Accelerating graph isomorphism queries in a graph database using the GPU

Simon Evertsson
Abstract

Accelerating graph isomorphism queries in a graph database using the GPU

Simon Evertsson

Over the last decade the popularity of utilizing the parallel nature of the graphical processing unit in general purpose problems has grown a lot. Today GPUs are used in many different fields where one of them is the acceleration of database systems. Graph databases are a kind of database systems that have gained popularity in recent years. Such databases excel especially for data which is highly interconnected. Querying a graph database often requires finding subgraphs which structurally matches a query graph, i.e. isomorphic subgraphs. In this thesis a method for performing subgraph isomorphism queries named GPUGDA is proposed, extending previous work of GPU-accelerating subgraph isomorphism queries. The query performance of GPUGDA was evaluated and compared to the performance of storing the same graph in Neo4j and making queries in Cypher, the query language of Neo4j. The results show large speedups of up to 470x when the query graph is dense whilst performing slightly worse than Neo4j for sparse query graphs in larger databases.
# Contents

1 Introduction ................................. 4  
  1.1 Report structure .......................... 4  

2 Background ................................. 6  
  2.1 Graphs .................................. 6  
  2.1.1 Isomorphism ............................ 7  
  2.1.2 Isomorphism in subgraphs ............... 7  
  2.2 NoSQL .................................. 7  
  2.3 The NoSQL subtype: Graph databases ....... 8  
  2.4 The graph database Neo4j ................. 8  
  2.5 Cypher, the query language of Neo4j ....... 9  
  2.6 GPU computing ........................... 9  
  2.6.1 What may theoretically be gained from GPU computing . 9  
  2.6.2 GPU design and GPGPU execution ....... 10  
  2.7 Related work .............................. 11  
  2.7.1 Accelerating databases with the GPU .... 11  
  2.7.2 Finding isomorphic subgraphs ......... 11  

3 The subgraph isomorphism finder GpSM .... 13  
  3.1 Data representation of the graphs ......... 13  
  3.2 Candidate vertex initialization ............ 14  
  3.2.1 The candidate checking kernel .......... 14  
  3.2.2 Candidate collection .................... 14  
  3.2.3 Candidate neighborhood exploration .... 15  
  3.3 Refinement of candidate vertices ........... 15  
  3.4 Candidate edge assembly ................... 16  
  3.5 Candidate edge joining ..................... 17  

4 Implementation .............................. 18  
  4.1 Expanding the graph representation of GpSM .... 19  
  4.1.1 Node labels ............................ 20  
  4.1.2 Relationship types ...................... 20  
  4.2 Conversion of Neo4j data .................. 20  
  4.3 Initialization of query node candidates .... 24  
  4.4 Further refinement of found node candidates .. 26  
  4.5 The generation of candidates for query relationships .... 26  
  4.6 Combining and pruning candidate relationships to form final solutions ............... 27  
  4.6.1 Representation of partial and final solutions .... 28  
  4.6.2 Initializing the partial solutions ........ 28  
  4.6.3 Combining candidate relationships with partial solutions .... 29  
  4.6.4 Pruning invalid solutions .............. 30  

5 Performance results ........................ 32  
  5.1 The databases used in the experiments .... 32  
  5.2 Generating experiment query graphs ....... 32  
  5.3 Measurement of run times and speedup ..... 33  
  5.4 Experiment environment .................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5</td>
<td>Experiments on the Dr Who database</td>
<td>34</td>
</tr>
<tr>
<td>5.6</td>
<td>Experiments on the Movies database</td>
<td>36</td>
</tr>
<tr>
<td>5.7</td>
<td>Experiments on the Yeast database</td>
<td>38</td>
</tr>
<tr>
<td>6</td>
<td>Discussion and future work</td>
<td>40</td>
</tr>
<tr>
<td>6.1</td>
<td>The performance of Cypher queries and when to use GPUGDA</td>
<td>40</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Neo4j indices</td>
<td>40</td>
</tr>
<tr>
<td>6.2</td>
<td>Out of memory errors</td>
<td>40</td>
</tr>
<tr>
<td>6.3</td>
<td>The necessity of candidate node refinement phase</td>
<td>41</td>
</tr>
<tr>
<td>6.4</td>
<td>Integration with Neo4j and beyond</td>
<td>41</td>
</tr>
<tr>
<td>6.5</td>
<td>Array gathering may be performed on the GPU</td>
<td>42</td>
</tr>
<tr>
<td>6.6</td>
<td>Difficulties and feasibility of GPU-acceleration</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>Conclusions</td>
<td>44</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Performance experiment results</td>
<td>48</td>
</tr>
<tr>
<td>A.1</td>
<td>Result files from experiments on the Dr Who database</td>
<td>48</td>
</tr>
<tr>
<td>A.2</td>
<td>Result files from experiments on the Movies database</td>
<td>49</td>
</tr>
<tr>
<td>A.3</td>
<td>Result files from experiments on the Yeast database</td>
<td>49</td>
</tr>
</tbody>
</table>
1 Introduction

When the paradigm shift from single-core to multi-core processors occurred, a number of new research areas were born. One such interesting subject is the usage of GPUs for other things than graphics, so called General Purpose GPUs (GPGPUs). One field which is growing with the increasing interest of GPGPUs is the usage of GPUs to increase database performance. Attempts to accelerate traditional SQL-databases has been made and have shown promising results of about 2-20x speedup for common database-operations[14] [2].

Relational databases have been the most common database type for many years. Today other options exists, namely NoSQL-databases. A sub-category of NoSQL-databases are graph databases. In a graph database, data is not stored in tables as in a relational database. Instead the data and relations are stored as labeled nodes and edges in a graph.

The popularity of graph databases is increasing [11]. Their popularity may be explained by the increasing interest in areas such as social network analysis, customer behavior analysis etc. [5][1]. Typical queries in these areas would include some form of search pattern matching, e.g. ”how many common friends does person x have with person y?”.

Such queries makes the need for efficient graph pattern matching, also known as graph isomorphism search, very important to maintain good performance. For this reason, this thesis work tries to answer the question: May graph pattern matching accelerated with a GPU gain any performance benefits over CPU-based pattern matching? Methods for GPU-based graph pattern matching algorithms are studied. Based on the study, a prototype of one method [27] was implemented to investigate GPU-based pattern matching query performance and compare it with the built in pattern matcher in the graph database system Neo4j [22]. The performance of the prototype was evaluated and compared against corresponding Neo4j pattern matching queries.

1.1 Report structure

This thesis report is structured in seven sections in addition to this introductory section.

In Section 2 the background to the thesis work is presented. Clarifications about graph databases, and more particularly the graph database Neo4j, graph isomorphism and GPU-computing are introduced. Section 2 furthermore discusses previous work in the field of utilizing the computing power of the GPU to accelerate databases in general.

Section 3 introduces GpSM which is the method which the prototype in this thesis work builds upon. It is an algorithm that uses the GPU to find all subgraphs which are isomorphic to, i.e. structurally matches, a specified query graph within a larger data graph.

The details of the prototype created in this thesis work, called GPUGDA, are explained in Section 4. Modifications and additions to the original GpSM algo-
rithm to make it work alongside an existing graph database as well as to explain how the algorithm was implemented can be found in that section.

The purpose of Section 5 is to compare the performance of GPUGDA in comparison to the performance of corresponding queries to Neo4j databases using Cypher, the query language of Neo4j. These performance experiments consists of several graph isomorphism queries with varying graph database- and query graph sizes.

In Section 6 the results of the performance experiments and the development process are discussed. Section 6 also raises some candidates for future work. And finally the work in this thesis work is concluded in Section 7.
2 Background

To give the reader a better understanding of the problem and what graphs, isomorphism, graph databases and GPU-computing is, this section will explain the concepts.

2.1 Graphs

The simplest from of a graph $G$ contain only two things, a set of vertices $V = \{v_1, v_2, \ldots \}$ and a set of edges $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), \ldots \}$ that describe connections between vertices. Such graphs are called undirected graphs since the order of vertices in an edge pair does not describe a direction, but only which vertices are connected to each other. An example may be seen in graph 1 in Figure 1.

The counterpart of undirected graphs are directed graphs. An example is seen in graph 2 in Figure 1. Directed graphs share the same structure as undirected graphs. The difference is that the edge pairs are interpreted in another way. The order of the vertices in an edge pair decides the direction of the edge. For example the edge pair $(v_1, v_2)$ tells us that the edge goes from vertex $v_1$ to vertex $v_2$. For the edge pair $(v_2, v_1)$ it is the other way around. The edge goes from vertex $v_2$ to vertex $v_1$.

The graphs that mostly resemble the property graph model are the directed labeled graphs as in graph 3 in Figure 1. They share the same structure and are interpreted in the same way as directed graphs, but with one addition. Each vertex in the vertex set $V$ has one or more labels in the label set $L = \{(v_1, A), (v_2, A), (v_2, B), \ldots \}$. A more formal definition of a directed labeled graph can be seen in Definition 1 based on the definition of an undirected labeled graph in [27].

**Definition 1.** A directed labeled graph $G$ is a quadruplet $G = (V, E, L, l)$ where $V$ is a set of vertices, $E$ is a set of ordered pairs of vertices, called edges, $E = \{(u, v) | u, v \in V \}$, $L$ is a set of labels and $l$ is a mapping function $l : V \rightarrow L$ which maps vertices in $V$ to labels in $L$. 

![Figure 1: An example of a simple undirected graph (1), a directed graph (2), and a directed labeled graph (3)](image-url)
2.1.1 Isomorphism

In graph theory [15] two graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are said to be isomorphic if they are structurally identical. That is, there exists a function $f$ and a function $g$ such that $f : V_G \rightarrow V_H$ and $g : E_G \rightarrow E_H$, where $f$ maps each vertex in $V_G$ to a single vertex in $V_H$ and $g$ maps each edge in $E_G$ to a single edge in $E_H$. Furthermore for each edge $e$, if $e$ contains a starting vertex $v$ then $g(e)$ contains a starting vertex $f(v)$. An example of two isomorphic graphs can be seen in Figure 2.

![Figure 2: An example of two isomorphic graphs](image)

2.1.2 Isomorphism in subgraphs

Even though isomorphism for entire graphs may be interesting in itself it is hard to find any practical uses for graph databases. Something that is more common in a graph database are searches for graph patterns in a large graph. This is called subgraph isomorphism. Given a graph $G$ and a graph pattern $Q$ we want to find all subgraphs in $G$ that match the graph pattern $Q$, i.e. all subgraphs of $G$ which are isomorphic to $Q$.

2.2 NoSQL

The first mention of the term NoSQL(Not only SQL) databases appeared in the beginning of the 21st century [29]. By then the relational databases had already been around for several decades. They came as a response to the increasing demand of web applications in Web 2.0 which required good scalability due to the increasing number of users and data. Today there are a number of different database categories which addresses themselves as NoSQL databases. These categories include, among others, document stores such as MongoDB or OrientDB and object databases such as Amos II.

The reason why so many different NoSQL-databases exist is because of the fact that no database type fits great for all applications. With the NoSQL-approach you choose your database to fit your application instead of trying to adapt the application to your database.
2.3 The NoSQL subtype: Graph databases

One category of NoSQL-databases is graph databases. In a graph database the data is stored in nodes with directed edges in-between. These nodes and edges may have labels which give them a conceptual context [32]. They may also have properties that describe them further. An example can be seen in Figure 3. A node labeled Car has a property Color being Red. Another node is labeled Person and has a property Name, which is set to Simon. An edge between the two nodes is labeled Owned by and has a property Since, which is set to 2015-05-08. This tells us that the person named Simon has owned a red car since 2015-05-08. This representation is called the property graph model.

![Figure 3: An example of a small graph that tells us that the red car is owned by person Simon since 2015-05-08](image)

The main difference between graph databases and traditional relational databases is how relationships are handled. To find relational or structural patterns in a relational database many join-operations may have to be performed. In a graph database edges, or relationships, may be seen as foreign keys in a relational database. Each node holds a reference to a list of relationships which the node has to other nodes. One could look at these relationship lists as pre-computed join-operations [20]. Since the joins are already stored along with the data in a graph database, structural queries made on a graph database outperforms similar queries made on a relational database [30]. This result is due to the fact that graph databases are designed to perform well for domains which have many relations in their data e.g. social networks [5] or e-commerce [1].

One drawback of graph databases are their maturity level. Relational databases have acquired both robustness and consistency over the years and most developers are familiar with the existing query languages. Another drawback is that the freedom of the schema-less approach of graph databases may lead to inconsistencies in the data. So if the data being stored is already well structured, not expected to evolve and does not contain many relationships between data elements, then graph databases may not be the best choice.

2.4 The graph database Neo4j

The graph database which was chosen to be accelerated in this thesis project was Neo4j [22]. Neo4j is open source and one of the most popular graph databases.
2.5 Cypher, the query language of Neo4j

To find complex patterns in a relational database many join-operations must be performed, as explained in Section 2.3. The strength in a graph database lies in the ability to efficiently find complex patterns in the stored data. In Neo4j graph searches are specified with the query language Cypher. Cypher is a language with SQL-similarities and describes graph patterns matched by the query.

Since a Cypher query searches for patterns that matches a supplied one, all Cypher queries could be seen as graph isomorphism queries. An example of an isomorphism query issued in Cypher may be seen in Figure 4. This query matches all occurrences in the database of nodes labeled Car with a relationship labeled OWNED_BY, pointing to a node labeled Person. The result of the query returns the nodes for each isomorphic sub graph.

```sql
MATCH (car:Car)-[:OWNED_BY]->(person:Person)
RETURN person, car;
```

Figure 4: An example of a Cypher query which returns all persons which owns a car

2.6 GPU computing

The GPUs were originally designed for drawing triangles of different colors really fast. They have mainly been used for gaming and visualization purposes.

The first GPU which had general purpose capabilities was the Nvidia 8800 [23]. With the 8800 came the CUDA library [24], which opened up the possibility to run not only graphics computations on the GPU, but also utilize the computing power of the GPU to do general purpose computations.

2.6.1 What may theoretically be gained from GPU computing

Today General Purpose GPU computing, or GPGPU computing, is a well established alternative to regular CPU-computing, because of very favorable price/performance ratio with high energy efficiency.
Figure 5: Specifications for the nVidia GTX 970 and Intel Core i7 4790K

<table>
<thead>
<tr>
<th></th>
<th>nVidia GTX 970</th>
<th>Intel Core i7 4790K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Cores</td>
<td>1664</td>
<td>4</td>
</tr>
<tr>
<td>Core frequency</td>
<td>1050MHz</td>
<td>4GHz</td>
</tr>
<tr>
<td>Peak SP performance</td>
<td>3494 GFLOPS</td>
<td>512 GFLOPS</td>
</tr>
<tr>
<td>Power usage</td>
<td>145W</td>
<td>88 W</td>
</tr>
<tr>
<td>GFLOP per Watt</td>
<td>24</td>
<td>5.8</td>
</tr>
<tr>
<td>Price</td>
<td>329$</td>
<td>339$</td>
</tr>
</tbody>
</table>

Figure 5 shows a comparison of specifications of the GPU nVidia GTX 970 and the CPU Intel Core i7 4790K. From the table we can see some interesting things. For roughly the same price we could theoretically gain almost 7x raw computing power from the GPU. Conversely this is a huge money saver for data centers. The same computing power could theoretically be achieved for 1/7th of the price if the data center was built with GPUs.

Another aspect is the power efficiency. The GPU in Figure 5 theoretically could yield the same performance as the CPU but at 1/4th of the power usage.

The reason why GPUs are not used for every task imaginable is because not every task suits well for GPU-processing. To get close to the theoretical speedup the task must work well with the GPU design.

2.6.2 GPU design and GPGPU execution

To successfully utilize GPU-computing, knowledge of its benefits and limitations is required. A GPU is designed to perform many computationally heavy operations really fast in parallel to achieve a high throughput. To do this, GPUs contain many slow cores as opposed to CPUs which contain a few fast cores.

GPGPU programs consists of so called kernels. Kernels can be seen as functions or procedures of a GPGPU program. A typical kernel execution is performed in four steps. The CPU-side of the program, which is also known as the host-side, allocates memory buffers on the GPU, which is also called the device. These memory buffers will contain the input data and, after the execution has finished, output data. The host fills the allocated input memory buffers with the input data that the kernel will do something with. When all of the needed input data has been copied to the device memory the host schedules an execution of the kernel with pointers to the allocated buffers as arguments.

The device will then execute the kernels in a specified number of GPU-threads. The result of the execution is written to one or more of the output buffers. To be able to read the result of the kernel execution, the host must copy the data in the output memory buffers back to the host memory.

The kernel execution allows for many parallel computations which may result in a performance increase depending on the algorithm. However the drawbacks of
GPUs lies in their limited memory management and the cost of thread synchronization and branch prediction. Another drawback is the low bandwidth while transferring data to and from the GPU. If a kernel execution involves many data transfers the data transfer bandwidth may become the performance bottleneck of the execution.

Another important aspect of GPU memory management is the way a GPU thread accesses the memory. Suppose the following scenario: When a GPU thread \( t_n \) reads data \( d[i] \) from the global GPU memory an entire segment of data is loaded along with \( d[i] \). If the subsequent threads \( t_{n+1}, t_{n+2}, \ldots \) does not access any data in the previously loaded segment but rather each starts loading data from addresses far away from each other, each thread will cause a new segment to be loaded from memory. This may have a considerable negative performance impact. A solution to this is issue is memory access coalescing. Perfectly coalesced memory accesses means that all data in a segment loaded from memory is utilized at once. To achieve good GPU performance one should strive to coalesce memory accesses.

### 2.7 Related work

In this subsection some related work on GPU-acceleration of databases and graph isomorphism algorithms are presented. Since the implementation solution in this thesis work is mostly based on the subgraph isomorphism finder GpSM [27] it is explained and summarized in its own section (Section 3).

#### 2.7.1 Accelerating databases with the GPU

Database acceleration is a wide area since the area of databases in itself is wide. In 2010 Bakkum and Skadron [2] published an article where they had accelerated SELECT-queries in a modified version of a SQLite database using the GPU with good results. Other work on acceleration of relational queries have also yielded fruitful results [9][14]

#### 2.7.2 Finding isomorphic subgraphs

In 1976 Ullman published his well cited article that described an algorithm to determine subgraph isomorphism [28]. More recent work has built upon his algorithm such as VF2 [4], GraphQL [10], QuickSI [25], GADDI [33], and SPath [34]. However these algorithms suffer from a common problem with the way the matching order of the query graph is selected. This affects the search performance greatly [12]. The algorithm TurboISO [6] tries to deal with this issue by proposing candidate vertex region exploration. The definition of a candidate vertex as defined in [27] can be seen in Definition 2.

**Definition 2.** Given a query graph \( Q = (V, E, L, l) \) and a data graph \( G = (V', E', L', l') \), a vertex \( v \in V' \) is called a candidate of a vertex \( u \in V \) if
\( l(u) = l(v) \), \( \text{degree}(u) \leq \text{degree}(v) \) where \( \text{degree}(u), \text{degree}(v) \) are the number of vertices connected to edges starting in vertex \( u \) and \( v \) respectively. The set of candidates of \( u \) is called candidate set of \( u \), denoted as \( C(u) \).
3 The subgraph isomorphism finder GpSM

In a recently published article Tran et al. [27] have built upon the results of TurboISO and created an algorithm they call GpSM for finding subgraph isomorphisms using a GPU. Their algorithm is based on a so called filter-and-join technique. The filtering stage consists of two steps. First candidate vertices are found for each vertex in the query graph, then the set of candidate vertices is refined to remove candidates which will yield invalid results. After the filtering phase has been performed the joining phase starts. In the joining phase candidate vertices are assembled to form candidate edges. These candidate edges are then combined and validated to form the final solutions. The details of the different phases are explained below.

3.1 Data representation of the graphs

In GPsM the data is stored as a directed, labeled graph. The vertices of the graph are enumerated and the labels and outgoing arcs of each vertex are stored in three arrays in vertex enumeration order: The label array stores the labels, the edge array stores the outgoing edges (adjacency lists), and the edge index array stores the indices of the starting positions of the outgoing edges in the edge index array.

To find the adjacency list of a certain vertex, enumerated \( n \), the \( n \)-th element of the edge index array contains the starting index, \( I_n \), of the adjacency list of vertex \( n \). The element \( n + 1 \) of the edge index array, \( I_{n+1} \) is the starting index of the adjacency list of vertex \( n + 1 \). Hence \( E_n = S_{n+1} - 1 \) is the end index of the adjacency list of vertex \( n \). The label of vertex \( n \) is the \( n \)-th element in the label array. Figure 7 shows an example of how the data in Figure 6 is represented. The data related to vertex 2 in the graph is highlighted.

![Figure 6: A simple undirected graph with three vertices and three edges. Vertex 1 is labeled A, vertex 2 is labeled B, and vertex 3 is labeled C](image-url)
3.2 Candidate vertex initialization

The purpose of the candidate vertex initialization step is to find the vertices, called candidates, which may contribute to valid solutions and to omit those which may not. This is done to minimize the number of possible edge combinations in the joining step (see Section 3.5). The initialization is performed by using three GPU-kernels for all vertices in the query graph.

3.2.1 The candidate checking kernel

The first kernel is the candidate checking kernel. This kernel is supplied the data graph, a query vertex and a boolean matrix-like structure called the candidateset where the result of the kernel call is put. The true elements of the candidate set are indicators for whether a vertex in the data graph is a candidate.

If candidateset[u][v] = true then the vertex v in the data graph is a candidate for the vertex u in the query graph.

Suppose that the candidate checking kernel is called with data graph G, query vertex u and a candidate set. The kernel will then apply Definition 2 for each vertex v ∈ G. The result will be the row candidateset[u]. After the kernel has been called the query vertex u is marked as initialized. The candidate checking kernel will only be called if the query vertex u has not been initialized.

3.2.2 Candidate collection

When the candidate vertices for a query vertex u have been marked in the candidateset[u] row these candidates are collected into an array. This collection is done by performing the two-step-output procedure. The first step of the procedure performs a form of prefix sum [8] to produce an array with the indices, in the output array, to which the elements of the data being collected will be written. An example of a prefix sum can be seen in Figure 8. The second step of the procedure is to write the data being gathered to the correct index using the result of the index array produced in the first step.
GpSM applies the prefix sum of the two-step-output procedure on \( \text{candidate}_\text{set}[u] \). Since the candidate set only contains boolean values, \text{true} is interpreted as 1 and \text{false} is interpreted as 0 when the prefix sum is computed. The result of the prefix sum is an array with the output indices for the candidate vertices in the candidate array. Using the new output index array the result is written, in a second pass, to the correct index in the candidate array.

### 3.2.3 Candidate neighborhood exploration

In the candidate checking kernel a vertex is considered a candidate if it has a similar structure as the query vertex (i.e. same label and greater than or equal degree). However a vertex is not a \text{true candidate} unless its neighborhood is similar to the neighborhood of the query vertex it is a candidate of. The role of the \text{neighborhood exploring kernel} is to verify this criteria.

Using the collected candidate array and the query vertex \( u \) the kernel begins comparing the adjacent vertices of each candidate vertex \( v \) to the adjacent vertices of \( u \). If there exists a vertex \( u' \in \text{adj}(u) \) such that no vertex \( v' \in \text{adj}(v) \) is a candidate of \( u' \) then the candidate vertex \( v \) is not a \text{true candidate} and \( \text{candidate}_\text{set}[u][v] \) is set to \text{false} again. Otherwise \( v \) is a true candidate where there exists at least one candidate for each vertex in \( u' \in \text{adj}(u) \) within \( \text{adj}(v) \).

The final step of the candidate neighborhood exploration kernel is to mark these candidates in the \text{candidate}_\text{set} in the same way as in the \text{candidate checking kernel}.

When the kernel returns, the false candidates have been removed from the \text{candidate}_\text{set}. The adjacent vertices of the query vertex \( u \) have also been initialized when the kernel has returned. This means that they do not need to be initialized with the \text{candidate checking kernel} hence they are marked as \text{initialized}.

The candidate vertex initialization is finished when all vertices in the query graph have been initialized and had their neighborhood explored.

### 3.3 Refinement of candidate vertices

When the candidate vertex initialization is done Tran et al. states that there may still be false candidates in the \text{candidate}_\text{set}. To prune these false candidates a refinement step is performed. This is done in the same way as in the first part of the \text{candidate neighborhood exploration kernel}(see Section 3.2.3). The
candidates for each query vertex is tested for the candidate criteria. This process is repeated a limited number of times to not affect the overall performance negatively.

3.4 Candidate edge assembly

After the candidate vertex refinement step the candidate set is considered to only contain true candidates. To be able to produce the final solutions, candidate edges must be assembled using the candidate vertices. Definition 3 defines what a candidate edge is.

Definition 3. Given a query edge \((u, v)\). A data edge \((u', v')\) is a candidate edge for \((u, v)\) if \(u'\) is a candidate vertex for query vertex \(u\) and \(v'\) is a candidate vertex for query vertex \(v\).

These candidate edges are stored in a lookup table-structure, see Figure 9. One such lookup table will be produced for each edge in the query graph. The lookup tables consists of three arrays containing the start vertices, the end vertices and the end vertex indices (which serves the same purpose as the edge index array explained in Section 3.1).

Suppose that the candidate edges for query edge \((u, v)\) are being assembled into a lookup table such as the one in Figure 9. Producing the start vertex array of the lookup table is trivial since the number of candidate vertices of \(u\) is already known. However the number of end vertices and where to put them is not initially known. To counter this, GpSM utilizes the two-step output procedure explained in Section 3.2.2 once more. The procedure creates the end vertex index array and end vertex array of the lookup table.

The first step of the procedure is the counting step. For each candidate vertex \(u' \in \text{candidate.set}[u]\), a kernel counts how many of the vertices \(v' \in \text{adj}(u')\) is a candidate for \(v\). By performing a prefix sum the end vertex index array will be produced and the last element of this array is the total size of the end vertex array. In the second step of the procedure the elements of the end vertex index array is used to determine where the valid end vertices should be written to in the end vertex array.
3.5 Candidate edge joining

The last step in a GpSM query is the joining step. In this step the candidate query edges in the previous step are combined to form valid solutions. First the candidate edges of the query edge \((u, v)\) with the least number of candidate edges are picked to form initial partial solutions. The vertices \(u\) and \(v\) are then marked as visited. Next another query edge \((u', v')\) is picked such that either \(u'\) or \(v'\) or both have been visited. The candidate edges for \((u', v')\) are then combined with the existing partial solutions. Query edges continue to be picked based on this criteria until all the query edges have been visited. The remaining partial solutions are the final solutions.
4 Implementation

Tran et al’s solution for finding subgraph isomorphisms using the GPU (GpSM) showed promising results compared to earlier methods. To validate if their solution works with an existing database it was chosen as the method which would be adapted for a Neo4j database, in this project. This section explains how the solution in this project, which is referred to as GPUGDA (GPU Graph Database Accelerator), was implemented.

![Figure 10: A system overview of the GPUGDA prototype.](image)

Figure 10 illustrates the steps of the GPUGDA algorithm. It consists of five
phases. A query is issued by supplying a connection to a Neo4j database and a query graph to GPUGDA. The connection is created by using the embedded Neo4j Java [18] version. It provides means to access a Neo4j database as well as interfaces for creating nodes and relationships as they are represented in Neo4j. These node and relationship interfaces are used in query graphs as the one shown in Figure 10.

In the first phase of GPUGDA the connection to the Neo4j database is used to copy all nodes and their relationships from the database. The nodes and relationships of the database graph and the query graph are then converted to a data representation which better fits the GPU. The details of how this conversion is performed can be found in Section 4.2.

When the query and data graphs have been converted the four phases of the query execution starts. These phases are similar to the four phases of the GpSM algorithm, with some additions. The details of their implementation can be found in Section 4.3, Section 4.4 Section 4.5 and Section 4.6. When the final phase has finished its execution the result is returned.

4.1 Expanding the graph representation of GpSM

The definition of the graphs in Neo4j is different from the one described in Definition 1 in Section 2.1. A Neo4j graph is defined by Definition 4 below. Node and relationship properties are omitted in the definition since they are not used in a GPUGDA query.

Definition 4. A Neo4j graph \( G \) is a 6-tuplet \( G = (N, R, L, l, T, t) \) where \( N \) is a set of nodes and \( R \) is a set of ordered pairs of nodes, called relationships, \( R = \{ (u, v) | u, v \in N \} \). \( L \) is a set of label lists and \( l \) is a mapping function \( l : N \to L \) which maps nodes in \( N \) to a list of labels in \( L \). \( T \) is a set of relationship types and \( t \) is a mapping function \( t : R \to T \) which maps relationships in \( R \) to a relationship type in \( T \).

The nodes, node labels, relationships, relationship types, and properties in Neo4j are represented with complex data types to improve search performance in Java or Scala. These include many linked lists which are traversed when a Cypher query is performed [13]. The memory access patterns of a linked list may potentially be very irregular and scattered, or uncoalesced. GPUs are optimized to run fast on aligned data using coalesced memory accesses, as mentioned in Section 2.6.1. Hence linked lists are not well fitted for GPU computations. Arrays on the other hand are data types capable of having unit strides, which can easily be transferred to a GPU memory buffer. For this reason the Neo4j data is first converted to an array format, explained further down.

In GpSM the graph is represented with three arrays as mentioned in Section 3.1. To avoid the use of lists this representation was used in GPUGDA as well. However, in GPUGDA some additional features were added to mimic the functionality of the Cypher query language in Neo4j. Furthermore the names of the arrays have been updated to match the naming of Neo4j. The edge array is called the relationship array, the edge index array is called the relationship index array.
4.1.1 Node labels

The labels in Neo4j are represented with strings. To decrease the number of data needed to be transferred to the GPU device memory and to increase comparison performance, the labels were added to a label-to-integer dictionary. This dictionary ensures that all labels receive a unique integer representation.

The nodes (vertices) in GpSM have one and exactly one label, however nodes in Neo4j may be unlabeled or have an arbitrary number of labels. Hence the label array in GpSM has been expanded to contain a list of labels for each node instead of only one label. Coupled with the label list array is a label index array. This label index array contains the start indices of the label list of each node and is used similarly as the edge array explained in Section 3.1.

4.1.2 Relationship types

The relationships in Neo4j may have an optional type. To be able to query for patterns containing a relationship with a type an additional relationship type array has been added. This array is of the same size as and used in the same way as the relationship array. Relationships may have an associated type. The relationship types are, similarly to labels, represented with strings. Hence for the same reason as with labels a relationship type-to-integer dictionary is used to avoid transferring and comparison of strings.

4.2 Conversion of Neo4j data

The purpose of the conversion phase is to convert the Neo4j representation of the data to the equivalent array representation, which is better adapted for the GPU.

Assume that the graph in Figure 11 is being converted. The first step of the conversion phase copies all nodes with their relationships from the Neo4j database to the main memory. The nodes and relationships in Neo4j are represented with interfaces called Node and Relationship, respectively. When all nodes and relationships have been copied the conversion begins.

In GpSM nodes are identified by an index from 0 to \( N \) where \( N \) is the total number of nodes in the database. This index is used to reference the node-, relationship-, and label arrays (see Figure 7 in Section 3.1). Neo4J assigns unique numeric identifiers to each node and relationship. For example, if a node is assigned the number \( n \) as identifier when it is created the next created node would be assigned the number \( n+1 \). At first glance this identifier would seem like a good choice as a node identifier also in the GPU data representation. However, if a node or relationship gets deleted from the database the identifiers would get fragmented. Using these identifiers as indices in the array representation would create faulty behavior and potentially array boundary errors since wrong indices may be accessed. For example, if four nodes are inserted to an empty Neo4j database Neo4j would assign the identifiers 0, 1, 2, and 3 to the nodes respectively. These identifiers could be used to access the array representation correctly.
However, if the node with identifier 2 is deleted from the database the node array would only be of length 3. This means that when you want to access the data of node with identifier 1, the data for node 3 is actually accessed. And if the data for node with identifier 3 is sought, the access will cause an array boundary error.

GPUGDA solves this issue by mapping the internal Neo4j identifiers to temporary enumerated values, one for each node and relationship, when a new query is issued. These values are referred to as node identifiers and relationship identifiers. These identifiers have a unit stride and will range from 0 to $N - 1$ where $N$ is the number of nodes or relationships in the database. This assures that the data belonging to each node and relationship can be correctly accessed in the array format.

To simplify the node label, and relationship type lookups the node labels and relationship types are also assigned temporary enumerated values (referred to as label identifiers and relationship type identifiers). An example of this can be seen in Figure 11 and Figure 12 where labels A and B are assigned label identifiers 0 and 1 respectively. The relationships types LOVES and KNOWS are also respectively assigned relationship type identifiers 1 and 2.

All of the identifiers explained above are stored in dictionaries to be able to use the same identifiers while converting a Cypher-query and to recover the original internal Neo4j identifiers from the result when the query execution has finished.

Figure 11: A simple data graph with node 2 labeled A and B, node 7 labeled B, and node 4. Node 2 has relationships with types LOVES and KNOWS to node 4 and 7 respectively. Node 7 has a relationship to 4 without a type.
Figure 12: Example of the result when the data graph in Figure 11 has been converted to the array representation. Note that the node, relationship, label, and relationship type identifiers have been translated to query identifiers. In this example node 2 is identified as 0, node 7 as 1 and node 4 as 2 to be able to retrieve the correct data from the arrays. Label A is identified as 1 and label B is identified as 2. In the same way, relationship type KNOWS is identified as 1 and relationship type LOVES is identified as 2. The data related to node 7 has been highlighted.

When the nodes have been copied to the main memory the nodes and their relationships are converted to arrays on the CPU in a similar fashion as in GpSM, with the additions explained above. To be able to query the converted data graph the query graphs are also required to be converted to the GPU-friendly representation. Cypher queries are stated in the Cypher query language as explained in Section 2.5. Neo4j parses these queries and forms the intermediate query representations internally. Since the scope of this thesis project is not to efficiently parse Cypher-queries this step was omitted in GPUGDA. GPUGDA uses the same Neo4j interfaces in which the data graph is represented in before the conversion.

By using GPUGDA-specific implementations of these two interfaces the query graphs can be formed programmatically. An example of how a simple Cypher query is represented pre- and post-conversion can be seen in Figure 13, Figure 14, and Figure 16.

```cypher
MATCH (a:A) -[:KNOWS]-> (b :B)
RETURN a, b;
```

Figure 13: An example of a Cypher query which returns all nodes a, b where a is labeled A has a relationship with type KNOWS to b which is labeled B.
Figure 14: The graph representation of the Cypher query in Figure 13.

```
// Create the query graph object
QueryGraph queryGraph = new QueryGraph();

// Create node 0:[A] and add it to the query graph
QueryNode A0 = new QueryNode(0);
A0.addLabel(new QueryLabel("A"));
queryGraph.addNode(A0);

// Create node 1:[B] and add it to the query graph
QueryNode B1 = new QueryNode(0);
B1.addLabel(new QueryLabel("B"));
queryGraph.addNode(A0);

// Create the relationship and add to query graph
QueryRelationship A0_B1 =
    A0.createRelationshipTo(B1, RelationshipTypes.KNOWS, 0);
queryGraph.addRelationship(A0_B1);
```

Figure 15: Example of how the query graph in Figure 14 is formed programatically for GPUGDA.
The query graph generator generates queries as subgraphs based on the GPU friendly data graph, making both the data and query graphs represented with the same interfaces (see Section 5.2).

When both the query and data graphs have been converted to the array format the actual GPUGDA query processing begins.

### 4.3 Initialization of query node candidates

The same steps which the GpSM implementation uses to query the database are used in GPUGDA. However the algorithms have been slightly modified to be able to handle multiple labels and typed relationships, as required by Neo4j databases. The modified definition of candidate nodes can be found in Definition 5:

**Definition 5.** Given a set of query nodes $V$ and a set of data nodes $V'$, a node $v \in V'$ is called a candidate of a node $u \in V$ if for each $l \in l(u)$ there exists $l' \in l(v)$ such that $l = l'$ and $\text{degree}(u) \leq \text{degree}(v)$ where $\text{degree}(u), \text{degree}(v)$ are the number of nodes connected to relationships starting in node $u$ and $v$ respectively. The set of candidates of $u$ is called candidate set of $u$, denoted as $C(u)$.

These changes affects most parts of the implementation. In the initialization step the data nodes are compared with the query nodes to determine which nodes are candidates for the query nodes. In the check-step the candidate node definition has been updated. For a node to be candidate it must at least have all the labels which the query node has. If the query node lacks labels, this label validation is omitted. Algorithm 1 shows how the candidate checking kernel works.
Algorithm 1 The candidate checking kernel. Given a data graph $G$ and a query node $u$ the kernel finds all candidates for $u$ in node array $N$ of $G$. The $\text{candidate\_indicators}$ is updated with the result.

1: procedure check_candidates_kernel($G$, $u$, candidate\_indicators)
2: \hspace{1em} v = G.N[thread\_id];
3: \hspace{1em} labels\_match = true;
4: \hspace{1em} if $l(v) \neq \emptyset$ then
5: \hspace{2em} for all $q\_node$\_label $\in l(u)$ do
6: \hspace{3em} if $q\_node$\_label $\notin l(v)$ then
7: \hspace{4em} labels\_match = false;
8: \hspace{3em} break;
9: \hspace{2em} end if
10: \hspace{1em} end for
11: \hspace{1em} end if
12: \hspace{1em} candidate\_indicators[$u$][$v$] = labels\_match AND degree($u$) $\leq$ degree($v$);
13: end procedure

In the candidate neighborhood exploration step the relationship type is considered. A node which is adjacent to the candidate node is only explored if the query node lacks a relationship type or if the relationship to that adjacent node is of the same type. Considering the case where candidate node $u'$ for query node $u$ is related to a node $v'$ which is a candidate of $v$. If $t((u', v')) \neq t((u, v))$ then $(u', v')$ is not a valid candidate relationship for $(u, v)$. The new algorithm can be seen in Algorithm 2:

Algorithm 2 The candidate neighborhood exploration kernel. Given a query node $u$ and an array $\text{candidate\_array}$ containing the candidate nodes for $u$, the kernel explores the neighborhood of each candidate node to determine if it is a false candidate or not. The $\text{candidate\_indicators}$ matrix is updated with the result.

1: procedure explore_candidates_kernel($u$, candidate\_array, candidate\_indicators)
2: \hspace{1em} $u' = \text{candidate\_array}[thread\_id]$;
3: \hspace{1em} for all $v \in \text{adj}(u)$ do
4: \hspace{2em} if there exists no candidate for $v$ in $\text{adj}(u)$ then
5: \hspace{3em} candidate\_indicators[$u$][$v$] = false;
6: \hspace{2em} return;
7: \hspace{1em} end if
8: \hspace{1em} end for
9: \hspace{1em} for all $v \in \text{adj}(u)$ do
10: \hspace{2em} for all $v' \in \text{adj}(u')$ do
11: \hspace{3em} if $v'$ is a candidate of $v$ AND $t(u', v') = t(u, v)$ then
12: \hspace{4em} candidate\_indicators[$v$][$v'$] = true;
13: \hspace{3em} end if
14: \hspace{2em} end for
15: \hspace{1em} end for
16: end procedure
4.4 Further refinement of found node candidates

The node candidate refinement step works in the same way as in GpSM: lines 2-8 in Algorithm 2 for the candidate node array of each query node. This process continues until the candidate indicator matrix does not change anymore. However, it was discovered during the development that no query would cause the refinement step to change the candidate indicator matrix.

4.5 The generation of candidates for query relationships

The step where the candidate nodes are joined to form query relationship candidates has been modified compared to GpSM. Initially the GpSM representation of candidate relationships was used to store the candidate relationships. However, issues in the joining-step (see Section 4.6) arose when using the original representation. The issue was that if a data relationship was a candidate for several query relationships a solution could contain the same data relationship for multiple query relationships. This yielded many invalid results. To counter this, the candidate relationship representation was updated with an identifier for each candidate relationship. This identifier is simply the index of the relationship end node in the data graph. The updated representation of the candidate relationships can be seen in Figure 17.

Figure 17: Example of the GPUGDA representation of the candidates for a query relationship \((u, v)\). The start node array contains the candidates for \(u\) and the end node array contains the candidates of \(v\). Each candidate relationship is represented with an identifier.

The generation of these candidate relationships for each query relationship works in a similar way as in GpSM. It utilizes the two step output scheme where the number of candidate relationships is counted in the first step. Using the result of the counting, the end node index array is generated and the end node array is allocated with the correct size. In the second pass the end nodes and ids for the candidate relationships are assigned to the correct indices. The algorithms for the two kernels executed in the candidate relationship generation can be seen in Algorithm 3 and Algorithm 4. The generated candidate relationships are stored in a hash table where the identifiers of the query relationships are the keys.
Algorithm 3 The candidate relationship counting kernel. Given a query relationship \((u, v)\), the \texttt{start\_nodes} array contains the candidates for \(u\). The kernel counts for each \(u' \in \texttt{start\_nodes}\) the number of candidate relationships for \((u, v)\) which starts in \(u'\). The result for each count is written to the \texttt{candidate\_relationship\_counts} array.

1: \textbf{procedure} \texttt{COUNT\_CANDIDATE\_RELATIONSHIPS\_KERNEL}((u, v), \texttt{start\_nodes}, \texttt{candidate\_relationship\_counts})
2: \hspace{1em} \texttt{u'} = \texttt{start\_nodes[thread\_id]};
3: \hspace{1em} \textbf{for all} v \in \texttt{adj(u)} \textbf{do}
4: \hspace{2em} \textbf{if} v' \text{ is a candidate of } v \text{ AND } t(u', v') = t(u, v) \textbf{then}
5: \hspace{3em} \texttt{candidate\_relationship\_counts[thread\_id]++};
6: \hspace{2em} \textbf{end if}
7: \hspace{1em} \textbf{end for}
8: \textbf{end procedure}

Algorithm 4 The candidate relationship counting kernel. Given a query relationship \((u, v)\), the \texttt{start\_nodes} array contains the candidates for \(u\), and the \texttt{end\_node\_indices} array contains the starting indices for the end nodes in each candidate. The kernel writes the end nodes and relationship identifiers to the correct index to the \texttt{end\_nodes} and \texttt{relationship\_ids} arrays, respectively, for each found candidate relationship.

1: \textbf{procedure} \texttt{FIND\_CANDIDATE\_RELATIONSHIPS\_KERNEL}((u, v), \texttt{start\_nodes}, \texttt{end\_node\_indices}, \texttt{end\_nodes}, \texttt{relationship\_ids})
2: \hspace{1em} \texttt{u'} = \texttt{start\_nodes[thread\_id]};
3: \hspace{1em} \texttt{output\_index} = \texttt{end\_node\_indices[thread\_id]};
4: \hspace{1em} \textbf{for all} v \in \texttt{adj(u)} \textbf{do}
5: \hspace{2em} \textbf{if} v' \text{ is a candidate of } v \text{ AND } t(u', v') = t(u, v) \textbf{then}
6: \hspace{3em} \texttt{end\_nodes[output\_index]} = v';
7: \hspace{3em} \texttt{relationship\_ids[output\_index]} = \text{The end node index of } (u', v') \text{ in the data graph};
8: \hspace{2em} \texttt{output\_index}++;
9: \hspace{2em} \textbf{end if}
10: \hspace{1em} \textbf{end for}
11: \textbf{end procedure}

4.6 Combining and pruning candidate relationships to form final solutions

Given the hash table of candidate relationships created in the previous step of the relationship candidate generation, the final step in the algorithm is to generate the final solutions. This is done by combining the relationship candidates to form partial solutions and validating these solutions. The candidate initialization and refinement are performed to minimize the number of invalid combinations generated in this joining step. The joining step is divided into three parts: Solution initialization, solution combination generation, and solution pruning. The goal of the joining step is to produce the arrays containing the valid final solutions.
4.6.1 Representation of partial and final solutions

The representation of the partial solutions are not mentioned in Tran et al’s article. Hence GPUGDA uses its own way to represent solutions on the GPU. To maintain performance the previous data representations are mimicked, by using two arrays. By storing the solution nodes and relationships in this way the GPU threads may efficiently figure out where to write or validate a solution.

The first array is the solution node array. This array contains the data nodes $u'_i$ for each query node $u_i$ of the solutions. This means that the nodes of a solution are written in a subarray with a length equal to the number of query nodes, i.e. $\text{size}(G_q,N)$. The first index of the node subarray for solution $j$ would be $\text{size}(G_q,N) \cdot j$ and the last index would be $\text{size}(G_q,N)(j + 1) - 1$. The found solution node $u'_i$ for the node $u_i$ would then be placed at index $i$ in the subarray. In a partial solution the query nodes for which a data node have not been found are represented with $-1$.

The second array stores the relationships that have been used to form the solution. This array works in a similar way, namely the relationships of a solution are written in a subarray with a length equal to the number of query relationships, i.e. $\text{size}(G_q,R)$. The relationships to which a solution has yet been found are represented as $-1$. An example of the solution representation can be found in Figure 18.

<table>
<thead>
<tr>
<th>(Partial) Solution node array</th>
<th>SolNodes$_1$</th>
<th>SolNodes$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_0$</td>
<td>$u_1$</td>
<td>$u_2$</td>
</tr>
<tr>
<td>$1$</td>
<td>$-1$</td>
<td>$2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution relationship array</th>
<th>SolRel$_1$</th>
<th>SolRel$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_0$</td>
<td>$r_1$</td>
<td>$r_2$</td>
</tr>
<tr>
<td>$1$</td>
<td>$-1$</td>
<td>$-1$</td>
</tr>
</tbody>
</table>

Figure 18: Example of the representation of both partial and final solutions. Each solution is stored in sub-arrays of a solution node array $S_N$ and a solution relationship array $S_R$. Each node subarray $S_N(i)$ will contain the identifiers for a found data node $u'_i$ for each query node $u_i$. Each relationship subarray $S_R(i)$ will contain the identifier for a found data relationship $r'_i$ for each query relationship $r_i$.

4.6.2 Initializing the partial solutions

The joining step needs a starting point. This starting point is given by the initialization part that is executed on the CPU. By using the candidate relationships generated in the previous step the query relationship $(u,v)$ with the least number of candidate relationships is chosen as the initial relationship. The partial solution arrays are created using these candidate relationships. If the number of candidate relationships is $n$ then the initial solution node array and
solution relationship will be of size $size(G_q.N) \cdot n$ and $size(G_q.R) \cdot n$ respectively. These arrays are filled with $-1$ initially.

Each candidate relationship is considered a partial solution. Hence the start node $u'$ and end node $v'$ of each candidate relationship $(u', v')$ are written to the correct index for query nodes $u$ and $v$ into the next uninitialized subarray of the solution node array. Similarly the relationship identifier of $(u', v')$ is written to the correct index of relationship $(u, v)$ in the next uninitialized subarray of the relationship node array.

When all the candidate relationships have been written as a partial solution the relationship $(u, v)$ and its nodes $u$ and $v$ are marked as visited. The starting point of the joining step is then considered generated and the algorithm can continue with the combination- and pruning parts.

4.6.3 Combining candidate relationships with partial solutions

After the partial solutions have been initialized the execution starts combining and pruning the partial solutions picking a previously unvisited query relationship $(u, v)$. Whether the partial solution should be combined or pruned using this query solution is determined by the set of visited query nodes. If only one of either $u$ or $v$ has been visited then $(u, v)$ is combined with the current set of partial solutions. The combination is also performed using the two-step output scheme. First the number of candidate relationships which validly can be combined with each partial solution in the current set of partial solutions, are counted. Then a new set of partial solutions are created and the new combinations are generated into it.

The counting kernel is supplied the previously unvisited relationship $(u, v)$, the candidate relationships for $(u, v)$, and the current set of partial solutions. Each thread is responsible for counting the number of candidate relationships, which can validly be combined with one solution $s$ of the partial solutions. A candidate relationship $(u', v')$ can be validly combined with $s$ if either $s.nodes[u] == u'$ or $s.nodes[v] == v'$ depending on whether $u$ or $v$ has been visited. Furthermore the identifier of $(u', v')$ must not be among $s.relationships$ identifiers. If these criteria hold $(u', v')$ can be combined with solution $s$ and the combination count for $s$ is increased. The implementation can be seen in Algorithm 5.
Algorithm 5  The solution combination counting kernel. Given a query relationship \((u, v)\), its candidate relationships \(R_c\) and the current partial solutions \(S\) the kernel counts the number of different combinations which can be generated for each solution in \(S\) using the candidates in \(R_c\). Each thread is responsible for combinations for one solution in \(S\) and writes the result to the \(combo\_counts\) array.

1:     procedure COUNT_SOLUTION_COMBINATIONS_KERNEL((u, v), \(R_c\), S, combo_counts)
2:          for all \((u', v') \in R_c\) do
3:               if \(u' = S[thread\_id].nodes[u] \text{ OR } v' = S[thread\_id].nodes[v]\) then
4:                   if \(id(u', v') \notin S[thread\_id].relationship\_ids\) then
5:                       combo_counts[thread\_id]++;
6:                   end if
7:               end if
8:          end for
9:     end procedure

When the number of combinations for each partial solution has been counted a prefix scan is performed on the array containing the combination counts. This prefix scan yields an index array, which defines the output index intervals for the combinations of each solution \(s\) in the current set of partial solutions. The last element of the index array contains the number of solution combinations that can be generated, i.e. the size of the buffer which will hold the new combined partial solutions.

When the output index array and new solution buffer have been created, the final step of the combined solution is to actually generate the combinations. This is done in a new kernel where each thread generates a new solution \(s'\) for each valid candidate relationship \((u', v')\), based on the same criteria as in the counting kernel. The elements of the new solution \(s\) are updated by either \(s.nodes[u] = u'\) or \(s.nodes[v] = v'\) depending on whether \(u\) or \(v\) was the visited node. In the same time the relationship identifiers of the solution is updated by \(s.relationship = id(u', v')\). The new solution \(s'\) is then written to the correct new position in the new set of partial solutions \(S'\) by using the output index array. When the kernel returns the query relationship \((u, v)\) and its nodes \(u\) and \(v\) are marked as visited.

4.6.4 Pruning invalid solutions

If a previously unvisited query relationship \((u, v)\) is picked, where both of the nodes \(u\) and \(v\) have been visited, the current set of partial solutions should be pruned. The pruning step is also performed with two kernels. The first kernel counts the number of valid solutions the current set of partial solutions, counting the number of valid ones. The second kernel uses the results of the validating kernel to generate a new set of partial solutions where the invalid solutions have been pruned.

The solution validation kernel is supplied by the query relationship \((u, v)\), its relationship candidates \(R_c\), and the current set of partial solutions \(S\). Each kernel
thread is responsible to validate one solution \( s \in S \). The solution is considered valid if there exists a relationship \((u',v') \in R_c\) such that \( s._{\text{nodes}}[u] = u'\) and \( s._{\text{nodes}}[v] = v'\). Furthermore, \( id(u',v') \) must not be among \( s._{\text{relationships}} \) identifiers. If one of these criteria fails, the solution is invalid. An example of how the kernel works can be seen in Algorithm 6.

**Algorithm 6** The solution validation kernel. Given a query relationship \((u,v)\), its candidate relationships \( R_c \) and the current partial solutions \( S \), the kernel validates each solution in \( S \) using the candidates in \( R_c \). Each thread is responsible for validating one solution in \( S \) and writes the result to the \( \text{valid\_solution\_indicators} \) array.

```
1: procedure VALIDATE_SOLUTIONS_KERNEL((u, v), R_c, S, valid_solution_indicators)
2:  for all \((u', v') \in R_c\) do
3:     if \( u' = S[\text{thread\_id}]._{\text{nodes}}[u] \) and 
4:        \( v' = S[\text{thread\_id}]._{\text{nodes}}[v] \) and 
5:        \( id(u', v') \notin S[\text{thread\_id}]._{\text{relationship\_ids}} \) then
6:         valid_solution_indicators[\text{thread\_id}] = true;
7:         return;
8:  end if
9: end for
10: end procedure
```

By performing a prefix scan on the results of the validation the output index array is generated for where the valid solutions should be written to in the pruned set of partial solutions. The last index of this array will also contain the total number of valid solutions. The generated output index array is therefore used to create the buffer that will hold the pruned set of partial solutions. The pruning kernel is then executed using the output index array to copy a valid solution \( s \) from the old set of partial solutions \( S \) to the pruned set of solutions \( S' \). The identifier of the query relationship \((u,v)\) with which the solutions are currently being pruned is written to \( s._{\text{relationships}} \) in the new position of \( s \) in \( S' \). When the kernel returns from its execution the query relationship \((u,v)\) is marked as visited.

When all the query relationships have been visited, either in a combination or a pruning step, the set of partial solutions is considered the set of final solutions. This set is returned as the result of the query.
5 Performance results

One of the goals of the thesis was to improve the query speed of Neo4j databases. In this section the performance of queries to the GPUGDA implementation is compared to the corresponding Cypher query language in Neo4j.

5.1 The databases used in the experiments

In Lee et al’s article [12], where different methods of finding subgraph isomorphisms are compared, four data sets with different sizes and densities are used [12]. However due to limited time these data sets were not parsed and inserted into a Neo4j database to be included in the experiments of this project. Instead three databases with similar characteristics were used. These databases are the Dr Who- and Movie-database which are featured as example databases for Neo4j [19] as well as the Yeast-database [3].

The Dr Who database is a fairly small database containing information about the episodes, characters, and settings of the Dr Who-world. The Movie database is a larger database containing information about movies with their actors and directors. Finally the Yeast database is a medium sized database.

The graph densities of the databases are calculated by the formula $R/N(N-1)$ where $R$ is the number of relationships and $N$ is the number of nodes in the database [31]. Whether a graph is deemed dense or sparse depends on the situation since graph density is relative. A comparison of the properties of the three databases can be found in Table 19. If the Dr Who is considered a dense database in this case, the Movies database may be seen as a sparse database.

<table>
<thead>
<tr>
<th></th>
<th>Dr Who</th>
<th>Movies</th>
<th>Yeast</th>
</tr>
</thead>
<tbody>
<tr>
<td># of nodes</td>
<td>1060</td>
<td>63042</td>
<td>2361</td>
</tr>
<tr>
<td># of relationships</td>
<td>2286</td>
<td>106651</td>
<td>7182</td>
</tr>
<tr>
<td>Graph density</td>
<td>2.04e-3</td>
<td>2.68e-5</td>
<td>1.29e-3</td>
</tr>
<tr>
<td>Average outdegree</td>
<td>2.16</td>
<td>1.69</td>
<td>3.04</td>
</tr>
<tr>
<td>Max. outdegree</td>
<td>165</td>
<td>71</td>
<td>60</td>
</tr>
<tr>
<td># of unique node labels</td>
<td>0</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td># of unique relationship types</td>
<td>16</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Average number of labels per node</td>
<td>0</td>
<td>1.81</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 19: A table showing the properties of the different databases used in the experiments.

5.2 Generating experiment query graphs

To test the performance of the databases against query graphs of different characteristics a query graph generator was created. The query graph generator generates query graphs, using the query graph format seen in Figure 15, based...
on the existing data in the database to ensure that the queries return at least one result.

When generating queries the query generator randomly picks a node $u_r$, from the queries data graph, to use as a root node. The generator then performs a depth-first traversal starting from the root node adding the nodes and relationships found on the path to the query graph being generated.

To control the preferred size and density of the query graphs the query graph generator is supplied a node count $N$, a relationship count $R$, and a maximum query graph count $Q$. When the depth-first traversal is performed these parameters are used such that the resulting query graph will have $N$ nodes and least $R$ relationships. If a query graph with $N$ nodes and at least $R$ relationships cannot be found with $u_r$ as the root, the generation is retried using the next node $u_{r+1}$ as root node. If the number of query graphs found reaches $Q$ the query graph generator stops and returns the found query graphs.

5.3 Measurement of run times and speedup

Each experiment generated query graphs using the query graph generator with experiment specific node counts. To measure the impact of query graph density, each experiment was executed twice, once with dense query graphs and once with sparse query graphs. The density was controlled with different settings on the number of relationships $R$ of the query graph generator. For the dense query graphs the generated query graphs have a number of relationships $R$ of at least $2 \cdot N$ where $N$ was the node count of the query graph. For the sparse query graph the query graph generator was supplied $R$ and $N$ where $R = N - 1$, which is the minimum number of relationships needed to generate a query graph with $N$ nodes.

The GPU query and Cypher query were then executed for each generated query graph. For these executions two run times were measured, the GPU query run time and the Cypher query run time. The run times were measured using the CPU-time in Java.

To get a good distribution of different query graphs the query graph generator tries to generate 30 query graphs using the same node count and relationship count for each experiment. The average speedups for each of the different query graphs is then calculated.

Since the purpose of the experiments was to compare the performance of the GPU friendly graph and GPUGDA query representation with the corresponding Neo4J performance, the data conversion time was omitted in the experiments.

5.4 Experiment environment

The experiments were performed on a machine running OS X Yosemite 10.10.5 with an Intel Core i7-4578U 3.0GHz quad core CPU and 16GB RAM. The GPU
on the machine was an Intel Iris Graphics 5100. The experiment program was compiled and run with Java version 1.8.0_92.

5.5 Experiments on the Dr Who database

The purpose of the experiments on the Dr Who database was to test how the algorithm performed on a small database with a relatively high density. The result data of the performance experiments on the Dr Who database can be found in Appendix A.1.

In the first experiment the query graphs were generated with node counts $N$ equal to 2-6 nodes and relationship counts of $2 \cdot N$. The result of the executions can be seen in Figure 20. It shows the average speedups including standard deviation when using the GPUGDA solution compared to the equivalent Cypher query when querying the Dr Who database with the generated query graphs.

![Figure 20: The average speedups and standard deviations gained when executing the dense experiment queries on the Dr Who database. The X-axis legend name shows the nodes/relationships of the generated query graphs of each experiment](chart.png)

The results show that GPUGDA performs increasingly better as the query graph size increased. This is due to the fact that the run times when the queries are executed with Cypher increases rapidly as the query graph grows larger. At the
same time when the queries are executed with GPUGDA the increase in query execution time remains roughly the same.

In the second experiment the Dr Who database was tested with sparse query graphs. These query graphs also had node counts $N$ ranging from 2-6 nodes. To get sparse query graph the number of relationships each query graph could have was set to $N - 1$. This is the minimum number of relationships required for a graph to have $N$ nodes. The result of the second experiment can be seen in Figure 21.

The difference between the two experiments is clear. The scaling increase in the Cypher run time as the query graph grows in size that was present in the first experiment is non-existent in the second experiment. With fewer relationships in the query graphs the Cypher run times are faster and more even. This causes the decrease in speedup that can be seen in Figure 21. Another observation is that the standard deviation of the results increased considerably. It even grew larger than the average speedup which causes the strange bars in Figure 21.
5.6 Experiments on the Movies database

The Movies database was used to test the performance of GPUGDA against a relatively large and sparse database. The query graphs were generated with similar characteristics as in the experiments on the Dr Who database. However, since the Movies database was much more sparse the query graph generator had trouble finding the more dense queries for the experiment. Hence the criteria that the query graphs would contain $2N$ relationships, where $N$ was the number of nodes in the query graphs, had to be dropped. Instead the dense query graphs was modified with the condition that it should contain between $N$ to $2N$ relationships. The result data of the performance experiments on the Movies database can be found in Appendix A.2.

The resulting speedups when comparing the run time of GPUGDA to the run time of Cypher when executing the different query graph sizes on the Movies database can be seen in Figure 22.

![Figure 22: The average speedups and standard deviations gained when executing the dense experiment queries on the Movies database. The X-axis legend name shows the nodes/relationships of the generated query graphs of each experiment](image)

The same phenomenon as when executing the more dense query graphs on the Dr Who database appears when executing more dense query graphs on the Movies database. As the query graph size grows the run times of the Cypher queries increases almost exponentially. However the speedups are not as high as in the Dr Who database, which may hint that GPUGDA performs better the
more dense the graph database is.

In the second experiment the Movies database was tested against sparse query graphs. Originally the query graphs were planned to have query node counts $N$ of 2-6 and relationship counts of $N - 1$ as in the experiments on the Dr Who database. However as the query graph size reached 5 nodes and 4 relationships the number of results found caused heap overflows when executing the queries with Cypher. The GPUGDA algorithm had issues as well since the buffer sizes which had to be allocated for the partial solutions got too large to fit in the GPU memory. Hence the experiments query graphs were only generated with 2-4 nodes and $N - 1$.

The resulting speedups when comparing the run times can be seen in Figure 23.

Similarly to the experiments with sparse query graphs on the Dr Who database, the result shows that speedup decreases when the sparse query graph grows in size. For a node count of 3 or 4 the number of queries that performed worse with GPUGDA increased, causing the increased standard deviation seen in Figure 23.
5.7 Experiments on the Yeast database

In the experiments performed on the Yeast database the purpose was to test how the GPUGDA performed compared to Cypher when the queries were executed on a database with more nodes than the Dr Who database but with a similar density. The query graphs were generated with similar characteristics as in the previous experiments. The result data of the performance experiments on the Yeast database can be found in Appendix A.3.

For the first experiment the more dense query graphs had node counts \( N \) with 2-6 nodes and relationships counts of \( 2 \cdot N \). However for the experiments where query graphs with 2 nodes and 4 relationships were to be generated, the query graph generator could not yield any valid query graphs. Hence that experiment was omitted. The resulting speedup when comparing the run time of the GPUGDA queries to the Cypher queries can be seen in Figure 24.

![Figure 24: The average speedups and standard deviations gained when executing the dense experiment queries on the Yeast database. The X-axis legend name shows the nodes/relationships of the generated query graphs of each experiment.](image)

The same scalable speedup pattern as in the previous experiments with more dense query graphs on the Dr Who- and Movies databases can be seen for the results of the equivalent experiments on the Yeast database as well. These results hints that the database density may be an indicator for whether the GPUGDA algorithm will perform well on the database.
The final experiment was to test the Yeast database against sparse query graphs. As previously mentioned these query graphs had the same characteristics as in the previous experiments with sparse query graphs. The node counts $N$ of the query graphs were 2-6 nodes and the relationship counts were $N - 1$. The resulting speedup when comparing the run times of the GPUGDA queries to the Cypher queries can be seen in Figure 25.

![Figure 25: The average speedup and standard deviation gained when executing the sparse experiment queries on the Yeast database. The X-axis legend name shows the nodes/relationships of the generated query graphs of each experiment.](image)

Similarly to the experiment with more sparse query graphs on the Dr Who database, the speedups decreased considerably when executing the experiment with sparse query graphs. The reason for this result can be seen when inspecting the run times of the experiments further (see Appendix A.3). While the run times of the GPUGDA queries decrease to some extent when the query graphs are more sparse, in the Cypher query run times the real change appears. The run times of the sparse Cypher queries are about 5-10x faster than the dense Cypher queries.
6 Discussion and future work

The performance comparisons of the GPUGDA implementation against Cypher queries in Neo4j shows some promising results. During these experiments some interesting observations and areas of improvement were discovered.

6.1 The performance of Cypher queries and when to use GPUGDA

The experiments with the more dense query graphs showed that GPUGDA outperformed Cypher in most cases. In particular, when more sparse query graphs were used, the run times of the Cypher queries were much lower. However, when sparse query graphs were executed on a larger database the speedup actually decreased (sometimes resulting in a slowdown) as the size of the query graphs grew larger, which can be seen in Figure 23. These results hints that Cypher may be better suited for performing matching of query graphs which are not too dense. On the other hand, if GPUGDA and Cypher could be integrated a query execution planner could take the query graph density and data graph size in consideration. Using this input it could choose to use the GPUGDA-algorithm for dense query graphs and Cypher for sparse query graphs on larger databases.

Another interesting observation was the fluctuation in run times when the number of matching graphs returned from the queries grew higher. This can be caused by a combination of many factors, though the main culprits are the lack of indices and heap tuning.

6.1.1 Neo4j indices

The subgraph isomorphism queries which GPUGDA can execute supports nodes, node labels, relationships, and relationship types (see Section 4.1). However it does not support node and relationship properties. Indices in Neo4j can only be applied on properties [17], hence indexing was omitted in the experiments. If property queries had been supported by GPUGDA and the Neo4j databases had been indexed on certain properties, the Cypher performance would have increased. This should be investigated in the future.

6.2 Out of memory errors

One of the drawbacks of using the GPU for larger data sets is the potential lack of memory. This gives a hard limit on the amount of memory which may be allocated. If the number of partial solutions grows too large in the final steps the memory allocation for the partial solutions will fail, causing the entire query to fail. This out of memory behavior could be observed in the experiments of the larger Movies-database (see Figure 23). When it was queried with a more sparse query graph the number of candidate relationships grew larger, which in
turn generated a large number of partial solutions causing an out of memory error to occur on the GPU.

These errors can be avoided by tweaking the algorithm. The step where the partial solutions are combined with candidate relationships of an unvisited query relationship does not need to be performed in one sweep. As long as the combinations are created using the same set of query candidate relationships the combinations can be generated using subsets of the whole set of partial solutions.

These changes would obviously have an impact on the performance since we need to call the kernels multiple with the subsets of the data. However, since Neo4j also had issues with memory resources for similar queries this performance loss may be within reasonable amounts. This solution was not included in the scope of this thesis work. To implement and measure the impact of such a change is a good candidate for future work.

6.3 The necessity of candidate node refinement phase

As previously mentioned in Section 4.4 it was discovered that the phase where candidate nodes were refined to remove false positives never yielded any changes to the set of candidate nodes. Tran et al. state that the refinement step prunes the majority of the false candidates in the first rounds and that a performance loss was observed if the candidate nodes were pruned until no more nodes were removed [27]. However, since no changes to the set of candidate nodes were observed when executing the refinement phase in GPUGDA an experiment where the refinement phase was entirely removed was conducted.

The result of that experiment showed that removing the refinement phase did not affect the end result. The query results were still correct. This change did not have any evident effects on the overall performance. However the code complexity was reduced.

6.4 Integration with Neo4j and beyond

One of the goals of this thesis work was to produce a prototype which could be used in a real graph database. To achieve this the prototype has functionality to use data which came from a real database. In Section 4.2 the process of how GPUGDA converts data from a Neo4j database to a representation which is better suited for the GPU was explained. Even though this conversion step was implemented to integrate with Neo4j data it is possible to rewrite the data conversion to use an adapter pattern. With such a solution GPUGDA would not be limited to only use data originating from a Neo4j database. Instead any graph data could potentially be used as long as an adapter is created for that particular data representation.

The GPUGDA prototype successfully queries data originating from a Neo4j database by converting the data. With this in mind the goal to produce a prototype which can be used with a real database has been achieved. Future
work could include the creation of the previously mentioned adapters, to be able to compare the performance against other graph databases.

A suggestion for future work could be to change how the Neo4j data is stored internally. As mentioned in Section 4.1 the internal data representation of Neo4j relies heavily on linked list-structures which is not optimal for the GPU due to their potentially uncoalesced memory accesses. To be more GPU-friendly the linked-list storage strategy would have to be changed. Parts of the proposed graph representation in GPUGDA could be used as inspiration for such a storage solution, which might remove the need to convert the data entirely. However, to do this the indexing issue, which was solved by assigning temporary query identifiers when the data was converted (see Section 4.2), must be solved in some way.

6.5 Array gathering may be performed on the GPU

The implementation heavily utilizes array data structures to store candidate nodes, relationships, and solutions. Many of these arrays are populated using the two-step-output procedure described in Section 3.2.2. In GPUGDA this procedure is performed between the kernel calls in the algorithm to prepare the data buffers on the GPU for the next step.

An example could be the gathering of the candidate nodes after the `check_candidates_kernel` (see Algorithm 1) has finished its execution. To gather the candidate nodes for the next step (see Algorithm 2) the buffer that contains the candidate indicators must be transferred back from the GPU memory to the main memory. When the data has been transferred back to the main memory the two-step-output procedure can be performed to create the array that contains the candidate nodes. When the candidate node array has been produced it is transferred from the main memory to the GPU memory.

The memory transfers between the GPU and the CPU are slow and costly. To reduce the amount of memory transfers the two-step-output procedure could be performed on the GPU. Algorithms for performing the first step of the procedure, the prefix sum, exists [7] but were omitted in the prototype implemented in this thesis. A good candidate for future work would be to implement the two-step-output procedure on the GPU and measure the performance impact.

6.6 Difficulties and feasibility of GPU-acceleration

Writing programs for the GPU is hard. Even though tools to make the development process easier exists, it is easy to make mistakes due to the massively parallel nature of GPGPU-computing. During the prototype development in this thesis work the library JavaCL [16] was used to handle the creation and execution of the OpenCL kernels. This library shrunk the development complexity since the boilerplate code which is needed to execute kernels was automatically generated. But the main difficulties when developing in OpenCL is the manual memory management. It is easy to make simple mistakes that takes a long time to discover.
When taking a decision to optimize code for performance the feasibility of such an optimization must be considered. The performance gain should outnumber the effort for developing that optimization. This is equally true for acceleration using GPGPU-computing. Since the main performance loss in GPGPU computing is extensive memory operations the arithmetic intensity of the algorithm that should be optimized must be high. Harris et al. describes arithmetic intensity as the ratio between memory operations to arithmetic operations in an algorithm [8]. If the arithmetic intensity is low the potential performance gain may be low, making the GPU-acceleration unfeasible. Another important aspect to take into consideration when accelerating algorithms using the GPU is the parallelizability of the algorithm. It can be seen as the ratio between the parallelizable and sequential parts of an algorithm.

In a perfect world the algorithms are perfectly parallel and perform little to no memory operations. With the arithmetic intensity and parallelizability in mind, the problem of parallelizing database operations is interesting. On one hand database operations are used to read and/or manipulate data. This may hint that the arithmetic intensity of such operations might be low, depending on the operation. On the other hand these operations may also be perfectly parallel. In this thesis a prototype for finding graph isomorphisms in a graph database using the GPU was developed. While the compute intensity of the algorithm may be lower the different phases of the algorithm are highly parallelizable. This made the problem a feasible candidate for GPU-acceleration and it yielded clear speedups when the algorithm was compared to Neo4j’s Cypher query language, which can be seen in the experiment results.
7 Conclusions

In this thesis work an algorithm for accelerating graph isomorphism queries on graph databases using the GPU has been created. The algorithm named GPUGDA builds upon the work of Tran et al. and their algorithm GpSM [27], which takes advantages of the many parallelizable parts of finding graph isomorphisms. GPUGDA adds additional functionality to the GpSM algorithm by allowing the vertices/nodes of a query graph to have multiple labels as opposed to only one label. Furthermore the edges/relationships of the query graphs can have a specific type. These additions were added to allow GPUGDA to be used to query the graph database Neo4j in a similar way as Cypher, the query language of Neo4j.

A prototype was created to show that the algorithm works with real data. The prototype was developed in OpenCL and Java to be able to support as many platforms as possible. The performance of the prototype was measured in a group of experiments. The experiments consisted of querying three graph databases of with different sizes and properties. The query graphs issued to the different databases also varied in size and density.

By comparing the run time of executing the same queries in the experiments with Cypher to GPUGDA some promising results were shown. The observed average speedups ranged up to 460x when the query graphs were dense. The speedups were much lower in the experiments were more sparse query graphs were used. For the largest database the sparse query graphs caused a slowdown because the Cypher query run time was lower than the run time for converting the graph data.

The results shows that the GpSM algorithm is feasible for use in a real database, with some modifications. A proposal for future work is to create more adapters to allow other graph databases to utilize the GPUGDA algorithm. Another proposal, if a graph database should take advantage of GPU acceleration, is to add an additional step in the execution planner to check the density of the query graph. If the density is high enough the execution planner could choose to use the GPUGDA algorithm to query the data on the GPU.
References


[34] Peixiang Zhao and Jiawei Han. “On Graph Query Optimization in Large Networks”. In: Proc. VLDB Endow. 3.1-2 (Sept. 2010), pp. 340–351. ISSN: 2150-8097. DOI: 10.14778/1920841.1920887. URL: http://dx.doi.org/10.14778/1920841.1920887 (visited on 06/26/2016).
Appendix A  Performance experiment results

In this appendix section the results of the performance experiments are listed. The results are represented as comma-separated value-files. Each row represents the result of executing a query against the given database using Cypher and GPUGDA. The measured times include the execution run time of the Cypher query, the GPUGDA query and the data conversion. It also contains information regarding the number of nodes, relationships and results returned of the query graph that was issued. The last column in the files contain the Cypher representation of the query which yielded that result row.

Table 1 specifies the order in which these values appear in the comma-separated value lists. All z files can be downloaded from a GitHub repository[26]. The filenames tells which database the experiment was performed on as well as the number of nodes and relationships the generated query graphs in the experiment had. For example, the file result-yeast-6-12-2016-05-25T1944.csv contains the results for the experiment where the Yeast database was queried with generated graphs that had 6 nodes and 12 relationships.

<table>
<thead>
<tr>
<th>Node count</th>
<th>Relationship count</th>
<th>Result count</th>
<th>Data conversion run time</th>
<th>GPUGDA run time</th>
<th>Cypher run time</th>
<th>Cypher query</th>
</tr>
</thead>
</table>

Table 1: The order in which the values of the comma-separated value lists appear.

A.1 Result files from experiments on the Dr Who database

result-drwho-2-1-2016-05-25T2014.csv
result-drwho-2-4-2016-05-25T2014.csv
result-drwho-3-2-2016-05-25T2014.csv
result-drwho-3-6-2016-05-25T2014.csv
result-drwho-4-3-2016-05-25T2014.csv
result-drwho-4-8-2016-05-25T2014.csv
result-drwho-5-4-2016-05-25T2014.csv
result-drwho-5-10-2016-05-25T2014.csv
result-drwho-6-5-2016-05-25T2016.csv
result-drwho-6-12-2016-05-25T2016.csv
A.2 Result files from experiments on the Movies database

result-movies-2-1-2016-05-26T1807.csv
result-movies-3-2-2016-05-26T2003.csv
result-movies-3-3-2016-05-26T2005.csv
result-movies-4-3-2016-05-26T2006.csv
result-movies-4-6-2016-05-26T2028.csv
result-movies-5-9-2016-05-26T2029.csv
result-movies-6-11-2016-05-26T2030.csv

A.3 Result files from experiments on the Yeast database

result-yeast-2-1-2016-05-25T1942.csv
result-yeast-3-2-2016-05-25T1942.csv
result-yeast-3-6-2016-05-25T1942.csv
result-yeast-4-3-2016-05-25T1942.csv
result-yeast-4-8-2016-05-25T1942.csv
result-yeast-5-4-2016-05-25T1942.csv
result-yeast-5-10-2016-05-25T1942.csv
result-yeast-6-5-2016-05-25T1944.csv
result-yeast-6-12-2016-05-25T1944.csv