# Practical and Theoretical Aspects of a Parallel Twig Join Algorithm for XML Processing using a GPGPU

Lila Shnaiderman*
Computer Science Department, Technion
Haifa 32000, Israel
lilas@cs.technion.ac.il

Oded Shmueli†
Computer Science Department, Technion
Haifa 32000, Israel
oshmu@cs.technion.ac.il

## ABSTRACT

With an increasing amount of data and demand for fast query processing, the efficiency of database operations continues to be a challenging task. A common approach is to leverage parallel hardware platforms. With the introduction of general-purpose GPU (Graphics Processing Unit) computing, massively parallel hardware has become available within commodity hardware.

XML is based on a tree-structured data model. Naturally, the most popular XML querying language (XPath) uses patterns of selection predicates on multiple elements, related by a tree structure. These are often abstracted by *twig patterns*. Finding all occurrences of such a (XML query) twig pattern in an XML document is a core operation for XML query processing.

We present a new algorithm, GPU-Twig, for matching twig patterns in large XML documents, using a GPU. Our algorithm uses the data and task parallelism of the GPU to perform memory-intensive tasks whereas the CPU is used to perform I/O and resource management. We therefore efficiently exploit both the high-bandwidth GPU memory interface and the lower-bandwidth CPU main memory.

## 1. INTRODUCTION

XPath, the XML Path Language [11], is a query language for selecting nodes from an XML [44] document. The XPath language is based on a tree representation of the XML document, and provides the ability to navigate within the tree, selecting nodes by a variety of patterns. For example, consider the following XPath expression: "movie[name = 'Harry Potter']//actor[first='Daniel' AND last = 'Radcliffe']". This expression matches *'actor'* elements that (i) have a child element named *'first'* with content *'Daniel'*, (ii) have a child element *'last'* with content *'Radcliffe'*, and (iii) are descendants of *'movie'* elements that have a child *'name'* element with content *'Harry Potter'*. This expression can be represented as a node-labeled tree called a *twig pattern*. The result of the matching between the given twig pattern and the XML database is the set of all the elements in the XML database with an *'actor'* label which satisfy the matching described above, blurring the distinction between a document and a database. Finding all occurrences of a twig pattern, or specific data nodes corresponding to a specific twig node, in an XML database are fundamental operations in XML query processing.

As XPath is a critical component in many XML-based applications, it is essential to maximize its performance. There are many works dealing with optimizing performance of a single query [15], and a few studies deal with parallelizing the execution of a single XPath query. Most state-of-the art XPath processors, such as Apache's Xalan, support parallel XPath queries, i.e., numerous XPath queries which are processed concurrently by different threads against the same query processor instance. Still, each XPath query is executed serially. By processing XPath queries concurrently, the overall latency is improved, but to enhance the overall performance we need to increase the speed of processing *a single* XPath query via parallelization. This is particularly important for large databases. Parallelization of XPath queries is also essential for parallelizing various host languages, such as XSL [46] or XQuery [45].

Recently, there have been attempts to use GPUs to boost the performance of various database algorithms. GPUs, also known as video cards, are regularly used to render graphical information. GPUs near universal use in desktop computers means that it is a cheap and ubiquitous source of processing power. There is a growing interest in applying this power to more general non-graphical problems through frameworks such as NVIDIA's CUDA, an application programming framework providing programmers a simple and standard way for executing general purpose programs on NVIDIA GPUs. Modern GPUs, with massively parallel execution architecture, have become powerful co-processors for many applications, including scientific computing [18] and databases [10, 47, 21].

Streams provide an important way for representing an XML document [7]. There is a dedicated stream for each element label in the document. Each stream contains the positional representations of the XML nodes whose label matches the particular element label of the stream.

The problem we address is how to use GPUs to speed up twig query processing. In this paper we present the GPU-Twig algorithm. The main idea underlying this algorithm is to copy the relevant parts of the database according to the input query to the GPU, to process the input query on the GPU, and to copy back to the CPU the query results. The key to parallelizing the query answering process, while using a stream representation scheme, is in the ability to expose some of the structural characteristics of the document tree within the streams. This is achieved by broadening the

*L. Shnaiderman was funded in part by ISF grant 1104/05.
†O. Shmueli was funded in part by ISF grant 1104/05.

classic stream scheme, such that each node in the stream contains additional information about its ancestors.

Previous work dealt with possible strategies for parallelizing XPath queries [6]. To the best of our knowledge, we present the first parallel algorithm (GPU-Twig) that uses GPUs to accelerate the processing of a single query, when a stream representation of documents is used.

The results of an extensive experimentation are presented in a separate publication.

**Organization**

The rest of the paper is organized as follows. Section 2 provides background and briefly reviews the GPU architecture. Section 3 presents the GPU-Twig algorithm and proved its correctness. Section 4 presents related work. Section 5 presents conclusions and future work directions.

# 2. BACKGROUND

In this section, we briefly introduce GPUs and CUDA (the underlying platform upon which our algorithm is implemented), the XML stream scheme we use, and twig pattern matching.

## 2.1 Graphics Processors (GPUs)

GPUs, originally designed for graphics rendering tasks, have evolved into massively multi-threaded many-core co-processors for CPUs, and are widely available as commodity components in modern machines [1, 34]. GPU programming languages include graphics APIs such as OpenGL [33] and DirectX [4], and GPGPU framework such as NVIDIA's CUDA [32]. Using these APIs, programs consist of two kinds of code, the kernel code and the host code. The host code runs on the CPU. The host part is in charge of transferring data between the GPU and main memory, and starting kernels on the GPU. The kernel code is executed on the GPU. Programmers write their algorithms so that one part of the algorithm's task runs on the CPU and the other part, which can be massively parallelized, runs on the GPU. In general, 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 the 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.

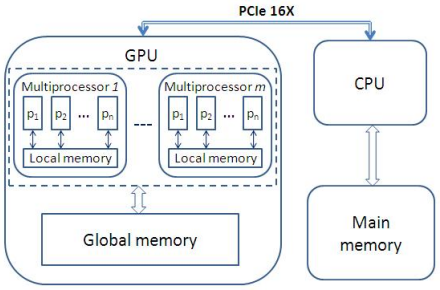


**Figure 1: GPU architecture model**

The GPU architecture is illustrated in Figure 1. This architecture is common for both AMD [22] and NVIDIA GPUs [32]. The GPU consists of many SIMD (Single Instruction, Multiple Data) multiprocessors (MPs), all sharing global memory (with size up to 4GB), which has both a high bandwidth and a high access latency. For example, the GTX480 GPU has access latency of 400-800 (according to [32]) cycles and a memory bandwidth of 177 GB/second. The latest GPUs, e.g., the GTX480, also have an L2 cache. CUDA, a general-purpose programming framework for NVIDIA GPUs, exposes a hierarchy of GPU threads. GPU threads execute the same code of a kernel, concurrently, on different data. The GPU supports thousands of concurrent threads. 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. Warps, each of which consists of the same number of threads, are scheduled across MPs (each warp belongs to one MP). Within each MP, warps are further grouped into thread blocks. Threads in the same thread block share resources on one MP, e.g., registers and local memory (also called *shared memory* in NVIDIAs term). The size of this shared memory is small and its access latency is low. GPU threads have both low context-switch and low creation time as compared to CPU threads.

The GPU has a hardware feature that is called *coalesced access* [1]. Coalesced access means that when many threads in a warp access consecutive global memory addresses, these memory accesses are grouped into one access. By that, the number of global memory accesses is significantly reduced, and memory bandwidth utilization is improved.

## 2.2 XML (Extended) Stream Scheme

An XML database is a forest of rooted, ordered, labeled trees, each node corresponding to an element or a value, and the edges representing (direct) element-sub element or element-value relationships. The ordering of sibling nodes implicitly defines a total order on the nodes in a tree, obtained by a preorder traversal of the tree nodes.

In the classic stream scheme, the position of a string or element occurrence in the XML database is represented as a 3-tuple (*DocId*, *LeftPos*:*RightPos*, *LevelNum*), where (i) *DocId* is the identifier of the document; (ii) *LevelNum* is the nesting depth of the element (or string value) in the document; and (iii) *LeftPos* and *RightPos* are the open and close indexes of the element in the data tree, which are assigned according to an order given by a DFS variation, as follows. Document *DocId* tree is scanned in DFS order. We use a variable called *counter* initialized to zero. When visiting some element $n$, we increment *counter*, then assign its value to the *LeftPos* of $n$, then scan the subtree of $n$ (in DFS order). After finishing the scan of the subtree of $n$, we increment *counter*, and assign its value to *RightPos*.

Let $lbl$ denote a label in a database. Associated with each distinct $lbl$ there is a stream $S_{lbl}$. The stream contains the positional representations of the database nodes with the $lbl$ label. The nodes in the stream are sorted by their (*DocId*, *LeftPos*) values.

For use in the algorithm, we extended the stream scheme, with additional structural information. The first additional field is added to each node is the $ancStreamsL$ list. Let $n$ be a node in the data tree, its $ancStreamsL$ list holds information about all the streams that contain at least one node that belongs to the path between $n$ and the root of the data tree (i.e., ancestors of $n$). The information that is held for each such stream is as follows: the name of the stream, and the index in the stream of a node that belongs to the path between $n$ and the root which is closest to the root. The second additional field that is added to each node is a Boolean array, *qArray*. Its size is equal to the number of nodes in the twig pattern (see definition bellow).

[1]This feature is not used in our particular application.

Each node in the document tree appears in a single stream (the stream of the node's label). In this scheme, each distinct leaf label also has a separate stream. This is very wasteful, as it causes a significant memory overhead. There are standard ways to overcome this problem. One is to assign a single stream for all labels of leaves; another one is to store all the leaves in a B-tree (according to the string order as the primary key and to the *LeftPos* order as the secondary key).

A stream is held in the memory as array of structs, while each struct contains the data as described above. This way is convenient while working with the GPU, as it is a compact way to store information, and the access to each node is easy and fast especially if the index of the node in the stream is known.

## 2.3 Twig Pattern Matching

The *twig pattern* is a rooted, ordered, labeled tree whose nodes' labels are either element tags, attribute-value comparisons or string values, and edges are either parent-child edges (depicted using a single line) or ancestor-descendant edges (depicted using a double line). For example, the XPath expression: movie[name='Harry Potter']//actor[last='Radcliffe']" which matches movie elements that (i) have a child element named *'name'* with content *'Harry Potter'*, (ii) have a descendant element *'actor'* with a child element named *'last'* with content *'Radcliffe'* can be represented as the twig pattern in Figure 2.

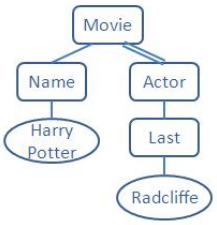


**Figure 2:** **Example of twig pattern**

We add two additional fields to each node in the twig pattern. The first field is the number (named *lSibNum*) of left siblings having the same label. The second field is a Boolean field named *isAnswer*.

For simplicity we assume that a twig pattern contains only ancestor-descendant edges. When two nodes are connected with an ancestor-descendant edge, we call the node that is closer to the root the parent node, and the node that is farther from the root the child node.

Given a twig pattern $Q$ with nodes ($q_1$,..., $q_n$) and an XML database $D$, a *match* of $Q$ in $D$ is a mapping from nodes in $Q$ to distinct nodes ($d_1$,..., $d_n$) in $D$ such that: (i) $d_i$ is matched with $q_i$, $1 \leq i \leq n$, (ii) $d_i$ and $q_i$ have the same label, and (iii) the structural (parent-child and ancestor-descendant) relationships between query nodes are satisfied by the corresponding database nodes.

In query $Q$, a single node is marked as the *target node* (technically, field *isAnswer* is set to $true$). A database node $d$ that is the image of the target node of $Q$ in a match is called an *answer node*. The problem we solve is that of computing all answer nodes for query $Q$ on database $D$. A related problem, called the *Twig Pattern Matching* problem, not addressed here, is that of computing all matches of $Q$ in $D$, where each match is represented as a vector ($d_1$,..., $d_n$) corresponding to $Q$'s nodes $q_i$, $i \leq i \leq n$.

# 3. THE GPU-TWIG ALGORITHM

The main special characteristic of the GPU-Twig algorithm is the ability to divide the work to hundreds or even thousands of threads that run in parallel. The GPU-Twig algorithm does not make use of most of the GPU features (like shared memory or coalesced memory access). The main GPU-specific features that we use are as follows. The first is the potential massive parallelization. The second is the fact that if there is massive work against the memory (such that at each moment there is data that has to be brought from the global memory), the overall time of bringing piece of data from the global memory by the GPU is smaller than bringing data from the RAM by the CPU due to the high bandwidth.

GPU-Twig processes only the document parts that are relevant to the input query. In other words, it processes only streams whose labels appear in the input twig pattern. In the first phase of the algorithm, it goes over all the relevant streams and derives additional structural information according to the input query. In the second phase of the algorithm, this structural information is used to produce the answer set.

To gain intuition about the algorithm, we start with some definitions. Next, we present the algorithm itself, and finally we present a correctness proof for the algorithm.

## 3.1 Definitions

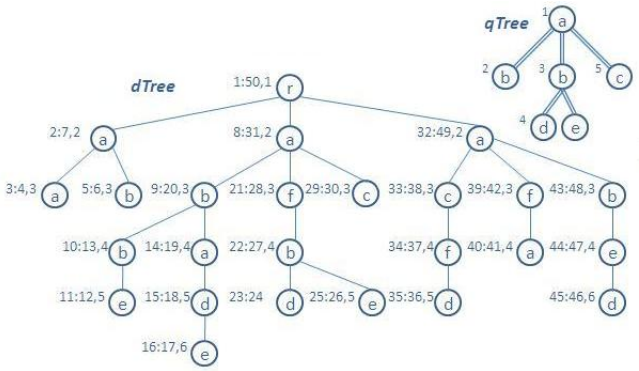


**Figure 3:** **Example for a *qTree* and a *dTree***

First, we define when a node $n$ of data tree $dTree$ is *qualifying* w.r.t. a node $q$ of twig pattern $qTree$. The intuition behind this definition is that if $n$ is *qualifying* w.r.t. $q$, then the subtree rooted at node $n$ satisfies the query requirements of the subtree rooted at node $q$. So, it means that $n$ is part of at least one match to the query represented by the subtree rooted at $q$ in $qTree$.

**Definition 1:** Let $q$ be a node in $qTree$ labeled $qLabel$, let $n$ be a node labeled $qLabel$ in $dTree$. We define, inductively on the height $h$ of $qTree$, when $n$ is *qualifying* w.r.t. $q$ as follows.

$h = 0$, $q$ is a leaf node in $qTree$: Node $n$ is *qualifying* w.r.t. node $q$ if they both have the same label, namely $qLabel$.

$h > 0$, $q$ is a non-leaf node in $qTree$: Node $n$ is *qualifying* w.r.t. node $q$ if: (1) $n$ and $q$ have the same label, namely $qLabel$. (2) there is a bijection between all children of node $q$ in $qTree$, and a subset of the descendants of node $n$ in $dTree$ such that each descendant in the above subset is *qualifying* w.r.t. the corr. child $qC$ of $q$. The order between the children does not have to be preserved by their bijection images. ■

*Example 1.* For the twig pattern $qTree$ and the $dTree$ in Figure 3, nodes (10:13,4), (9:6,3), (22:27,4) and (9:20,3) in *dTree*are *qualifying* w.r.t. the node with index 2 in *qTree*. Another example is node

(8:31,2) in *dTree*. Node (8:31,2) is *qualifying* w.r.t. the node with index 1 in *qTree*. This is because it has child labeled $c$ (29:30,3) that is *qualifying* w.r.t. the node labeled $c$ with index 5, a descendant labeled $b$ ((22:27,4) or (9:20,3)) that is *qualifying* w.r.t. the node labeled $b$ with index 3, and an additional descendant labeled $b$ ((9:20,3) or (10:13,4)) that is *qualifying* w.r.t. node $b$ with index 2. Node (32:49,2) has descendants that are *qualifying* w.r.t. nodes with index 3 and 5, but it has no additional distinct descendant labeled $b$, i.e., node (32:49,2) does not have 3 distinct descendants, that are *qualifying* w.r.t. nodes with index 2, 3 and 5. Therefore, node (32:49,2) is not qualifying w.r.t. the node with index 1. □

Now, we define when a node $n$ in $dTree$ has $k$ *relatives* labeled $qLabel$ w.r.t. node $p$ labeled $pqLabel$ in $qTree$.

**Definition 2:** For twig pattern nodes $q$ labeled $qLabel$ and node $pq$ labeled $pqLabel$ in $qTree$ (where $pq$ is the parent of $q$), and for node $n$ labeled $qLabel$ and node $p$ labeled $pqLabel$ in $dTree$, $n$ has $k$ *relatives* w.r.t. $p$ if $n$ is a descendant of $p$ and there are at least $k$ distinct nodes, different than $n$, labeled $qLabel$ in the subtree rooted at $p$. These nodes are called *relatives* of $n$. ■

Consider a query that is represented by the subtree rooted at $pq$ (labeled $pqLabel$), where $pq$ is the parent of $q$, and some node $n$ labeled $qLabel$ in $dTree$ that has an ancestor node $p$ labeled $pqLabel$. In order for node $n$ to be part of at least one match to the query represented by the subtree rooted at $pq$, three conditions must be satisfied. The first is that node $n$ has *to be qualifying* w.r.t. $q$. The second is that there has to be a bijection from a subset of the relatives of node $n$ w.r.t. $q$ and $p$ to the set of left siblings of $q$ with the same label as $q$. And, the last condition is that $p$ must have an additional subset of distinct descendants (that are different from the descendants used in the second condition) that are *qualifying* w.r.t. all the other siblings of $q$ (which are not left siblings with the same label as $q$). The next definition formally defines the second condition.

**Definition 3:** For twig pattern nodes $q$ labeled $qLabel$ and $pq$ labeled $pqLabel$ in $qTree$ (where $pq$ is the parent of $q$) and for node $p$ labeled $pqLabel$ in $dTree$, node $n$, a descendant of $p$ in $dTree$ (labeled $qLabel$) is *left qualifying* w.r.t. nodes $q$ and $p$, if $q$ has exactly $lSibNum$ distinct left siblings labeled $qLabel$, and if node $n$ has at least $lSibNum$ distinct relatives labeled $qLabel$ w.r.t. $p$, and there is a bijection between the left siblings of $q$ and a subset of said relatives of $n$ such that each relative of $n$ is *qualifying* w.r.t. the corr. left sibling of $q$. The order between the left siblings does not have to be preserved by their images. ■

Observe that node $n$ *is* not necessarily qualifying w.r.t. $q$.

*Example 2.* For the twig pattern $qTree$ and the $dTree$ in Figure 3, nodes (9:20,3) and (22:27,4) in $dTree$ are *qualifying* w.r.t. the node with index 3, and *left qualifying* w.r.t. the node with index 3 (matches the $q$ node in the definition) and node (8:31,2) (matches the $p$ node in the definition). The node with index 3 (in $qTree$) has one left sibling (the node with index 2). Each one of nodes (9:20,3) and (22:27,4) has more than one distinct relatives that are *qualifying* w.r.t. the node with index 2. For example, the relatives of (22:27,4) are ((9:20,3), (10:13,4)). Node (10:13,4) does not satisfy the requirements of *being qualifying* w.r.t. the node with index 3 and *left qualifying* w.r.t. the node with index 3 and node (8:31,2), as it is *not qualifying* w.r.t. the node with index 3 even though it is *left qualifying* w.r.t. the node with index 3 and node (8:31,2). □

As we saw in Section 2, each node in a stream has a field named *qArray* which is a Boolean array. The *qArray* is used to encode the additional structural information that is derived in the first phase of the algorithm. In the following definition we define when bit $i$ in the $qArray$ of node $n$ is *set correctly* w.r.t. some node $q$ in $qTree$, where $0 \leq i <$ the number of nodes in $qTree$. Intuitively, if bit $i$ in the $qArray$ of node $n$ labeled $qLabel$ is *set correctly* w.r.t. node $q$ labeled $qLabel$, then $n$ has some descendant node $m$ whose subtree satisfies the query requirement w.r.t. the subtree rooted at node $qChild$ with index $i$ that is a child of node $q$.

**Definition 4:** Let $n$ be an arbitrary node in $dTree$ labeled $qLabel$. Let $q$ be a node labeled $qLabel$ in $qTree$. Let $qChild$ be a child of $q$ indexed $i$ and labeled $decLabel$. Bit $i$ in the $qArray$ of node $n$ is *set correctly* w.r.t. node $q$ if the following holds. The value of bit $i$ is $true$ iff: (1) node $n$ has a descendant node $m$ labeled $decLabel$, (2) node $m$ is *qualifying* w.r.t. $qChild$, and (3) node $m$ is *left qualifying* w.r.t. $qChild$ and $n$. ■

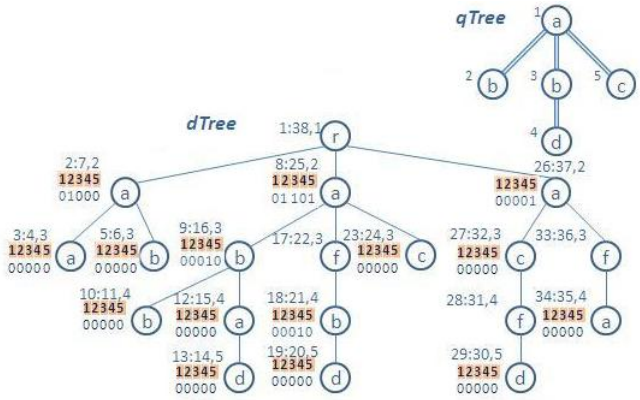


**Figure 4:** **Example of a *qTree* and a *dTree***

To indicate if the subtree rooted at node $n$ (in $dTree$) satisfies the query requirements w.r.t. the subtree rooted at node $q$ (in $qTree$) using the $qArray$, we define the *subtree-correctness*.

**Definition 5:** The $qArray$ structure of node $n$ labeled $qLabel$ is *subtree-correct* w.r.t. node $q$, labeled $qLabel$ in $qTree$, if for every child of $q$, say with index $i$, bit $i$ of the $qArray$ is *set correctly*, and its value is $true$. ■

*Example 3.* For the twig pattern $qTree$ and for the $dTree$ of Figure 4, node (8:25,2) is *subtree-correct* w.r.t. the node with index 1. This is because, for every child of node with index 1 in $qTree$, say with index $i$ (children indices are 2,3,5), bit $i$ of the $qArray$ of node (8:25,2) is *set correctly*. Bit 2 is *set correctly* to $true$ because node (8:25,2) has descendants that are *qualifying* w.r.t. the node with index 2 and *left qualifying* w.r.t. the node with index 2 in $qTree$ and node (8:25,2) (nodes (10:11,4), (9:6,3) and (18:21,4)). Bit 3 is *set correctly* to $true$ because node (8:25,2) has descendants that are *qualifying* w.r.t. the node with index 3 and *left qualifying* w.r.t. the node with index 3 and node (8:25,2) (nodes (9:16,3) and (18:21,4)). Bit 5 is *set correctly* to $true$ because it has a child that is *qualifying* w.r.t. the node with index 5 and *left qualifying* w.r.t. the node with index 5 and node (8:25,2) (node (23:24,3)). □

## 3.2 The Algorithm

The algorithm has two phases. The goal of the first phase is to derive the additional structural information for all the nodes in the streams that correspond to the twig pattern nodes. The first phase of the algorithm works in a bottom up manner over the $qTree$ twig pattern. It chooses some node $q$ such that all its children and left siblings are already processed. I.e., the first nodes to be processed are the leaves that have no left siblings. For every node $q$ in $qTree$, the algorithm processes all the nodes in the stream that corresponds to $q$. Suppose that the label of $q$ is $qLabel$, the index of $q$ in $qTree$ is $qIdx$, and the parent of $q$ is $pq$ labeled $pqLabel$. For every node

in the $qLabel$ stream, the first phase of the algorithm updates all the relevant $qArray$ structures of nodes in the $pqLabel$ stream, such that after processing all the nodes in the stream, for each node $n$ in the $pqLabel$ stream that has a descendant in the $qLabel$ stream that is *qualifying* w.r.t. $q$, bit $qIdx$ in the $qArray$ of node $n$ is *set correctly* w.r.t. $pq$. Note that we can not process $qTree$ nodes in parallel, as the result of processing node $q$ relies on the fact that all its children and left siblings are already processed. Another important issue is to understand the bottleneck of our algorithm. Our algorithm makes a very small amount of computations per data unit, while most of the work is to go over the algorithm's data which means that it is memory intensive. So the bottleneck of our algorithm is the global memory bus bandwidth.

Before starting the first phase, all the relevant information for the query is copied using $cudaMemcpy$ calls. The relevant information includes the streams of each query node and the query nodes data itself.

Figure 5 depicts the first phase of the GPU-Twig algorithm. The input to the first phase are the data tree $dTree$, and the twig pattern $qTree$. Line 4 contains the invocation of the CUDA kernel function $gpuTwigFirstPhase$ which is processed on the GPU. I.e., the derivation of the additional structural information of $dTree$ is executed on the GPU, while the task of the CPU is to run the outer loop over $qTree$ nodes, and to prepare the information for the GPU.

The $gpuTwigFirstPhase$ kernel call *sets correctly* bit $qIdx$ in the $qArray$ of all nodes $pn$ in the $pqLabel$ stream w.r.t. node $pq$. The code of $gpuTwigFirstPhase$ is run for every node in $qStream$, which is the "Multiple Data", while the "Single Instruction" is the code of the $gpuTwigFirstPhase$ function itself. This provides the potential for an enormous number of parallel threads, as the number of nodes in such a stream can be in the tens and even hundreds of thousands. In GTX 480, the maximum number of resident threads per MP (multiprocessor) is 1536 (i.e., 1536*15 for all the MPs), while the maximum number of threads that can actually run in parallel at any point of time is 480 (32 on each of the 15 MPs). As the number of nodes in a stream is usually much larger than the number of compute units in the GPU, the utilization of the GPU is very high, i.e., the throughput of processing the work is high in comparison to multi-threaded CPU systems. The number of nodes in different streams can vary. This fact does not present any special problem, as usually the number of nodes in a stream of a large document is much larger than the number of GPU compute units.

Note that, for speeding up job processing, our algorithm uses mainly one feature of the GPU, namely that of potentially high parallelism. It does not use the shared memory, as each piece of data has to be read only once. The coalesced access data feature is also not used as this feature requires data accesses to be ordered, which does not happen, as the algorithm works with data that represents a tree, which is unordered data. The loop in line 4 is run only for nodes labeled $pqLabel$ that are ancestors of node $n$. If the query pattern is allowed to contain the child axis, then line 4 of the algorithm has to be changed as follows. In case that the edge between node $q$ and its parent is parent-child (and not ancestor-descendant), then instead of the $FOREACH$ loop in line 4, we need to run lines 5-8 only for a single node, which is the parent node of $n$ labeled $pqLabel$ (if such a node exists). Parent nodes can be identified easily, by checking that the level of a potential parent node is one less than the level of node $n$. Line 7 checks if node $n$ fulfills the conditions needed to set bit $qIdx$ in the $qArray$ of node $pn$ according to Definition 4. I.e., whether node $n$ is *qualifying* w.r.t. $q$ (checked by the $subtreeCorrect$ function) and is *left qualifying* w.r.t. $q$ and

**Input**: 1) Data tree $dTree$. 2) Twig pattern $qTree$.
**Goal**: Build the additional structural information for all nodes in the streams that correspond to $qTree$ nodes.
**Method**:
1. WHILE there are unprocessed nodes in $qTree$:
2. Choose node $q$ from $qTree$ such that all its child nodes and all its left siblings were already processed
3. SET $qStream$ to $q$'s stream, $qIdx$ to $q$'s index, $pqStream$ to $q$'s parent stream
4. Invoke CUDA kernel call for function: $gpuTwigFirstPhase(qStream, pqStream, qIdx)$
5. Mark $q$ as processed
6. End WHILE

$gpuTwigFirstPhase$ **kernel function (runs on GPU):**
**Input**:
1) $qStream$: index of $q$ stream.
2) $pqStream$: index of $q$'s parent stream.
3) $qIdx$ the index of $q$.
textbfGoal: For each node $n$ in the $pqStream$ that has a descendant in the $qStream$ that is *qualifying* w.r.t. $q$, bit $qIdx$ in the $qArray$ of node $n$ is *set correctly* w.r.t. $q$'s parent.
**Method**:
1. set $idx$ to $blockDim.x * blockIdx.x + threadIdx.x$
2. IF $idx \geq$ number of nodes in $qStream$ then RETURN
3. set $n$ to node at index $idx$ of $qStream$
4. FOREACH node $pn$ in the $pqLabel$ stream that is an ancestor of $n$
5. SET $set_1$ to contain all the distinct left siblings of $q$ labeled $qLabel$
6. SET $set_2$ to contain all the descendants of $pn$ labeled $qLabel$ except for $n$
7. IF $subtreeCorrect(q, n) == true$ and $leftSiblingMatch(set_1, set_2) == true$ THEN
8. $pn.qArray[qIdx] = true$
9. END FOREACH

// $leftSiblingMatch$ **function:**
**Input**:
1) $set_1$: subset of $qTree$ nodes.
2) $set_2$: subset of $dTree$ nodes.
**Output**: Boolean value.
**Method**:
1. FOREACH subset $ss_2$ of $set_2$ of cardinality $|set_1|$
2. FOREACH bijection $bjc$ from $ss_2$ to $set_1$
3. IF for each pair of nodes $x \in ss_2$ and $y \in set_1$ in $bjc$ $subtreeCorrect(y, x) == true$ THEN
4. RETURN $true$
5. END FOREACH
6. END FOREACH
7. RETURN $false$

// $subtreeCorrect$ **function:**
**Input**: 1) $q$: node in $qTree$. 2) $n$: node in $dTree$.
**Output**: Boolean value.
**Method**:
1. $res = true$
2. FOREACH child $qC$ of $q$
3. SET $i$ to the index of $qC$
4. IF $n.qArray[i] == false$ THEN
5. $\{res = false$
6. BREAK$\}$
7. END FOREACH
8. RETURN $res$

**Figure 5: The first phase of the GPU-Twig Algorithm**

$pn$ (checked by the $leftSiblingMatch$ function). In case it is, we set bit $qIdx$ of the $qArray$ of node $pn$ to $true$ (line 8).

The $subtreeCorrect(q, n)$ function checks if node $n$ is *subtree-correct* w.r.t. $q$ according to Definition 5. The invariant maintained

by this function is that the value of all the bits in the $qArray$ of $n$ that correspond to the children of $q$ are already *set correctly* w.r.t. $q$. In Section 3.3, we prove that when we call the $subtreeCorrect$ function for some $q$ and $n$, the above invariant holds. So, to check if node $n$ is *subtree-correct* w.r.t. $q$, according to Definition 5, it suffices to check if for every child of $q$, say with index $i$, bit $i$ of the $qArray$ of $n$ is $true$. This is exactly what is done in the $subtreeCorrect$ function in lines (2-7). When we find that some bit $i$ in the $qArray$ of $n$ that corresponds to some child of $q$ is $false$ (line 4), it means that $n$ is *not subtree-correct* w.r.t. $q$.

The $leftSiblingMatch(set_1, set_2)$ function checks if there is a bijection from some subset $ss_2$ of $set_2$ to $set_1$ such that for each pair of nodes $x \in ss_2$ and $y \in set_1$ in the bijection the following holds: for each child of $y$ indexed $i$, the value of bit $i$ in the $qArray$ structure of $x$ is $true$. This is in fact the problem of finding a perfect matching in a bipartite graph. The best known complexity of the problem for graph $G$ with $V$ nodes and $E$ edges, is $O(\sqrt{V}E)$. According to the first phase of the algorithm, $set_1$ contains all the distinct left siblings of $q$ labeled $qLabel$ while $set_2$ contains all the descendants of $pn$ labeled $qLabel$ except for $n$. So, as proved in Section 3.3, the $leftSiblingMatch$ function checks if node $n$ is *left qualifying* w.r.t. nodes $q$ and $pn$ according to Definition 3. $set_1$ is always very small (a few nodes), $set_2$ is usually small (a few tens of nodes), so according to the above complexity, the run time of this function is usually small. In very rare cases in which $set_2$ is large, the run time of the function begins to be significant.

The goal of the second phase of the algorithm is to find the answer nodes of a $qTree$ twig pattern in $dTree$ using the $qArray$ structures of nodes in $dTree$ that were derived in the first phase of the algorithm.

The second phase of the algorithm processes all nodes in stream $s$ corresponding to the label of the $qAnsN$ target node whose $isAnswer$ field (see Section 2) is set to $true$. This is because each node $currSN$ in the $s$ stream is potentially an answer. We define a sub-query $qPath$ as the path between the answer node $qAnsN$ and the twig pattern root node. In order for a node $currSN$ to be an answer, we need to check that there is at least one match of $qPath$ in $dTree$ such that the $qAnsN$ node maps to $currSN$ and the $qArray$ of each node in the match (according to match definition) is *subtree-correct* w.r.t. the matched node in $qPath$. The algorithm works bottom up. It starts from node $currSN$ and checks if $currQN$ that is initialized to $qAnsN$'s parent ($pCurrQN$) maps to some ancestor of node $currSN$, called $ancN$, such that the $qArray$ of $ancN$ is *subtree-correct* w.r.t. $pCurrQN$. In case that such $ancN$ is found, we update $currSN$ to be $ancN$ and $currQN$ to be $pCurrQN$, otherwise we stop. This is because not finding such an $ancN$ means that the match we are looking for does not exist. We continue this process until we find the desired match, or until we fail to find it (as explained above).

Figure 6 presents the second phase of the GPU-Twig algorithm. The input to the second phase are the source document tree $dTree$, and the twig pattern $qTree$. The output of the second phase is a set $ansSet$ with all the answer nodes for $qTree$ in $dTree$. Line 3 contains the invocation of the CUDA kernel function called $gpuTwigSecondPhase$ that is processed on the GPU. In line 4, we prepare the $ansSet$ set, by scanning the $ansLabel$ stream for nodes whose $isAnswer$ bit is set to $true$.

The $gpuTwigSecondPhase$ kernel call sets the $isAnswer$ bit of all the answer nodes for $qTree$ in $dTree$ (this bit is initialized to $false$). The code of $gpuTwigSecondPhase$ is run for every node in $qStream$. As before, this provides a potential for an enormous number of parallel threads. Lines (5-20) check that there is at least one match of $qPath$ (defined above) in $dTree$ such that

**Input**: 1) Data tree $dTree$. 2) Twig pattern $qTree$.
**Output**: $ansSet$, the set of all answer nodes of $qTree$ in $dTree$.
**Method**:
1. SET $qAnsN$ node to be the node in $qTree$ whose $isAnswer$ field value is $true$ (the target node).
2. SET $ansLabel$ to be the label of $qAnsN$
3. Invoke CUDA kernel call for function: $gpuTwigLastPhase(qStream, pqStream, qIdx)$
4. Insert all nodes from the $ansLabel$ stream whose $isAnswer$ bit is $true$ to $ansSet$

$gpuTwigSecondPhase$ **kernel function (runs on GPU):**
**Input**:
1) $qAnsStream$: index of $qAnsN$ stream.
**Goal**: find the answer nodes in $qAnsStream$.
**Method**:
1. set $idx$ to $blockDim.x * blockIdx.x + threadIdx.x$
2. IF $idx \geq$ number of nodes in $qAnsStream$ then RETURN
3. set $n$ to node at index $idx$ of $qStream$
4. SET $currQN$ to $qAnsN$, and $currSN$ to $n$
5. IF $subtreeCorrect(currQN, currSN) == false$ THEN BREAK
6. WHILE the index of $currQN >$ the index of the root node $rq$ in $qTree$
7. SET $pCurrQN$ to the parent of $currQN$
8. SET $pCurrQL$ to the label of $pCurrQN$
9. IF $currSN$ has no ancestor labeled $pCurrQL$ THEN BREAK
10. SET $upperL$ (respectively, $lowerL$) to the node with the smallest (respectively, largest) *LeftPos* value which is an ancestor of $currSN$ and is in the $pCurrQL$ stream
11. SET $ancN$ to $NULL$
12. FOREACH node $n$ between $lowerL$ and $upperL$ in the $pCurrQL$ stream
13. IF $subtreeCorrect(currQN, currSN) == true$ THEN
14. {SET $ancN$ to $n$
15. BREAK}
16. END FOREACH
17. IF $ancN == NULL$ THEN BREAK
18. SET $currSN$ to $ancN$
19. SET $currQN$ to $pCurrQN$
20. END WHILE
21. IF $currQN$ == $rq$ and $ancN$ is not $NULL$ THEN
22. set $n.isAnswer$ to $true$.
23. END FOREACH

**Figure 6: The second phase of the GPU-Twig Algorithm**

the $qAnsN$ node maps to $n$ (line 5) and the $qArray$ of each node in the match is *subtree-correct* w.r.t. the matching node in $qPath$ (line 13). Line 21 checks if such a match was found, and in case of a positive answer, line 22 sets the $n.isAnswer$ bit to $true$. If the query pattern is allowed to contain the child axis, then if the edge between $pCurrQN$ and $currQN$ is parent-child (and not ancestor-descendant), then instead of lined 9-12, we need to check if $CurrSN$ has a parent node in $pCurrQN$. In case of a positive answer, we need to run lines 13,14 and to continue at line 17, otherwise, to go to line 17 (without running lines 13,14).

*Example 4.* Consider the twig pattern $qTree$ and the $dTree$ presented in Figure 4. After processing the first phase of the algorithm for all the leaves of $qTree$, the $qArray$ of nodes (2:7,2), (8:25,2), (9:16,3), (18:21,4), and (26:37,2) in $dTree$ are changed as shown in Figure 7. During the first phase of the algorithm for node with index 3, bit 3 in $qArray$ of node (8:25,2) is set to $true$, because the $qArray$ of node (18:21,4) is *subtree-correct* w.r.t. node with index 3, and because node (18:21,4) is *left qualifying* w.r.t. the node with index 3 and node (8:25,2). Node (9:16,3), for example, is the node that makes the node (18:21,4) *left qualifying* w.r.t. the node with index 3 and node (8:25,2). By this, the first phase of the algorithm

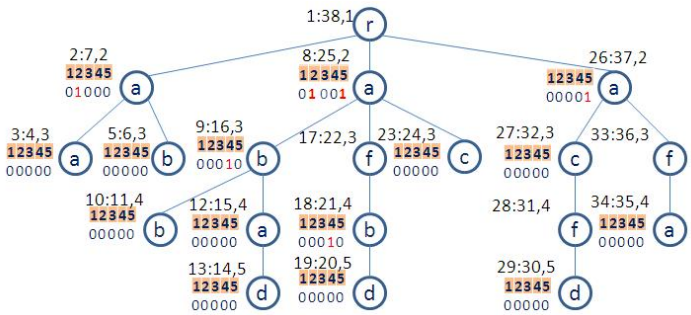


**Figure 7:** **The view of $dTree$ after processing the first phase of the algorithm for all the leaves of $qTree$, for $dTree$ and $qTree$ from Figure 4**

is finished.

The value of the $isAnswer$ field in the node labeled $c$ (with index 5) is $true$. So, according to the definition of answer nodes, the potential answer nodes are placed in the $c$ stream. The $c$ stream contains two nodes: (23:24,3) and (27:32,3).

First we describe the run of the second phase for node (23:24,3). The parent of the node labeled $c$ in $qTree$ is the node labeled $a$. The only ancestor node of (23:24,3) that is labeled $a$ is (8:25,2). Node (8:25,2) has the biggest *LeftPos* value in $a$'s stream such that it is an ancestor of node (23:24,3) and is *subtree-correct* w.r.t. the node with index 1 labeled $a$. We know it because the $qArray$ structure of node (8:25,2) is *subtree-correct* w.r.t. the node labeled $a$ (in $qTree$). The *left qualifying* definition is not relevant for node (8:25,2) because it is the root node. So, we update $currSN$ to be (8:25,2). As the node labeled $a$ is the root of $qTree$, the while loop finishes and $currSN$ is labeled $a$, we add node (23:24,3) to $ansSet$ (set of all answer nodes).

Lastly, we describe the run of the second phase for node (27:32,3). The parent node of the node labeled $c$ in $qTree$ is the node labeled $a$. The only ancestor node of (27:32,3) with label $a$ is (26:37,2). Node (26:37,2) has the highest *LeftPos* value in $a$'s stream but, it is *not qualifying* w.r.t. $currQ$, because the $qArray$ of node (26:37,2) is *not subtree-correct* w.r.t. the node labeled $a$ (in $qTree$). This is because bits 2,3 in the $qArray$ of (26:37,2), that correspond to nodes with index 2,3 in $qTree$, are $false$. As node $a$ is the root of $qTree$, the while loop finishes. As we did not update $currSN$ to be (26:37,2), we do not add node (27:32,3) to $ansSet$. □

## 3.3 Correctness Proof

**Correctness proof of the first phase of the algorithm**

**Lemma 1:** For node $q$ labeled $qLabel$ in $qTree$. If the $qArray$ structure of $dTree$ node $n$ labeled $qLabel$ is *subtree-correct* w.r.t. $q$ then $n$ is *qualifying* w.r.t. $q$, and $n$ is *part of* at least one match of the $qTree$ sub-query rooted at $q$ in $dTree$.

**Proof:**

Let $q$ be a node in $qTree$ labeled $qLabel$, let $n$ be a node labeled $qLabel$ in $dTree$, and let $h$ be the height $h$ of $qTree$.

For $h = 0$ and $q$ a leaf node in $qTree$: By Definition 1, any $dTree$ leaf node $n$ having the same label as $q$ is *qualifying* w.r.t. $q$. So, as node $q$ is a leaf node, it is *qualifying* w.r.t. $q$ no matter how $qArray$ is filled.

For $h > 0$ and $q$ a non-leaf node in $qTree$: If the $qArray$ structure of node $n$ labeled $qLabel$ is *subtree-correct* w.r.t. $q$ then according to the definition of *subtree-correct*, for every child of $q$, say with index $i$, bit $i$ of the $qArray$ is *set correctly* and its value is $true$. According to the *set correctly* definition, the value of bit $i$ in the $qArray$ of node $n$ is $true$ iff node $n$ has a descendant node $m$ labeled with the same label as child $qC$ with index $i$ such that $m$ is *qualifying* w.r.t. $qC$ and *left qualifying* w.r.t. $qC$ and $n$. The *left qualifying* requirement assures that in case we have a few children with the same label, their images in $dTree$ are distinct nodes. If the rightmost child node in the group of children with the same label is *left qualifying* w.r.t. $qC$ and $n$, then all the images of the children with the same label are distinct nodes. So, in order to assure that all the images of the children with the same label are distinct nodes, we only need to look at the rightmost child node. Thus, as according to the definitions, for each child $qC$ with index $qIdx$ of node $q$, the bit $qIdx$ in $qArray$ of node $n$ is $true$, it means that for each child $qC$ of node $q$ in $qTree$ (including the rightmost in the group of children with the same label), node $n$ has a distinct descendant node $m$ in $dTree$ such that node $m$ is qualifying w.r.t. node $qC$. I.e., there is a bijection from a subset of the descendants of $n$ to all children of $q$ such that each descendant in the above subset is *qualifying* w.r.t. the corr. child of $q$. So, according to Definition 2, $n$ is *qualifying* w.r.t. $q$. ■

**Definition 6:** For twig pattern $qTree$ and for node $q$ labeled $qLabel$ in $qTree$, $dTree$ is *tree-correct* w.r.t. $q$, if for each node $n$, labeled $qLabel$, in $dTree$ that is *qualifying* (respectively, *not qualifying*) w.r.t. $q$, the $qArray$ of node $n$ is *subtree-correct* (respectively, *not subtree-correct*) w.r.t. $q$. ■

**Lemma 2:** At the end of the run of the first phase of the algorithm, for all nodes $q$ in $qTree$, $dTree$ is *tree-correct* w.r.t. $q$.

**Proof:** By induction over the structure of $qTree$.

**Base:** For leaf node, $dTree$ is tree-correct w.r.t. $q$, because according to the *tree-correct* definition, for each node $n$, labeled $qLabel$ in $dTree$ that is *qualifying* w.r.t. $q$, the $qArray$ of node $n$ is *subtree-correct* w.r.t. $q$. According to the *qualifying* definition, each leaf node $lfN$ that is labeled $qLabel$ is qualifying w.r.t. $q$. I.e., there are no special requirements for the $qArray$ of $lfN$. So, no matter what is the value of the $qArray$ of $lfN$, because $lfN$ is labeled $qLabel$, it is also vacuously *subtree-correct* w.r.t. $q$.

Now, we prove that if $dTree$ is tree-correct w.r.t. $qC$ for each child $qC$ of some node $q$ in $qTree$, then $dTree$ is *tree-correct* w.r.t. $q$.

**Induction assumption:** $dTree$ is *tree-correct* w.r.t.all children $qC$ of $q$.

We show that the algorithm marks $dTree$ such that $dTree$ is *tree-correct* w.r.t. $q$. Assume that the label of $q$ is $qLabel$. Assume that $pq$ is the parent of $q$ with label $pqLabel$. Note that before starting the algorithm, all $qArray$ structures, of all nodes, were initialized to $false$. Node $n$ in $dTree$ is *part of a match*, if $n$ is the image of some node in $qTree$ in some match of $qTree$ in $dTree$.

**Claim 1:** Assume $qC$, indexed $qIdx$, is a child of $q$. Assume $n$ is a node labeled $qLabel$ in $dTree$. We claim that bit $qIdx$ in the $qArray$ of $n$ is *set correctly* during the processing of $qC$.

**Proof:** While child $qC$ (with index $cqIdx$ and label $cqLabel$) of $q$ was considered, we processed each node $nD$ in the $cqLabel$ stream. By the algorithm, we set bit $cqIdx$ to $true$ in all nodes $n$ in the $qLabel$ stream that are ancestors of $nD$ in case that the following conditions hold.

(1) For each child of $qC$ indexed $i$, the value of bit $i$ in the $qArray$ structure of $nD$ is $true$. According to Definition 5, if for each child of $qC$, say with index $i$, bit $i$ of the $qArrayof\ nD$ is set correctly w.r.t. $qC$ and its value is $true$, then the $qArray$ structure of node $nD$ is *subtree-correct* w.r.t. $qC$. By the assumption, $dTree$ is *tree-correct* w.r.t. $qC$. So, based on Definition 5 and the assumption, if this requirement holds, it means that $qArray$ of $nD$ is *subtree-correct* w.r.t. $qC$.

(2) The $leftSiblingMatch(set_1, set_2)$ function returned $true$

where $set_1$ contains all the distinct left siblings of $qC$ labeled $cqLabel$, and $set_2$ contains all the descendants of $n$ that are labeled $cqLabel$ except for node $nD$. According to Definition 3, $nD$ is *left qualifying* w.r.t. $qC$ if $qC$ has exactly $lSibNum$ distinct left siblings labeled $cqLabel$ and $nD$ has at least $lSibNum$ distinct relatives labeled $cqLabel$ w.r.t. $n$ and there is a bijection between the left siblings of $q$ and a subset of said relatives of $n$ such that each relative of $n$ is qualifying w.r.t. the corr. left sibling of $q$. Function $leftSiblingMatch$ checks exactly these requirements. It checks if there is a bijection from some subset $ss_2$ of $set_2$ to $set_1$, such that for each pair of nodes $x \in ss_2$ and $y \in set_1$ in the bijection the following is $true$: for each child of $y$ indexed $i$, the value of bit $i$ in the $qArray$ structure of $x$ is $true$. By the assumption, $dTree$ *is* tree-correct w.r.t. $qC$ for each child $qC$ of $q$. So, based on Definition 3, the fact that $leftSiblingMatch$ function returned $true$, and the assumption, the existence of such bijection means that $nD$ is *left qualifying* w.r.t. $qC$ and $n$.

So, the fact that we set bit $cqIdx$ to $true$ is correct according to Definition 4., as we found that $nD$ is *subtree-correct* w.r.t. $qC$ and is *left qualifying* w.r.t. $qC$ and $n$, and according to Definition 4. in that case bit $cqIdx$ has to be $true$.

If after finishing processing all nodes in the $cqLabel$ stream, bit $cqIdx$ in a $dTree$ node $n$ labeled $qLabel$ was not set to $true$ and its value at the end of the process remained $false$, it means that no node was found such that it satisfies the above requirements. The fact that no such node was found, means that node $n$ labeled $qLabel$ does not have any descendant that is labeled $cqLabel$ such that it satisfies the above requirements. This is because, if such a node exists, it must be part of the $cqLabel$ stream, but we checked all nodes in $cqLabel$ stream, and did not find such a node. So, if bit $cqIdx$ in node $n$ was not set to $true$ during processing nodes in $cqLabel$ stream, it must be because node $n$ labeled $qLabel$ does not have any descendant that is labeled $cqLabel$ that satisfies the requirements of Definition 4.

So, the above process *sets correctly* the $cqIdx$ bit in $qArray$ of all nodes $n$ with label $qLabel$. Note that when some bit in $qArray$ of any node is set to $true$, its value does not change during the continuation of the algorithm. □

**Claim 2:** Now we claim that before starting to process node $q$ in $qTree$, $dTree$ is *tree-correct* w.r.t. $q$.

**Proof:** By the algorithm, we start processing node $q$ only after finishing to process all the children of $q$. Based on Claim 1, for each child node $qC$ of $q$, the bit that corresponds to $qC$ in the $qArray$ structure of nodes labeled $qLabel$ in $dTree$ is *set correctly* during the processing of node $qC$. So, after finishing processing all children of $q$, all bits that correspond to children of $q$ in the $qArray$ structure of nodes labeled $qLabel$ in $dTree$ are *set correctly*. So, according to Definition 6, $dTree$ is *tree-correct* w.r.t. $q$. □

According to Claim 2 and to the fact that the algorithm continues until it processes all nodes in $qTree$, at the end of the execution of the algorithm, for all nodes $q$ in $qTree$, $dTree$ is *tree-correct* w.r.t. $q$. ■

**Correctness proof of the second phase of the algorithm**

**Lemma 3:** At the end of the run of the algorithm, all the answer nodes of twig pattern $qTree$ in $dTree$ are added to the output.

The Lemma is established using the following three Claims.

**Claim 1:** For the subtree rooted at node $n$ in $dTree$, for node $m$ in this subtree, and for the subtree rooted at $q$ in $qTree$, if the $qArray$ of node $n$ is *subtree-correct* w.r.t. $q$, and $m$ is *part of* a match $mtch$ of the twig pattern defined by the subtree rooted at $q$ in the subtree rooted at $n$, and there exists some ancestor $ancN$ of node $n$ whose $qArray$ is *subtree-correct* w.r.t. the parent node $pq$ labeled $pqLabel$ of $q$, then $mtch$ can be extended to a match $mtchExt$ of the query defined by the subtree rooted at $pq$ in the subtree rooted at $ancN$ such that $m$ is *part of* the match $mtchExt$ matching the same $qTree$ node as in $match$.

**Proof:** According to problem definition, the $qArray$ of $ancN$ is *subtree-correct* w.r.t. $pq$, so according to Lemma 2, $ancN$ is *part of* at least one match of the twig pattern defined by the subtree rooted at $pq$ in the subtree rooted at $ancN$. Assume that the index of node $q$ is $qIdx$. Bit $qIdx$ in the $qArray$ of node $ancN$ is $true$, as according to the problem definition the $qArray$ of node $ancN$ is *subtree-correct* w.r.t. $pq$. According to the claim definition, the $qArray$ of node $n$ is *subtree-correct* w.r.t. $q$, so according to Lemma 2, $n$ is *part of* at least one match of the twig pattern defined by the subtree rooted at $q$ in the subtree rooted at $n$. Based on the above, we can extend the match of $q$ in the subtree rooted at $n$ to the match of $pq$ in the subtree of $dTree$ rooted at $ancN$. The mapping of nodes of $qTree$ subtree rooted at $q$ to nodes of $dTree$ subtree rooted at $n$ remains the same, and we add to this mapping the pair ($pq$,$ancN$), i.e., $pq$ maps to $ancN$. So, as we found a new (extended) match $mtchExt$, $m$ is *part of* a match of the $qTree$ subtree rooted at $pq$ in the subtree of $dTree$ rooted at $ancN$. □

**Claim 2:** For a node $n$ labeled $qLabel$ and for a node $q$ labeled $qLabel$, if the $qArray$ of node $n$ is *subtree-correct* w.r.t. $q$, and $n$ does not have any ancestor whose $qArray$ is *subtree-correct* w.r.t. $pq$, where $pq$ is the parent of $q$, then $n$ does not participate in any match of sub-query $pq$ in $dTree$.

**Proof:** Suppose, for the sake of deriving a contradiction, that the $qArray$ of node $n$ is *subtree-correct* w.r.t. $q$, and $n$ does not have any ancestor whose $qArray$ is *subtree-correct* w.r.t. $pq$ where $pq$ is the parent of $q$, and that $n$ participates in match $mtch$ of query $pq$ in $dTree$. Match $mtch$ looks as follows: $(..., (q : n), (pq : pn))$. According to the definition of a match, each $qTree$ node maps to a distinct node in $dTree$, and the structural relationship between each parent child pair of nodes in $qTree$ is satisfied by the corresponding $dTree$ nodes. In match $mtch$, $q$ maps to $n$ and $pq$ maps to $pn$. According to match definition, there has to be ancestor descendant relationship between nodes $n$ and $pn$, as $q$ and $pq$ have a parent child relationship in $qTree$. According to the assumption, $n$ does not have any ancestor whose $qArray$ is *subtree-correct* w.r.t. $pq$ which contradicts the fact that $pn$ is an ancestor of $n$. Therefore, $n$ does not participate in any match of sub-query $pq$ in $dTree$. □

**Claim 3:** At the end of the run of the algorithm, all the answer nodes of twig pattern $qTree$ in $dTree$ are added to the output.

**Proof:** We consider every node that can potentially be an answer node as we consider all nodes that belong to the $ansLabel$ stream. For each such node, say $leafN$, we run a while loop. In each loop iteration, we take the current node (namely, $currS$) and look for an ancestor (namely, $ancN$) such that the $qArray$ of node $currS$ satisfies the following condition: for each child of $pCurrQ$ (the parent of node $currQ$) indexed $i$, the value of bit $i$ in the $qArray$ structure of $currS$ is $true$. Based on the correctness proof of phase one of the algorithm and on Definition 5, it means that the $qArray$ structure of $currS$ is *subtree-correct* w.r.t. $pCurrQ$. Each time we look for such an ancestor, we choose the ancestor with the highest $lIdx$ value, which means the closest one to $currS$. This ensures that we never miss any potential matching data tree path. So, if such an ancestor does not exist, then based on Claim 2, $currS$ is *not part of* any match, which means that $leafN$ *is* also *not part of* any match. If we found such an ancestor $ancN$, according to Claim 1, node $leafN$ is *part of* at least one match to the target node of the twig pattern defined by the subtree rooted at $pCurrQ$ in the subtree rooted at $ancN$. □

At the end of the algorithm (after ending the while loop) we check if $currQ$ is equal to the root of $qTree$ ($rq$) and that $currS$

is not NULL, which means (according to Claim 1) that $leafN$ is *part of* at least one match of of the twig pattern defined by the subtree rooted at $rq$ in the subtree rooted at $currS$. I.e., $leafN$ is an answer node in $dTree$. ■

# 4. RELATED WORK

Recently, there have been research efforts aiming to parallelize relational DBMS query processors [28, 39, 38, 9]. This is mainly due to the emergence of multi-core processors. One example is InfiniteDB, which is a PC-Cluster-based Parallel Massive Database Management System [25]. Another example is an extensive study of SQL query parallelization in the context of both distributed and centralized repositories [23, 24, 29].

Parallelization of SQL queries differs from that of XPath parallelization. (1) XPath processing is read-only, while SQL allows inplace updates. (2) SQL is associated with a relational data model which leads to natural partitioning along either rows or columns. XPath is associated with a tree data model that is less naturally partitioned. Hence, XML query paralelization is challenging.

As XPath is a critical component in many XML-based applications, it is important to maximize its performance. There has been extensive work on optimizing performance of a single XPath query by improving its traversal pattern [16, 27, 30]. Studies on parallelization of XML processing began to appear only recently; for example, studies regarding parallel XML parsing [31, 37, 35, 36], parallel data placement in XML databases [41], and parallel processors for XML databases [13, 42, 43]. Most existing XML processing engines are thread-safe and allow multiple threads to issue concurrent XPath queries against an XML database. Distributed XML processing is discussed in [8, 12]. Almost no published works deal with parallelization of a single query.

Parallelization of a single XPath query using the Xalan XPath engine on shared address space multi-core systems is presented in [6]. It evaluates opportunities for parallelizing a single XPath query, in a shared-address space environment, on commodity multi-core processors. It proposes three strategies for parallelizing individual XPath queries. *Data Partitioning approach:* executes the same (sub)query on different sections of the same XML document. *Query Partitioning approach:* executes different (sub)queries on the same XML dataset. The third is the *Hybrid Partitioning approach* that combines both the data and query partitioning schemes. Continuation work by the same authors is presented in [5]. It proposes a parallelization framework that determines the optimal way of parallelizing an XML query. They demonstrated that it is possible to accelerate XPath processing using commodity multi-core systems.

As GPUs emerged, they were mainly used to accelerate scientific, geometric, and imaging applications. State-of-the-art General Purpose GPU (GPGPU) techniques can be found in [34]. Lately, there are efforts to use GPUs to improve the performance of database operations. For example, in [40] the rendering and search capabilities of GPUs are used for spatial selection and join operations. In [3] GPU-based spatial operations are implemented as external procedures in a commercial DBMS. Govindaraju et al. [20, 19, 17] are novel GPU based algorithms for relational operators. Most recently, in [26], a similarity join was implemented on CUDA.

# 5. CONCLUSIONS AND FUTURE WORK

The GPU-Twig algorithm is a novel efficient algorithm for matching XML twig patterns, using a stream representation scheme, in a parallel multi-threaded computing platform, while using a GPU as a CPU co-processor. GPU-Twig employs techniques that allow it to run hundreds of threads in parallel.

The results of an extensive experimentation are presented in a separate publication.

As part of future work we intend to extend our algorithms to work more efficiently with larger terabyte sized files. We also intend to examine certain features, and their associated trade-offs, of the algorithm and determine whether we can further optimize it. An example of such a tradeoff is the checking of the left qualifying property for each child rather than just for the rightmost child for each particular label. We also plan to check potential parallelism in other aspects of XML and semi-structured databases in general (for example, parallelization of DataGuides [14], Path Summaries [2] and more).

# 6. REFERENCES


[1] A. Ailamaki, N. K. Govindaraju, S. Harizopoulos, and D. Manocha. Query co-processing on commodity processors. In *VLDB*, page 1267, 2006.

[2] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern xml databases. *World Wide Web*, 11(1):117–151, 2008.

[3] N. Bandi, C. Sun, A. E. Abbadi, and D. Agrawal. Hardware acceleration in commercial databases: A case study of spatial operations. In *VLDB*, pages 1021–1032, 2004.

[4] D. Blythe. The direct3d 10 system. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 724–734, New York, NY, USA, 2006. ACM.

[5] R. Bordawekar, L. Lim, A. Kementsietsidis, and B. W.-L. Kok. Statistics-based parallelization of xpath queries in shared memory systems. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 159–170, New York, NY, USA, 2010. ACM.

[6] R. Bordawekar, L. Lim, and O. Shmueli. Parallelization of xpath queries using multi-core processors: challenges and experiences. In *EDBT*, pages 180–191, 2009.

[7] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

[8] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis. Using partial evaluation in distributed query evaluation. In *VLDB*, pages 211–222, 2006.

[9] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[10] S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. ACM, 2006.

[11] J. Clark and S. DeRose. Xml path language (xpath). www.w3.org/TR/xpath.

[12] G. Cong, W. Fan, and A. Kementsietsidis. Distributed query evaluation with performance guarantees. In *SIGMOD Conference*, pages 509–520, 2007.

[13] Q. G. Parallel processing of xml databases. In *CCECE*, 2005.

[14] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[15] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing xpath queries. In *VLDB*, pages 95–106, 2002.

[16] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing xpath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[17] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD Conference*, pages 325–336, 2006.

[18] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. Memory—a memory model for scientific algorithms on graphics processors. *Proceedings of the 2006 ACMIEEE conference on Supercomputing SC 06*, page 89, 2006.

[19] N. K. Govindaraju, B. Lloyd, W. W. 0010, M. C. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD Conference*, pages 215–226, 2004.

[20] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD Conference*, pages 611–622, 2005.

[21] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4), 2009.

[22] J. Hensley. Amd ctm overview. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[23] M. F. Khan, R. A. Paul, I. Ahmad, and A. Ghafoor. Intensive data management in parallel systems: A survey. *Distributed and Parallel Databases*, 7(4):383–414, 1999.

[24] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[25] J. Li, H. Gao, J. Luo, S. Shi, and W. Zhang. Infinitedb: a pc-cluster based parallel massive database management system. In *SIGMOD Conference*, pages 899–909, 2007.

[26] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *ICDE*, pages 1111–1120, 2008.

[27] Z. H. Liu, M. Krishnaprasad, H. J. Chang, and V. Arora. Xmltable index an efficient way of indexing and querying xml property data. In *ICDE*, pages 1194–1203, 2007.

[28] D. B. Lomet and B. Salzberg. Access method concurrency with recovery. In *SIGMOD Conference*, pages 351–360, 1992.

[29] H. Lu. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[30] J. Lu, T. Chen, and T. W. Ling. Efficient processing of xml twig patterns with parent child edges: a look-ahead approach. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, CIKM '04, pages 533–542, New York, NY, USA, 2004. ACM.

[31] W. Lu, K. Chiu, and Y. Pan. A parallel approach to xml parsing. In *GRID*, pages 223–230, 2006.

[32] NVIDIA. Nvidia cuda c programming guide.

[33] OpenGL.org. Opengl: The industry's foundation for high performance graphics. http://www.opengl.org.

[34] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, Mar. 2007.

[35] Y. Pan, Y. Zhang, and K. Chiu. Hybrid parallelism for xml sax parsing. In *ICWS*, pages 505–512, 2008.

[36] Y. Pan, Y. Zhang, and K. Chiu. Parsing xml using parallel traversal of streaming trees. In *HiPC*, pages 142–156, 2008.

[37] Y. Pan, Y. Zhang, K. Chiu, and W. Lu. Parallel xml parsing using meta-dfas. In *eScience*, pages 237–244, 2007.

[38] N. W. Paton, J. B. Chávez, M. Chen, V. Raman, G. Swart, I. Narang, D. M. Yellin, and A. A. A. Fernandes. Autonomic query parallelization using non-dedicated computers: an evaluation of adaptivity options. *VLDB J.*, 18(1):119–140, 2009.

[39] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1):610–621, 2008.

[40] C. Sun, D. Agrawal, and A. E. Abbadi. Hardware acceleration for spatial selections and joins. In *SIGMOD Conference*, pages 455–466, 2003.

[41] N. Tang, G. Wang, J. X. Yu, K.-F. Wong, and G. Yu. Win: An effcient data placement strategy for parallel xml databases. In *ICPADS (1)*, pages 349–355, 2005.

[42] L. W. and G. D. Parallel xml processing by work stealing. In *SOCP*, 2007.

[43] L. W. and G. D. A Parallel XML Processing Model on the Multicore CPUs. Technical report, School of Informatics, Indiana University, 2008.

[44] W3C. XML: Extensible Markup Language . http://www.w3.org/XML/.

[45] W3C. Xquery: Xml query. http://www.w3.org/XML/Query/.

[46] W3C. Xsl: The extensible stylesheet language family. http://www.w3.org/XML/XSL/.

[47] G. Weikum, A. C. König, and S. Deßloch, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM, 2004.