Measures Computation and Pattern Detection in Social Networks with GPU Acceleration

Gonçalo Nuno da Costa Teixeira de Sousa

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Alexandre Paulo Lourenço Francisco
Prof. Luís Jorge Brás Monteiro Guerra e Silva

Examination Committee
Chairperson: Prof. João António Madeiras Pereira
Supervisor: Prof. Alexandre Paulo Lourenço Francisco
Member of the Committee: Prof. Vasco Miguel Gomes Nunes Manquinho

November 2015
Acknowledgments

To family and friends, for always being there
Abstract

Straightforward application of known graph measures to social network graphs can produce high execution times due to the extremely large size of their graphs. One solution to reduce the computation times to take advantage of parallel systems to implement these measures.

GPUs have a complex hardware that is capable of running thousands of concurrent threads, but the GPU memory is limited. Understanding the architecture of a GPU, the thread and memory organization is necessary to develop efficient GPU kernels. CUDA is a framework to work with CUDA GPUs, it greatly simplifies performing both memory operations and kernel invocations across multiple threads. These operations can be done using the provided CUDA extension to the C/C++ programming languages.

This work discusses the implementation of two node centrality measures, the PageRank and betweenness centrality, on a CUDA platform. It also studies the possibility of applying these measures on large graphs, inside a GPU.

The implementation of these measures consider possible optimizations on the CUDA kernels, that promote balanced threads workloads, a low number of idle threads, as well the full usage of all GPU resources. Graph partition schemes and CUDA Streams are used to allow for concurrent kernel executions and memory transfer operations from host.

The PageRank results prove that it is possible to process large graphs on a GPU, using graph partitioning and CUDA Streams. Another interesting result was found by caching the rank transfer for each node, in each iteration. In the PageRank $9 \times$ speedups where achieved for a graph with $7.0\text{GB}$ of size.

The betweenness centrality had less positive results. Not only no partition scheme was found that would allow graphs larger than the GPU memory, but it also noted that to make full usage of the GPU resources, the necessary auxiliary data structure severely limit the graph sizes, for this measure. The betweenness centrality GPU implementation achieved speedup is similar to a multi core system.

Keywords: graphs, GPU, CUDA, PageRank, Betweenness centrality
Aplicar medidas conhecidas a grafos de redes sociais tem temos de execução altos devido ao grande tamanho do grafo. Uma solução para reduzir os tempos de execução é usar sistemas paralelos na implementação destas medidas.

As GPUs têm um hardware complexo, capaz de executar milhares de threads concorrentes, no entanto a memória disponível na GPU é limitada. Entender a arquitetura da GPU, e a organização das threads e memória é necessário para desenvolver kernels para a GPU eficientes. CUDA é uma framework para trabalhar com GPUs CUDA, que simplifica bastante a realização de operações de memória e invocações das kernels em várias threads. Estas operações podem ser feitas usando as extensões para as linguagens de programação C/C++.

Este trabalho discute a implementação de duas medidas de centralidade de nós, o PageRank e a betweenness centrality, numa plataforma CUDA. Também estuda a possibilidade de aplicar as estas métricas a grafos grandes, dentro da GPU.

A implementação destas medidas considera possíveis optimizações às kernels CUDA, que promovam distribuição de trabalho pelas threads, um baixo número de threads paradas, bem como a total utilização dos recursos da GPU. Esquemas de partição de grafos e Streams CUDA são usados para permitir concorrentemente a execução de kernels e operações de transferência de memória a partir do sistema anfitrião.

Os resultados do PageRank provam que é possível processar grafos grandes numa GPU, usando para isso particionamento do grafo e Streams CUDA. Um outro resultado interessante foi encontrado ao fazer caching da transferência do rank de cada nó, em cada iteração. No PageRank foi atingindo um speedup de \(9 \times\) para grafos de 7,0 GB de tamanho.

Para a betweenness centrality não foi encontrado um esquema de particionamento que permitisse grafos maiores que a memória da GPU, e também foi notado que para utilizar totalmente os recursos da GPU, as estruturas de dados auxiliares necessárias limitam severamente o tamanho dos grafos, para esta medida. A implementação da betweenness centrality na GPU obteve um speedup equivalente ao de um sistema multi-núcleo.

**Keywords:** grafos, GPU, CUDA, PageRank, Betweenness centrality
Contents

List of Tables 1

List of Figures 3

1 Introduction 5
   1.1 Objectives .................................................. 6
   1.2 Document Structure ........................................ 6

2 Background and related work 9
   2.1 Graphs ...................................................... 9
       2.1.1 Shortest Paths ........................................ 10
       2.1.2 Breadth First Search - BFS .......................... 11
       2.1.3 Floyd-Warshall ........................................ 11
       2.1.4 Dijkstra ................................................. 12
   2.2 Graph storage ............................................. 12
       2.2.1 Coordinate Format (COO) ............................... 13
       2.2.2 Compressed Sparse Row, CSR .......................... 14
       2.2.3 Ellpack-Itpack Format, ELL ............................ 14
       2.2.4 Hybrid Formats, HYB ................................... 14
       2.2.5 Compressed sparse web graph, CSWG ................ 14
   2.3 Parallel programming in the CUDA platform ............ 15
       2.3.1 GPU architecture ......................................... 16
       2.3.2 CUDA Programming ....................................... 17
       2.3.3 Thread Management ....................................... 18
       2.3.4 GPU Memory ............................................. 19
       2.3.5 CUDA Streams ........................................... 20
   2.4 Existing graph frameworks ................................. 21
   2.5 Social Networks ............................................ 22
       2.5.1 Social network graphs .................................. 22
   2.6 Network Measures .......................................... 22
       2.6.1 Centrality of a vertex .................................. 23
       PageRank .................................................... 24
       Betweenness Centrality ..................................... 25
   2.7 Conclusion of the background ............................. 28

3 Parallel GPU implementation 29
   3.1 PageRank .................................................... 29
       3.1.1 Sequential implementation .............................. 29
3.1.2 Parallel SpMV

- Implementing from SNAP
- Removing atomic operations by transposing the graph matrix
- SpMV without atomic operations
- SpMV kernels for different node sizes
- Precomputing the rank contribution (rank cache)
- Optimizing the SpMV kernels
- Supporting graphs larger than the GPU memory
- Concurrent copies to GPU memory and computations

3.1.3 Parallel Reduce

3.1.4 Computing the rank difference

3.1.5 Data structures

3.1.6 PageRank with graphs larger than GPU memory

3.2 Betweenness Centrality

3.2.1 Parallelizing the Brandes’ algorithm
- Shortest Paths
- Dependency accumulation

3.2.2 Data structures

3.2.3 Parallel source node processing

3.3 Conclusion

4 Experimental Results

4.1 Test Environment

4.2 PageRank

- Partitioning graphs
- Node Classes
- Caching the rank values
- CUDA Streams

4.3 Betweenness Centrality

4.4 Summary

5 Conclusions

5.1 PageRank

5.2 Betweenness Centrality

5.3 Future work

Bibliography
List of Tables

4.1 Used datasets and respective sizes ........................................ 46
4.2 LiveJournal PageRank execution (wall) time with different rank parameters ............ 46
4.3 Friendster PageRank execution (wall) time with different rank parameters ................. 47
4.4 Execution (wall) time with different partitioning parameters ............................... 48
4.5 Execution (wall) time of the rank cache operation ............................................. 49
4.6 CPU vs CUDA PageRank execution (wall) time ............................................... 49
4.7 Betweenness centrality execution (wall) time in CUDA .................................... 50
4.8 LiveJournal betweenness centrality execution (wall) time, the last value is estimated .... 53
List of Figures

2.1 An example directed unweighted graph ............................................. 10
2.2 Shortest pair from node 0 to node 5 ................................................. 10
2.3 Example adjacency matrix and respective graph .................................. 13
2.4 GPU Architecture of a “Maxwell” card .............................................. 17
2.5 CUDA SMM Architecture ................................................................. 18
2.6 CUDA Thread Hierarchy ................................................................. 19
2.7 Parallel operations using CUDA Streams ............................................. 21
3.1 Example graph and respective adjacency matrix ................................... 31
3.2 Example graph in CSR format .......................................................... 31
3.3 Transposed example graph adjacency matrix and respective CSR format .... 32
3.4 Example graph divided in two node classes ......................................... 33
3.5 Shuffle down operation ................................................................. 35
3.6 CUDA block reduce ................................................................. 37
3.7 Global reduce operation ................................................................. 38
4.1 PageRank speedup increase with rank cache ...................................... 49
4.2 A PageRank iteration of soc-livejournal using streams ......................... 51
4.3 A PageRank iteration of com-friendster using streams ......................... 52
4.4 Betweenness centrality speedup comparison ...................................... 53
Chapter 1

Introduction

As online social networks have grown in size over the last decade, current networks reach the tens of millions of users. In these networks each user has, on average, hundreds of connections to other users. In order to keep users engaged, companies behind these networks use several metrics to customize the content served to each user. Graphs are best suited to represent this networks, as each user can be represented by a graph node and the connections between users are mapped to the graph edges. Processing these large graphs is necessary to better understand social interactions between users.

The computation of network related measures requires complex operations. Some measures are performed over all network elements to detect patterns and identify the most important and/or relevant nodes in the graph. Often, the required computations are so complex that given the huge size of the network, they take days if not even months to complete [1]. Reducing the computation times of network measures enables a better understanding of the evolution of the network topology and dynamics. One way to reduce the execution time is to use faster hardware, like faster processors or lower latency storage. Another possible solution is to take advantage of highly parallel systems when implementing these algorithms.

Parallel systems, can be roughly divided into: clusters, multiprocessors and multicore systems. In clusters data is processed throughout several linked machines. This type of systems have the advantage that each machine has full control over its execution and is not directly limited by the other machines. In practice these systems work best when the data is well partitioned across the machines and suffer if there are many synchronization points, due to the high cost of inter-machine communication.

While multicore parallel systems have a lower synchronization cost, they only allow for a low degree of parallelization with at most a dozen cores on a single processor. Multiprocessor systems are not only multiple processors on running at the same time in the same board. Systems like the Xeon Phi coprocessors or the GPGPUs from NVIDIA and AMD, that have hundreds of independent cores are also considered multiprocessor systems.

The first has the advantage of running on the x86 architecture with only a few extra instructions over a dozen of cores. GPUs have a special instruction set, to run and control the thousands of parallel cores. In a GPU threads runs the same instructions simultaneously but with different data.

The use of specialized hardware with hundreds of parallel cores enables very fast computations. These systems are usually limited by their own internal memory, as these systems do not have direct access to the host's memory. GPU's only have a couple of GBs in the device memory, where the host memory can be in the orders of hundreds of GB's. A solution to work with graphs whose size is larger
than the GPU memory is to divide it in smaller partitions. These division are sized so that can they fit inside the GPU memory. The division scheme might also be applied to auxiliary data structures for each measure if necessary.

1.1 Objectives

This work will be focused on implementing an efficient solution capable of computing measures for large social networks using a single CUDA GPU card. The implementation will take into consideration the latest features available in the CUDA API, such as the CUDA Streams to allow memory operations concurrent to kernel execution, in the GPU.

In order to demonstrate the advantage of using the highly parallel architecture of a GPU to process graphs, two different measures where chosen. The choice was based on the procedures necessary for each measure, and how they process the graph. The two chosen measures are the PageRank and the betweenness centrality. The PageRank implementation uses an iterative process to obtain an approximation of the rank value, by accessing the rank of each node neighbors. The betweenness centrality, on the other side, uses the number of paths that pass through that node of all the shortest paths between every other two nodes, to compute each node centrality score. These two measures have distinct access patterns to the graph, creating different opportunities to explore the limits of the GPU.

To achieve optimum performance on a CUDA application it is necessary to assign work to all available cores. When processing partitioned graphs, copying a graph partition to the GPU can cause CUDA threads to wait until the transfer is done. It is necessary to understand the memory access patterns of these algorithms, in order to always keep data available on the GPU.

The implementation of these algorithms will reiterate the feasibility of using the GPGPU to process graphs, while aiming to reduce the computation time of the larger graphs in a GPGPU.

1.2 Document Structure

The rest of this work is organized as follows:

Chapter 2 describes graphs, common graphs operations and storage model. Including storage formats specifically tailored for sparse graphs. It also contains a general introduction to the GPGPU and CUDA architecture and programming model. How CUDA threads and memory are organized and how to use CUDA Streams to perform multiple operations in the GPU at the same time. This chapter includes a description of both the PageRank and betweenness centrality algorithms, and how previous works have implemented these measures on a GPU architecture.

Chapter 3 contains a detailed implementation for both measures. It describes the parallelization of each of the PageRank steps, with emphasis on the best partition models for large graphs and improving SpMV performance by caching the rank contribution of each node. It also details the use of CUDA Streams to perform data transfers to the GPU simultaneous with running CUDA kernels. For the parallel implementation of the betweenness centrality it discusses how to compute the shortest paths and dependency accumulation steps. It will also show how several nodes can be processed in parallel to fully occupy all the GPU’s cores. It also points what factors could limit an implementation of the betweenness centrality on the GPU.
Chapter 4 evaluates the performance of a parallel implementation of both measures comparing them to a sequential implementation. For the PageRank it discusses on how the partition scheme affects the GPU performance and the impact of performing several data transfers from the host to the GPU. In particular if these transfers cause any delay on the kernel invocations or if the transfers finish faster than the GPU kernel. It is also shown that introducing the rank cache yields a significant speedup on all tested graphs. Regarding the betweenness centrality it explains the reason for the betweenness centrality achieved speedup and limits to the graph size.

Chapter 5 describes the work that was done and its conclusions, results achieved and how further development can improve the performance of the studied measures in a GPU.
Chapter 2

Background and related work

This chapter will present the concepts necessary to understand the basis of this work, the problems that motivated this work and also the proposed solutions. The main points will be common graph measures and storage methods, the CUDA architecture and the computation of measures in the context of social network. The graph measures will include the computation of the shortest paths and the centrality of a vertex. Both common graph storage methods and compressed formats will be described. The most relevant parts of the CUDA architecture and programming API will be referenced. Lastly there will a description of the PageRank and betweenness centrality measures in the context of processing social graphs in a GPGPU.

2.1 Graphs

A graph $G = (V, E)$ is a representation of connected objects, where $V$ is the set of vertices, the objects, and $E$ the set of edges. An edge is defined as a connection between any two vertices. In the context of this work $n$ will refer to the number of nodes in a graph, or $|V|$ and $m$ will denote the number of edges in a graph, or $|E|$.

In directed graphs, edges are replaced by directed edges that connect one vertex to another, these are respectively the source and the destination vertices. Edges can have an associated cost, or weight, that should be considered when traversing the edge. A graph whose edges have a weight is called a weighted graph.

Graphs can also be classified as dense or sparse, according to the number of edges. In a dense graph the number of edges is close to the maximum possible number of edges, while in sparse graphs the number of edges is far lower that the maximum possible number of edges.

A graph can be either vertex labeled or edge labeled, depending to which set the data is mapped to. For example a social graph is vertex labeled, where individuals are mapped to a node, and edges represent the connections between them. On the other side a street map is edge labeled graph, each edge takes the name of the street that his mapped to it, and the intersections are the nodes. In this work only vertex labeled graphs will be used.

Another important concept in graph theory is the node neighbor. A neighbor of a node is any node that is connect to it by an edge. In the case of directed graphs it is possible to distinguish incoming neighbors where the connecting edge has the node as its destination from outgoing neighbors where
the connecting edge has the node as its destination. The node degree is the number of edges of a node, there is also the concepts of in-degree and out-degree, for respectively incoming and outgoing edges.

2.1.1 Shortest Paths

A path in a graph is a sequence of edges that connect a sequence of vertices, where all vertices are distinct. In the case that the first and last vertices are the same it is considered a walk. On weighted graphs the weight of the path is the sum of the weight of all traversed edges, for unweighted graphs the path has a length equal to the number of traversed edges (for simplicity it is usually considered that on unweighted graphs every edge has a weight of one).

Many of the measures that analyze the topology of a network make use of the shortest path between network nodes. One in particular is the betweenness centrality, that will be used in the proposed solution. The shortest path is a special path that represents the minimum distance between two vertices. In particular the shortest path is the path between any two vertices \( s \) and \( t \) that has the lowest weight. This is the path that minimizes the sum of all the edge weights. The shortest paths algorithms can be classified into single source, single pair or all pairs, according to the type of the shortest paths returned by each algorithm. In this work the weight of an edge connecting \( v \) to \( u \) will be referred by \( w(v, u) \) and the cost of the shortest path, from \( s \) to \( t \), will be denoted by \( d(s, t) \).

There are many algorithms that solve the shortest path problem in different ways, some compute the shortest path for a single vertex pair as the A*. Dijkstra and BFS algorithms return the all shortest paths from a single source vertex, commonly referred as SSSP, other algorithms like the Floyd-Warshall compute the shortest path for all possible pairs at the same time.

The choice of algorithm also depends on graph characteristics, like it the graph is weighted or if the graph has edges with negatives weights. Although the Breadth First Search algorithm only works in unweighted graphs, it is the most efficient solution to compute the all SSSPs of a node, with a time complexity of \( O(n + m) \) and requiring \( O(n) \) space. For weighted graphs it is necessary to use either the Dijkstra algorithm that has a time complexity of \( O(m + n \log n) \), or the Bellman-Ford for graphs with negatives edge weights but runs with a complexity of \( O(nm) \).

While the next sections will contain a brief description of the relevant algorithms for this work, a more detailed description of these algorithms, related data structure, and other graph related algorithms can
be found in Skiena, 2008 [2].

2.1.2 Breadth First Search - BFS

The BFS is a search algorithm for unweighted graphs, it can also be used to discover the SSSP from a source vertex $s$. It uses an auxiliary queue $Q$ and vertices are marked as discovered as they are added to it. The queue is initialized with the source vertex $s$. In every iteration the top element of the queue is removed, and all his undiscovered neighbors are added to queue and set as discovered. The algorithm stops when the target element is removed from the queue.

To solve the SSSP problem, the BFS algorithm ends instead when the queue is empty, meaning that all reachable vertices from $s$ have been processed. In this case when adding a undiscovered vertex to the queue, the shortest path distance from $s$ to the new vertex is set. The value is the distance of $s$ to the vertex that is being processed, plus one. See Algorithm 1 for a possible implementation of this SSSP algorithm.

While Breadth First Search works both for directed and undirected graphs, because it does not take into consideration the edge weights it does not work with weighted graphs. The path with the fewest number of edges on a weighted graphs might not be the path with the lowest total cost.

**Algorithm 1** BFS Algorithm for SSSP

\[
\begin{align*}
d[v] & \leftarrow -1, v \in V \\
d[s] & \leftarrow 0 \\
Q.enqueue(s) \\
\text{while } Q \text{ is not empty do} \\
\quad v & \leftarrow Q.dequeue() \\
\quad \text{for all neighbor } u \text{ of } v \text{ do} \\
\quad \quad \text{if } d[u] == -1 \text{ then} \\
\quad \quad \quad Q.enqueue(u) \\
\quad \quad \quad d[u] = d[v] + 1 \\
\quad \quad \end{align*}
\]

2.1.3 Floyd-Warshall

The Floyd-Warshall algorithm is capable of computing the all pairs shortest paths on weighted graphs. This algorithm iteratively computes the all the shortest paths with $k$ intermediate vertices, where $k$ increases from 0 to $n$. This results on a matrix $n \times n$ with the weight of all the shortest paths between every pair of vertex.

The Floyd-Warshall algorithm is able handle negative edge weights, and runs with complexity of $O(n^3)$, on pair with Dijkstra algorithm when applied to all vertices as the source vertex. However, the matrix with all the path weights requires so much space can not fit inside the GPU memory, so this algorithm can not be used in this work.

Considering that social graphs do not contain negative edges, the Dijkstra algorithm should be appropriate to implement the betweenness centrality on weighted graphs.
2.1.4 Dijkstra

The Dijkstra is a graph search algorithm that can be used on weighted graphs, as long all edges (or vertices) have non-negative weights. This algorithm computes the shortest path from $s$ to every other vertex on the graph. In each iteration it computes the shortest path from $s$ to a new undiscovered vertex $x$ minimizing $d(s, v) + w(v, x)$, where $v$ is a previously visited vertex that still has edges to visit. This works because if the shortest path from $s$ to $t$ contains the vertex $x$, then it also contains the shortest path from $s$ to $x$. The algorithm is presented in Algorithm 2.

**Algorithm 2 Dijkstra’s Algorithm**

```plaintext
d[v] ← −1, v ∈ V
p[v] ← −1, v ∈ V
d[s] ← 0
Q.enqueue(s)
while Q is not empty do
  v ← Q.extract_min()
  mark v as visited
  for all neighbor u of v do
    if d[u] == −1 then
      Q.add(u, d[v] + w[v, u])
      d[u] = d[v] + w[v, u]
    end if
    if d[u] > d[v] + w[v, u] then
      d[u] = d[v] + w[v, u]
      p[u] = v
      Q.decrease_(u, d[u])
    end if
  end for
end while
```

This algorithm requires two data structures: the shortest path tree from $s$ to all other processed vertices, and a queue of vertices that are waiting to be processed, sorted by the lowest score so far. The shortest path tree can be replace by an array that stores only the predecessor vertex on the shortest path to the source $s$.

The complexity of this algorithm varies with the data structures used for the queue. Using a simple array the time complexity is $O(n^2)$, however using a Fibonacci heap [3] (a data structure similar to a binomial heap, but lazily postponing the heap construction until the next delete) it is possible to achieve $O(n + m \times \log(n))$.

2.2 Graph storage

A graph is usually stored in one of two ways: an adjacency matrix or adjacency list. Adjacency matrices, are useful to store dense graphs. These matrices have $n$ row and $n$ columns, where $n$ is the number of vertices in $V$ and an edge from $v$ to $u$ is represented in matrix cell $[v, u]$ with the value $w(v, u)$. For example the following matrix:

Adjacency lists are usually preferred to represent sparse graphs. For each $v$ in $V$ a list of outgoing edges from $v$ is maintained. Each entry of the list containing the destiny vertex $u$ and $w(v, u)$. More information about the graph structures and properties and can be found in [4]. The previous adjacency matrix $A$ could also be store as the following adjacency list:
While the adjacency lists do not waste space by reserving space for edges that do not exist. Not only the cost of checking if an edge exists can be $O(V)$ but there are other storage formats that have a small space requirement and at the same time provide fast access times. Plus they also increase the data locality, that is an important factor in GPU applications, as described in 2.3.4.

For following descriptions of alternative storage schemes, the graph represented in the adjacency matrix $A$ will be used for demonstration purposes. $nnz$ will refer to the number of nonzero elements in the matrix, this value corresponds to the number edges $m$ on the graph. Only a few formats will be described in detail, while other formats also exist and can be used in CUDA applications[5], they are not as relevant for this work.

### 2.2.1 Coordinate Format (COO)

A sparse matrix is stored in three arrays with $nnz$ length, thus the COO format has space $O(3m)$. The first, values, holds all values of the matrix in row-major format, where elements in the same row are stored in continuous memory positions. The second rowIndices contains the row indices of the corresponding elements of values. Lastly columnIndices that stores the column indices of the corresponding elements in values. In the case of unweighted graphs the values can be removed.

$$
\text{values} = \begin{bmatrix}
1 & 4 & 2 & 3 & 3 & 5 & 7 & 8 & 9 & 6
\end{bmatrix}
$$

$$
\text{rowIndices} = \begin{bmatrix}
0 & 0 & 1 & 1 & 2 & 3 & 3 & 3 & 4 & 4
\end{bmatrix}
$$

$$
\text{columnIndices} = \begin{bmatrix}
0 & 1 & 1 & 2 & 4 & 0 & 3 & 5 & 3 & 5
\end{bmatrix}
$$
2.2.2 Compressed Sparse Row, CSR

This format is similar to COO, but only uses $O(2m + n + 1)$. Instead of storing the `rowIndices`, it uses `rowStart` with $n + 1$ elements. This array stores index of the first nonzero element in each column. The last element of the array, `rowStart[n]` stores `nnz`. The only drawback of this storage scheme is that edges cannot be trivially traversed backwards as it is the case with the COO format. There is also an equivalent format for column major matrices, appropriately called `CSC`.

\[
\text{values} = \begin{bmatrix} 1 & 4 & 2 & 3 & 3 & 5 & 7 & 8 & 9 & 6 \end{bmatrix} \\
\text{columnIndices} = \begin{bmatrix} 0 & 1 & 1 & 2 & 4 & 0 & 3 & 5 & 3 & 5 \end{bmatrix} \\
\text{rowStart} = \begin{bmatrix} 0 & 2 & 4 & 5 & 8 & 10 \end{bmatrix}
\]

2.2.3 Ellpack-Itpack Format, ELL

This format uses two matrices of size $n$ and $k$, where $k$ is the maximum number of non zeros in a single row. The rows on these matrices have correspondence with the rows on the original matrix, this makes another array like the `rowStart` or `rowIndices` unnecessary. The first matrix `data` stores the values of the nonzero elements, while the second matrix `indices` contains the corresponding column indices. All rows that do not have $k$ vertices will be padded with 0 on the `data` matrix and padded with the $-1$ on the `indices` matrix. When the row sizes are not all similar, in particular if only some rows are large, this format will also suffer from using space in excess.

\[
data = \begin{bmatrix} 1 & 4 & 0 \\ 2 & 3 & 0 \\ 3 & 0 & 0 \\ 5 & 7 & 8 \\ 9 & 6 & 0 \end{bmatrix} \quad \text{indices} = \begin{bmatrix} 0 & 1 & -1 \\ 1 & 2 & -1 \\ 1 & -1 & -1 \\ 0 & 3 & 4 \\ 2 & 4 & -1 \end{bmatrix}
\]

2.2.4 Hybrid Formats, HYB

Not exactly a format but instead a composition of two or more formats, usually by partitioning the matrix in two or more. One common combination is ELL with COO or CSR. The ELL format stores the partition with the rows that have the similar sizes. The CSR partition stores the irregular rows, in particular those with many non zeros, that would otherwise force the ELL partition to use extra space.

2.2.5 Compressed sparse web graph, CSWG

Proposed by [6] as an improvement to CSR, specially for web graphs. The first two arrays `source` and `incoming` work the same way as in CSR. This format uses an extra arrays of size $n$, `outgoing` to store the number of outgoing edges on each vertex.
2.3 Parallel programming in the CUDA platform

A general rule when parallelizing any algorithm is that it is necessary to find pieces of code that can be executed independently from each other. An alternative is to find a block of code that is repeated so many times that executing those repetitions in parallel takes less time than executing them sequentially. Usually parallelizing a piece of code is not a simple operation. Sometimes it requires the use of auxiliary structures or some kind of data preprocessing to make it fit the parallel algorithm. This introduces an overhead on the parallel implementation. Considering that in most cases there are always some steps that cannot be parallelized, it is easy to conclude that parallelizing an application might not always yield benefits.

In light of this fact the speedup is introduced as a measure to compare the performance of a sequential application with the parallel implementation. To be more precise, the speedup is the ratio between the execution time of the parallel program, run on $N$ processors, and the sequential version of the same program. The ideal speedup value is equal to the number of parallel processors $N$. The reason is that to achieve this value it is necessary a fully parallel program without any kind of overheads. In some rare cases it is possible to achieve a superlinear speedup, meaning that the speedup is bigger than $N$.

The actual speed values usually are lower than $N$, due the existence of communication overheads between parallel processors, the initialization cost and the existence of inherently sequential code. However, a speedup value of 1 or lower means that there is a decrease in the performance resulting from parallelizing the original code.

Amdahl's law predicts the maximum speedup for a parallel problem executed in $N$ processors, taking into consideration both the parallel, $P$, and sequential $(1 − P)$ parts of the program:

$$S(N) = \frac{1}{(1 − P) + \frac{P}{N}}$$

A conclusion of the Amdahl's law is that the sequential part of the program will limit the speedup. As $N$ increases, the effect of the sequential code will reduce the performance per processor $P/N$.

Problems that can be easily parallelized and require no communication between tasks being executed simultaneously are called, embarrassingly parallel problems. The GPGPU architecture works better with this kind of problems as is composed by hundreds of parallel processors, that execute the same code on different data.

More information on parallel systems and performance metrics is available in Quinn, 2003 [7].
2.3.1 GPU architecture

A GPU is a SIMD (Single Instruction Multiple Data) system, where all threads execute the same instructions, but each thread is dealing with different information. This architecture is useful for image and video processing, as it applies the same steps to a large amount of pixels or vertices at the same time. Considering that the pixels are independent from each other it is easy to understand how hundreds of threads can collaborate to speed up the data processing.

The GPU architecture is optimized to perform highly parallel computations. Compared to a CPU, the GPU has less control flow operations and also less powerful caching mechanisms, focusing instead on a large number of simpler parallel cores, making it able to process a much larger amount of data in the same amount of time. This architecture was developed for graphic problems, like texture processing and 3D rendering, meaning that a GPU is appropriate to solve data parallel problems. Cryptography, computational biology and other areas have found the GPU capacity to execute large parallel tasks useful to improve the performance of their algorithms. This usage of GPUs as a general purpose computing unit led to the evolution of GPUs to provide a better support for these applications. Not only by creating cards specialized in High Performance Computing but also by releasing tools that reduce the work necessary to develop kernels that can run on a GPU.

There are specialized high-level APIs that allow developers to use compatible GPUs for general purpose processing. The two most widely used are CUDA [8] and OpenCL [9]. Both languages are based on the C/C++ languages, and the computational kernels can be used in and called from a C/C++ program. This makes easier for programmers that are familiar with the C/C++ languages to also understand parallel programs.

OpenCL an open standard that supports multiple platforms and architectures. The OpenCL is only a standard specification, where each vendor has to make his own implementation for usage with their products. The OpenCL support depends on the capabilities of each platform, some features may not be supported in some platforms. This heterogeneous ecosystem reduces the code portability between platforms. The CUDA API was developed as part of the CUDA platform and programming model. Because was developed alongside the NVIDIA GPU architecture, it is not available for other platforms. But as a result CUDA applications usually has better performance than the OpenCL implementation for NVIDIA GPUs.

For a CUDA developer is important to consider carefully two of the most important components of the GPU: the memory and the processing unit. The GPU memory is part of the GPU device, but is located off-chip and connected to the chip by a bus. It behaves just like the RAM present in the host system. It has a large capacity but also has a large latency when compared with the GPU cache’s latency and processing speed. The GPU chip contains all the parallel cores and control logic. Figure 2.4 details these GPU components in a CUDA card with compute capability 5.2.

The CUDA GPU is built around an array of Streaming Multiprocessors (SM). Each SM is independent from each other, this means that different kernels can be run at the same time in the GPU. A GPU kernel is a function run across multiple threads at the same time. This architecture also allows for the applications to scale with the available number of SMs. When a program invokes a GPU kernel, it is distributed through the available Streaming Multiprocessors. Each SM is assigned to up to 1024 threads (on CUDA 3.0), when no more SMs are available the remaining threads will be scheduled to run when one, or more, SMs are available again. Section 2.3.3 explains in more detail on how this distribution is done from the perspective of a developer.

Each SM contains 128 CUDA cores, these are actual cores that run the instructions. However, as
represented in figure 2.5 there are only a few control structures per SM. These, along the cache and registers have to be shared across all the cores. This results from the SIMD architecture, where most of the cores are executing the same instruction at the same time. As a result of not needing to control each core individually it is possible to increase the total number of cores per chip, leading to a greater parallelism.

2.3.2 CUDA Programming

As described in the previous section, the CUDA language was developed so it can be used as an extension to the C/C++ language, meaning that both CPU and GPU code can be mixed in the same application. CUDA kernels can be invoked directly from the CPU code, it is only necessary to specify the total number of concurrent threads that are going to run that specific kernel. CUDA kernels are defined just like regular C functions, with the difference that a __global__ declaration is used to specify a CUDA kernel. Information about the threads running the kernel or manipulation of shared memory are only possible inside the respective kernel context.

To develop a CUDA application and obtain the best performance, is necessary to first understand how the CUDA platform works. In particular the way the platform handles parallel threads and how the memory organization impacts the latency and computing times.
2.3.3 Thread Management

When invoking threads it is possible to organize them in up to three dimensions, this is useful when splitting vectors, matrices or volumes across the running threads. In a CUDA application threads are organized in blocks. Each block has a limit of 1024 threads. This limit is due to the fact that each block will be assigned to a single SM, and will be limited to the number of threads, registers and cache of that SM. Each SM can run multiple blocks at the same time, as long the total number of threads is under the SM's thread limit, 2048 in CUDA 3.0 and later. SM's are also able to handle block scheduling in order to maximize thread occupancy, for example when a block is waiting for a memory request to complete.

In turn blocks are organized into a grid, that can also have up to three dimensions. The maximum limit of blocks in a grid is $2^{31} - 1$, large enough that it is not a limitation for the most applications. This organization of thread, blocks and grids is illustrated in Figure 2.6. The actual number of blocks and threads per block depends on the application and the target system. It is important to keep each SM fully occupied but also to consider that each GPU has multiple SMs.

The execution of two blocks is independent, because of that it is not known whether they are run at the same time or if one will run before the other. This constraint enables the creating of applications that
are scalable as they can run in GPUs with a different number of SMs, without changing the code.

Threads inside the same block can share data through the shared memory (Section 2.3.4) and synchronize their execution to coordinate memory accesses. Synchronization points are specified by calling the `__syncthreads()` function, that creates a barrier that all threads much reach before any can proceed. This function has a very low impact on thread execution, but can improve the overall performance if used before accessing memory.

A thread block is further divided in warps, a group of at most 32 threads. Warps that are less than 32 threads will still require 32 threads for execution, with the extra threads marked as disable. If there is branching of the execution path inside the warp, all paths will be executed serially by the warp, disabling threads that are not in the path.

Each multiprocessor support several warps, and can switch between them whenever one is stalled, for example when is waiting for a memory access to complete. The total amount of warps and blocks inside a multiprocessor depend on the amount of registers and shared memory used by kernel and the availability of these resources on the multiprocessor.

### 2.3.4 GPU Memory

A GPU has a few different types of memory, global memory that is usually the largest memory on the device and is accessible by any thread, shared memory with a speed equivalent to a cache and is shared by the threads inside the same block, and local memory that is private to each thread. There is also two read-only memory types, constant memory and texture memory that, along with the global memory, are persisted from one kernel launch to another. Each of these memories have a specific usage pattern, and it is important to understand them to optimize CUDA programs.
• The global memory is optimized to be accessed in memory transactions of 32, 64 or 128-byte in size. These transactions have to be aligned to their size, meaning that the first address has to be a multiple of the size. When accessing global memory warps coalesce the accesses into one or more memory transactions, depending on the word size and distribution of memory accesses. Global memory throughput is achieved by coalescing memory accesses, through the usage of appropriate data types that meet the size and alignment requirements, or by padding two dimensional arrays such that memory accesses are aligned with the transaction size. In more recent devices, accesses to this memory are cached in a cache shared across all multiprocessors.

• The local memory is managed by the compiler, and used to store automatic variables, like large structures or arrays that uses too much register space, or any variable when the kernel uses more registers than available. This memory does not reside on-chip, so it has high latency and low bandwidth, just like the global memory. Local memory is organized such way that consecutive 32-bit words are accessed by consecutive threads, making these accesses fully coalesced for threads accessing the same relative address. Just like global memory, in more recent devices, accesses to this memory are cached in a cache shared across all multiprocessors.

• Shared memory is an on-chip memory, so it has higher bandwidth and lower latency than local or global memory. This memory is divided into banks that can be accessed in parallel, as long each request goes to a different bank. If two requests are made to the same bank these requests are serialized and thus the throughput decreases.

• Constant memory is a device memory that is optimized for the case where all the requests are made to the same memory address.

• Texture and surface memory are device memories, but they are cached on the chip. The cache is optimized for 2D spacial locality, this means that threads that read addresses that are close together in the 2D space will exhibit the best performance.

2.3.5 CUDA Streams

Commonly a CUDA program has three phases. Copy the data to the device, invoke the kernel on the GPU and finally copy the data to the host memory. While these three phases have to be run sequentially in relation to each other, it is possible to perform multiple GPU operations simultaneously. The CUDA architecture allows to run one kernel per Stream Multiprocessor, plus two memory operations, one from the host to the device and the other from the device to the host. It also allows for independent CPU computations as all operations are non blocking. This behavior is demonstrated on Figure 2.7 with six streams each copying data to the GPU, invoking three GPU kernels, and finally copying the results to the host, at the same time three CPU kernels are also being run. This is useful when running several CUDA kernels, each with its own data.

CUDA allows for concurrent operations by using CUDA Streams. A stream is defined as a sequence of operations that execute in issue-order on the GPU. Streams are independent from each other, meaning that multiple streams can run simultaneously and also that there is no guarantee on the operation order between streams. Even when not using explicit streams all CUDA operations are run on a default stream, that is mostly a synchronous stream with respect to the device and host. The exception being explicit asynchronous operations, that return the program control to the host before completing.

The usage of CUDA Streams allows for faster computation times by hiding the memory operation with kernel invocations, as well running different kernels at the same time in the GPU, as long as
there are enough SM available. Streams also allow for multiple types of synchronization, like global synchronization across all streams with `cudaDeviceSynchronize()`, a single stream synchronization with `cudaStreamSynchronize(stream)`, or using events to synchronize across multiple streams. When using streams it is important to consider times of both kernel invocations and memory operations, as well the invocation order of each operation.

### 2.4 Existing graph frameworks

The analysis of sparse networks is already a well studied problem. Libraries like Webgraph [10] have a high efficiency on single processor systems. Not only these frameworks are heavily optimized but they also have the benefit of having the entire graph accessible in local memory. This is possible by using heavy compression techniques and also access to larger local memory.

Some other frameworks are tailored for high performance multiple processors and distributed systems. For example the GraphX framework[11], from the Apache Spark project. It provides an API to seamlessly perform graph exploration and computations inside a machine cluster. The GraphX framework employs two alternative methods to implement parallelism. The first is a message passing interface, a super-step implementation with vertices sending messages to each other. Where in each step, every vertex computes the new value and sending a message to their neighbors. Synchronization is performed at the end of each step. The second is a Gather Apply Scatter, GAS, mechanism. The GAS accesses all the vertices in each iteration and in the end performs a synchronization operation across all nodes involved.

While the GraphX is focused on a different type of parallel system, it introduces interesting strategies for graph partitioning. For example using vertex-cuts to split the number of edges evenly among all the processors, thus reducing the communication overhead and minimizing the number of processors that perform work with each vertex.

A recent framework for processing graphs in CUDA GPUs is the MapGraph, formerly MPGraph, [12], which achieves a high performance due to the high memory bandwidth available on GPUs. The implementation of algorithms in MapGraph is done by following a template structure. It implements a design pattern similar to the Gather Apply Scatter API seen in GraphX, with data structures that can represent the graph on the GPU. The MapGraph is a rigid approach to the parallelization problem. As such it will fail to provide the best results in every problem when compared with specific solutions for
each different problem. It also raises concerns regarding the ability to deal with graphs that do not fit completely inside the GPU memory.

2.5 Social Networks

Social networks are easily represented by graphs, users are represented by the vertices and the connections between them are represented by the edges. While not always necessary these connections can be weighted accordingly to different parameters, like the proximity or activity between two users. Depending on the social network the graph can be directed or not, for example a LinkedIn graph is not directed, but the Twitter graph is. While you can follow a user on Twitter without him following you back, on LinkedIn the connection is only made when one user sends an invitation and recipient accepts it.

Modeling social networks through graphs enables the use of the known graph algorithms to obtain information about the topology and characteristics of the network, for example to determine the most relevant users of a network, or identify communities and influence groups inside a network. Depending on what information is going to be analyzed it is possible to use different techniques to map the network to a graph. For example by focusing on pages, photos, status updates or any other kind of interaction between two or more users. Although this type of analysis can be equally interesting, it falls outside the scope of this work as it would involve more complex data structure to support the extra information and would also require a different set of measures. This work is focused in implementing the computation of existing network measures in a GPU system with graphs that are larger than the GPU memory.

2.5.1 Social network graphs

A social network graph is a sparse graph. This means that the number of connections to each vertex is low compared to the total number of vertices. Each user only has a small number of connections when compared to the total number network users.

As stated in Section 2.2, storing this kind of graph in a matrix is impractical due to the average amount of memory required. Not to mention that it would be very inefficient, as most values in the matrix would be zero, and only the non-zero values are actually relevant for the network. An alternative would be to store the graph as an adjacency list. In this work the network graph will not change during the computation of the measures, thus it is only necessary to consider the access times to the graph nodes and edges. For this scenario formats like the Compressed Sparse Row, CSR, or the ELL-Pack, described in Section 2.2 are far more appropriate, as they have linear access times, and are also more convenient for some graph algorithms.

2.6 Network Measures

There are several properties relevant to network analysis [13] that theoretically should benefit from a significant speedup when implemented in a GPU. Those that involve repeating the same set of operations over different data with little or no special cases and dependencies between data points, are better suited to the GPU architecture.

There are several types of information that can be obtained from a social network. For example finding the set of connected nodes at a distance of at most \( k \) hops, usually employed to find the friends
of a friend. Or more complex measures like detecting clusters and cycles, using clustering coefficients, and others used for community identification.

While most of these properties can improve from a parallel system, they all work with the graph in different ways. Some are trivial to implement in a parallel system, while others create problems when dealing with partitioned graphs. For example in properties that iterate several times over the graph it is common to partition the graph across the available processors or cores. This way each core will only need to access a small part of the entire graph.

In a system like a CUDA, with thousands of parallel threads, it is common to assign a vertex to each thread, and have a block of threads handle a partition. For even better performance, results from one partition should only influence that same partition, otherwise it will be necessary to synchronize the execution across the partitions.

Some interesting measures to be implemented in a GPU are PageRank [14] and betweenness centrality [15], both are considered centrality measures. The first is the original Google algorithm to compute the importance of a web page based on the number and quality of connections to other webpages. The betweenness centrality for a node is defined as the number of times messages between two other nodes, have to pass through that node.

2.6.1 Centrality of a vertex

Networks are a set of elements connected by a set of paths through which something flows. The contents of the flow are particular to each network type, for example in a street network the flow is the vehicle traffic and on a social network the flow is the information. A network can be mapped to a graph, with elements being the nodes and the path between two adjacent nodes being an edge. The importance of each node in the network is called centrality, this is a measure used to rank nodes by their importance in the network.

There are many ways to measure the centrality of a node, each one focusing on different aspects of the network. A classification for these measures was proposed in Borgatti and Everett, 2006, [16], with the introduction of three categories for these measures: walk type for the type of walks considered in the measure analysis. Walk property distinguishes measures that count the number of walks or the volume, from those that focus instead on the walk length. Lastly, walk position separates measures in two categories: those that use walks that start or end on the node, from those that use walks that pass through a node, these classes names are respectively radial and medial.

These categories help us to better understand the differences between centrality measures. It is beneficial to use centrality measures from different categories, as measures from different categories are complimentary. This way it is possible to obtain, in the same application, a more accurate value of the importance of a node in the network.

This work will use two centrality measures to demonstrate how these can be computed in a GPGPU system. First the PageRank, that uses the number and quality of links to a node. According to the previous categorization this measure’s walk property is volume and the walk position is radial. An also the betweenness centrality, that counts the number of shortest paths that pass through a node, thus falling into volume class regarding walk property and medial class of the walk position. A better description and implementation details for the two measures is in section 2.6.
PageRank

The PageRank was developed to rank websites according to their relevance. It considers the number and quality of connections to pages to rank each page, this measure was created assuming that more important websites are more likely to be liked from other sites. Each PageRank is computed as the sum of the contributions of all pages that link to it, the page contribution is calculated from the page rank equally divided by all links from that page. Therefore, pages linked from many pages are more likely to have a higher rank, and pages that have a high rank will have a bigger influence on their linked pages’ rank.

The PageRank of a page \( v \) can also be represented as \( PR(v) = \sum_{u \in I_v} \frac{PR(u)}{|O_u|} \), where \( I_v \) is the set of pages with incoming links to \( v \) and \( |O_u| \) is the number of outgoing links from \( u \). There are two possible implementations of the PageRank: an algebraic and an iterative.

The algebraic solution involves finding the solution of a linear system with \( n \) variables. Considering a graph with only 4 nodes, the solution is given by the following equation:

\[
A. \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}
\]

Where \( A \) is the transition matrix, representing the links between pages and what fraction of the PageRank is transferred to each linked page. Solving this problem for graphs with several millions of nodes is not only impractical but even specialized tools are not capable of solving this linear algebra problem.

Due to the difficulty in finding an exact solution, the PageRank is solved for an approximate value. This approach is found using a dynamic system, that finds an equilibrium value for the PageRank value. This approximation is most commonly computed using the Power Method, an iterative algorithm that computes the PageRank value with an error lower than \( \epsilon \). In each iteration a SpMV, a sparse matrix-vector multiplication, is performed between the graph matrix and the PageRank vector, as seen in Algorithm 3.

**Algorithm 3 Power Method for PageRank**

1: Initialize \( R \) randomly to be \( R_0 \), and let \( k = 0 \)
2: repeat
3: Compute \( Y = qAR_k \)
4: \( d = ||R_k||_1 - ||Y||_1 \)
5: \( R_{k+1} = Y + dE \)
6: \( k \leftarrow k + 1 \)
7: until \( ||R_{k+1} - R_k||_1 < \epsilon \)

The PageRank also considers the possibility that at some point a random surfer stops following links on a page. This also models the existence of disconnected components and pages without outgoing links. The possibility of a random surfer going to another unlinked page is represented in the PageRank with the introduction of the dampening factor, \( p \). The dampening factor can represented by a value between 0 and 1, where a typical value is 0.85. When taking into consideration this new dampening factor the new PageRank matrix is \( M_{pr} \), and is computed with the following way:

\[
M_{pr} = pA + (1 - p).I/n
\]
There are already several studies in implementing the PageRank in a GPU system, only a few considered using graphs larger than the available GPU memory. While in the end this possibility was dismissed, one of the common solutions was to partition the graph into partitions to increase both memory efficiency and balanced workloads across all threads.

One study done on the AMD platform [17], uses the CSR format to store the graph. Their implementation involves sorting the graph by row length, and then proceeding to divide the graph into three partitions, according to the number of non zeros in each row. The exact number of non zeros per row used to define the thread assignment was determined by analysis of web graphs. Each partition is processed by a different kernel. The exact number of non zeros per row used to define the thread assignment was determined by analysis of web graphs. The partitions where created because social and web graphs present a power law distribution and using this partition scheme allows for a better thread occupancy.

The proposed division was:

- Nodes with no more than 6 non-zero elements are processed by a single thread.
- Nodes with more than 6 but less than 96 non-zero elements are processed by quarter of a wavefront, or 16 threads.
- Nodes with more than 96 non-zero elements are processed by an entire wavefront, or 64 threads.

If each row was processed by a single thread, threads that were processing small rows would have to wait for threads of the same wave that were processing large rows. Assigning a full wave for each row would leave many threads idle in each wave as only a small percentage of rows have a large number of threads. Using an entire wave for large threads allows memory accesses to be optimized, edges for the same node are stored sequentially on the global memory, with only one memory access enough data for all threads will be obtained, reducing the number of transferred unused words and idle threads.

The wavefront, or wave, is the AMD equivalent of the warp on the CUDA architecture, with the difference that a wavefront has 64 threads and the warp has only 32 threads.

Other works focused instead on using hybrid formats [18], in the SpMV computation. Both CSR and ELL-pack formats are used to store graph partitions, called workloads. Each of these workloads is stored in either CSR or ELL-pack format, accordingly to the size of the workload, to leverage on these formats capabilities in processing row-major and column-major matrices. In both cases the matrices are padded with zeros to improve the memory accesses by aligning the data with the warps. In the dense partitions, the vector can be cached from one multiplication to the next, because the values of this array are constant through the computation of the SpMV's.

When the datasets do not fit inside the GPU memory, it is suggested to implement a pipelining mechanism to transfer data from the host to the GPU [17, 18]. The only drawback of this method is that both implementations present a throughput between 16GB/s and 40GB/s. Comparing these values to the 15.75GB/s maximum transfer speed of the PCI-E bus, indicate a possible bottleneck on the data transfer between the host and the GPU.

**Betweenness Centrality**

Betweenness centrality [15], is a measure of a node’s centrality, used to determine what are the nodes responsible for the flow of information inside a network. It measures how much a network is dependent
on a node to pass information from one side to another. For example the betweenness centrality can be used to measure how much a user controls the communication path between two other users.

The betweenness centrality for a node $v$ is defined as the number of paths that contain $v$ out of total number of shortest paths between $s$ and $t$. Taking $\sigma_{st}$ as the number of shortest paths between $s$ and $t$, and $\sigma_{st}(v)$ as the number of shortest paths that pass through $v$, the betweenness centrality can be also defined as follows:

$$BC(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

For a graph with $n$ vertices and $m$ edges, the current fastest known algorithm to compute the betweenness of all vertices [19], represented in algorithm 4, has a time complexity of $O(nm)$ for unweighted graphs and $O(nm + n^2 \log n)$ for weighted graphs. It is generally believed this to be the asymptotically optimal solution. Other solutions, mainly based on the Floyd-Warshall algorithm, for obtaining the all pairs shortest paths, have a complexity of $O(n^3)$.

The Brandes’ algorithm [19] implementation introduces the concept of dependency of a vertex $s$ on any other vertex $v$. This dependency is defined by:

$$\delta_{ss}(v) = \sum_{w,v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{ss}(w))$$

The computation dependency makes use of a set of predecessor vertices on a the single source shortest path, from $s$. This set is defined by:

$$P_s(v) = \{ u \in V : \langle u, v \rangle \in E, d(s, v) = d(s, u) + w(u, v) \}$$

The implementation of Brandes’ algorithm follows two main steps. These steps are repeated taking into consideration a different source vertex, in line 2. These two steps are the graph traversal to find all shortest paths and computation of the centrality score for all vertices.

The graph traversal, in line 9, in unweighted graphs is based on the BFS algorithm, Algorithm 1. It has only a few changes in order to create a predecessor set of each vertex, $P$, and also keep the number of shortest paths from $s$ that reach the each vertex, in line 20.

The second step is the to compute the centrality score relative to the source $s$. In this step nodes are traversed in the same order as before. During this traversal, predecessor of a node, $v$, have their partial centrality score updated with the information of the fraction of the shortest paths to $v$ that go through them, in line 30. When nodes will no longer be visited on the current source, their score can be added to their final centrality score, in line 33.

There are already some work in implementing the betweenness centrality in GPUs with some degree of success [1, 20], presenting a few interesting solutions in optimizing the betweenness centrality on a GPU.

One of the solutions[20] was dividing the source nodes through the available GPU Stream Multiprocessors, and then performing the graph traversal in parallel inside each SM. Another improvement over the Algorithm 4 was to remove the need of the predecessor vertex set, by traversing all neighbors on the centrality computation step, thus reducing the space complexity from $m$ to $n$. Looking into how the graph is processed one level at a time it was possible to use only two queues. One with nodes in the
Algorithm 4 Betweenness centrality

1: \( C_B[v] \leftarrow 0, \forall v \in V \)
2: for \( s \in V \)
3: \( S \leftarrow \) empty stack
4: \( P[w] \leftarrow \) empty list, \( w \in V \)
5: \( \sigma[t] \leftarrow 0, t \in V; \sigma[s] \leftarrow 1 \)
6: \( d[t] \leftarrow -1, t \in V; d[s] \leftarrow 0 \)
7: \( Q \leftarrow \) empty queue
8: enqueue \( s \rightarrow Q \)
9: while \( Q \) not empty do
10: dequeue \( v \leftarrow Q \)
11: push \( v \rightarrow S \)
12: for all \( w \) of \( v \)
13: if \( d[w] < 0 \)
14: enqueue \( w \rightarrow Q \)
15: \( d[w] \leftarrow d[v] + 1 \)
16: end if
17: if \( d[w] < d[v] + 1 \)
18: \( \sigma[w] \leftarrow \sigma[w] + \sigma[v] \)
19: append \( v \rightarrow P[w] \)
20: end if
21: end for
22: end while
23: \( \delta[v] \leftarrow 0, \forall v \in V \)
24: while \( S \) not empty do
25: pop \( w \leftarrow S \)
26: for \( v \leftarrow P[w] \)
27: \( \delta[v] \leftarrow \delta[v] + \frac{|v|}{\delta[w]} \cdot (1 + \delta[w]) \)
28: end for
29: if \( w \neq s \)
30: \( C_B[w] \leftarrow C_B[w] + \delta[w] \)
31: end if
32: end while
33: end for
current level and another with the nodes that are going to be processed in the next level. To reduce the complexity of using the stack, a data structure similar to CSR was used to store the vertices that were processed at each level.

The other solution [1] focused instead on improving the data access patterns and preprocessing the graph in order to achieve the best speedups. One of their solutions is the virtual-CSn, a format where vertices with a high degree are replaced by several virtual vertices, each one with at most $ndeg$ edges. Replacing large vertices with smaller ones allows for each thread to have working sets with similar size, and thus having a better load balance across all threads. To further increase the data access patterns to the edge list, edges from different nodes are grouped together, so when threads access this list a single access will be able to provide information for all thread on the warp.

One other improvement made was to compress the graph by removing vertices with degree 1. These vertices can be safely removed when computing the betweenness centrality, because since they only have one connection it means that they centrality score will be the same as the vertex that it is connected to. This step is applied until no single vertex has only one edge. The betweenness centrality computation then takes into account the number of vertices that were removed from the graph. While the effect of this compression step depends on the actual data, on most data sets it represented an actual reduction on the number of vertices and edges.

Finally, not only they compare the running times of their implementation on a GPU versus a single CPU thread and also 8 CPU threads, but also versus a heterogeneous implementation using both GPU and CPU.

### 2.7 Conclusion of the background

This chapter explored relevant concepts and algorithms for graph exploration, like the shortest paths, breadth first search, Dijkstra's and also the vertex centrality. The most common graph storage schemes, adjacency matrices and adjacency lists, are compared to alternative schemes that are more suitable for GPU processing, like compressed sparse row (CSR) and Ellpack.

Considering that this work focus on the GPU architecture, the CUDA programming model is described in detail, in particular the thread hierarchy and memory management, and how these relate to the actual hardware implementation of a CUDA GPU. Some frameworks that are used for graph processing on multicore architectures, like the GraphX and MapGraph, are also reviewed, in an effort to understand some other techniques for graph partitioning and synchronization across multiple threads.

Finally, it explains the importance of graphs for social network analysis, and how to apply network measures to these graphs. Both network measures that are going to be used in this work, the PageRank and betweenness centrality, are described in detail, discussing previous works that focused on implementing said measures in a GPU, with particular emphasis on what improvements they made to increase the performance on a GPU and in how graphs larger than the GPU memory could be used.
Chapter 3

Parallel GPU implementation

This chapter will focus in the proposed implementation of both measures, PageRank and betweenness centrality, in a GPU architecture, using the algorithms described in Section 2.6.1.

The implementation will be presented using an iterative process, starting from a CPU implementation and through successive steps implement a parallel solution for a CUDA system, and then further optimize it. The PageRank implementation will start from the SNAP library. An initial sequential implementation will be used to compare against the parallel GPU implementation, and evaluate the improvement of the parallel implementation. The sequential implementation will also serve as a starting point for a parallel CUDA implementation. The last step will be to implement a solution capable of dealing with graphs larger than the available GPU memory.

The betweenness centrality will use previous CUDA implementations [20] as its basis and study the possibility of processing graphs larger than the GPU memory.

Graph preprocessing steps and relevant data structures that improve the PageRank performance and allow the graph to fit inside the GPU memory will also be discussed. For the betweenness centrality, most of the improvements follow the Bader work [21], and a possible implementation using CUDA Streams.

3.1 PageRank

As referred in Section 2.6.1, one of the best solutions to compute the PageRank is to use the Power-Method. Besides the graph matrix it only takes two extra vectors of size $n$, that are compared at the end of each iteration. Related work also pointed out that this method allows for easy graph partitioning [17, 6, 18]. A drawback of using the PowerMethod is that it requires access to the entire graph in each iteration to perform the SpMV computation, posing a challenge for graphs larger than the GPU memory.

3.1.1 Sequential implementation

Looking into the SNAP implementation of PageRank, [22], the parallelization on a GPU is straightforward, and does not need any kind of manipulation of the graph. A simplified sequential implementation, in Algorithm 5, using less complex data structures than SNAP, was used to create the first parallel implementation.

Besides the initialization of the PageRank vector, $prank$ on line 1, and the new PageRank vector,
temp on line 6, there are three main steps that can be parallelized. They are: the actual sparse matrix vector multiplication, SpMV, on line 9, the sum of the new vector on line 18 and the computation of the difference in the PageRank vector between the previous iteration vector and the new vector, on line 22.

In PageRank the sum of all ranks at the end of each iteration is always 1, meaning that after the SpMV the sum should be equal to the dampening factor $p$. Due to the floating point operations, on the SpMV, not being exact, a small value can be leaked during the computation. For this reason, at the end of the SpMV, all ranks are summed and the leaked value is determined, and used as the $1 - p$ probability of user randomly stop clicking links on a web page.

The computation of rank difference in the PageRank vector is also responsible of stopping the execution of the program when the difference between the newly computed rank vector and the old one is smaller than the specified $error$.

### 3.1.2 Parallel SpMV

#### Implementing from SNAP

Parallelizing the SpMV step from the SNAP implementation can be easily achieved by assigning each node to a different thread. This requires launching as many threads as the number of nodes in the graph. For each node obtain all the outgoing edges, and for each one compute the percentage of the node rank that is going to be transferred. Lastly that percentage is added to the new rank vector on the position corresponding to the destination node.

This last step has the possibility of inducing write conflicts when two different threads try to write into the same position. These conflicts will happen every time two edges being processed in the GPU have
the same destination node. Using this implementation where threads are assigned to source nodes, the only way to avoid this conflict is to use an atomic write for this operation. The atomic operations in CUDA are not cached, meaning that every atomic operation requires at least one read and one write operation from the global memory. As seen in Section 2.3.4 these accesses have a high latency, and will impact the execution of the program.

Another important detail is that this write operation has very low locality, as each thread will try to perform a different set of writes to memory, meaning that it will not be taking advantage of the large word size of each global memory access, but instead performing several memory accesses.

**Removing atomic operations by transposing the graph matrix**

The need for atomic operations in the previous implementation has due to the mapping of a source node to each thread. Considering that no data is written back to the source node, it is not relevant if it is the same thread that processes all node neighbors. Assigning instead each thread to a destination node and then obtaining the set of incoming edges, removes the need for the atomic operation. Using this mapping each thread will only write to its assigned node position, thus having no write conflicts.

The CSR format, Section 2.2.2 stores all the edges in a continuous array, with another array storing a pointer to the start of each node edge set. On directed graphs only the outgoing edges are stored, meaning that on CSR following an edge backwards is a costly operation compared to obtaining that same information from the the adjacency matrix.

![Example graph and respective adjacency matrix](image)

Considering the graph and respective matrix represented in Figure 3.1. It is pretty straightforward to discover what edges arrive to the \( n \) node by looking into the \( n \) column of the matrix. The same is not possible when the same graph is represented in the CSR format, as it is in Figure 3.2. Accessing the set of incoming edges for each node would require iterating through the columnindices to find all edges arriving to a given node, and then search in RowStartIndices for the origin node.

\[
A = \begin{bmatrix}
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{bmatrix}
\]

![Example graph in CSR format](image)

Considering that the CSR, compressed sparse row, stores matrices in row major format, obtaining an equivalent representation of the matrix but in column major should provide easy access to the set of
incoming edges of each node. One way to achieve this transformation is to obtain the CSR representation of the transposed matrix, $A^T$. Looking back at line 14 of the code represented in Algorithm 5, the number of outgoing edges for each node is also required, therefore that information is kept in the \textit{values} array. This representation is very similar to the CSWG format [6], described in the Section 2.2.5.

In practice obtaining the transposed matrix of a large sparse graph stored in CSR format is not trivial due to space concerns, but many datasets provide both the matrix and respective transposed matrix. It is also noteworthy that most of the PageRank implementations already assume that the graph is provided in a transposed format.

$$A^T = \begin{bmatrix}
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}$$

$$Values = [3 \ 3 \ 0 \ 1 \ 1 \ 1 \ 1]$$

$$RowStartIndices = [0 \ 1 \ 2 \ 4 \ 7 \ 8 \ 9 \ 10]$$

$$ColumnIndices = [3 \ 0 \ 0 \ 1 \ 0 \ 1 \ 4 \ 1 \ 6 \ 5]$$

Figure 3.3: Transposed example graph adjacency matrix and respective CSR format

**SpMV without atomic operations**

Using the transposed matrix enabled improvements on the SpMV kernel, removing atomic operations and increasing the locality of the writing operations. With this implementation each thread in a warp will perform write operations into the continuous memory positions, reducing the number of memory accesses. While this will increase the number of read accesses, for the most connected nodes it will be likely that the edge information is already present in L1 or L2 cache inside the GPU chip. The main advantage of this implementation is the absence of atomic operations, that were triggering several extra memory accesses as well stalling several threads until the atomic operation ended.

**Algorithm 6** CUDA PageRank SpMV without atomic operations

1: for all node of graph do in parallel
2: for edge = rowStart[node] to rowStart[node+1] do
3: srcNode = columnIndices[edge]
4: rankChange = $R_k[srcNode] \times \text{dampening/values[srcNode]}
5: $R_k + 1[node] = R_k + 1[node] + rankChange
6: end for
7: end for

With this implementation it was noted that graphs larger than the GPU memory were still not supported but also with the existing graphs there where many idle threads. Threads became idle while waiting for only a few threads on each warp to finish. The reason for this behavior is the structure of social network graphs, where most nodes have only a few connections, but a few nodes are highly connected. Considering this implementation, a few threads will have a great impact on the overall execution time, as they will force threads that are only processing a small number of edges to become idle, until they finish.

Two solutions have been found in literature that help to minimize this problem. One is to sort all the nodes first by the length of edge list, the other is to create graph partitions containing vertices with a similar number of edges. Sorting the CSR format by node size is costly and would require reindexing
the entire column indices array. On the other size partitioning the graph is computationally faster, and allows for optimizing the SpMV kernels for nodes with different sizes.

**SpMV kernels for different node sizes**

By splitting the graph into partitions that have only nodes of the same size it is possible to largely improve the thread occupancy, and also to have a better workload distributions. One possible way to do this division is to split the graph into partitions, one with small nodes, other with medium nodes and a third with the largest nodes, [17], and for each of these partitions a slightly different SpMV kernel was implemented.

Nodes classified as small if their size is less than 8 incoming edges and medium nodes have at least 8 but less than 96 edges, nodes with more than 96 edges are considered large. The associated kernels for these partitions are described as follows: for small nodes no changes are made into the previous implementation, meaning that each thread will be assigned one node; for medium nodes, the warp is divided into 4 parts, and each will node will be divided across that warp partition, in other words one node is assigned to a quarter of warp; large node are assigned to an entire warp. Not only this division reduces most of the unbalanced workloads of each thread present in the previous implementation but it also improves the access to global memory. Accesses made to get the edges of medium and larger nodes will serve several threads at once, thus reducing the time each warp is waiting for them to complete.

These first values were based on previous work, that implemented this partitioning model in a GPU [17], but after experimentation values that exhibited better performance for the graphs in this study where found, in Section 4.2.2. In any case the initial values allowed to understand the behavior of each kernel.

In this solution the rowStart and columnIndices vectors are replaced by three rowStart and columnIndices vectors, one pair for each of the node classes. These vectors only contain information pertaining to nodes on that class. On Algorithm 6 the node id was the same as the position on the Rk index. After dividing the nodes by their size this mapping is no longer true, making necessary to introduce an index of node id’s to the corresponding index in the new rowStart vectors.

The resulting data structures from this division are represented in figure 3.4. Comparing with the previous structure, figure 3.3, only one data structure was added: the NodeIds vectors, storing the previous index of the node.

\[
Values = [3 \ 3 \ 0 \ 1 \ 1 \ 1 \ 1]
\]

\[
\begin{align*}
SmNodeIds &= [0 \ 1 \ 2 \ 5 \ 6] & LgNodeIds &= [3 \ 4] \\
SmRowStart &= [0 \ 1 \ 2 \ 3 \ 4 \ 5] & LgRowStart &= [0 \ 2 \ 5] \\
SmColumnIdx &= [3 \ 0 \ 1 \ 6 \ 5] & LgColumnIdx &= [0 \ 1 \ 0 \ 1 \ 4]
\end{align*}
\]

(a) Small nodes, with 1 edge

(b) Large nodes, with more than 1 edge

Figure 3.4: Example graph divided in two node classes

It was also considered the use of shared memory on the SpMV kernels, as a way to store data that could be accessible by all threads. The only data structure that could benefit from this memory
was the `columnIdx` array, that is accessed different times by each thread. Using the shared memory would be possible to have all threads copy a small partition of this array to the shared memory and then individually consume it in the SpMV computation. With the implementation of the different kernels for each node size, this would only be applicable to partitions with small nodes. For these partitions the number of reads from this vector is already limited by the node size. Moreover, for these nodes, each global memory access is already serving several threads at once, and values that are not used immediately are most likely still present one of the two available levels of cache on the GPU chip.

**Precomputing the rank contribution (rank cache)**

After further observation of the SpMV kernels, it was noted that the computation of the transferred rank is repeated for each outgoing edge of each node. With the graph transposed the computation is performed from the point of view of the destination node, where it receives different contributions from different nodes, in line 4 of Algorithm 6. To compute the contribution it is necessary to obtain the source node rank and number of edges. This reflects in two global memory accesses for each edge. Each node gives the same contribution through all of his edges, independently of the destination node.

Considering not only the difference in the number of nodes and the number of edges in the graph, but also limited caches of the GPU, it was chosen to compute each node contribution before the SpMV kernels. This computation has to be done in every PageRank iteration, as the rank contribution changes from one iteration to another.

**Algorithm 7** computing the rank cache in the GPU

1:    for all node of graph do in parallel
2:        rankCache = Rk[srcNode] * dampening/values[srcNode]
3:    end for

This computation can be simply done by iterating through all nodes and storing the result in a new vector, as seen in Algorithm 7. The resulting vector is called rank cache as in some sense it acts like a cache for the computed rank. The main benefits of using this rank cache is that the SpMV kernel will have to perform one less global memory access per edge, increasing the likelihood that the read value is still available on the GPU cache. The introduction of the this new kernel to compute the rank cache should have a minimal impact, as all global memory accesses are aligned and coalesced.

**Optimizing the SpMV kernels**

When computing the new rank of each node it is possible to delay the writing of the new rank until all node incoming edges are processed. This is because all threads that are processing a node are guaranteed to be on the same warp. Writing to the PageRank vector, on global memory, can be done at the end of the SpMV computation of that node. Not only this reduces the number of write operations to global memory of each thread to exactly one, but also helps to synchronize that one access across all thread in the same warps.

Besides using an entire warp for the SpMV computation of each node, the Algorithm 8 also performs a warp reduce to get the sum of result across all threads on each warp. This way only the thread 0 of each warp needs to perform the write operation. The reduce operation is done on line 14 by the `__shfl_down` instruction. This instruction is capable of accessing a value from another thread in the same warp. This usage of the `__shfl_down` operation to reduce across a warp, is illustrated in Figure 3.5 perform the reduce on blocks of 8 threads.
**Algorithm 8** CUDA SpMV kernel for large nodes

```c
__global__ void multMVWarp (double* prank, int nNodes, int* rowStart, int* edgeList, int* nodeIds, double* output) {
    int threadNode = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
    double result = 0.0;

    if (threadNode < nNodes) {
        int start = rowStart[threadNode] + threadIdx.x % warpSize;
        int end = rowStart[threadNode + 1];
        for (int edge = start; edge < end; edge += warpSize) {
            result += prank[edgeList[edge]];
        }
        __syncthreads();
        result += __shfl_down (result, 16);
        result += __shfl_down (result, 8);
        result += __shfl_down (result, 4);
        result += __shfl_down (result, 2);
        result += __shfl_down (result, 1);
        __syncthreads();
        if (!(threadIdx.x % warpSize)) {
            output[nodeIds[threadNode]] += result;
        }
    }
}
```

**Supporting graphs larger than the GPU memory**

The GPU memory size is smaller than some social graphs available. To support these a partition scheme was implemented by dividing the graph in rows. Other alternatives were to divide the graph by columns or creating a set of tiles combining the division of rows and columns.

Dividing by columns would not only lead to multiple writes to the resulting rank vector, but when iterating through the nodes, there would be no guarantee that each partition would contain at least one edge. Therefore, using this partition type is not ideal. On the other hand, dividing by rows requires access to the entire original PageRank vector on every partition. Otherwise row division does not seem to have any relevant drawback. Finally, using a tiling partition would combine both drawbacks of row and column partitioning. While also viable, tiling partitioning is more appropriate for datasets larger than the ones considered in this work.

To apply a row partitioning scheme to a graph in CSR format is only necessary to know the number of nodes in each partition. There are two ways to determine number of nodes. $n_{part}$, in each partition, the simpler one is to divide the nodes equally among the partitions, the other ways is to balance the...
total number of edges across all partitions. To implement the SpMV with this partition scheme, each partition can be considered as an independent graph of \( n \) columns and \( n_{\text{part}} \) rows. These partitions can be either applied to the entire graph, or after splitting nodes as described in Section 3.1.2, apply this partition scheme to each one of the node classes. Considering that nodes of the same class have a similar number of edges, dividing the node equally among the partitions will provide a satisfactory balance of the edges per each partition.

Algorithm 9 CUDA SpMV kernel for small node partitions

There is no need to use additional data structures to store each partition, these can be created when invoking the SpMV kernels, like the one in Algorithm 9. Knowing the number of nodes in each partition, it is possible to copy to the GPU memory only the fraction of the CSR data structure relevant to the current partition. The copy of the partition is done immediately before invoking the kernel that is going to process it. When the kernel is finished, the partition information on the GPU is replaced by the next partition.

When invoking the SpMV kernel with this partition scheme, it is also necessary to send the original index of the first node \( \text{nodeOffset} \) and edge \( \text{edgeOffset} \), so that the kernel can still write to the correct position on the PageRank vector.

Concurrent copies to GPU memory and computations

When dealing with multiple partitions that are swapped in and out of the GPU memory, it is necessary to also consider the memory transfer time. This transfer can have a large impact in the overall performance. It was this step that was pointed as a limitation in previous work [17, 18], as the data transfer would take longer than the actual computation.

As referred in Section 2.3.5, using CUDA Streams it is possible to perform more concurrent operations in a CUDA architecture. Using CUDA Streams it is possible to perform a copy to the GPU memory and invoke a kernel at the same time. In effect hiding the communication costs between CPU and GPU behind the computation. While several streams can be instantiated and run at the same time, in this implementation only two were used to maximize the GPU occupancy.

To achieve this effect, it is necessary to use the partition scheme presented in the previous section. In addition, it is necessary to have at least two partitions in the GPU at the same time. One that is being read by the running kernel, and another that is being copied from the host memory to the GPU memory. The control of each partition is assigned to a different stream, and each stream will alternately copy a partition from the host and invoke the SpMV kernel on the partition that was just copied over, until all
partitions have been processed.

For this process to work it is necessary to have two partitions and two streams. Considering the graph sizes, it is expected that a single kernel will occupy all available SMs, if that is not the case more streams can be used to fully occupy all SMs.

### 3.1.3 Parallel Reduce

In a parallel system, the reduce operation can be done iteratively with the contribution of all threads. In each iteration each threads applies the reduce operation to two different values. The best approaches follow a tree-like structure to apply the reduce operation to all elements. Not only reduction operation can be implement in a very efficient way in a CUDA architecture, achieving $O(\log N)$ complexity on a block-wide reduce operation. Such implementation for a block-wide reduce is provided and documented in the CUDA framework [23] as part of their collection of sample programs.

To summarize the optimal implementation of a CUDA reduce operation, on a block-level, it uses threads on the same warp to access continuous memory positions. The two values to reduce in each iteration are chosen using the `threadId` and a `stride` value. The `stride` value is the same as the number of active threads. This reduces the number of inactive threads, and also achieves optimal efficiency on the memory accesses. This access pattern is represented on Figure 3.6.

One other important detail is that because that there is no explicit global synchronization instruction that can be invoked by a kernel, this implementation is only capable to apply the reduce to one block. Taking advantage of this fact, and that a block has at most 1024 threads, it is possible to apply advanced optimization techniques to the reduce kernel, like: loop unwrapping; templating all (10) possible block sizes and using shared memory.

![Figure 3.6: CUDA block reduce](image)

The PageRank requires two reduce operations: one to compute the leaked rank and the other to get the convergence of the PageRank vector between two iterations. By reducing several blocks at the same time and using the kernel invocations as synchronization points, it is possible to use the previous reduce implementation on vectors larger than the block size. This can be achieved by calling the reduce kernels recursively, as show in Figure 3.7.
3.1.4 Computing the rank difference

This last step is responsible for stopping the execution of the program when the difference between the newly computed rank vector and the old one is smaller than the specified error. It is also responsible for adding the probability that a user stops following links, while ensuring that the sum of the vector is always 1. Both operations are done by the same kernel, using both the old rank and new rank vectors. For each node, the new rank value is computed from the sum of the SpMV result and the probability of stopping following links. The new rank vector will temporarily store the rank difference in the current iteration for each node. Lastly, the new value is copied to the old rank vector.

When this kernel is done, a reduce operation is applied to the new rank vector, that is now storing the change in rank for every node. This will provide the overall change on rank vectors, if this value is smaller than the error parameter, the PageRank execution will finish, otherwise the new rank value is reset to 0 and the PageRank computation will proceed.

3.1.5 Data structures

Focusing only on the data structures used in PageRank to store both the graph and the rank values. The host memory stores the entire graph using the format represented in Figure 3.4. On the GPU the graph is stored in a format similar what is stored in host memory, but considering the partitioning scheme.

- **rowStart** this is divided in three separate arrays, one for each node partition, corresponding to the rowStart array in CSR.
- **values** contains the number of outgoing edges for each node.
- **p1ColumnIdx** stores a graph partition, contains only the graph edges, it is a partial copy of CSR format’s columnIdx array.
- **p2ColumnIdx** stores a second graph partition.

Also on the GPU, three different arrays are used to store the rank information, two are required by the PageRank algorithm, the third is used to cache the rank transfer. Note that while the graph only uses arrays with integers, the rank information is stored using floating points values, in particular doubles for extra precision.

- **rank** with the current rank of each node.
- **newRank** where the new rank is stored.
- **rankCache** for caching each node rank transfer, or rank \* values \* dampening factor.
3.1.6 PageRank with graphs larger than GPU memory

In the previous sections each one of the PageRank steps was analyzed and discussed how it could be implemented in CUDA. Not only every step has its own kernels but also some have dependencies to data that has to be copied to the GPU beforehand. It is important to optimize the scheduling of CUDA kernels to maximize the GPU usage, and reduce the idle times. This is achieved by a careful management of the kernel invocations through the existing streams. The kernel invocations and respective streams are outlined in Algorithm 10

Algorithm 10 PageRank in CUDA using partitions and streams

1: copy rowStart and nodeId arrays to GPU on stream1
2: initPrank(pageRankVector, nNodes) stream2
3: computePRankValues(rankV, values, p, nNodes, rankCacheV) stream2
4: set newRankV to 0 on stream2
5: synchronize device
6: while true do
7:     for i ← i, nPartitions do
8:         copy partition i to GPU, on stream1
9:         multMV(p1RowStart, p1ColumnIdx, rankCacheV, newRankV) stream1
10:        i ← i + 1
11:     copy partition i to GPU, on stream2
12:     multMV(p2RowStart, p2ColumnIdx, rankCacheV, newRankV) stream2
13: end for
14:     for i ← i, nPartitions do
15:         copy partition i to GPU, on stream1
16:         multMV dividedWarp(p1RowStart, p1ColumnIdx, rankCacheV, newRankV) stream1
17:        i ← i + 1
18:     copy partition i to GPU, on stream2
19:     multMV dividedWarp(p2RowStart, p2ColumnIdx, rankCacheV, newRankV) stream2
20: end for
21:     for i ← i, nPartitions do
22:         copy partition i to GPU, on stream1
23:         multMV warp(p1RowStart, p1ColumnIdx, rankCacheV, newRankV) stream1
24:        i ← i + 1
25:     copy partition i to GPU, on stream2
26:     multMV warp(p2RowStart, p2ColumnIdx, rankCacheV, newRankV) stream2
27: end for
28: synchronize device
29: sum ← reduce(newRankV) stream1
30: getDiff(sum, newRankV, rankV) stream1
31: recordEvent(event, stream1)
32: diff ← reduce <<< stream1 >>> (newRankV)
33: waitEvent(stream2, event)
34: computePRankValues(rankV, values, p, nNodes, rankCacheV) stream2
35: set newRankV to 0 on stream1
36: if diff < error then
37:    break
38: end if
39: synchronize device
40: end while

The usage of streams in the SpMV computation is already described in Section 3.1.2. Partitions of nodes of similar size are computed in concurrent streams, this serves to hide the communication cost
behind the computation of another partition of similar size. In Algorithm 10 some operations related to creation of partitions and kernel parameters are omitted for simplicity.

The reduce and difference computation is synchronized so it only starts after all partitions are processed. The reset for the next iteration is done by using one stream to compute the next PageRank values while the other stream resets the $newRankV$ to 0. This is possible because the reset is done using a CUDA memset call that doesn't require computation by the GPU. Lastly the device is synchronized again, to ensure that both streams have completed before starting the new iteration.

### 3.2 Betweenness Centrality

The best betweenness centrality algorithm is the Brandes’ [19], explained in Section 2.6.1. The available datasets are all unweighted, therefore it will only be necessary to use the SSSP implementation of the betweenness centrality. There are two previous works that present an efficient approach to the betweenness centrality implementation in CUDA. One focusing in a parallel implementation of CUDA and other more focused in using effective data structures.

The Brandes’ algorithm computes the betweenness centrality scores by calculating all the shortest paths from a source node to every other node, and then calculating a partial centrality score for each node. This process is repeated for every node as a source node. The final centrality score is obtained by summing all partial scores. This algorithm requires several auxiliary data structures, for the shortest paths, with the BFS, it needs both a queue of vertices to be processed, the stack of visited nodes, the distance of each node from source, the number of shortest paths to that node as well the predecessor set of each node. The dependency accumulation step requires access of most of the previous structures and one to store the dependency of each node. Finally, it is also necessary to store the betweenness centrality score of each node.

#### 3.2.1 Parallelizing the Brandes’ algorithm

There are three possible parallelization possibilities in the Brande’s algorithm:

**Source node**

All sources are processed in parallel. This solution has the advantage that each source can be processed individually as there are no dependencies between the associated computations. On the other side the required data structures will require a large amount of memory, also parallelizing the computation of each source in CUDA would not be practical due to the many conditions and different execution paths.

**Nodes at the same distance**

The computation of shortest paths, using the BFS is done by traversing the graph as a tree. Nodes at the same depth, or level, in the tree all have the same distance to the source node. In each iteration of the BFS the frontier advances one level, nodes at the frontier level are processed in that iteration. These nodes can be processed in parallel as long the writes to the distance and number of paths from source are performed atomically.

**Neighbors of the same node**

On the BFS when processing a node it is necessary to traverse all his edges. This step can also
be parallelized. Considering the structure of social graphs, there isn’t guarantee that every thread will have work assigned, as some nodes only have a few edges.

The first option is a coarse grain approach, where the processing of each source is assigned to a different thread. The other two options are more fine grained as they further break-down the processing of source node and distribute the work across multiple threads. The last option is not ideal as there is no solution for the thread occupancy problem, and it would reduce the performance of the final solution.

As noted on Section 2.6.1 there are already some implementations of the betweenness centrality on the CUDA architecture [20]. In this implementation each source node is processed by a different Stream Multiprocessor. The processing of a each source node is done by a single block, with nodes at the same distance from source being processed in parallel by the block threads. This approach is a combination of the first two parallelization options described before. The parallel implementation requires only small changes to the Brandes’ algorithm in order to allow parallel threads and reduce the space requirements.

This work will review the proposed work-efficient CUDA implementation in McLaughlin [20], and study how it scales with large graphs and also how it compares against a single core CPU implementation.

Shortest Paths

The computation of the shortest paths, using the BFS algorithm is slightly modified from the original Brande’s algorithm. It no longer needs to keep the predecessor vertex as the centrality accumulation step will traverse each node individually.

To enable the parallel computation of nodes at the same distance from source, two queues are used. One with the set of nodes that are being processed in the current iteration and another with the nodes that are going to be processed in the next iteration. These queues represent the frontier, containing nodes that are not yet processed but have already been discovered. This queue scheme is possible due to the usage of the BFS to process the graph, where all visited nodes are at most in a one hop distance from the current node, so the order that nodes, in the same queue, are processed is indifferent. At the end of each iteration the current queue is replaced by the next frontier queue.

One other change to enable parallel processing inside the same block is the to track the iteration where each node is processed. This information is associated to the stack of processed nodes by using an auxiliary array. This auxiliary array marks the positions of the first node of each iteration in the stack, just like the rowStart in the CSR format marks the first edge of each node in columnIndices.

Iterations have to be synchronized across all participating threads. This synchronization is done after processing all nodes in the current frontier and also just before the next iteration. Other necessary synchronization points are after the data initialization and just before sequential steps in the kernel. The synchronization points can be done by invoking the __syncthreads call, as the source is only being processed by a single block.

Dependency accumulation

This step is more straightforward as it uses the previous step results to compute the partial centrality scores for each source node. In the previous step, nodes where stored in stack grouped by the distance to the source node, using the ends vector. In this step, nodes at the same distance can be processed at the same time by different threads. For each node, all edges are traversed once again to find the predecessor nodes, as the predecessor sets are no longer created in the previous step. This not only greatly
reduces the memory requirements, but is also enabled interesting speedups. Lastly, it is necessary for the partial centrality scores to be added to the betweenness centrality vector. This addition needs to be atomic, as other blocks can also be performing the partial writes to the betweenness centrality vector.

### 3.2.2 Data structures

In the betweenness centrality all relevant data is stored inside the GPU without any partitioning schemes. The graph is stored using the CSR format, with the rowStart and columnIdx arrays. To perform the shortest paths and dependency accumulation steps for each source node, extra data structures are used, each one requiring $O(n)$:

- `queue` containing the nodes in the current frontier
- `queue_nxt` with nodes on the next frontier
- `d` storing the distance of each node to the source
- `sigma` with the number of shortest paths from source to each node
- `stack` used to keep track of nodes visited in each iteration of the BFS
- `ends` pointing to the first node of each iteration in stack, similar to the rowStart in CSR format
- `dependency` the dependency accumulation for each node

### 3.2.3 Parallel source node processing

Due to the several synchronization points necessary in the BFS kernel, this step can only be assigned to one block. The alternative would be to do the synchronization using several smaller kernel calls. This alternative would need a more complex implementation, and there is no guarantee that having more threads available would improve the performance. Recalling that in CUDA each block is mapped to a single SM, and each card has several SM’s using only one block is not optimal.

The best solution is to have each SM process a different source node. This effect can be obtained in two ways. One is to assign each block to a different source based on the blockId of each block. This solution requires each kernel to perform several arithmetic operations to get each source and respective data structures. One alternative is to use CUDA Streams, where each stream only processes one source node at a time. Using streams, the auxiliary data structures can be directly passed to kernel invocations. In both cases is only necessary to have enough space on the device to store one set of data structures for each block being processed in parallel.

The major limitation of this implementation is that it is not capable of dealing with graphs larger than the GPU memory. Considering the number and size of the auxiliary data structures, like the `queue`, `stack` and `dependency` vectors, these will occupy a significant part of the GPU memory, further reducing the limit on the graph size.

### 3.3 Conclusion

In this chapter GPU implementations for both PageRank and betweenness centrality where presented. Both implementations aim to achieve the best possible performance in a CUDA environment and also to
support graphs larger than the total GPU memory.

The PageRank implementation was obtained from parallelizing an initial sequential implementation. Peak performance was obtained by transposing the graph to eliminate write conflicts. A better balancing thread workloads was achieved using different kernels for different node sizes, as suggested by Wu [17]. Lastly repeated rank computations were removed from the SpMV kernels and instead computed at the beginning of each iteration. To support graphs larger than the GPU memory the graph was partitioned by rows. Each partitioned is then copied to the GPU and processed individually.

By using CUDA Streams most of the communication costs between the GPU and host is eliminated by hiding them behind the SpMV computations. Only the first partition communication can not be hidden as there are no kernel running on the device. Using CUDA Streams also allows for memory operations simultaneous to kernel executions, which creates a small speedup in each iteration.

The betweenness centrality follows the proposed implementation of McLaughlin [20]. Most notably by not using the predecessor sets and using two queues for storing nodes in the current iteration and in the next one. For the parallelization of this algorithm an extra data structure is necessary to track what nodes are at each depth. The parallelization of this algorithm also requires synchronization of threads several times across its execution.

While no changes were made to the algorithm, the parallel processing of source nodes where considered. Both in the aspect of which solution would have the best speedup and also managing the GPU memory to maximizing the number of parallel source node tasks. No solution was found that could fit graphs larger than the GPU memory.
Chapter 4

Experimental Results

This chapter presents the on the experimental evaluation and discussion of the results relating both to
the PageRank and betweenness centrality.

After presenting both the test environment and test datasets, the execution times of the CUDA im-
plementation of each measure will be compared against their respective sequential execution times. For
each measure, the impact of parameters related to graph partitioning on the usage of the GPU resources
will be discussed.

The PageRank will be tested against large datasets and also against a graph larger than the GPU
memory. The performance with different sizes for the node classes will also be analyzed, as well the
effect of the number of partitions. To understand if the choice of these parameters can be influenced by
the graph structure, these tests will be run in different graphs.

The betweenness centrality will be also be evaluated across large datasets. Considering the high
complexity of computing the betweenness centrality it is only possible to obtain result in a useful time
for medium graphs. Large graphs can take up to months to obtain the betweenness centrality results
in a single core. Unless a high speedup is achieved, obtaining results for betweenness centrality on a
CUDA architecture will also have an execution time in the order of weeks or months. In these cases the
computation will be only run for a percentage of the total source nodes, this should provide valid results
for both the speedup and allow for an accurate estimation of the execution time for all source nodes.

4.1 Test Environment

The testing system is composed by an i7 4790K CPU with four cores with hyper-thread and a base clock
of 4.0GHz, 16GB of RAM and a GTX 970 GPU with 4GB of VRAM and 13 SM for a total 1664 CUDA
cores. The parallel CPU implementation is achieved through the OpenMP API. The used datasets are
obtained from the SNAP Large Network Dataset Collection [24], the datasets where chosen based on
their size and the represented network type. Details on each of the datasets are on table 4.1.

To guarantee the validity of the results, unless stated otherwise, each test case was run five times.
The presented results are the average of all five runs. Plus the computing times for both sequential and
CUDA implementations will only consider the actual execution time. It excludes the loading of the graph
to memory and on the case of CUDA implementation it does not consider the time taken to transpose the
CSR as most datasets already provide the transposed graph. The speedups presented in the evaluation
<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
<th>CSR Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>soc-Slashdot0811</td>
<td>77 360</td>
<td>905 468</td>
<td>20</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>4 847 571</td>
<td>68 993 773</td>
<td>282</td>
</tr>
<tr>
<td>com-Orkut</td>
<td>3 072 441</td>
<td>117 185 083</td>
<td>459</td>
</tr>
<tr>
<td>com-Friendster</td>
<td>65 608 366</td>
<td>1 806 067 135</td>
<td>7140</td>
</tr>
</tbody>
</table>

Table 4.1: Used datasets and respective sizes

<table>
<thead>
<tr>
<th>Dampening</th>
<th>Error</th>
<th>Iterations</th>
<th>CPU time (s)</th>
<th>CUDA time (s)</th>
<th>SpeedUp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.850</td>
<td>1·10^-4</td>
<td>24</td>
<td>6.4489</td>
<td>1.2697</td>
<td>5.0791</td>
</tr>
<tr>
<td>0.850</td>
<td>1·10^-5</td>
<td>36</td>
<td>9.5676</td>
<td>1.8353</td>
<td>5.2131</td>
</tr>
<tr>
<td>0.850</td>
<td>1·10^-10</td>
<td>105</td>
<td>27.3441</td>
<td>5.0987</td>
<td>5.3629</td>
</tr>
<tr>
<td>0.850</td>
<td>1·10^-15</td>
<td>175</td>
<td>45.3952</td>
<td>8.4079</td>
<td>5.3991</td>
</tr>
<tr>
<td>0.900</td>
<td>1·10^-4</td>
<td>35</td>
<td>9.2886</td>
<td>1.7880</td>
<td>5.1950</td>
</tr>
<tr>
<td>0.900</td>
<td>1·10^-4</td>
<td>309</td>
<td>80.0007</td>
<td>14.7421</td>
<td>5.4267</td>
</tr>
<tr>
<td>0.999</td>
<td>1·10^-4</td>
<td>2,313</td>
<td>597.4695</td>
<td>109.4677</td>
<td>5.4580</td>
</tr>
<tr>
<td>0.990</td>
<td>1·10^-15</td>
<td>2,804</td>
<td>740.5829</td>
<td>130.2306</td>
<td>5.6867</td>
</tr>
</tbody>
</table>

Table 4.2: LiveJournal PageRank execution (wall) time with different rank parameters

are obtained by comparing the CUDA implementation to a single-core sequential implementation.

### 4.2 PageRank

It is not only important to check if the PageRank implementation described in the previous chapter is capable of achieving any speedup compared with the sequential execution. Additional tests with different PageRank parameters where also run to understand and exemplify how these affect the overall execution times. Due to the nature of the PageRank, where these parameters only affect the number of iterations, it was expected that a constant speedup over all test cases.

For this test, the graph partition and node classes where set with large graphs as more than 96 edges, and small edges having up to 8 edges. For medium size nodes, the warp is divided into 8 groups, with 4 threads each. Plus each one of those three node classes was divided into 12 partitions. Although a smaller number of partitions could be used for graphs that fit inside the GPU, after inspection of the execution of CUDA programs it was found to have no negative effects on the performance.

With the soc-livejournal1 graph it was achieved a speedup of about $5.7 \times$. This low value for speedup can be attributed to the cost on initializing the GPU and launching the multiple kernels. It is also noteworthy that the speedup has a slight increase in test cases that have a higher number iterations. This behavior can be explained by the small impact of the preprocessing step in splitting the nodes by their size.

Using different values for the dampening factor and the approximation error lead to a different number of iterations of the Power Method Algorithm. As expected, the achieved speedup is independent of the number of iterations. The difference in the CUDA Total Time and CUDA Execution Time results from transposing the matrix in the step.
<table>
<thead>
<tr>
<th>Test #</th>
<th>Dampening</th>
<th>Error</th>
<th>Iterations</th>
<th>CPU Total Time (s)</th>
<th>CUDA time (s)</th>
<th>SpeedUp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.850</td>
<td>1·10^{-4}</td>
<td>20</td>
<td>351.0906</td>
<td>39.3774</td>
<td>8.9160</td>
</tr>
<tr>
<td>1</td>
<td>0.850</td>
<td>1·10^{-5}</td>
<td>26</td>
<td>393.4610</td>
<td>50.0138</td>
<td>7.8670</td>
</tr>
<tr>
<td>2</td>
<td>0.850</td>
<td>1·10^{-10}</td>
<td>55</td>
<td>811.0283</td>
<td>100.2703</td>
<td>8.0884</td>
</tr>
<tr>
<td>3</td>
<td>0.850</td>
<td>1·10^{-15}</td>
<td>84</td>
<td>1,230.2332</td>
<td>149.0471</td>
<td>8.2540</td>
</tr>
<tr>
<td>4</td>
<td>0.900</td>
<td>1·10^{-4}</td>
<td>23</td>
<td>347.5900</td>
<td>44.7532</td>
<td>7.7668</td>
</tr>
<tr>
<td>5</td>
<td>0.990</td>
<td>1·10^{-4}</td>
<td>33</td>
<td>492.5164</td>
<td>62.1141</td>
<td>7.9292</td>
</tr>
<tr>
<td>6</td>
<td>0.999</td>
<td>1·10^{-4}</td>
<td>34</td>
<td>507.2943</td>
<td>63.9134</td>
<td>7.9372</td>
</tr>
<tr>
<td>7</td>
<td>0.990</td>
<td>1·10^{-15}</td>
<td>135</td>
<td>1,967.8159</td>
<td>239.5791</td>
<td>8.2136</td>
</tr>
</tbody>
</table>

Table 4.3: Friendster PageRank execution (wall) time with different rank parameters

4.2.1 Partitioning graphs

The friendster graph has over 1800 Million edges, which takes around 7GB. The graph alone cannot be copied entirely to the GPU memory, and it is also necessary to have space to store the other structures necessary for the PageRank computation.

As discussed in previous sections one solution is to partition the graph into smaller pieces and have all necessary data structures copied into the GPU memory. While each of the node classes can be seen as a graph partition, these are not enough. Each class was further divided into twelve partitions with the same number of nodes. The total GPU memory necessary was reduced to around 3.5GB. This size includes all data structures as well two partitions, one being copied to the GPU and another being processed.

A smaller number of partitions could also be used, without increasing the required memory. In this graph the majority of the used memory is storing the non partitioned arrays, containing the rank information, the nodeIds and the number of outgoing edges of each node.

Using more partitions than the strictly necessary to the GPU computation can improve the performance. This is a result of the usage of CUDA Streams, where the first copy to GPU memory is the only one not hidden behind computation. With more partitions, the copy operation times are smaller, leading to a slight increase in performance in each iteration.

4.2.2 Node Classes

The division of nodes according to their size is done to reduce the number of global memory accesses and the number of idle threads. This reduction is achieved by having threads on the same warp processing edges of the same node, that are on continuous memory positions. Currently only three classes are used, where the large nodes are processed by a single warp, medium size nodes are processed by only some threads of the warp, lastly small nodes are processed by a single thread.

The best number of edges for each partition depends on the structure of the graph. Initially the partition limits were defined as 96 or more of large edges and 8 or more edges for medium edges. When using medium graphs the warp was divided in 8 parts, meaning that each node is processed by only 4 threads. In social graphs only a few nodes have a large number of edges, meaning that the best performance kernel has a low usage.

In Table 4.4 other possible partition parameters where tested for both soc-liveljournal and com-friendster. For large nodes it was noted that reducing the limit from 96 to 16 edges yielded improvements on both tested graphs. This indicates that if a node has more than 16 edges, or half a warp, it is
Table 4.4: Execution (wall) time with different partitioning parameters

<table>
<thead>
<tr>
<th>Warp Division</th>
<th>Parameters</th>
<th>CUDA execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Small nodes</td>
<td>Medium nodes</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>96</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>

beneficial to use the entire warp for that thread. For the limit of medium edges it was tested with both 4 and 8 edges, both giving very similar results on both graphs. This test was repeated for a warp division of just 4, meaning that each node would be processed by 8 threads instead. A slight improvement was achieved on the soc-livejournal1 graph, but slightly worst performance with the com-friendster. In the end the combination of 8 divisions per warp, 8 edges as the limit for medium nodes and 16 edges as the limit for large nodes was chosen, as it yield the best performance on the largest graphs.

4.2.3 Caching the rank values

As described in Section 3.1.2 the implementation of the PageRank was changed. The rank transference done through each node is computed before the beginning of the SpMV operation. This process was called rank cache, as in effect it caches the multiplication of the previous rank with the dampening factor and diving by number of edges of the source node. When measuring the performance increase of this step, Table 4.5, it was found to not only increase the speedup across all graphs but also being more relevant on larger graphs, Figure 4.1.

The rank cache process was also used in the parallel CPU implementation, where it also resulted in some performance improvement. The parallel CPU implementation also used the transposed graph’s matrix to avoid the atomic operations in the SpMV as described in Section 3.1.2, thus benefiting from precomputing the new rank transfer in each iteration.

This performance increase can be explained by reducing the number of memory accesses required by each node, and also reducing the total number of operations. With larger graphs it becomes less likely that each memory access is serving multiple threads at the same time, as only few nodes have an edge from a common node. In CUDA a global memory access will read (or write) several sequential positions. Sequential positions, on the columnIdx array, will likely refer to edges of the same node. Except for the large node class, where each memory access will likely serve all 32 threads of each warp. On the other node classes, different threads are processing different nodes, so of each global read only a few positions are actually needed at that time. In the saving fewer memory access represent an increased chance of having the information for the next edge on the CPU chip cache.

4.2.4 CUDA Streams

The usage of CUDA Streams to perform memory operations and computation concurrently was described in 3.1.6. Figures 4.2 and 4.3 show how most of the memory operations are hidden behind a kernel invocation, when processing two different graphs. Except for the first memory copy from host to
Table 4.5: Execution (wall) time of the rank cache operation

<table>
<thead>
<tr>
<th>Graph</th>
<th>CPU (s)</th>
<th>CUDA (s)</th>
<th>CUDA - cached (s)</th>
<th>Speed up</th>
<th>Speed up (cached)</th>
</tr>
</thead>
<tbody>
<tr>
<td>soc-Slashdot0902</td>
<td>0.0424</td>
<td>0.0258</td>
<td>0.0220</td>
<td>1.6424</td>
<td>1.9297</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>6.4489</td>
<td>1.7980</td>
<td>1.1164</td>
<td>3.5867</td>
<td>5.7766</td>
</tr>
<tr>
<td>com-orkut</td>
<td>1.9838</td>
<td>1.7830</td>
<td>1.0870</td>
<td>4.4822</td>
<td>7.3525</td>
</tr>
<tr>
<td>com-friendster</td>
<td>351.0906</td>
<td>67.1656</td>
<td>37.8621</td>
<td>5.2272</td>
<td>9.2729</td>
</tr>
</tbody>
</table>

Table 4.6: CPU vs CUDA PageRank execution (wall) time

<table>
<thead>
<tr>
<th>Graph</th>
<th>CPU (s)</th>
<th>Parallel CPU (s)</th>
<th>CUDA (s)</th>
<th>Speedup (parallel CPU)</th>
<th>Speedup (CUDA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>soc-Slashdot0902</td>
<td>0.0424</td>
<td>0.0290</td>
<td>0.0220</td>
<td>1.4630</td>
<td>1.9297</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>6.4489</td>
<td>3.3148</td>
<td>1.1164</td>
<td>1.9455</td>
<td>5.7766</td>
</tr>
<tr>
<td>com-orkut</td>
<td>1.9838</td>
<td>7.9919</td>
<td>1.0870</td>
<td>2.2594</td>
<td>7.3525</td>
</tr>
<tr>
<td>com-friendster</td>
<td>351.0906</td>
<td>164.3603</td>
<td>37.8621</td>
<td>2.1361</td>
<td>9.2729</td>
</tr>
</tbody>
</table>

Figure 4.1: PageRank speedup increase with rank cache
<table>
<thead>
<tr>
<th>Graph</th>
<th>CPU (s)</th>
<th>CUDA (s)</th>
<th>Speedup (CUDA)</th>
<th>Speedup (parallel CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA-HepTh</td>
<td>3.8621</td>
<td>0.8886</td>
<td>4.3465</td>
<td>2.7385</td>
</tr>
<tr>
<td>Soc-Slashdot</td>
<td>431.6054</td>
<td>175.7743</td>
<td>2.4555</td>
<td>1.4796</td>
</tr>
<tr>
<td>Soc-LiveJournal1(est.)</td>
<td>4.9384 · 10^6</td>
<td>1.5393 · 10^6</td>
<td>3.2081</td>
<td>2.5219</td>
</tr>
</tbody>
</table>

Table 4.7: Betweenness centrality execution (wall) time in CUDA

the device and a few others, all memory operations happen at the same time that a kernel is running.

In Figure 4.2 there are two spots where the memory operations actually delay the invocation of the respective kernels. This happens because partitions are not guaranteed to have the same number of edges, so one partition can take more time to copy than another. These are partitions with large nodes, with no upper bound on the node size, therefore nodes in those partitions should have a high number of edges, taking for that reason a longer time to copy. A solution for this would be to create the partitions based on the number of edges, and also to sort the nodes by size, as it would contribute to similar sized partitions that would have a similar execution size.

It is also interesting to note that in the majority of the pairs of memory copy and kernel invocations there is a lapse between the end of the copy and the beginning of the corresponding kernel, with the exception of the partitions referred above. This time interval happens because the previous kernel is still being run. Meaning that the memory operations will not represent a bottleneck even with faster kernels.

### 4.3 Betweenness Centrality

With the betweenness centrality the achieved speedups over the single CPU implementation are small. It could be argued that the same values could be achieved with a multicore CPU. In fact this is the case with current GPU betweenness centrality implementations [1]. Relevant speedups over a multicore CPU where only achieved when using optimized data structures to improve thread utilization.

Looking in how the graphs are processed in the betweenness centrality and the necessary data structures, no apparent partition scheme would allow for graphs that are not completely inside the GPU memory to be processed. In Section 3.2.3 it is described a way to process each source in a different Stream Multiprocessor. While this solution is scalable across multiple GPU's [20] by taking advantage of the CUDA SM architecture, it is not capable of processing larger graphs as com-frienster1 due to the high memory required for each source.

To achieve maximum performance, using the 2048 threads per multiprocessor, the number of source nodes is necessary to be simultaneous processor is twice the number of SM of the GPU. Considering the size of the data structures used for the betweenness centrality and the number of simultaneous source nodes, the required total memory greatly increases when compared with the graph size. The soc-livejournal1 graph only has around 282MB, but with 20 simultaneous source nodes it is necessary to have 3.2GB of memory available on the GPU. This solution does not seem practical with the current hardware.

In spite of the low speedups achieved with the betweenness centrality, using a parallel approach for this problem might still be beneficial. This is evident on Table 4.8, where the CPU execution would take about two months to complete, while a parallel approach to the same problem would take just over two weeks. Using a more efficient approach, with specialized data structures, it should be possible to further reduce the parallel execution times [1].
Figure 4.2: A PageRank iteration of soc-livejournal using streams
Figure 4.3: A PageRank iteration of com-friendster using streams.
Figure 4.4: Betweenness centrality speedup comparison

<table>
<thead>
<tr>
<th>nNodes</th>
<th>CPU (s)</th>
<th>Parallel CPU (s)</th>
<th>CUDA (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>537.3114</td>
<td>215.4100</td>
<td>1.6971</td>
</tr>
<tr>
<td>32,768</td>
<td>33,329.2340</td>
<td>13,236.8417</td>
<td>10,402.7689</td>
</tr>
<tr>
<td>65,536</td>
<td>66,782.4897</td>
<td>26,477.0303</td>
<td>20,916.9217</td>
</tr>
<tr>
<td>1.3107 · 10^6</td>
<td>NaN</td>
<td>NaN</td>
<td>41,652.0177</td>
</tr>
<tr>
<td>2.6214 · 10^6</td>
<td>NaN</td>
<td>NaN</td>
<td>83,225.0354</td>
</tr>
<tr>
<td>4.8476 · 10^6</td>
<td>4.9384 · 10^6</td>
<td>1.9582 · 10^6</td>
<td>1.5393 · 10^6</td>
</tr>
</tbody>
</table>

Table 4.8: LiveJournal betweenness centrality execution (wall) time, the last value is estimated
4.4 Summary

This chapter described how the parallel implementations were evaluated and how the obtained results fared against the initial assumptions.

The PageRank implementation provided very interesting results. It allows graphs larger than the GPU memory to be processing in a GPU environment, with almost no drawbacks in the communication costs. This is achieved by using CUDA Streams to execute memory operations concurrent to kernel invocations. The achieved speedup on the CUDA PageRank implementation is compared against a CPU parallel implementation in Figure 4.1, where it not only consistently outperforms the CPU parallel implementation but also has better performance on larger graphs. Using the rank cache to precompute the rank transferred across each edge, the parallel implementation can be further improved as shown in the speedups of the CUDA (cached) implementations. By having less global memory accesses done by each kernel it reduces the time threads are idle waiting for a global memory access to complete.

The betweenness centrality CUDA implementation did not prove to be usable for larger graphs. To achieve optimal GPU thread occupancy it requires a very large amount of memory. It also did not provide a considerable speedup against the single core CPU implementation, nevertheless the CUDA implementation was still faster than the parallel CPU implementation, as shown in Figure 4.4.
Chapter 5

Conclusions

This work studies processing graph processing on a GPU, with special emphasis on graphs larger than the GPU memory. One of the main problems with processing large graphs on a GPU is that the graph has to be partitioned first. The chosen partitioning scheme has to take into consideration the memory access pattern of implemented algorithms.

The CUDA Streams are used in this work to hide memory copy operations with the execution of the kernels, processing other graph partitions already on the GPU memory. This approach has not been referred in any previous work addressing graph processing on GPUs, but it is a common practice in CUDA environments [23]. Furthermore, previous work in computing the PageRank, on a GPU, speculated that computing the kernels would always be faster than the memory transfers, [17, 18].

This work shows that, for the PageRank, it is possible to process graphs larger than the GPU memory. This is achieved by partitioning the graph so that each partition can fit inside the GPU memory. Regarding the betweenness centrality, the implementation described in this work only achieves small speedups against a single CPU core and also is not capable to process graphs larger than the GPU memory.

To test the performance of the GPU implementations, these implementations were compared against a single core CPU implementation, as well a parallel CPU implementation running on CPU 8 threads. The dataset contained both graphs that where far smaller than the GPU memory but also graphs, that even when compressed, had a size larger than the total GPU memory.

The PageRank results are satisfactory as they are better than a parallel multi-core CPU implementation, and could be even further optimized. On the other hand, processing the betweenness centrality in the GPU has only a small speedup compared to an equivalent parallel CPU implementation. Nevertheless, this small speedup is significant when considering the total execution time of the algorithm.

5.1 PageRank

The parallel PageRank implementation on a CUDA architecture needs the transposed graph, to have easy access to all incoming edges of each node. Splitting the nodes in different classes according to their size allows for specialized SpMV kernels with more coalesced memory operations and less idle threads. The graph is further partitioned by rows so that two partitions can be simultaneously stored in the GPU. CUDA Streams are used to allowing data transfer from the host to the GPU at the same time a running kernel is processing another partition.
The resulting CUDA implementation is capable of dealing with graphs that are bigger than the available GPU memory, in the same way that smaller graphs are handled in previous works. The communication cost between host and GPU is hidden by kernel computations, except for the initial transfer in each iteration. For most of the partitions, the communication time is lower than the time it takes to process the previous partition. Meaning that if the kernels are improved the communication will still not be a bottleneck for the problem.

This partition scheme works as long as all auxiliary data structures, with a space complexity of $O(n)$, are fully stored inside the GPU memory, plus enough space for two partitions. While increasing the number of partitions may allow for larger graphs, using small partitions the GPU may not be fully occupied, and launching a high number of kernels will impact the performance. In the implemented partition model only the edge list is partitioned, meaning that there is a limit on the number of nodes on the graph, at around $10^5$ nodes. For graphs that large, it may be necessary to apply a tiling partition scheme to the graph, where the graph is not only partitioned in rows but also by columns.

Regarding the implementation of the SpMV kernels, a mechanism to reduce non coalesced memory accesses was implemented. In the PageRank each node transfers equal partitions of its rank through all of its outgoing edges. It makes sense to compute this transfer only once and cache the value to be used when visiting each node edges. Just by using this optimization it was possible to increase the GPU implementation $1.77 \times$.

Comparing both the parallel CPU and the CUDA implementations it was possible to achieve speedups of $2 \times$, and up to $9.2 \times$ against a sequential CPU implementation.

### 5.2 Betweenness Centrality

The betweenness centrality traverses a large part of the graph several times, each time starting from a different node. This behavior does not allow for partition schemes as used in the PageRank, because it is necessary to have the entire graph accessible in memory. Even before consider auxiliary data structures the graph size is limited to the GPU memory size.

Both the graph traversal and the dependency accumulation steps require several synchronization points. The CUDA platform only allows for synchronization inside a block or on kernel invocations on the host. As processing a single source node does not have enough individual tasks to occupy all GPU SMs, it is better to use only one SM per source node and perform the synchronization at the block level.

With each block processing a different source node, it is necessary to have at least as many blocks as SM in the GPU chip. Ideally it should be at least two source nodes per SM, as each SM can manage up to 2048 threads and only 1024 threads are launched per block. Each source node being processed needs several auxiliary data structures. When processing several nodes at the same time, the required memory space quickly exceeds the available GPU memory for graphs with just a couple million nodes.

With in this betweenness centrality implementation, of the proposed work – balanced approach in [20],only small speedups were achieved. The execution time of the betweenness centrality can take months on graphs with million nodes in single CPU core. A speedup of three or four times, when using a GPU, significantly reduces the execution time.
5.3 Future work

There are some possible improvements to increase the GPU performance of both measures. In the PageRank it should be possible to use better balanced partitions with equivalent number of edges, as well presorting the graph by node size. These two steps should increase kernel efficiency, by having similar workloads for threads in the same warps. Sorting the graph by node size and reindexing all edges information to the new node order will also reduce the need for a redirection array used in SpMV kernels. Nodes within the same node size class will have consecutive ids. Applying these optimizations should not interfere with the proposed solution, as memory transfers can still be performed during the kernel computations. These transfers can also be further improved by using pinned memory on the host to improve transfer speeds on large memory operations.

In order to achieve better performance with the betweenness centrality possible improvements [1] are: balancing the number of edges on each node by using virtual nodes and also removing nodes with only one edge from the graph. To allow graphs larger than the GPU memory to be handled it is first necessary to allow the processing of each source to use more than just one block.
Bibliography


