Abstract

The paper aims to provide a comparative analysis on the performance of in-memory databases as opposed to a customised graph database written ground up whose joins(searches) are performed on a GPGPU. This is done primarily to serve as a proof of concept on how databases that are represented as graphs can benefit by fostering the raw parallel processing power of GPGPUS.

1. Introduction

With the increasing need to connect to resources and people via the Internet, the need for coupling data models to the real world has increased sharply in recent times. Traditional relational models do not adequately represent the highly connected nature of the data that they contain both in terms of being intuitive and efficient. Relational Databases have matured out over a long period of time and programmers and analysts are trained to think of data models in terms of how they are represented as tables and tuple structures. However representing data as tables comes at the cost of logical disconnection. For example, consider a model that represents Restaurants (R) and People (P). A relationship like 'people who visit the restaurant' needs to be represented as a peripheral entity in this model either as a separate table or as part of R or P. Representing such relationships as graphs are is not only more intuitive but also has several performance benefits.

Graph Databases like Neo4j have gained popularity recently for changing the way social networks are represented. Linear/logarithmic searches for field values in tables has been replaced by graph searches that are beneficial many a time. This paper explores the possibilities of accelerating such graph traversals on the GPU.

Accelerating graph traversals on the GPU is directly related to improving data retrieval operations on Graph Databases. This is a particularly pertinent problem to solve in the context of social-media applications that would benefit out of such a performance improvement.

Previous work done on running graph algorithms on GPGPUs does not calibrate and analyze performance from the perspective of information-retrieval analogous to graph database queries. This paper attempts to fill this missing gap.

2. Introduction to Graph Databases

A graph database stores data in a graph, the most generic of data structures, capable of elegantly representing any kind of data in a highly accessible way. Consider the following example. Well read a graph by following arrows around the diagram to form sentences. A Graph records data in Nodes which have Properties. A Graph records data in - Nodes which have - Properties. Nodes are organized by - Relationships which also have - Properties. Relationships organize Nodes into arbitrary structures, allowing a Graph to resemble a List, a Tree, a Map, or a compound Entity any of which can be combined into yet more complex, richly inter-connected structures.

As shown in Fig 2 a Traversal is how you query a Graph, navigating from starting Nodes to related Nodes according to an algorithm, finding answers to questions like what barbecue restaurants do my friends visit that I have not yet visited or if this power supply goes down, what web services are affected? Often, there is a need to find a specific Node or Relationship according to a Property it has. Rather than traversing the entire graph, one can use an Index to perform a look-up, for questions like find the Account for username master-of-graphs. An Index maps from Properties to either Nodes or Relationships as in Fig 2.

In this context, the following section explains model that has been used to evaluate the problem.

3. Evaluation Model

The previous section shows how retrieving information from graph databases can be reduced to traversing/ search-
ing graph data structures. In the interest of having a simple model, I have assumed that the entire graph is In-Memory. This may initially seem like an over simplification since disk storage and retrieval is a key part of database systems. Although this is largely true, in recent times, performance demands of Internet application have forced firms like Facebook to have in memory databases that result in RAM sizes on the order of TeraBytes [1]. Also, if the graph data-structure can be visualised as just an index that supplements a relational database, it can be seen why these structures need not be as large as the database itself [4].

In order to evaluate traversal, it is useful to visualize the graph as an alternate representation of a sparse matrix. Consider a situation where we have dense index generated on every possible column on a database (only single column indices) so that all queries that filter by some criteria on one column can be answered in logarithmic time. What this really means is that NATURAL JOINS can be performed efficiently since each record is aware of the physical storage location of the record it is to be ‘joined with’. It is important to bear in mind that this self-awareness is essentially what a graph database tries to establish. Hence from that perspective, it is only fair that the evaluation be carried out by comparing to a relational DB with dense index column. Also the entire data-structure is in-memory and hence comparisons also are to be made to in-memory relational databases.

The Relational Data Model used for the experiment is simple. Consider a model that represents all Restaurants visited by a Person. It also represents all Friends of the Person. The query being evaluated is to retrieve all the barbecue restaurants visited by a particular person’s friend’s friends. The entity Relationship Diagram for the schema being used is in Fig 3. The BCNF of the relational schema is in Fig 3. It is useful to note that the data model under consideration is simple (it has just 2 types of entities and 3 types of relationships) for the sake of clarity. However, from section 4 it should be clear as to why a simplified model is sufficient to demonstrate performance.

4. Breadth First Search on CUDA

The algorithms for CUDA implementation of Breadth First Search(BFS) was obtained from [8]. In this paper, the BFS problem is solved using level synchronization. BFS traverses the graph in levels. Once a level is visited it is

![Figure 1. Index of the Graph Database](image1)

![Figure 2. Traversing the Graph Database](image2)

![Figure 3. Evaluation Schema](image3)
not visited again. The BFS frontier corresponds to all the nodes being processed at the current level. A queue for each vertex during BFS execution is not efficient because it will incur additional overheads of maintaining new array indices and changing the grid configuration at every level of kernel execution. This slows down the speed of execution on the CUDA model.

The implementation gives one thread to every vertex. Two boolean arrays, frontier and visited, Fa and Xa respectively, of size —V— are created which store the BFS frontier and the visited vertices. Another integer array, cost, Ca, stores the minimal number of edges of each vertex from the source vertex S. In each iteration, each vertex looks at its entry in the frontier array Fa. The vertex removes its own entry from the frontier array Fa and adds to the visited array Xa. It also adds its neighbors to the frontier array if the neighbor is not already visited. This process is repeated until the frontier is empty. This algorithm needs iterations of order of the diameter of the graph G(V,E) in the worst case. Algorithm 1 runs on the CPU while algorithm 2 runs on the GPU. The while loop in line 5 of the Algorithm terminates when all the levels of the graph are traversed and frontier array is empty.

Create vertex array Va from all vertices and edge array Ea from all edges in G(V,E)

Create frontier array Fa, visited array Xa

Initialize Fa, Xa to False

Fa[S] = True

while Fa not Empty do
    for each vertex V in parallel do
        Invoke CUDA_BFS_KERNEL(Va, Ea, Fa, Xa, Ca) on the grid
    end for
end while

Figure 4 represents search stages for the query: All Barbecue Restaurants Visited by Blue’s Friends’ Friends. Once

<table>
<thead>
<tr>
<th>No of Nodes</th>
<th>101000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outdegree</td>
<td>200</td>
</tr>
<tr>
<td>Step-Count</td>
<td>2</td>
</tr>
<tr>
<td>Size of Record</td>
<td>10 bytes</td>
</tr>
</tbody>
</table>

Blue has been identified, stage 1 of the iteration node identifies Green Nodes (i.e Blue’s Friends). In the second iteration, Green Nodes (i.e Blue’s Friends’ Friends) are identified. In the third iteration, the Red and Orange Nodes are identified (Restaurants visited by emphBlue’s Friends’ Friends) with Red Nodes being the barbecue restaurants. This example clearly shows the search involved in answering a query and also demonstrates how the overheads of table joins are overcome.

From the algorithm, it can be seen that a simple data model is sufficient to demonstrate performance since increasing the number of Entities or Relationships has no worse than a linear effect on performance. Even this linear factor tends to be further less pronounced on account of the parallel nature of the problem.

Figure 5. Searching a Graph to answer queries

5. Implementation notes

The values of the parameters when not under observation are given in table 5.
5.1. Tweaking the BFS Algorithm

The algorithm in [4] is incapable of handling queries that involve cycles in the graph since the Frontier array ensures that no node is visited more than once. For this, the algorithm needs to be tweaked to accommodate cases where even visited nodes must be revisited in following iterations to cover queries that involve cycles. The query given in section [3] can have such cycles if 2 Persons have a mutual friend. In this case, the mutual friend is eligible to be opened when either of the friends is visited in an iteration.

6. Results

As mentioned in [5], a graph database can be conveniently understood to be dense index generated on all meaningful relationships. For a convenient comparison, the experiment results provided contain the running times for the following

- Indexed Inmemory SQLite Database (2G Ram)
- Non-Indexed Disk SQLite Database
- GPU Total Time
- GPU Computation Time

The graphs are represented as adjacency lists in arrays. UTHash was used to maintain a hash-table of the nodes on the CPU. This structure could be used to identify the root of the search. Once the first node is identified, the search is carried out as mentioned in [4] on the GPU. The following subsections demonstrate the performance of the parallelised graph traversals.

6.1. Cardinality of the Graph

The Graph for the considered example consists of 2 types of nodes Person and Restaurant. The experiments were performed by varying the number of Person nodes. It was found that the in-memory version of sqlite closely matched the performance of the GPU at lower cardinalities. However, the GPU implementation appears more scalable. This is because the inherently sequential portion [5] of the parallel algorithm (as indicated by the difference in Total GPU Time and the GPU Computation Time as shown in Fig. 6.1) becomes a worthwhile investment at higher cardinalities where greater data-parallelism can be exploited.

6.2. Outdegree of a Node

The outdegree of the graph was varied by changing the number of Friends and Visits made by a Person. The Connectedness of the Graph would be a function of its outdegree. The GPU implementation once again outperforms the CPU implementations. It should be noted here that the SQLite considerations did not return for over 60 seconds at an outdegree of 600 and are hence not plotted. There connectedness of this data-model causes relational databases to be unusable at a fairly reasonable outdegrees itself. This result also demonstrates why Relational Databases fail even with highly simplified data models like the one in consideration.

6.3. Number of Steps

The example considered so far is 2-step in the sense that it find the restaurants visited by Friends of Friends. By changing the number of steps, the size of the problem increases exponentially. This is by far the biggest advantage of graph models as opposed to relational models.

In the relational model, increasing the step size entails increasing the number of joins involved in the computation. The number of joins performed increases linearly with step size. As seen, the GPU graph implementations scale with larger step sizes exceedingly well. Once again, for step sizes 4 or greater, the Relational Models do not return results for over 60 seconds and are hence not plotted. These results are also fairly consistent with results indicated in [10].

The source code of Grafight – the prototype built to obtain these results can be obtained and built from https://github.com/ashwinraghav/Grafight.
7. Interpreting the Results

As shown in section 6, performing graph traversals on the GPU outperforms in-memory relational databases database. It is worthwhile noting here that the exceedingly good results are a combination of the inherent benefits of Graph Data Models as well as the parallel nature of the traversals.

It can also be seen that a major portion of the time involved in GPU computation was on account of the data transfer between the CPU and GPU. It is commonly knowledge that this can be solved by housing the data in GPU memory or by improving data transfer speeds of the PCI Express bus. In terms of sheer computation speed, the brute force algorithm and the GPU outperform relational models by big margins.

On the whole it is found that using GPGPUS and graph databases on highly connected data models can improve retrieval efficiency by a factor of 10X. It is also demonstrated in section 6.3 how operations that force relational database joins to increase linearly with increase in complexity successfully scale in Graph Data Models on the GPU.

8. Conclusion

Grafight, a simple graph traversal engine was written to obtain the calibrations that are shown in this paper. The primary objective of developing a proof of concept that demonstrates graph traversal in GPUs is fulfilled and a comparison to in memory database performance is also provided. In addition to what was initially planned, the first step of finding the start node was also implemented using simple indexing. Further details of this portion of the project can be obtained in [9]. The results obtained are highly encouraging.

SQLite is an open-source free database that can be installed from source at http://www.sqlite.org/