[13] Stephen Borgatti. “Centrality and Network Flow”. In: *Social Networks* 27 (Jan. 2005), pp. 55–71. DOI: 10.1016/j.socnet.2004.11.008.

[14] Pedro Domingos. “Mining social networks for viral marketing”. In: *Journal of Retailing and Consumer Services* 20 (Jan. 2005). DOI: 10.1016/j.jretconser.2007.02.005.

[15] Business Wire. In: (2005). URL: https://www.businesswire.com/news/home/20050420006168/en/AMD-Announces-Worlds-64-Bit-X86-Multi-Core-Processors.

[16] A. Yoo et al. "A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L". In: vol. 2005. Dec. 2005, pp. 25–25. ISBN: 1-59593-061-2. DOI: 10.1109/SC.2005.4.

[17] Jeff Parkhurst, John Darringer, and Bill Grundmann. "From Single Core to Multi-Core: Preparing for a New Exponential". In: *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design.* ICCAD '06. San Jose, California: Association for Computing Machinery, 2006, pp. 67–72. ISBN: 1595933891. DOI: 10.1145/1233501.1233516. URL: https://doi.org/10.1145/1233501.1233516.

[18] R.M. Ramanathan. *Intel® Multi-Core Processors: Making the Move to Quad-Core and Beyond.* 2006. URL: http://web.cse.ohio-state.edu/~panda.2/775/slides/intel_quad_core_06.pdf.

[19] Pawan Harish and P. J. Narayanan. "Accelerating Large Graph Algorithms on the GPU Using CUDA". In: *HiPC.* 2007.

[20] Greg Humphreys. *How GPUs Work.* 2007. URL: https://research.nvidia.com/sites/default/files/pubs/2007-02_How-GPUs-Work/04085637.pdf.

[21] Akshay Java et al. "Why we Twitter: Understanding microblogging usage and communities". In: *of the 9th WebKDD and 1st SNA* 43 (Jan. 2007), pp. 56–65. DOI: 10.1145/1348549.1348556.

[22] Jure Leskovec and Eric Horvitz. "Planetary-Scale Views on an Instant-Messaging Network". In: Mar. 2008, pp. 915–924. DOI: 10.1145/1367497.1367620.

[23] Steven S. Skiena. *The Algorithm Design Manual.* London: Springer, 2008. ISBN: 9781848000704 1848000707 9781848000698 1848000693. DOI: 10.1007/978-1-84800-070-4.

[24] Leavitt et al. *The Influentials, New Approaches for Analyzing Influence on Twitter.* 2009. URL: http://www.webecologyproject.org/wp-content/uploads/2009/09/influence-report-final.pdf.

[25] Markus Billeter, Ola Olsson, and Ulf Assarsson. "Efficient Stream Compaction on Wide SIMD Many-Core Architectures". In: *Proceedings of the Conference on High Performance Graphics 2009.* HPG '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 159–166. ISBN: 9781605586038. DOI: 10.1145/1572769.1572795. URL: https://doi.org/10.1145/1572769.1572795.

[26] Yangdong (Steve) Deng, Bo David Wang, and Shuai Mu. "Taming Irregular EDA Applications on GPUs". In: *Proceedings of the 2009 International Conference on Computer-Aided Design.* ICCAD '09. San Jose, California: Association for Computing Machinery, 2009, pp. 539–546. ISBN: 9781605588001. DOI: 10.1145/1687399.1687501. URL: https://doi.org/10.1145/1687399.1687501.

[27] Schneider M. *Twitter + Stimulus = Conservative Stupidity.* 2009. URL: https://www.dailykos.com/stories/2009/3/26/713407/-Twitter-+-StimulusConservative-Stupidity.

[28] EVGENY MOROZOV. *Swine Flu: Twitter's Power To Misinform.* 2009. URL: https://www.npr.org/templates/story/story.php?storyId=103562240&t=1592433219167.

[29] Virat Agarwal et al. "Scalable Graph Exploration on Multicore Processors". In: Nov. 2010, pp. 1–11. DOI: 10.1109/SC.2010.46.

[30] M. Cha et al. "Measuring user influence in twitter: The million follower fallacy". In: *4th International AAAI Conference on Weblogs and Social Media (ICWSM)*. 2010. URL: http://scholar.google.de/scholar.bib?q=info:rqhbqWEH79kJ:scholar.google.com/&output=citation&hl=de&as_sdt=0&ct=citation&cd=10.

[31] P. R. Chamberlain. "Twitter as a Vector for Disinformation". In: *Journal of Information Warfare* 9.1 (2010), pp. 11–17. ISSN: 14453312, 14453347. URL: https://www.jstor.org/stable/26480487.

[32] Cynthia Chew and Gunther Eysenbach. "Pandemics in the Age of Twitter: Content Analysis of Tweets During the 2009 H1N1 Outbreak". In: *PloS one* 5 (Nov. 2010), e14118. DOI: 10.1371/journal.pone.0014118.

[33] Haewoon Kwak et al. "What Is Twitter, a Social Network or a News Media?" In: vol. 19. Jan. 2010. DOI: 10.1145/1772690.1772751.

[34] Lijuan Luo, Martin Wong, and Wen-mei Hwu. "An Effective GPU Implementation of Breadth-First Search". In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California: Association for Computing Machinery, 2010, pp. 52–55. ISBN: 9781450300025. DOI: 10.1145/1837274.1837289. URL: https://doi.org/10.1145/1837274.1837289.

[35] Js Weng et al. "Twitterrank: Finding Topic-Sensitive Influential Twitterers". In: Jan. 2010, pp. 261–270. DOI: 10.1145/1718487.1718520.

[36] Zhang et al. "IDENTIFYING INFLUENTIAL USERS OF MICRO-BLOGGING SERVICES: A DYNAMIC ACTION-BASED NETWORK APPROACH". In: 2011. URL: https://eprints.qut.edu.au/79289/15/pacis2011_submission_543.pdf.

[37] Eytan Bakshy et al. "Everyone's an Influencer: Quantifying Influence on Twitter". In: Jan. 2011, pp. 65–74. DOI: 10.1145/1935826.1935845.

[38] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. "Efficient Parallel Graph Exploration on Multi-Core CPU and GPU". In: Oct. 2011, pp. 78–88. DOI: 10.1109/PACT.2011.14.

[39] Sungpack Hong et al. "Accelerating CUDA Graph Algorithms at Maximum Warp". In: *SIGPLAN Not.* 46.8 (Feb. 2011), pp. 267–276. ISSN: 0362-1340. DOI: 10.1145/2038037.1941590. URL: https://doi.org/10.1145/2038037.1941590.

[40] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[41] Shaomei Wu et al. "Who Says What to Whom on Twitter". In: Jan. 2011, pp. 705–714. DOI: 10.1145/1963405.1963504.

[42] Scott Beamer, Krste Asanović, and David Patterson. "Direction-Optimizing Breadth-First Search". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012. ISBN: 9781467308045.

[43] Duane Merrill, Michael Garland, and Andrew Grimshaw. "Scalable GPU Graph Traversal". In: *SIGPLAN Not.* 47.8 (Feb. 2012), pp. 117–128. ISSN: 0362-1340. DOI: 10.1145/2370036.2145832. URL: https://doi.org/10.1145/2370036.2145832.

[44] World Economic Forum. *Digital Wildfires in a Hyperconnected World, Global Risks Report*. 2013. URL: http://reports.weforum.org/global-risks-

2013/risk-case-1/digital-wildfires-in-a-hyperconnected-world/#view/fn-4.

[45] William Stallings. *Computer Organization and Architecture, Designing for performance, Ninth Edition, International Edition.* Pearson, 2013.

[46] Andy Adinets. *CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics.* 2014. URL: https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/.

[47] Fang Jin et al. "Misinformation Propagation in the Age of Twitter". In: *Computer* 47 (Dec. 2014), pp. 90–94. DOI: 10.1109/MC.2014.361.

[48] Ty McKercher John Cheng Max Grossman. *Professional CUDA C Programming.* Wrox, 2014.

[49] Marc S. Orr et al. "Fine-Grain Task Aggregation and Coordination on GPUs". In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 181–192. ISSN: 0163-5964. DOI: 10.1145/2678373.2665701. URL: https://doi.org/10.1145/2678373.2665701.

[50] Jim Edwards. *Twitter just made it harder to figure out how many inactive users it has.* 2015. URL: https://www.businessinsider.com/twitter-just-made-it-harder-to-figure-out-how-many-inactive-users-it-has-2015-5?r=US&IR=T.

[51] Hang Liu and H. Howie Huang. "Enterprise: Breadth-First Graph Traversal on GPUs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450337236. DOI: 10.1145/2807591.2807594. URL: https://doi.org/10.1145/2807591.2807594.

[52] Peter Zhang et al. "Dynamic parallelism for simple and efficient GPU graph algorithms". In: Nov. 2015, pp. 1–4. DOI: 10.1145/2833179.2833189.

[53] Tokyo Institute of Technology Matthias Springer. "Breadth-first Search in CUDA". In: (2017). URL: https://m-sp.org/downloads/titech_bfs_cuda.pdf.

[54] Yeimer Ortiz-Martínez and Luisa Jiménez-Arcia. "Yellow fever outbreaks and Twitter: Rumors and misinformation". In: *American journal of infection control* (Mar. 2017). DOI: 10.1016/j.ajic.2017.02.027.

[55] Shazia Tabassum et al. "Social network analysis: An overview". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8 (Apr. 2018), e1256. DOI: 10.1002/widm.1256.

[56] Soroush Vosoughi, Deb Roy, and Sinan Aral. "The spread of true and false news online". In: *Science* 359.6380 (2018), pp. 1146–1151. ISSN: 0036-8075. DOI: 10.1126/science.aap9559. eprint: https://science.sciencemag.org/content/359/6380/1146.full.pdf. URL: https://science.sciencemag.org/content/359/6380/1146.

[57] Alexandre Bovet and HernĂ¡n Makse. "Influence of fake news in Twitter during the 2016 US presidential election". In: *Nature Communications* 10 (Jan. 2019). DOI: 10.1038/s41467-018-07761-2.

[58] Twitter Investor Relations. *Q1 2019 Earnings Report.* 2019. URL: https://s22.q4cdn.com/826641620/files/doc_financials/2019/q1/Q1-2019-Slide-Presentation.pdf.

[59] Felix Richter. "Twitter User Growth". In: (2019). URL: https://www.statista.com/chart/10460/twitter-user-growth/.

[60] Omnicore Agency. *Twitter by the Numbers: Stats, Demographics and Fun Facts.* 2020. URL: https://www.omnicoreagency.com/twitter-statistics/.
[61] Wasim Ahmed et al. *COVID-19 and the 5G Conspiracy Theory: Social Network Analysis of Twitter Data.* Apr. 2020.
[62] Revive Digital. *most popular social media networks (updated for 2020).* 2020. URL: https://revive.digital/blog/most-popular-social-media/.
[63] Shona Ghosh. *Twitter is adding fact-checking labels on tweets that link 5G with the coronavirus.* 2020. URL: https://www.businessinsider.com/twitter-factchecks-tweets-5g-coronavirus-2020-6?r=US&IR=T.
[64] Shona Gosh. *Facebook blocked 5G conspiracy groups with thousands of members after users celebrated the destruction of phone masts.* 2020. URL: https://www.businessinsider.com/facebook-blocks-anti-5g-groups-2020-4?r=US&IR=T.
[65] The Guardian. *Fake coronavirus tweets spread as other sites take harder stance.* 2020. URL: https://www.theguardian.com/world/2020/mar/04/fake-coronavirus-tweets-spread-as-other-sites-take-harder-stance.
[66] Simon Kemp. *Digital 2020 Global Overview Report.* 2020. URL: https://datareportal.com/reports/digital-2020-global-digital-overview.
[67] Johannes Langguth. *personal communication.* 2020. URL: https://www.simula.no/research/projects/umod-understanding-and-monitoring-digital-wildfires.
[68] Alfred Lua. *21 Top Social Media Sites to Consider for Your Brand.* 2020. URL: https://buffer.com/library/social-media-sites/.
[69] Tom Warren. *British 5G towers are being set on fire because of coronavirus conspiracy theories.* 2020. URL: https://www.theverge.com/2020/4/4/21207927/5g-towers-burning-uk-coronavirus-conspiracy-theory-link.
[70] BBC. *2020.* Coronavirus: Twitter bans incitement to attack 5G towers. URL: https://www.bbc.com/news/technology-52395158.
[71] Greg Bowman. In: (). URL: https://foldingathome.org/2020/03/15/coronavirus-what-were-doing-and-how-you-can-help-in-simple-terms/.
[72] DeepMind. In: (). URL: https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go.
[73] Edgar Gilbert. *Random Graphs.* URL: https://projecteuclid.org/download/pdf_1/euclid.aoms/1177706098.
[74] Khronos Group. *OpenCL Overview.* URL: https://www.khronos.org/opencl/.
[75] Geoff Koch. In: (). URL: https://software.intel.com/en-us/articles/multi-core-introduction.
[76] Nvidia. URL: https://docs.nvidia.com/cuda/thrust/index.html.
[77] Nvidia. *CUDA C++ Programming Guide.* URL: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf.
[78] Charlie Osborne. In: (). URL: https://www.zdnet.com/article/cryptocurrency-miners-bought-3-million-gpus-in-2017/.
[79] NVIDIA Research. *CUB Documentation.* URL: https://nvlabs.github.io/cub/.

[80] The Tesla team. "All Tesla Cars BEing Produced Now Have Full Self-Driving Hardware". In: (). URL: https://www.tesla.com/no_NO/blog/all-tesla-cars-being-produced-now-have-full-self-driving-hardware.

[81] TUDatasets. *File Format*. URL: https://chrsmrrs.github.io/datasets/docs/format/.

[82] Twitter. URL: https://www.theguardian.com/technology/2014/nov/19/new-twitter-search-makes-every-public-tweet-since-2006-findable.

[83] Twitter. URL: https://twitter.com/carterjwm/status/849813577770778624.

[84] Twitter. URL: https://twitter.com/TheEllenShow/status/440322224407314432.

[85] Twitter. URL: https://twitter.com/yousuck2020/status/1081544630754103296.

[86] Twitter. *How To Use Hashtags*. URL: https://help.twitter.com/en/using-twitter/how-to-use-hashtags.

[87] Xufei Wang et al. *Identifying Information Spreaders in Twitter Follower Networks*.

[88] Pete Warden. *The March of Twitter: Analysis of How and Where Twitter Spread*. URL: https://blog.hubspot.com/blog/tabid/6307/bid/6505/the-march-of-twitter-analysis-of-how-and-where-twitter-spread.aspx.