A Parallel Tree Pattern Query Processing Algorithm for Graph Databases using a GPGPU

Lila Shnaiderman
Computer Science Department, Technion
lilas@cs.technion.ac.il

Oded Shmueli
Computer Science Department, Technion
oshmu@cs.technion.ac.il

ABSTRACT

Large amounts of data are modeled and stored as graphs in order to express complex data relationships. Consequently, query processing on graph structures is becoming an important component in real-world applications. The most commonly used query format is that of tree pattern queries. We present a new parallel SIMD algorithm, GGQ (GPU Graph data base Query), for answering tree pattern queries on graph databases, using a GPU. We present the results of extensive experimentation of GGQ on large graph databases using known benchmarks that show that GGQ is an effective and competitive algorithm.

1. INTRODUCTION

Graph databases are widespread in many areas, including the semantic web and social/biological networks, as a graph is a more flexible and expressive structure than a tree. One of the most important and practically most interesting query formats for graph databases is a tree pattern query (TPQs - Tree Pattern Queries). In most known query languages for XML and RDF (such as XQUERY and SPARQL [19]), many queries can be regarded as TPQs on graphs. An example of a TPQ query is presented in Figure 2 (right end side). Finding all occurrences of matching a TPQ query to an isomorphic sub-graph of a given data graph is a fundamental operation in graph query processing. Related basic problems are (a) determining if a matching exists, and (b) providing part of the matched data nodes, corresponding to query target nodes, as the result.

Lately, there has been much research on using GPUs to speedup database operations. The standard use of GPUs is to render graphical information. GPUs are a cheap and ubiquitous source of processing power, as at least one GPU can be found in almost any computer. GPUs follow a SIMD (Single Instruction, Multiple Data) architecture, while multi-core systems follow a MIMD (Multiple Instructions, Multiple Data) architecture. In SIMD, multiple processing elements perform the **same** operation on multiple data elements, simultaneously.

We focus on processing TPQ queries and not on general graph structured queries as, in practice, TPQ queries seem to be the most frequently used type. Few research projects have addressed parallelizing query processing over graph databases. The main idea has been to use the data partitioning strategy, i.e., methods to partition the data between many computing elements, for example see [9]. To the best of our knowledge, there have been no attempts to parallelize the query processing of a *single* TPQ, or to design a parallel algorithm that exploits GPUs (or any other SIMD-based device) to accelerate the processing of a *single* TPQ.

In this paper we present the GGQ algorithm (GPU-Graph data base Query). The problem we address is how to use GPUs to accelerate the processing of a *single* TPQ query. The main idea underlying GGQ is to copy the relevant parts of the graph document, according to the input query, to the GPU global memory, to process the query using *all* the threads of the GPU in parallel, and to copy the query results back to the CPU memory. The key to parallelizing the query processing is in the ability to efficiently coordinate the query processing tasks between thousands of working units. In GGQ, each thread checks a different potential matching between the TPQ pattern and the data graph. In case that the checked potential matching actually exists, the thread reports this matching as one of the answers to the query.

GGQ is novel in that thread identifiers (IDs) are used to determine the choices made in attempting to match the tree pattern to actual database graph nodes and edges. As the space of possibilities that can be represented by an ID is limited, methods are presented to practically increase this space.

To minimize the amount of data that has to be copied to the GPU for a particular query execution, we designed a new graph lists storage scheme, GLS, that is based on a XML stream representation scheme [11]. A section describing GLS is not included in the paper due to lack of space.

For documents that can fully reside in the global memory, we gain speedup of up to 1000 times in comparison to Gremlin [17][1] (counting the time of copying the results from the GPU to the CPU but not counting copying from the CPU to the GPU). If a whole document is loaded to the GPU, many queries on this document can be processed one after the other, thus eliminating the need to copy the document, for each query, from the CPU to the GPU. For documents that can not fully reside in the GPU global memory, according to our experiments, we still gain a significant improvement of up to 100 times in comparison to Gremlin (counting the time of copying the data from the CPU to the GPU and the time of copying the results from the GPU to the CPU). In experiments with an extra-large query, we obtained speedup of up to 50 times in comparison to Gremlin (while counting just the copying time of the results from the GPU to CPU), and up to 35 times in comparison

[1]There may by now be tools for public use that are more efficient than Gremlin.

to Gremlin (while counting the time of copying the data from the CPU to the GPU and the time of copying the results from the GPU to the CPU).

2. BACKGROUND

In this section, we briefly introduce GPUs and CUDA (the underlying platform upon which the GGQ algorithm is implemented), and TPQ pattern matching.

2.1 Graphics Processors (GPUs)

GPUs were originally designed for dealing with graphics rendering. In recent years GPUs are also used as multi-threaded multi-core co-processors for CPUs. GPUs have a SIMD (Single Instruction, Multiple Data) architecture. In the SIMD architecture, there are multiple processing elements that perform the same operation on multiple data, simultaneously. Any algorithm for GPUs has to fit the SIMD scheme; hence an original CPU (or multi-core) algorithm **should not** be run as is on a GPU. If run as is, it will most probably be extremely inefficient. Programmers write their algorithms so that the part of the algorithm that does not have to be massively parallelized runs on the CPU and the other part, which can be massively parallelized, runs on the GPU.

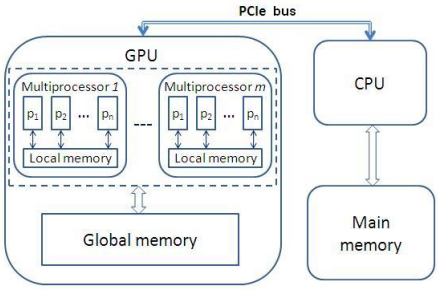

Figure 1: GPU architecture model

GPU Hardware Architecture. The GPU architecture is shown in Figure 1. This architecture is common to both NVIDIA [16] and AMD [13]. The NVIDIA GTX processor is a collection of multiprocessors (in GTX480 there are 15 multiprocessors), each with a group of processors (32 in GTX480). Each multiprocessor has its own shared memory which is common to all the processors within it. It also has a set of registers, texture, and constant memory caches. At any given cycle, each processor in the multiprocessor executes the same instruction on different data. A *warp* is a collection of threads that run simultaneously on a multiprocessor. The warp size is fixed for a specific GPU. Communication between multiprocessors (i.e., processors from different multiprocessors) is through the device memory (also called global memory), which is available to all the processors of the multiprocessors. The size of the global memory is limited. The GTX480, for example, has 1.5GB memory. The global memory has both a high bandwidth and high access latency. GPU threads have both low context-switch and low creation time as compared to CPU threads. The global memory is available to all the threads, so any thread can access any memory location.

CUDA Programming Model. Programmers use two types of code, the **kernel code** and the **host code**. The kernel code is executed on the GPU. The host code runs on the CPU. The host part is in charge of transferring data between the GPU and main memory, and starting kernel-code instances (kernels) on the GPU. A computation task on the GPU is divided into three separate steps. First, the host code allocates GPU memory for input and output data, and copies input data from the main memory to the GPU memory. Second, the host code starts threads each executing the kernel code, kernels, on the GPU. The kernels perform the required task on the GPU. Third, when the kernels finish their work, the host code copies results from the GPU memory to main memory. For the programmer the CUDA model is a collection of threads running in parallel. A collection of threads (called a *block*) runs on a multiprocessor at a given time. One can assign multiple blocks to a single multiprocessor and then the blocks execution on the multiprocessor is time-shared.

Execution. All threads of all blocks executing on a single multiprocessor share its resources. Each thread and block has a unique ID. In addition, each thread has a program counter, registers, per-thread private memory, and inputs that can be used by the thread during its execution. Each thread in a set of parallel threads executes an instance of the kernel code, in parallel. Blocks are further organized into grids of thread blocks by the programmer. Each grid is a 2 or 3-dimensional arrangement of blocks. When a block is executed, it is further divided into warps. Using the thread and block IDs each thread can perform the kernel code on different set of data. In some cases, during some operations, for example an *if else* statement, some of the threads in a multiprocessor are idle (during the *if* block or the *else* block), as according to the *if else* statement, they do not have to process the body of the *if* block or the *else* block of the statement.

2.2 TPQ (Tree Pattern Query) pattern matching

Tree Pattern Queries (TPQs) are represented as directed trees, where (1) the nodes and edges of a TPQ Q are labelled by labels from an alphabet $\sum$. The label of a node u is denoted by $\tau(u)$, and the root node of Q is denoted by $root(Q)$. The size of Q, denoted by $|Q|$, refers to the number of nodes in Q. (2) The nodes in Q are connected by parent-child edges (pc-edges) labeled by a label from $\sum$. Consider an edge $e = (u, v)$ with parent node u and child node v, we say that v is a child of u and u is the parent of v.

Given a TPQ Q with nodes (q_1,..., q_n) and a directed graph document D, a *match* of Q in D is a mapping from the nodes of Q to nodes (d_1,..., d_n) in D s.t.: (1) d_i is matched with q_i, $1 \leq i \leq n$, (2) d_i and q_i have the same label except that nodes labeled with the special label '*' may be matched with data nodes that can have any label from alphabet $\sum$. (3) the edges, i.e., structural (parent-child) relationships between query nodes are satisfied by the corresponding D nodes and the label of both of the edges (in Q and D) have to be exactly the same (again, with the '*' exception). The ordering of sibling nodes in a TPQ query imposes no constraints on the matching. Also, pattern nodes need not be mapped to *distinct* D nodes (the algorithm can be extended to enforce such distinct mappings).

The *TPQ pattern matching problem* is defined as finding all the possible matches of a given TPQ Q in a given graph document D.

3. THE GGQ (GPU GRAPH DATA BASE QUERY) ALGORITHM

The GGQ algorithm is a SIMD algorithm. The main advantage of the GGQ algorithm is the ability to divide the matching work to hundreds or even thousands of threads that run in parallel, and that the work of each thread is exactly of the same length. The idea of the basic version of the algorithm is to use the ID of a thread to determine the portion of the data to which a pattern matching attempt will be executed by the particular thread. Then, as the number of bits in a thread ID is bounded, we designed an extension that al-

lows the algorithm to be efficient also in cases when the query tree or the data graph are more complex. GGQ processes mainly the document parts that are relevant to the input query by processing only edge streams that are relevant for the input query.

The inputs of the algorithm are a labeled directed graph $G = (V, E)$, a TPQ Q, and a set of nodes V_q, subset of V, containing all data graph nodes which are part of legal possible matches for the root node of Q. The algorithm finds all possible matches between Q and G subject to the V_q constraint.

Next, we explain the main idea of the algorithm. For ease of explanation, assume that set V_q has just one node, v_1. Each GPU thread has a unique ID. For example, the ID of thread th is $thNum$, and in binary $thNumb =< b_m, b_{m-1}, ..., b_0 >$. Each node v_i in V has at most $outgNum(lbl)$ outgoing edges with label lbl for each lbl label where $outgNum(lbl)$ is the maximum number of edges labeled lbl connected to a node in the database. I.e., we need $log_2(outgNum(lbl))$ bits to represent $outgNum(lbl)$. For ease of exposition, we assume that $outgNum(lbl)$ for any label is a power of 2. The bits of $thNumb$ define which edge has to be chosen at each step of checking for a match against the data graph.

For example, assume that we have just two types of labels, $lblA$ and $lblB$, in the graph. $outgNum(lblA) = 4, outgNum(lblB) = 16$. Assume that we have a query pattern Q with 3 edges, the first and third edges are with label $lblA$, and the second edge is with label $lblB$. Assume that the maximal thread ID is 255, thus we have 8 bits $< b_7, b_6, ..., b_0 >$ to represent any possible thread ID. Bits b_0 and b_1 represent the index of all possible data edges with label $lblA$, for the first edge in the query pattern. Bits b_2, b_3, b_4, and b_5 represent the index of all possible data edges with label $lblB$, for the second edge in the query pattern. And finally, bits b_6 and b_7 represent the index of all possible data edges with label $lblA$, for the third edge in the query pattern.

We have also tried to reverse the ordering of enumerating the thread ID bits (i.e., to extract the bits from left to right - from MSB to LSB), so that the MSB bit will correspond to the top of the tree. However, the effectiveness of this will fully depend on the nature of the data tree. For the data we used which induces a flat structure on the data tree, this scheme turned out to have inferior performance.

To gain intuition about the algorithm, we start off with an example.

3.1 An Intuitive Example

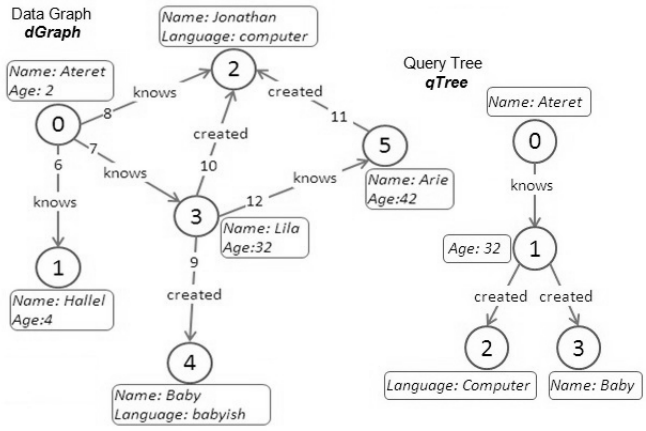

Figure 2: Example of a query tree *qTree* and a data graph *dGraph*

Consider[2] $qTree$ and $dGraph$ presented in Figure 2. Based on $dGraph$, we see that each node $v_i \in V$ has no more than $outgNum(created) = 2$ outgoing edges labelled $created$ and no more than $outgNum(knows) = 3$ outgoing edges labelled $knows$, thus we need 1 bit to represent $outgNum(created)$ and 2 bits to represent $outgNum(knows)$, here bit=0 means edge number 1 and bit=1 means edge number 2.

In total, we need 4 bits to represent all the possible potential matchings with graph $gData$ (shown in Figure 2). During the run, we have 16 different running threads. For ID with bits $< b_3, b_2, b_1, b_0 >$, bits b_0 and b_1 apply to the first edge (between query nodes 0 and 1), bit b_2 applies to the second edge (between query nodes 1 and 2) and bit b_3 applies to the last edge (between query nodes 1 and 3). V_q contains the data node with $ID = 0$. Now, we go over all the threads and explain what happens at run time with each of them.

The thread with ID 0000, finds that the data node with ID 0 has 3 outgoing edges labelled $knows$, according to the first 2 bits "00" of the thread's ID, we choose the first edge which leads us to the data node with ID 1. While checking the data of this node, we find that it does not have label and data "Age: 32". So, this partial matching is not part of an answer, thus the thread terminates with no match. The same behavior happens for threads with ID: 0100, 1000, and 1100.

The thread with ID 0010, chooses the 3-rd outgoing edge (corresponding to the "10" bits) of the data node with ID 0. Thus, it matches the query node with ID 1 to the data node with ID 2. While checking the data of this node, we find that it does not have label and data "Age: 32". Thus, the thread 0010 terminates with no match. The same behavior happens for the threads with ID: 0110, 1010, and 1110.

The thread with ID 0011, chooses the 4-th outgoing edge (corresponding to the "11" bits) of data node with ID 0, but such an edge does not exist. Thus, the thread terminates with no match. The same behavior happens for the threads with ID: 0111, 1011, and 1111.

The thread with ID 0001, chooses the 2-nd outgoing edge of the data node with ID 0. Thus, it matches the query node with ID 1 to the data node with ID 3. The data node with ID 3 has label and data "Age: 32"İ. Next, the thread chooses the 1-st outgoing edge, with label "created", of the data node with ID 3. Thus, it matches the query node with ID2 with the data node with ID 4. The data node with ID 4 has no label and data "Language: computer"İ. Thus the thread terminates with no match. The same behavior happens for the thread with ID 1001.

The thread with ID 0101, chooses the 2-nd outgoing edge of the data node with ID 0. Thus, it matches the query node with ID 1 to the data node with ID 3. The data node with ID 3 has label and data "Age: 32"İ. So, the thread now chooses the 2-nd outgoing edge of the data node with ID 3. Thus, it matches the query node with ID 2 with the data node with ID 2. The data node with ID 2 has label and data "Language: computer"İ. Now, the thread chooses the 1-st outgoing edge of the data node with ID 3 (corresponding to the leftmost "bit" with value 0). Thus, it matches the query node with ID 4 with the data node with ID 4. The data node with ID 4 has label and data "Name: Baby"İ. At this point the tread has finished to match all the nodes and edges. Thus the thread reports that the currently identified assignment of query nodes to data nodes is an answer to $qTree$ in $dGraph$, and terminates with a match. The last thread does not find a match.

3.2 Base algorithm

We specify how the algorithm operates for query Q, graph G, set

[2]In the experiments we used simpler documents in which a single label may be associated with a graph node.

V_q, and a thread with ID $thNum$. For ease of explanation, assume that set V_q has just one node, v_1.

Let $maxthNum$ be the maximal possible thread ID. Let p be $log_2(maxThNum)$, without loss of generality, assume that $maxThNum$ is power of 2. Sort the edges of Q: $e_0, ..., e_w$ so that if edge e_x is on the path from Q's root to the left vertex of edge e_y, then e_x precedes e_y in the order (i.e., "higher" in the tree).

Input: 1) Data graph G. 2) TPQ query Q. 3) V_q set.
Output: $ansSet$, the set of all thread IDs that encode patterns that are an answer to query Q in data graph G.
Method (runs on the CPU):
1. $ansSet = \{\}$
2. Invoke CUDA kernel call for function:
   $GpuGraphQuery(G, Q, V_q, ansSet)$
3. RETURN $ansSet$

// $GpuGraphQuery$ kernel function (runs on the GPU):
Input: 1) Data graph G. 2) TPQ query Q.
3) V_q set of nodes in G that match to the root of Q.
4) $ansSet$ set of all answers (thread IDs).
5) $idConst$. Has default value of 0. Used for algorithm extensions
Goal: In case that current thread's ID encodes an answer to query Q in data graph G, add it into $ansSet$.
Method:
1. Set $thID$ to a system assigned index of the current thread.
2. Set $maxID$ to a system value of the maximal thread index.
3. $thNum = (idConst * (maxID + 1) + thID)$
4. Set $cqIdx$ to 0. /* current query edge index */
5. Express $thNum$ in binary notation as $< b_p, b_{p-1}, ..., b_0 >$.
6. Set $bitIdx$ to 0. /*represents the currently processed bit in the binary notation of $thNum$*/
7. Create $dataNodeArray$ of size $|Q|$ and initialize all its entries to nil. /* the element with index a_i will be data graph node d_n, that corresponds to query node with index a_i */
8. $dataNodeArray[1] = v_r$ /* w.l.o.g. the root index of Q is 1 and v_r is some matching data node (according to V_q).*/
9. FOREACH edge $e_{cqIdx} = (q_a, q_b)$ in Q's edges in order
10. $lbl = e_{cqIdx}.getLabel()$
11. $k = Q.getNumBits(lbl)$ /* According to the graph definition, there are no more than 2^k outgoing edges labeled lbl from any node in G*/
12. Set num to the integer represented by bits $< b_{bitIdx+k-1}, b_{bitIdx+k-2}, ..., b_{bitIdx} >$ of $thNum$. /* These bits corresponds to edge number $cqIdx$ in Q */
13. $currV = dataNodeArray[q_a.idx]$. /*the data graph node to which q_a is mapped*/
14. $currE = currV.getEdge(lbl, num)$ /*gets edge number num out of outgoing edges labeled lbl of node $currV$*/
15. IF ($currE == nil$) THEN RETURN
16. $currV = currE.getTargetNode()$ /*find q_b*/
17. IF (NOT $isMatching(currV, q_b)$) THEN RETURN
18. $dataNodeArray[q_b.idx] = currV$ /*update the mapping array*/
19. $bitIdx = bitIdx + k$ /*prepare $bitIdx$ to read next edge data*/
20. END FOREACH
21. $ansSet.add(thNum)$ /*current thread encodes an answer*/

Figure 3: The base GGQ algorithm

Figure 3 presents the base version of the GGQ algorithm. The input to the algorithm are the data graph G, the TPQ Q and the set V_q that contains the matching data node of the TPQ query root node. Line 2 contains the invocation of the CUDA kernel function $gpuGraphQuery$ which is processed on the GPU. I.e., the query processing algorithm itself is executed on the GPU.

The $gpuGraphQuery$ kernel call finds all the matchings between the TPQ Q and the data graph G. The code of $gpuGraphQuery$ is run in all the threads. They process exactly the same code (i.e., the code of the $gpuGraphQuery$ function itself) at the same time ("Single Instruction") over different data ("Multiple Data") in G. As the possible number of pattern matchings between Q and G is very large, there is a potential for an enormous number of parallel threads. According to the GPU GTX 480 architecture, the maximum number of resident threads per MP (multiprocessor) is 1536 (i.e., 1536*15 for all the MPs), while the number of threads that are processed at any moment of time in the MP is 32 (other threads may be waiting for data from the global memory, or just waiting for their turn to be run). The threads in the GPU are arranged in blocks where each block can have a maximum of 1536 threads. If the requested (by the algorithm) number of threads exceeds 1536*15, then the GPU first handles the 15 first blocks, and then continues to process the next 15 blocks, and so on until all the blocks are processed. Note that maximum number of threads that can actually run in parallel at any point of time is 480 (32 on each of the 15 MPs). The potential number of pattern matchings between Q and G is very large, and is usually much larger than the number of compute units in the GPU. Thus the utilization of the GPU is usually very high, i.e., the throughput of processing the work is high in comparison to multi-threaded CPU systems. By running a profiling tool on GGQ, we found that the execution uses the coalesced memory access feature of the GPU[3]. This is due to an apparent matching between the structure of the storage and the way the algorithm traverses the data.

Next, we explain the $gpuGraphQuery$ kernel function. Line 1 computes the thread's ID, namely $thNum$, according to CUDA's semantics. In line 3 we compute the index for which the current thread is responsible. In the base algorithm, $idConst$ is always 0. Thus, $thNum == thID$. $cqIdx$ that is defined in line 4, indicates the index of the currently processed edge. Line 6 defines the $bitIdx$ variable. $bitIdx$ points to the bit that is currently processed in the binary presentation of $thNum$. The $dataNodeArray$ array which is defined in line 7 holds the data nodes that are matched against query Q by the current thread. I.e., the element with index a_i of the $dataNodeArray$ is the data graph node d_i that is matched to the query node with index a_i. The size of $dataNodeArray$ is the number of nodes in Q, i.e., $|Q|$. In line 8, $dataNodeArray[1]$ is initialized. This is the data node that is matched to the root node of Q. This data node named v_r is taken from set V_q which is one of the parameters of the $gpuGraphQuery$ function.

In line 9 the algorithm starts a FOREACH, that tries to perform a matching between the pattern that is encoded by $thNum$ (the index of the current thread) and G according to TPQ Q. Note that before starting the algorithm, the edges of Q are sorted in a way that if edge e_x is on the path from Q's root to the source vertex of edge e_y, then e_x precedes e_y in the order. And this is the order in which they are processed during the FOREACH. In lines 10-12, the algorithm finds the edge number num that has to be chosen out of the outgoing edges labeled lbl of node $currV$ (a value is assigned to $currV$ in line 13). $currV$ is the node that is matched to the q_a node, which is the source node of the e_{cqIdx} edge. $currV$ is taken out of the $dataNodeArray$ according to the index of the q_a node. The ordering of Q edges (described above) guarantees that $currV$ exists. To find num, the algorithm first extracts the bits of the binary representation of $thNum$ that correspond to the e_{cqIdx} edge. The decimal value that is encoded by these bits is inserted to num. In line 14 the algorithm gets the outgoing lbl labeled edge number num of node $currV$ and assigns it to edge $currE$. If the value of $currE$ is nil, it means that such an edge does not exist, thus according to line 15 the algorithm terminates the run, as this thread does not encode a matching pattern in graph G. In line

[3]This means that when many threads in a warp access consecutive global memory addresses, these memory accesses are grouped into one access.

16-17, using edge $currE$, the algorithm finds the data node that matches to the query node q_b (the target node of edge e_{cqIdx}) and inserts it to $currV$, then it checks the matching between the data of the new $currV$ and the data of q_b. In case it finds that there is no matching between the data of $currV$ and q_b, it terminates the run, as this thread encodes a pattern that does not exist in graph G. In lines 18-19 the data of $dataNodeArray$ and $bitIdx$ is updated, as preparation to the next iteration of the FOREACH. If the algorithm finishes successfully the FOREACH loop for all the edges, without returning in lines 15 or 17, it means that the current thread encodes a pattern that exists in the graph and that fully matches Q. That is why in line 21, the algorithm inserts $thNum$ to $ansSet$.

There are possible optimizations of the basic scheme. As pointed out by a reader, one can base thread addressing on a simple algorithm that takes into account the maximum number of edges with a particular label emanating from a node and, based on the query and the thread ID, deduce the thread's search pattern. This will often result in fewer threads.

3.3 First algorithm extension (Brute Force Looping)

Input: 1) Data graph G. 2) TPQ query Q. 3) V_q set.
Output: $ansSet$, the set of all thread IDs that encode patterns that are an answer to query Q in data graph G.
Method (runs on the CPU):
1. $ansSet = \{\}$
2. $maxIDbitNum = getBinBitsNum(getMaxID())$
   /*$getMaxIDbits$ is a system function*/
3. $maxQBinNum = getBinBitsNum(Q)$
4. FOR ($i = 0; i < 2^{(\lfloor \frac{(maxBin+maxIDbits)}{maxIDbits} \rfloor)}; i++$)
5. Invoke CUDA kernel call for function:
6. $GpuGraphQuery(G, Q, V_q, ansSet, i)$
7. END FOR
8. RETURN $ansSet$

Figure 4: The first extension of the GGQ algorithm

There can be situations in which the maximal number of bits that may be required to represent query patterns is larger than the number of bits of maximal thread ID. Thus, we extend the algorithm as presented in Figure 4. Assume that the maximal thread ID is $maxID$ and that we need $maxIDbits$ to represent it, that $maxBin$ bits are required to represent the query pattern, and that $maxBin > maxIDbits$. In line 4 we start a FOR loop. The number of iterations is: $2^{(\lfloor \frac{(maxBin+maxIDbits)}{maxIDbits} \rfloor)}$. At each loop iteration (line 5), we run the base algorithm, where each thread in the current iteration will take care of the pattern represented by the following number: $(i * (maxID + 1) + threadID)$, where $maxID$ is the ID of the maximal thread ID, and $treadID$ is the system ID of the current thread. This computation can be seen in line 3 of the base algorithm (Figure 3). Note that this way, conceptually, we extend the thread's ID bit representation to the left by placing there the bits corresponding to i in the current loop iteration.

Often, when a query is posed, the desired answer is whether there exist any matching between the query tree and the data graph. In such cases, it is sufficient to find one matching in order to provide a positive answer. In a slightly modified version of the algorithm, the run is stopped the moment a first match is found. This feature decreases the running time of the algorithm in such cases. Sometimes, the desired answer to a query corresponds to only one specific query node and not to all nodes corresponding to the whole set of query nodes. This does not affect the GGQ algorithm as an answer provided by GGQ to a query is an ID of the thread.

3.4 Second algorithm extension (Multi Phase)

A substantial possible improvement, in case that the number of possible patterns is larger than the maximal thread ID, is a two phase exploration (and, in general, a multi phase exploration using the same principle). Here, we first limit the pattern by removing subtrees (actually edges leading to the roots of subtrees) so as to be left with the original rooted pattern with portions removed so that the remaining new pattern Q' is a "prefix" of the original pattern Q. The idea is that we have sufficiently many bits in $maxIDbits$ to explore the smaller Q' (with no need to use the first extension). A Q' node is called a *contact node* if it is a node in Q from which an edge leading to a subtree was removed along with the whole subtree. When evaluating Q' we record for each solution the images in the data graph of the contact nodes of Q' which we call a *recorded solution vector*. Then, we run the second phase in which, for each recorded solution vector, we explore the rest of Q using all the threads we can utilize. In case two phases are not sufficient, we grow Q' to Q in more than 2 phases. Each such phase will produce a collection of recorded solution vectors in which additional Q nodes are assigned values. The advantage of this two phase (and in general multi phase) scheme is that (a) We employ many threads in the first phase working on a smaller query derived from the original query and obtain all the relevant prefixes, encoded in recorded solution vectors, out of the data graph. (b) In the second phase, for each recorded solution vector, we employ all threads on a relevant portion of the data graph that can potentially lead to a solution to Q.

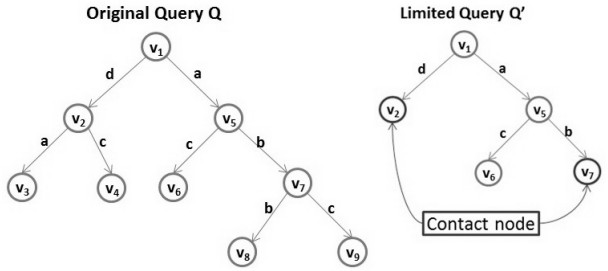

Figure 5: Example of limited Query

For example, consider the query Q as presented on the left side of Figure 5. Suppose that an edge representation requires 4 bits for any label, namely the whole pattern requires 32 bits. Suppose that $maxID$ requires 16 bits. So, we are "missing"İ 16 bits. We can transform Q to the limited query Q' with 4 less edges, as presented on the right side of Figure 5. This way we can handle Q' with all threads (whose $maxID$ requires 16 bits). Once we evaluate Q' we obtain recorded solution vectors. Each recorded solution vector encodes a partial matching of the full matching, and determines the *data contact nodes* v_a and v_b that are matching to the *query contact nodes* v_2 and v_7. When phase 2 is carried out for each recorded solution vector, each thread will operate on the subtrees rooted at v_2 and v_7 where the $dataNodeArray$ will be initialized with v_a in the location corresponding to v_2 and v_b in the location corresponding to v_7. As the subtrees rooted at the contact nodes have a total of 4 edges, 16 bits will suffice to represent all possible navigations. This means that in phase 2, when considering a particular recorded solution vector, **all** threads will be employed in checking possible continuations for this recorded solution vector. Thus, the computing power is fully utilized in (the short) phase 1 and later on throughout phase 2. Note that there is an advantage here over the loop scheme (that is presented in the first extension) in that for a loop index that corresponds to a non-prefix of the data graph, all

GPU threads are activated in vain. Here, the first phase guarantees that the sequence of GPU activations is done for recorded solution vectors that correspond to potentially extendable matchings. The disadvantage is that the recorded solution vectors need be stored so that they are available for the second phase.

Input: 1) Data graph G. 2) TPQ query Q. 3) V_q set.
Output: $ansSet$, the set of all thread IDs that encode patterns that are an answer to query Q in data graph G.
Method (runs on the CPU):
1. $prelimAnsSet = \{\}$
2. $prefixQ = getPrefixQ(Q)$ /*$getPrefixQ$ returns the "prefix" of the query Q*/
3. Invoke CUDA kernel call for function:
4. $\quad GpuGraphQuery(G, prefixQ, V_q, prelimAnsSet, 0)$
5. $ansSet = \{\}$
6. $remainQ = getRemainQ(Q, prefixQ)$ /*$getRemainQ$ returns $Q \setminus prefixQ$*/
7. FOREACH ans in $prelimAnsSet$
8. $\quad currAnsSet = \{\}$
9. $\quad$ Invoke CUDA kernel call for function:
10. $\quad\quad GpuGraphQueryExt(G, remainQ, currAnsSet, ans)$
11. $\quad ansSet = ansSet \cup currAnsSet$
12. END FOREACH
13. RETURN $ansSet$

// $GpuGraphQueryExt$ kernel function (runs on the GPU, just the differences from $GpuGraphQuery$ presented):
Input: 1) Data graph G. 2) forest $remainQ$.
3) $ansSet$ set of all answers (thread IDs).
4) $baseAns$ is the ID that encodes the matching between $prefixQ$ and G
Goal: In case that current thread's ID encodes an answer to query $remainQ$ based on the matching presented by $baseAns$ in data graph G, add it into $ansSet$.
Method:
...
3. $thNum = thID$
...
8. $initNodesArray(dataNodeArray, baseAns)$
/* $initNodesArray$ extracts $baseAns$, and fill all the nodes that already matched in dataNodeArray by answering $prefixQ$ in the first phase */
...

Figure 6: The second extension of the GGQ algorithm

Figure 6 presents the second extension to the algorithm. The function $getPrefixQ$ (line 2), decides which part of Q is going to be the "prefix" query. It makes the decision based on the number of bits $bLimit$ required to present the maximal thread ID, and on the structure of Q. Basically, it chooses the "upper" part of the tree (the part with the smallest depth), up to the limit of $bLimit$. I.e, it sums the number of bits that are required to present all the edges of the chosen part, and enlarges the chosen part up to the limit of $bLimit$. Lines 3,4 run the base algorithm on $prefixQ$, and insert the answer into $prelimAnsSet$. Line 5 initialize $ansSet$, the set of the final answers. $remainQ$ that is computed in line 6, is the remainder part of Q after removing $prefixQ$ out of it. Line 7 starts a FOREACH that computes the final answers for Q based on the preliminary answers from $prelimAnsSet$. The set of answers of the current iteration, $carrAnsSet$ is defined in line 8. Lines 9-10, contain the invocation of the CUDA kernel function $gpuGraphQueryExt$ which is processed on the GPU, and is slightly different from $gpuGraphQuery$ (as defined in Figure 3). In line 11 we add the answers that were found in the current iteration to the final answers set, namely $ansSet$.

Next we describe $gpuGraphQueryExt$. This function has slight differences from the base algorithm GPU function, $gpuGraphQuery$. Thus, we describe just these differences. The first difference is in line 8, in which the $thNum$ is defined. $thNum$ is equal to the system value of the ID of the current thread. The second difference is in line 8, in which the $initNodesArray$ function initializes $dataNodeArray$. The function extracts from $baseAns$ the matchings between nodes in Q and nodes in G that were found during the first phase, and assigns the found data nodes into the appropriate places in $dataNodeArray$. Except for the described two changes, the function operates exactly as the base $gpuGraphQuery$ function.

The number of edges that can be represented by one phase is n such that $\sum_{edge=1}^{n} outgNum(lbl(e)) \leq 2^{bitsNum(maxThreadID)}$ where $bitsNum(maxThreadID)$ is the number of bits that are used by the GPU to represent the maximal thread ID. For example, assume that a GPU thread ID is represented with 32 bits. Assume that for each edge e, on average, there are 16 potential $outgNum(lbl(e))$ from each node. Thus, on average, we need 4 bits to represent each edge of the query. Based on the above, each phase allows us to represent $32/4 = 8$ edges on average. Having 2 phases in the multi-phase extension described above allows us to represent fairly large TPQs with about 16 edges. The multi-phase extension can be easily extended to more than 2 phases. Based on this analysis, if we extend the multi-phase extension to 3 phases, we can represent a TPQ with 24 edges, which is a very large query. It is important to note that without the multi phase extension, experiments involving very large queries give very poor results that are worse than Gremlin's performance on these queries.

Input: 1) query edges ($qEdges$). 2) $maxQdepth$, the max depth of Q
3) $maxBitsNum$, the number of bits required to represent maximal thread ID
Goal: to set the field $phaseNum$ of each query edge
Method (runs on the CPU):
1. $currPhase = 1$
2. $currBitsSum = 0$
3. FOR $depth$ FROM 1 TO $maxQdepth$
4. $\quad$ FOREACH $edge$ IN $qEdges$
5. $\quad\quad$ IF $edge.depth == depth$
6. $\quad\quad\quad$ IF $currBitsSum + edge.bitsNum > maxBitsNum$
7. $\quad\quad\quad\quad currPhase++$
8. $\quad\quad\quad\quad currBitsSum = 0$
9. $\quad\quad\quad$ END IF
10. $\quad\quad\quad currBitsSum += edge.bitsNum$
11. $\quad\quad\quad edge.phaseNum = currPhase$
12. $\quad\quad$ END IF
13. END FOREACH
14. END FOR

Figure 7: Query phase ordering algorithm

Figure 7 presents the algorithm for breaking the query into phases.

4. EXPERIMENTAL EVALUATION

We compared GGQ to Gremlin [17] in terms of run time (to completion). Gremlin is the only query processor that we found that uses the *native graph* approach and that supports XPath-style queries over graph documents. Using Gremlin's query language, one can easily express TPQs. We are not aware of any parallel graph query processor to which we can currently compare our results. We used the GLS storage scheme to store the data. We implemented the GGQ algorithm from scratch on CUDA [7]. We experimented with GRR [10], a benchmark tool for generating random RDF documents. We also experimented with the Geospecies data document [2], and a representative data document example of the Census database [8]. We checked different TPQ query patterns. [4]

[4]Queries and data are available upon request.

	Path Q1	Path Q2	Speedup	
			Q1	Q2
$Gremlin$	114	84		
$GPU-full$	0.8	7.4	148	11
$GPU-ans$	0.09	0.08	1267	1050
$GPU-alg$	0.085	0.075	1425	1200

Figure 8: Results of GGQ on a document with size 125MB, for path queries with 5 nodes. The right two columns contain the speedup of GGQ run in comparison to a Gremlin run.

	Path Q1	Tree Q2	Speedup	
			Q1	Q2
$Gremlin$	81	75		
$GPU-full$	0.86	0.67	94	112
$GPU-ans$	0.09	0.12	900	625
$GPU-alg$	0.08	0.11	1012	682

Figure 10: Results of GGQ on a document with size 180MB, for a path query with 4 nodes and a tree query with 6 nodes. The right two columns contain the speedup of GGQ run in comparison to a Gremlin run.

All experiments were run on an 3 GHz Intel S5520SC ShadyCove 5520 12DDR3 6SATA/R 2LAN1000 EATX workstation having an NVIDIA GTX 480 GPU (with 1.5GB global memory), and having two Intel Xeon 6C X5650 processors (with 24GB of RAM in total). Each Xeon processor has 6 cores so altogether the workstation has 12 cores. We used the actual run time in various scenarios as the main metric of performance.

4.1 Experiments Description

Setting Up. An experiment run has two input files: an RDF document, and a text file with query (TPQ) patterns to run against the given document. An experiment begins with loading the input document into the *GLS storage* system by the parser. Then, we parse the queries, and process them against the input document. We used different TPQ patterns. The patterns we used have different length and of different tree structures.

Experiment Description. The document is first loaded to the GLS storage system (the time of loading is not measured, as it is a one time procedure). Every experiment has the following runs:

1. *Gremlin Run* - We process the queries in the queries text file using Gremlin [17]. Information regarding the run time of the query is collected in the result log file.

2. *GPU Run* - This run is performed using the GPU. We process the queries in the queries text file. The queries are processed by the GGQ algorithm as described in Section 3. Information regarding start and end times of processing the queries is collected in the result log file.

We compare the performance of GGQ to Gremlin by comparing the run time of these algorithms in three different ways. In the first way we start the time measurement for the GGQ algorithm before copying the data from the CPU to the global memory of the GPU, and stop after copying the result data from the GPU to the CPU (namely, *GPU-full*). In the second way we start the time measurement for the GGQ algorithm right after copying the data from the CPU to the global memory of the GPU, and before the query execution begins, and stop the time measurement right after finishing the query processing, but before copying the results data from the global memory of the GPU to the CPU (namely, *GPU-alg*). In the third way we start the time measurement for the GGQ algorithm right after copying the data from the CPU to the global memory of the GPU, and before the query execution begins, and stop the time measurement after copying the result data from the GPU to the CPU (namely, *GPU-ans*). *GPU-full* reflects the potential time improvement of the GPU for large documents that cannot fully reside in the global memory of the GPU. *GPU-ans* reflects the potential time improvement for documents that can fully reside in the global memory of the GPU. This is an important measurement as in a case that the document can fully reside in the GPU, we have to copy it to the GPU only once and then we can run many queries over this document in a row, by this eliminating the need for copying the document to the GPU per each query. *GPU-alg* is appropriate for GPUs in which the global memory and RAM are merged, i.e., in more recent processors such as NVIDIA's planned PASCAL GPU family. Time is measured is milliseconds. Each experiment is characterized by the size of the input RDF document. We experimented with documents sized as follows: 40MB, 125MB,180MB, 600MB. We did not use larger files, as the GRR benchmark tool was not able to create larger files. Also, the main factor that influences the complexity of GGQ is the size of the query and not the size of the RDF database document. Note that only the relevant edge streams have to be copied to the global GPU memory, so ordinarily the amount of data that is copied to the global GPU memory is usually much smaller than the document size.

	Path Q1	Tree Q2	Speedup	
			Q1	Q2
$Gremlin$	76	72		
$GPU-full$	3.2	3.17	24	23
$GPU-ans$	0.08	0.09	950	800
$GPU-alg$	0.075	0.095	1085	900

Figure 9: Results of GGQ on a document with size 600MB, for a path query with 5 nodes and a tree query with 6 nodes. The right two columns contain the speedup of GGQ run in comparison to a Gremlin run.

	Tree Q1	Tree Q2	Speedup	
			Q1	Q2
$Gremlin$	187	165		
$GPU-full$	49	5.4	3.8	30.6
$GPU-ans$	48	2.6	3.9	63.5

Figure 11: Results of GGQ on a document with size 180MB, for two different tree queries with 11 nodes. The right two columns contain the speedup of GGQ run in comparison to a Gremlin run.

4.2 Experiments

Figures 8 and 9 show the results of GGQ on GRR documents with sizes 125MB and 600MB respectively for different TPQ queries. The GGQ run with full memory transferring time (both directions) has speedup with respect to Gremlin of about 147 and 11 for a document with size 125MB and of 24 and 23 for a document with size 600MB, for Q1 and Q2 respectively. The GGQ run with result transferring time (from the global memory to the CPU) has speedup with respect to Gremlin of 1267 and 1050 for a document with size 125MB and of 950 and 800 for a document with size 600MB, for Q1 and Q2 respectively. The GGQ pure run (without transferring times) has speedup with respect to Gremlin of 1425 and 1200 for a document with size 125MB and of 1086 and 900 for a document with size 600MB, for Q1 and Q2 respectively. Due to lack of space, we shall not elaborate on all the experimentation Figures.

5. RELATED WORK

There are a growing number of initiatives to implement and commercialize Graph databases, such as Neo4j [6], HyperGraphDB [4] and DEX [3] and many RDF solutions such as Jena [5] and AllegroGraph [1]. There are other initiatives to create graph querying languages that enable a simplified user view of querying such as SPARQL [19] and Gremlin [17]. Another initiative for a graph query language is GraphQL that is presented in [12] in which the base node is a graph, so it deals with a graph of graphs. Thus, the answer to this query is a set of graphs; further, this work is not dealing with parallel query processing. Works in the area of parallelization of graph databases have started to appear. For example, parallelGDB [9] and papers that address parallelization that is based on graph partitioning. GPU-based work is [14] which proposes an efficient subgraph matching algorithm. It presents an implementation of the STwig algorithm [18] in which the third (join) step of the algorithm is performed in parallel on a GPU.

Lately, there are efforts to use GPUs to improve the performance of DBMSs. There are also new framework proposals, such as Medusa, a programming framework for parallel graph processing on GPUs. Medusa enables developers to leverage the massive parallelism and other hardware features of GPUs by writing sequential C/C++ code for a small set of APIs. Recent works, [15], propose efficient XML path processing algorithms using GPUs, which deal with path patterns. The current paper, on the other hand, deals with TPQs, which are more complex query patterns, looked for on more complex database structures.

6. CONCLUSIONS

We present the GGQ algorithm, a novel efficient algorithm for processing TPQ queries on graph documents. We use a new storage scheme, GLS, in a parallel multi-threaded computing platform, using a GPU as a CPU co-processor. GGQ employs techniques that allow it to run hundreds of threads in parallel.

We conducted extensive experimentation with GGQ. We compared, in terms of run time, GGQ to Gremlin [17], currently the only available tool for comparison, that supports XPath-style queries over graph documents. We checked performance for varying document sizes and for different queries. Experimental results indicate that using GGQ significantly reduces the run time of queries in comparison to Gremlin.

As part of future work, we plan to adapt the multi-phase scheme to oddly shaped graphs, e.g., ones with a few nodes, each having a multitude of edges. We also plan to extend GGQ to handle queries that are in the form of a directed graph. The idea is to first build a spanning forest out of the query graph. Then, to run the above algorithm on each tree in the forest. As the last step, to check all the answers for compatibility (namely, that the same query node is not mapped to different data graph nodes) and retain the answers that conform to the graph query structure.

7. REFERENCES

[1] Allegrograph: Modern, high-performance, persistent graph database. http://franz.com/agraph/allegrograph/.

[2] The apache xalan project. http://stats.lod2.eu/rdfdocs/769.

[3] Dex: High-performance and scalable graph database management system. http://www.sparsity-technologies.com/dex.

[4] Hypergraphdb. http://www.hypergraphdb.org/index.

[5] Jena: Java framework for building semantic web and linked data applications. http://jena.apache.org/.

[6] Neo4j: WorldŠs leading graph database. http://www.neotechnology.com/neo4j-graph-database/.

[7] Nvidia cuda c programming guide.

[8] Rdf data sets repository links. http://www.w3.org/wiki/DataSetRDFDumps.

[9] L. Barguñó, V. Muntés-Mulero, D. Dominguez-Sal, and P. Valduriez. Parallelgdb: a parallel graph database based on cache specialization. IDEAS '11.

[10] D. Blum and S. Cohen. Grr: Generating random rdf. In *ESWC (2)*, 2011.

[11] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD'02*.

[12] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. SIGMOD '08.

[13] J. Hensley. Amd ctm overview. In *SIGGRAPH'07*.

[14] X. Lin, R. Zhang, Z. Wen, H. Wang, and J. Qi. Efficient subgraph matching using gpus. In *Databases Theory and Applications*, Lecture Notes in Computer Science. 2014.

[15] R. Mousalli, R. Halstead, M. Salloum, W. Najjar, and V. J. Tsotras. Efficient xml path filtering using gpus. In *ADMS - VLDB Workshops*, 2011.

[16] NVIDIA. What is gpu-computing? http://www.nvidia.com/object/what-is-gpu-computing.html.

[17] M. A. Rodriguez. Gremlin, 2010.

[18] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *VLDB'12*.

[19] W3C. Sparql. http://www.w3.org/TR/rdf-sparql-query/.