High-Performance Holistic XML Twig Filtering Using GPUs

Ildar Absalyamov
UC Riverside
Riverside, CA, 92507
iabsa001@cs.ucr.edu

Roger Moussalli
IBM T.J. Watson Research Center
Yorktown Heights, NY, 10598
rmoussal@us.ibm.com

Walid Najjar
UC Riverside
Riverside, CA, 92507
najjar@cs.ucr.edu

Vassilis J. Tsotras
UC Riverside
Riverside, CA, 92507
tsotras@cs.ucr.edu

ABSTRACT

Current state of the art in information dissemination comprises of publishers broadcasting XML-coded documents, in turn selectively forwarded to interested subscribers. The deployment of XML at the heart of this setup greatly increases the expressive power of the profiles listed by subscribers, using the XPath language. On the other hand, with great expressive power comes great performance responsibility: it is becoming harder for the matching infrastructure to keep up with the high volumes of data and users. Traditionally, general purpose computing platforms have generally been favored over customized computational setups, due to the simplified usability and significant reduction of development time. The sequential nature of these general purpose computers however limits their performance scalability. In this work, we propose the implementation of the filtering infrastructure using the massively parallel Graphical Processing Units (GPUs). We consider the holistic (no post-processing) evaluation of thousands of complex twig-style XPath queries in a streaming (single-pass) fashion, resulting in a speedup over CPUs up to 9x in the single-document case and up to 4x for large batches of documents. A thorough set of experiments is provided, detailing the varying effects of several factors on the CPU and GPU filtering platforms.

1. INTRODUCTION

Publish-subscribe systems (or simply pub-subs) are timely asynchronous event-based dissemination systems consisting of three main components: *publishers*, who feed a stream of documents (messages) into the system, *subscribers*, who register their profiles (queries, i.e., subscription interests), and a *filtering infrastructure* for matching subscriber interests with published messages and delivering the matched messages to the interested subscriber(s).

Early pub-sub implementations restricted subscriptions to pre-defined topics or channels, such as weather, world news, finance, among others. Subscribers would hence be "spammed" with more (irrelevant) information than of interest. The second generation of pub-subs evolved by allowing predicate expressions; here, user profiles are described as conjunctions of (attribute, value) pairs. In order to add further descriptive capability to the user profiles, third-generation pub-subs adopted the *eXtensible Markup Language* (XML) as the standard format for data exchange, due to its self-describing and extensible nature. Exchanged documents are now encoded with XML, while profiles are expressed with XML query languages, such as XPath [2]. Such systems take advantage of the powerful querying that XML query languages offer: profiles can now describe requests not only on the document values but also on the structure of the messages [1] allwoing to match complex twig-like messages.

Currently, XML-based Pub-Sub systems have been adopted for the dissemination of *Micronews* feeds. These feeds are typically short fragments of frequently updated information, such as news stories and blog updates. The most prominent XML-based format used is RSS. In this environment, the RSS feeds are accessed via HTTP through URLs and supported by client applications and browser plug-ins (also called feed readers). Feed readers (like Bloglines and NewsGator), periodically check the contents of micronews feeds and display the returned results to the user.

The complex process of matching thousands of profiles against massive amounts of published messages is performed in the filtering infrastructure. From a user/functionality perspective, filtering consists of determining, for each published message, which subscriptions match at least once. The novelty lies in exploring new efficient filtering algorithms, as well as high-performance platforms on which to implement algorithms and further accelerate their execution. Several fine-tuned CPU-based software filtering algorithms have been studied [3, 12, 14, 19]. These memory-bound approaches, however, suffer from the Von Neumann bottleneck and are unable to handle large volume of input streams. Recently, various works have exploited the obvious embarrassingly parallel property of filtering, by evaluating all queries in parallel using Field Programmable Gate Arrays (FPGAs) [22, 24, 25]. By introducing novel parallel-

[1] In this paper, we use the terms "profile", "subscription" and "query" interchangeably; similarly for the terms "document" and "message".

Path	::=	Step | Path Step
Step	::=	Axis TagName | Step "[" Step "]"
Axis	::=	"/" | "//"
TagName	::=	Name | "*"

Figure 1: Production rules to generate an XPath expression; Name corresponds to an arbitrary alphanumeric string

hardware-tailored filtering algorithms, FPGAs have been shown to be particularly well suited for the stream processing of large amounts of data, where the temporary computational state is not offloaded to off-chip (low-latency) memory. These algorithms allowed the massively parallel streaming matching of complex path profiles that supports the */child:: axis* and */descendant-or-self:: axis* [2], wildcard ('*') node tests and accounts for recursive elements in the XML document. Using this streaming approach, inter-query and intra-query parallelism were exploited, resulting in up to two orders of magnitude speed-up over the leading software approaches.

In [23] we presented a preliminary study on the mapping of the path matching approach on GPUs, providing the flexibility of software alongside the massive parallelism of hardware. In this paper we detail the support of the more complex twig queries on GPUs in a massively parallel manner. In particular, the novel contributions of this paper are:

- The first design and implementation of XML twig filtering on GPUs, more so in a holistic fashion.
- An extensive performance evaluation of the above approach is provided, with comparison to leading software implementations.
- A study on the effect of several query and document factors on GPUs and the software implementations through experimentation is depicted.

The rest of the paper is organized as follows: in Section 2 we present related work in software and hardware XML processing. Section 3 provides in depth description of the holistic XML twig filtering algorithm. Section 4 details the implementation of the XML filtering on GPUs. Section 5 presents an experimental evaluation of the parallel GPU-based approach compared to the state of the art software counterparts. Finally conclusions and open problems for further research appear in Section 6.

2. RELATED WORK

The rapid development of XML technology as a common format for data interchange together with the emergence of information dissemination through event notification systems, has led to increased interest in content-based filtering of XML data streams. Unlike the traditional XML querying engines, filtering systems experience essentially different type of workload. In a querying system, the assumption is that XML documents are fixed (known in advance), which allows building efficient indexing mechanisms on them to facilitate the query process; whereas queries are adhoc and can have arbitrary structure. On contrary, XML filtering engines are fed a series of volatile documents, incoming in a streaming fashion while queries are static (and known in advance). This type of workload prevents filtering systems from using document indexing, thus requiring a different paradigm to solve this problem. Moreover, since filtering systems return only binary result (match or no match, whether a particular query was matched or not in a given document), they should be able to process large number of queries. This is in contrast to XML querying engines, which need to report every document node that was matched by an incoming query.

Software based XML filtering algorithms can be classified into several categories: (1) FSM-based, (2) sequence-based and (3) others. XFilter [4] is the earliest work studying FSM-based filtering, which builds a single FSM for each XPath query. Each query node is represented by an individual state at the FSM. Transitions in the automaton are fired when an appropriate XML event is processed. An XPath query (profile) is considered matched if an accepting state is reached at the FSM. YFilter [12] leverages path commonalities between individual FSMs, creating a single Non-Deterministic Finite Automaton (NFA) representation of all XPath queries and therefore reducing the number of states. Subsequent works [16, 29, 14] use a unified finite automaton for XPath expressions, using lazy DFAs and pushdown automata.

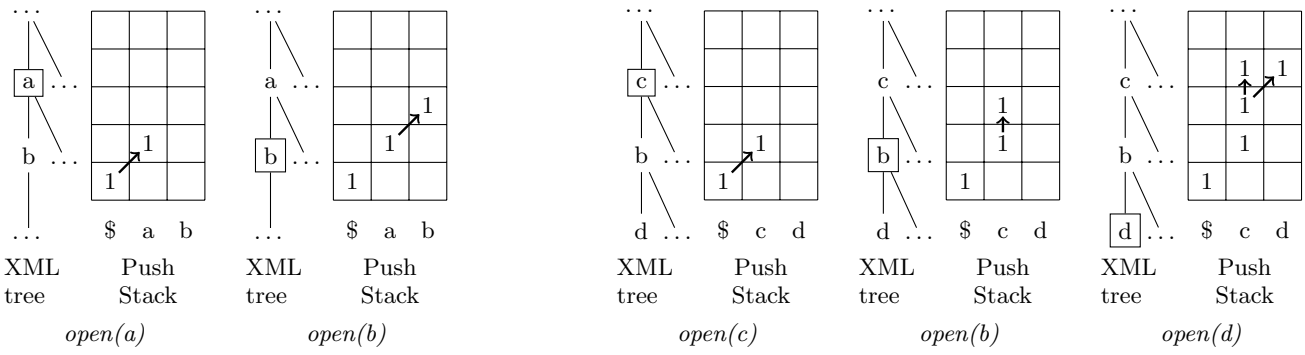
FiST [19] and it's successors are the examples of the sequence-based approach for XML Filtering, which implies a two-step process: (i) the streaming XML document and the XPath queries are converted into sequences and (ii) a subsequence match algorithm is used to determine if a query had a match in the XML document.

Among the other works, [10] builds a Trie index to exploit query commonalities. Another form of similarity is explored in the AFilter [9] which uses prefix as well as suffix sharing to create an appropriate index data structure. [13] introduces stream querying and filtering algorithms LQ and EQ, using a lazy and eager strategy, respectively.

Recent advances in the GPGPU technology opened a possibility to speedup many traditional applications, leveraging the high degree of parallelism offered by GPUs. A large number of research works explore the usage of GPUs for improving traditional database relational operations. [5] discusses improving the SELECT operation performance, while [17] focuses on implementing efficient join algorithms. Recently [11] addressed all commonly used relational operators. There are also research works concentrated on improving indexing operations. [18] introduces a hierarchical CPU-GPU architecture for efficient tree querying and bulk updating and [7] proposes an abstraction for tree index operations.

There has also been much interest in using GPUs for XML related problems. [8] introduced the Data partitioning, Query partitioning and Hybrid approaches to parallelize XPath queries on multi-core systems. [31] used these strategies to create a cost model, which decides how to process XPath queries on GPUs in parallel. [15] processed XPath queries in a way similar to Yfilter, by creating end executing a NFA on a GPU, whereas [30] studied twig query processing on a large XMLs with the help of GPUs. All these works focus on the problem of indexing and querying XML documents using XPath queries. However none of them address filtering XML documents, which an orthogonal problem to query processing, thus requiring a different implementation

[2] In the rest of the paper we shall use '/' and '//' as shorthand to denote the */child:: axis* and */descendant-or-self:: axis*, respectively.



(a) Matching path {*/a/b*}

(b) Matching path {*/c//d*}

Figure 2: Step-by-step overview of stack updates for (a) parent-child and (b) ancestor-descendant relations. Each step corresponds to *open*(tag) event, with respective opened tag highlighted. As the XML document tree is traveled downwards, content is pushed onto the top of the stacks. The leftmost stack column is set to be always in a matched state, corresponding to a dummy root node (denoted by $).

approach.

As mentioned, our previous work on the XML filtering problem concentrated on using FPGAs. In [24] we presented a new dynamic-programming based XML filtering algorithm, mapped to FPGA platforms.

To support twig filtering, a post-processing step is required to identify which path matches correspond to twig matches. In [25] we extended this approach to support the holistic (no post-processing) matching of the considerably more complex twig-style queries. While providing significant performance benefits, these FPGA-based approaches were not without their disadvantages. Matching engines running on the FPGA relied on custom implementations of the queries in hardware and did not allow the update, addition, and deletion of user profiles on-the-fly. Although the query compilation to VHDL is fast, re-synthesis of the FPGA, which includes translation, mapping and place-and-route are expensive (up to several hours) and must be done off-line. Furthermore, solutions on FPGAs cannot scale beyond available resources, where the number of queries is limited to the amount of real-estate on the chip.

GPU architectures do not have these limitations thus offer a promising approach to the XML filtering problem. In this paper we extend our previous work [23] that considered only filtering linear XPath queries on GPUs and provide a holistic filtering of complex twig queries.

3. PARALLEL HOLISTIC TWIG MATCHING ALGORITHM

We proceed with the overview of the stack-based holistic twig filtering algorithm and respective modifications as required for the mapping onto GPUs.

3.1 Framework Overview

The structure of an XML-encapsulated document can be represented as a tree, where nodes are XML tags. Opening and closing tags in the XML document translate to the traveling (down and up) through the tree. SAX parsers process XML documents in a streaming fashion, while generating *open(tag)* and *close(tag)* events. XML queries expressed in XPath relate to the structure of XML documents, hence, rely heavily on these open/close events. As will be clearer below, stacks are an essential structure of XML query processing engines, used to save the state as the structure of the tree is visited (push on *open*, pop on *close*).

Figure 1 shows the XPath grammar used to form twig queries, consisting of nodes, connected with parent-child ("/"), ancestor-descendant ("//") or nested path relationships ("[]"). We denote by L the length of the twig query, representing the total number of nodes in the twig, in addition to a dummy start node at the beginning of each query (the latter being essential for query processing on GPUs).

Such a framework allows us to filter the whole streaming document in a single pass, while capturing query match information in the stack.

In [25] we show that matching a twig query is a process consisting of 2 interleaved mechanisms:

- Matching individual root-to-leaf paths. Here, each such path is evaluated (matched) using a stack whose contents are updated on push (open(tag)) events.
- Carefully joining the matched results at query split nodes. This task is performed using stacks that are mainly updated on pop (close(tag)) events. The main reasoning lies in that a matched path reports its match state to its ancestors using the pop stack.

In both mechanisms, the depth of the stack is equal to the maximum depth of the streaming XML document, while the width of the stack is equal to the length of the query. The top-of-stack pointer is referred to as TOS in the remainder of this paper.

Detailed descriptions of the operations and properties of these stacks are presented next.

3.2 Push Stack Algorithm

As noted earlier, an essential step in matching a twig query lies in matching all root-to-leaf paths of this query. We achieve this task using *push stacks*, namely stacks holding query matching state, updated solely on push events as the XML document is parsed.

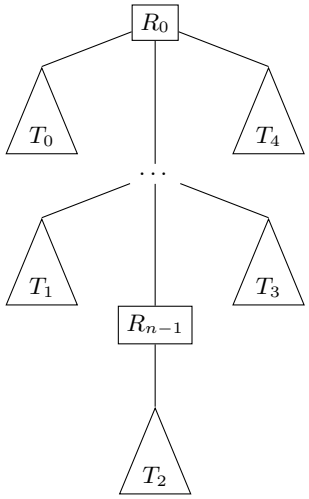


Figure 3: Abstract view of the structure of an XML document tree, with root R_0, and several subtrees such as T_0, T_1, etc.

We describe the matching process as a dynamic programming algorithm, where the push stack represents the dynamic programming table. Each column in the push stack represents a query node from the XPath expression; when mapping a twig to the GPU, each stack column needs a pointer to its parent column (*prefix* column). A split node acts as a prefix for all its children nodes.

The intuition is that the root-to-leaf path with length L can be in a matched state only if its prefix of length *L-1* is matched. This optimal substructure property can be trivially proved by an induction over the matched query length (assuming a matched dummy root as base case).

The L^{th} column is matched if '1' is stored on the top of the stack for this particular column. When a column, corresponding to a leaf node in the twig query, stores '1' on the top-of-stack, then the entire root-to-leaf path is matched.

The following list summarizes the properties of the push stack:

- Only the entries in the top-of-stack are updated on each event.
- The entries in the push stack can be modified only in response to a push event (open(tag)).
- On a pop event, the pointer to the stop-of-stack is updated (decreased). The popped state (data entries) is lost.
- A dummy root node is in a matched state at the beginning of the algorithm (columns corresponding to root nodes are assumed to be matched even before any operation on the stack occurs).
- If a relationship between a node and its prefix is parent-child, then a '1' can diagonally propagate from the parent's column to the child column, only on a push event if the opened tag matches that of the child. Figure 2a depicts the diagonal propagation for a parent-child relationship.

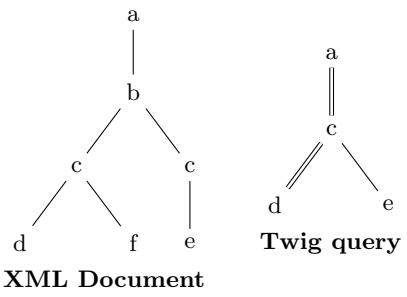


Figure 4: Sample XML document tree representation and twig query. The twig query can be broken down into two root-to-leaf paths, namely /a//c//d and /a//c/e. While each of these paths is individually matched (found) in the tree, the twig /a//c[//d]/e is not.

- Columns corresponding to wildcard nodes allow the propagation from a prefix column without checking whether the column tag is the same as the tag respective to the push event.
- If the relationship between node and its prefix is ancestor-descendant, then the diagonal propagation property applies, as in parent-child relationships. In addition, in order to satisfy ancestor-descendant semantics (the descendant lies potentially more than one node apart from the parent), a match is carried in the prefix column vertically upwards, even if the tag, which triggered the push event does not correspond to the descendant's column tag. Using this technique, all descendants of the prefix node can see the matched state of the prefix. This matched info is popped with the prefix.

  Figure 2b depicts the matching of the path */c//d*, where d is a descendant but not a child of c. Note that upward match propagation for prefix node c will continue even after d is matched. This process will stop only after c will be popped out of the stack.
- A match in an ancestor column will only be seen by descendants. This is visualized through Figure 3, depicting an abstract XML tree, with root R_0, and several subtrees such as T_0, T_1, etc. Assuming node R_{n-1} is a matched ancestor, this match will be reported in child subtree T_2 using push events. Subtrees T_0 and T_1 would not see the match, as they would be popped by the time R_{n-1} is visited. Similarly, subtrees T_3 and T_4 will not see the match, as R_{n-1} would be popped prior to entering them.

Considering twigs: in the case of a split query node having both ancestor-descendant and parent-child below it, then two stack columns are allocated for the split node; one would allow vertical upwards propagation of 1's (for ancestor-descendant relationships), and another would not (for parent-child relationships). These two separate columns will also later be used to report matches back to root in the pop stack (Section 3.3 describes this situation in details).

Recurrence equation, applied at each (binary) stack cell $C_{i,j}$ on push events is shown on Figure 5.

$$C_{i,j} = \begin{cases} 1 \quad if \begin{cases} C_{i-1,j-1} = 1 \quad AND \begin{cases} \text{relationship between } j^{th} \text{ column and its prefix is ``/''} \\ AND \\ \begin{cases} j^{th} \text{ column tag was opened in push event} \\ \text{OR} \\ j^{th} \text{ column tag is wildcard symbol ``*''} \end{cases} \end{cases} \\ OR \\ C_{i-1,j} = 1 \text{ AND relationship between } j^{th} \text{ column and its prefix is ``//''} \end{cases} \\ 0 \quad otherwise \end{cases}$$

Figure 5: Recurrence relation for push stack cell $C_{i,j}$. $1 \leq i \leq d, 1 \leq j \leq l$, where d - maximum depth of XML document, l - length of the query.

3.3 Pop Stack Algorithm

Matching of all individual root-to-leaf paths in a twig query is a necessary condition for a twig match, but it is not sufficient. Figure 4 shows the situation when both paths */a//c//d* and */a//c/e* of the query */a//c[//d]/e* report a match, but holistically the twig fails to match. In order to correctly match the twig query we need to check that the two root-to-leaf paths, branching at split node *j* report their match to the same node *j*, by the same time it will be popped out of stack.

In order to report a root-to-leaf match to the nearest split node in [25] we introduced a *pop stack*. As for the push stack, the reporting problem could be solved using a dynamic programming approach, where the pop stack serves as a dynamic programming table. Columns of the pop stack similarly represent query nodes of the original XPath expression.

Column j would be considered as matched in pop stack if '1' is stored on TOS for a particular column after a pop event. The optimal substructure property could again be easily proved, but this time the induction step for a split node should consider optimality of all its children paths, rather than just one leaf-to-split-node path, in order to report a match. When a column, corresponding to the dummy root node, stores '1' on the top of the pop stack, the entire twig query is matched.

The subsequent list summarizes the pop stack properties:

- Unlike the push stack, the pop stack is updated both on push and pop events: push always forces rewriting the value on the top of the pop stack to its default value, which is '0'. A pop event on the other hand reports a match, propagating '1' downwards, but will not force '0' to be carried in the same direction.

- If the relationship between a node and its prefix is parent-child, then '1' is propagated diagonally downwards from the node's column to its parent. As with the push stack, propagation occurs only if the pop event that fired the TOS decrement, closes the tag, corresponding to the column's tag in the query (unless the column tag is the wildcard).

- The propagation rule for the case when the relationship between a node and its prefix is ancestor-descendant, is similar to the one described for the push stack. This time however a match is propagated in the descendant node and downwards, rather than in its prefix and upwards.

  Recalling Figure 3, only nodes $R_0, \ldots, R_{n-1}$ would be reported about a matched ancestor. In this case subtree T_2 is not matched, since we report a match only during the pop event of the last matched path node R_{n-1}, which in turn occurs after T_2 has been processed. Subtrees T_0 and T_1 again do not observe the match, because it was not yet reported at the moment they are encountered. Although subtrees T_3, T_4 are processed after the pop event, they still cannot see the match, since the top of the stack has grown by processing additional push events.

- Since the purpose of the pop stack is to report a matched path back to the root, reporting starts at the twig leaf nodes. Stack columns corresponding to leaf nodes in the original query are matched only if they were matched in the push stack.

- A split node reports a match only if all of its children have been matched. If the latter is true, relationship propagation rules are applied so as to carry the split match further.

Figure 6 shows a recurrence relation, which is applied at each stack cell $D_{i,j}$ on a pop event.

4. FROM ALGORITHM TO EFFICIENT GPU IMPLEMENTATION

The typical workload for an XML based publish-subscribe system consists of a large number of user profiles (twig queries), filtered through a continuous document data stream. Parallel architectures like GPUs offer an opportunity for much performance improvement, by exploiting the inherent parallelism of the filtering problem. Using GPUs, the implementation of our stack-based filtering approach leverages several levels of parallelism, namely:

- **Inter-query parallelism:** All queries (stacks) are processed independently in a parallel manner, as they are mapped on different SMs (streaming multiprocessors).

- **Intra-query parallelism:** All query nodes (stack columns) are updating their top-of-stack contents in parallel. The main GPU kernel consists of the query node evaluation, and each is allocated a SP (streaming processor).

$$D_{i,j} = \begin{cases} 1 \quad if \begin{cases} C_{i+1,j} = 1, \text{ if a node corresponding to } j^{th} \text{ column is a leaf in twig query} \\ \forall c \in \{children\ of\ j^{th}\ column\} D_{i+1,c} = 1, \text{ if a node corresponding to} j^{th} \text{ column is a split node} \\ D_{i+1,j+1} = 1 \quad AND \begin{cases} j+1^{th} \text{ column tag was closed in pop event} \\ \text{OR} \\ j+1^{th} \text{ column tag is wildcard symbol “*”} \end{cases} \\ OR \\ D_{i+1,j} = 1 \text{ AND relationship between } j^{th} \text{ column and its prefix is “//”} \end{cases} \\ 0 \quad otherwise \end{cases}$$

Figure 6: Recurrence relation for pop stack cell $D_{i,j}$. $1 \leq j \leq l$, $1 \leq i \leq d$, where d - maximum depth of XML document, l - length of the query, $C_{i,j}$ - a corresponding cell of push stack.

FIELD	SIZE
Event type (pop/push)	1 bit
Tag ID	7 bits

Table 1: Encoding of XML events at parsing (pre-GPU). The size of each encoded event is 1 byte.

FIELD	SIZE
IsLeaf	1 bit
Prefix relationship	1 bit
Query children with “/” relationship	1 bit
Query children with “//” relationship	1 bit
Prefix ID	10 bits
...	11 bits
Tag name	7 bits

Table 2: GPU Kernel personality storage format. This information is encoded using 4 bytes. Some bits are used for padding, and do not encode any information (depicted as “...”).

- **Inter-document parallelism:** This type of parallelism, provided by the Fermi NVidia GPU architecture [27], is used to process several XML documents in parallel, thus increasing the filtering throughput.

In the following subsections, we provide a detailed description of the implementation of our stack approach on GPUs, alongside with hardware-specific optimizations we deployed.

4.1 XML Stream Encoding

XML parsing is orthogonal to filtering, and has been thoroughly studied in the literature [20, 28, 21]. In this work, parsed XML documents are passed to the GPU as encoded open/close events, ready for filtering. In order to minimize the information transfer between CPU and GPU (which is heavily limited by the PCIe interface bandwidth), XML documents are compressed onto an efficient binary representation, that is easily processed on the GPU side.

Each open/close tag event is encoded as a 1 byte entry, whose most significant bit is reserved to encode the type of event (push/open internally viewed as pop/close) while the remaining 7 bits represent the tag ID (Table 1). Using this encoding, the whole XML document is represented as an array of such entries, which is then transferred to the GPU.

4.2 GPU Streaming Kernel

Every GPU kernel, executed in a thread on a GPU Streaming Processor (SP), logically represents a single query node (evaluated using one stack column). Each of the thread updates the top-of-stack value of its column according to the received XML stream event.

On the GPU side, multiple threads are grouped within a *thread block*. Each thread block is individually scheduled by the GPU to run on a Streaming Multiprocessor (SM). The latter have a small low-latency memory, which we use to store the stack and other relevant state. The deployed kernels within a block perform filtering of whole twig queries by updating the top-of-stack information in a parallel fashion.

Algorithm 1 represents a simplified version of the GPU kernel: the process of updating a value on the top-of-stack.

Algorithm 1 GPU Kernel

```
1: level ← 0
2: for all XML events in document stream do
3:     if push event then
4:         level++
5:         prefixMatch ← pushStack[level − 1][prefixID]
6:         if prefixMatch propagates diagonally then
7:             pushStack[level][colID] → childMatch ← 1
8:         end if
9:         if prefixMatch propagates upwards then
10:            pushStack[level][colID] → descMatch ← 1
11:        end if
12:    else
13:        level − −
14:        prevMatch ← popStack[level + 1][colID]
15:        if prevMatch propagates upwards then
16:            pop_stack[level][colID] → descMatch ← 1
17:        end if
18:        if prevMatch propagates diagonally then
19:            if node is leaf && pushStack[level+1][colID]
   then
20:                popStack[level][colID] → childMatch ←
   1
21:            end if
22:        end if
23:        popStack[level][prefix] → childMatch ←
   popStack[level][prefix] && popStack[level][colID]
24:    end if
25: end for
```

Note that the *ColID* variable refers to the column ID index, unique within a single thread block (*threadId* in CUDA primitives). Similarly, *prefix* serves as a pointer to the colID

of the prefix node, within the thread block (twigs are evaluated within a single block).

The match state on lines 6 and 18 propagates diagonally upwards and downwards respectively, if:

- The relationship between node and its prefix is "/".
- The *childMatch* value of the entry from the respective level in the push/pop stack is matched (lines 5 or 14 respectively).
- The (fixed) column tag corresponds to the open/closed XML tag, or the column tag is a wildcard.

The match on lines 9 and 15 propagates strictly upwards and downwards respectively, if:

- The relationship between node and its prefix is "//".
- The *descMatch* value of the entry from the respective level in the push/pop stack is matched (lines 5 or 14 respectively).

To address the case of a split node with children having different relationship types, the push stack needs to keep two separate fields: *childMatch* to carry the match **for** the children with parent-child relationship and *descMatch* **for** the ancestor-descendant case.

The same occurs with the pop stack: *descMatch* will propagate match **from** the node's descendants, and *childMatch* will carry the match **from** all its nested paths. The latter is especially meaningful for split nodes, as they are matched only when *all* their split-node-to-leaf paths are matched. This reporting is done as the last step on processing of pop event on line 23.

4.3 Kernel Personality Encoding

As every GPU kernel executes the same program code, it requires a parameterization in the form of a single argument which we call *personality*. A personality is created once offline, by the query parser on CPU, which encodes all information about the query node.

A personality is encoded using a 4 byte entry, whose attribute map is shown in Table 2.

The most significant bit indicates whether the query node is a leaf node in the twig. This information is used in pop event processing to start matching leaf-to-split-node paths on line 19 of Algorithm 1.

The following bit encodes the type of relationship that the query node has with its prefix, which is needed to determine match is propagated on lines 6,9,15 and 18.

Consider the case where a split node has two children with different types of relationship connecting them to their prefix. Instead of duplicating the following query nodes, we use the 3^{rd} and 4^{th} most significant bits of personality entry to encode whether a node has children with parent-child, ancestor-descendant relationship, or both respectively. Note that for other types of query nodes (including split nodes, which have several children, but all of them are connected with one type of relationship) only one of those bits will be set. This information would be later used to propagate push value into either *childMatch* or *descMatch* fields for ordinary node, or both of them is addressed special case on lines 7 and 10.

A query node also needs to encode its prefix pointer, which is the prefix ID in a contiguous thread block. Since in the Fermi architecture [27] the maximum number of threads allowed to be accommodated in a single thread block along one dimension is 1024, ten bits would be enough to encode a prefix reference.

Finally the last 7 bits represent the tag ID of the query node. Tag ID is needed to determine if a match should be carried diagonally on lines 6 and 18.

FIELD		SIZE
Push stack	Value for children with "/" relationship	1 bit
	Value for children with "//" relationship	1 bit
	...	2 bits
Pop stack	...	2 bits
	Value obtained from descendant	1 bit
	Value obtained from nested paths	1 bit

Table 3: Merged push/pop stack entry. The size of each entry is 1 byte. Padding bits are depicted as "...".

4.4 Exploited GPU Parallelism Levels

Since a GPU executes instructions in the SIMT (Single instruction, multiple threads) manner, at each point in time multiple threads are concurrently executed on SM. CUDA programming platform executes the same instruction in groups of 32 threads, called a *warp*. As each GPU kernel (thread) evaluates one query node, intra-query parallelism is achieved through the parallel execution of threads within a warp. Parrallel evaluation is beneficial in comparison to serial query execution not only when the number of split nodes is large, but also in a case, when most of the split nodes appears in the nodes which are close to query root.

Threads are grouped into blocks, which are executed each on one SM; having said that, the amount of parallelism is not bounded to the number of SMs. It could be further improved by scheduling warps from different logical thread blocks on one physical SM. This is done in order to fully utilize the available hardware resources. But the number of co-allocated blocks depends on available resources (shared memory and registers), which are shared between all blocks executing on SM.

SM uses simultaneous multithreading to issue instructions from multiple warps in parallel manner. However GPU architecture, as well as resource constraints, limit the maximum number of warps, available for execution. The ratio between achieved number of active warps per SM and maximum allowed limit, is known as the *GPU occupancy*. It is always desirable to achieve 100% occupancy, because a large number of warps, available to be scheduled on SM helps to mask memory latency, which is still a problem even if we use fast shared memory. There are two main ways to increase occupancy: increasing the block size or co-scheduling more blocks that share the same physical SM. The trade-off between these two approaches is determined by the amount of resources consumed by individual threads within a block.

Inter-query parallelism is achieved by parallel processing of thread blocks on SMs and sharing them in time multiplex fashion.

MEMORY TYPE	SCOPE	LOCATION
Global	All threads	Off-chip
Shared	One thread block	On-chip
Register	One thread	On-chip

Table 4: Characteristics of memory blocks available as part of the GPU architecture. The scope describes the range of accessibility to the data placed within the memory.

Because typical queries in XML filtering systems are relatively short, we pack query nodes related to different queries into a single thread block. This is done in order to avoid having large number of smal thread blocks, since GPU achitecture limits maximum number of thread blocks, that could be scheduled on SM. Therefore packing maximizes the number of used SPs within a SM. However depending on query size some SPs could be unoccupied. We address this issue by using a simple greedy heuristic to assign queries to blocks.

Finally, the Fermi architecture [27] opened a possibility to leverage inter-document parallelism, by executing several GPU kernels concurrently, every kernel processes a single XML document. It is supported by using asynchronous memory copying and kernel execution operations together with fine-grained synchronization signaling events, delivered through *GPU event streams*. This feature allows us to increase GPU occupancy even in cases when there is a shortage of queries, which in normal situation would lead to underutilization, hence lower GPU throughput. Benefits from the concurrent kernel invocation are later discussed in the experimental section 5.

4.5 Efficient Memory GPU Memory Hierarchy Usage

The GPU architecture provides several hierarchical memory modules (summarized in Table 4), depicting varying characteristics and latencies. In order to fully benefit from the highly parallel architecture, good performance is achieved by carefully leveraging the trade-offs of the provided memory levels.

Global memory is primarily used to copy data from CPU to GPU and to save values, calculated during kernel execution. Kernel personailies and XML document stream are examples of such data in our case. Since global memory is located off the chip its latency penalty is very high. If a thread needs to read/write value from/in global memory it is desirable to organize those accesses in *coalesced* patterns, when each thread within a warp accesses adjacent memory location, thus combining reads/writes into a single contiguous global memory transaction, avoiding memory bus contention.

In our case, global memory accesses are minimized such that:

- The thread reads its personality at the beginning kernel of execution, then stores it in registers.
- Only threads, corresponding to root nodes, write back their match state at the end of execution.

Unlike the kernel personality XML document stream is shared among all queries within a thread block, which makes it a good candidate for storing in shared memory. However XML event stream is too big to be stored into shared memory, which has very limited size. It is hence placed in global memory and is read by all threads throughout execution. We optimize accesses by reading the XML event stream into shared memory in small pieces, looping over the document in a strided manner.

Experimental evaluation showed that the main factor that limits achieving high GPU occupancy is the SM shared memory. In order to save this resource we merged the pop and push stacks into a single data structure. This merged stack contains compressed elements, each of which consumes of 1 byte of shared memory. The most significant half of this byte stores information needed for the push stack, and the least significant half encodes the pop stack value. The detailed encoding scheme is shown in Table 3. Both the push and pop parts of the merged stack value contain the fields: $childMatch$ and $descMatch$, described in Section 4.2.

4.6 Additional Optimizations

Processing of pop event makes each node responsible for reporting its match/mismatch to the prefix node through the prefix pointer, because the split node does not keep information about its children. In case of a massively parallel architecture like a GPU, this reporting could lead to race conditions between children, reporting their matches in an unsynchronized order. In order to avoid these race conditions the CUDA Toolkit provides a set of atomic operations [26], including atomicAnd(), needed to implement the match propagating logic, described in Section 4.2.

Experimental results showed that the usage of atomic operations heavily deteriorates performance, up to several orders of magnitude in comparison to implementations having non-atomic counterparts of that operations. To overcome this issue each merged stack entry is coupled with an additional *childrenMatch* array of predefined size. All children report their matches into different elements within this array, therefore avoiding race conditions. After children reporting is done, the split node can iterate through this array and collect matches, "AND"-ing elements of the childrenMatch array and finally reporting its own match according to the obtained value. The size of childrenMatch array is statically set during compilation, such that the algorithmic implementation is customized to the query set at hand.

This number should be chosen carefully, since large arrays could significantly increase shared memory usage. Each query is coupled with exactly the number of stack columns needed, depending on the number of respective children. Customized data structures are compiled (once, offline) from an input set of queries. This is in contrast to a more generic approach which would greatly limit the scalability of the proposed solution, where each query node would be associated with a generic data structure, accommodating for a max (rather than custom) set of properties. For most practical cases, twig queries are not very "bushy", hence this optimization could significantly decrease filtering execution time, as opposed to the general solution leveraging atomic operation semantics.

5. EXPERIMENTAL RESULTS

Our performance evaluation was completed on two GPU devices from different product families, namely:

- NVidia Tesla C2075: Fermi architecture, 14 SMs with 32 SPs, 448 computational cores in total.
- NVidia Tesla K20: Kepler architecture, 13 SMs with 192 SPs each, 2496 compute cores.

Measurements for CPU-based approaches are produced from a dual 6-core 2.30GHz Intel Xeon E5-2630 server with 30GB of RAM, running on CentOS 6.4 Linux. As a representative of software-based XML filtering methods, we used the state-of-the-art YFilter [12].

Wall-clock execution time is measured for the proposed approach running on the above GPUs, and for YFilter executed on the 12-core CPU. In the case of YFilter, parsing time is not measured, since it is done on CPU and uses similar streaming SAX techniques. Execution time starts from having the parsed documents in the cache. With regards to the GPU setup, execution time includes sending the parsed documents, filtering, and receiving back filtering results.

Documents of different sizes are used, obtained by trimming the original DBLP XML dataset [1] into chunks of different lengths; also, synthetic XML documents are generated using the DBLP DTD schema with the help of ToXgene XML Generator [6]. Experiments were performed on single documents of sizes ranging from 32kB to 2MB. Furthermore, to capture the streaming nature of pub-subs, another set of experiments was carried out, filtering batches of 500 and 1000 25kB synthetic XML documents.

Several performance experiments while varying the block size were carried out, with conclusion that a block size of 256 threads is best, maximizing utilization hence performance (data omitted for brevity).

As for the profile creation, unique twig queries are generated using the XPath generator provided with YFilter [12]. In particular, we made use of a fairly diverse query dataset:

- Number of queries: 32 to 2K.
- Query size: 5, 10 and 15 nodes.
- Number of split points: 1, 3 or 6 node respectively (to the above query sizes).
- Maximum number of children for each split node: 4 nodes.
- Probability of '*' and '//': 10%, 30%, 50%

5.1 GPU Throughput

Figure 7 shows the throughput of Tesla C2075 GPU using a 1MB XML document, for an increasing number of queries with varying lengths. Throughput is given in MB/s and in thousands of XML Events/s, which is obtained assuming that on average every 6.5 bytes of the XML document encodes push/pop event (due to document design).

Data for different wildcard and //-probabilities and other document sizes is omitted for brevity, since they do not affect the total throughput.

The characteristics of the throughput graph are correlated with the results reported in [23]: starting off as a constant, throughput eventually reaches a point, where all computational GPU cores are utilized. After this point, which we refer to as *breaking* point, the amount of parallelism available by the GPU architecture is exhausted. Evaluating more queries will result in serialized execution, which results in an almost linear relation with the added queries (e.g. a 50% decrease in throughput as the number of queries is doubled).

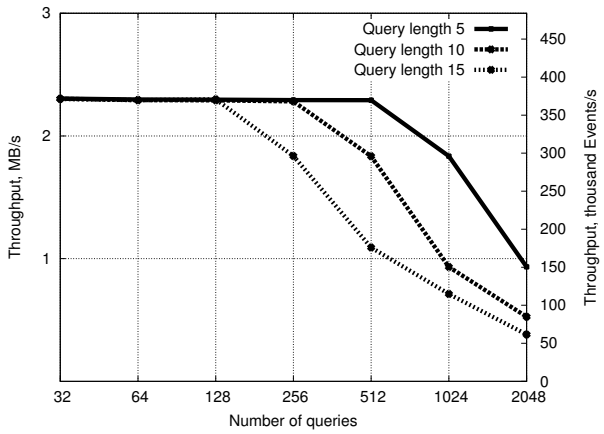


Figure 7: Tesla C2075 throughput (in MB/s and thousands of XML Events/s) of filtering 1MB XML document for queries of length 5,10 and 15.

5.2 Speedup Over Software

In order to calculate the speedup of filtering over YFilter, we measured the execution time of YFilter running on a CPU and our GPU approach running on Tesla C2075 while filtering 256 queries of length 10. This particular query dataset was picked to study the speedup of a fully utilized GPU, because it corresponds to one of the breaking points shown on Figure 7. As mentioned earlier, the execution times for both systems do not include the time spent on parsing the XML documents.

Figure 8 shows that the maximum speedup is achieved for documents of small size. As the size increases, speedup gradually lowers and flattens out for documents $\geq$ 512kB. This effect could be explained by the latency of global memory reads, since number of strided memory accesses grows with increasing document size.

The existence of wildcard and // has a positive effect on the speedup: the execution time of the YFilter depends on size of the NFA automaton, which, in turn, grows with the aforementioned probability.

5.3 Batch Experiments

To study the effect of inter-document parallelism we performed experiments with batches of documents. These experiments used synthetically generated XML documents.

Since the YFilter implementation is single-threaded and cannot benefit from the multicore/multiprocessor setup that we used for our experiments, we also perform measurements for a "pseudo"-multicore version of YFilter. This version equally splits document-load across multiple copies of the program, the number of copies being equal to the number of CPU cores. Query load is the same for each copy of the program. Note that the same tecnique cannot be applied to split query load among multiple CPU cores in previous experiments, since this could possibly deteriorate YFilter performance due to splitting single unified NFA into several automata.

For this experiment we have measured the batched execution time not only on the Tesla C2075 GPU, but also on the Tesla K20, which has six times more available computational cores.

FACTOR	CPU	GPU	FPGA
Document size	Decreases	Minimal effect	No effect on the filtering core
Number of queries	Slowly decreases	No effect prior breaking point, decreases after	Slowly decreases
Query length	Slowly decreases	No effect prior breaking point, decreases after	Minimal effect
'*' and //-probability	Decreases on 15% on average per 10 % increase in probability	No effect	No effect
Query bushyness	Minimal effect	No effect until maximum number of query node children exceeds predefined parameter	Minimal effect
Query dynamicity	Minimal effect	Minimal effect	**No support**
Batch size	Slowly decreases	Minimal effect	No effect on the filtering core

Table 5: Summary of factors and their effects on the throughput of CPU-, FPGA- and CPU-based XML filtering. Query dynamicity refers to the capability of updating user profiles (the query set). The FPGA analysis is added using results from our study in [25].

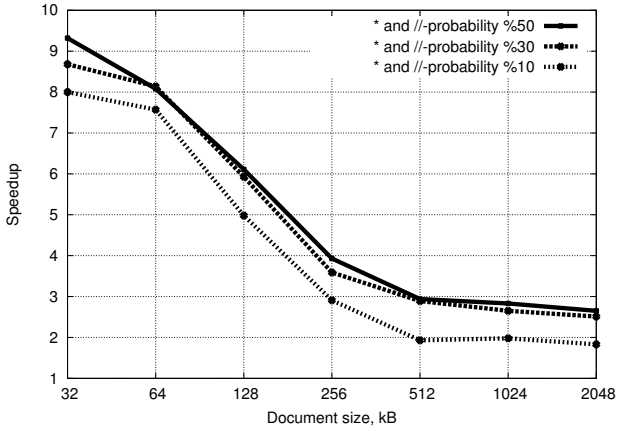


Figure 8: Speedup of GPU-based version running on Tesla C2075 for fixed number of queries (256) and query size (10). Data is shown for queries, having wildcard and //-probabilities equal to 10%, 30% and 50 %.

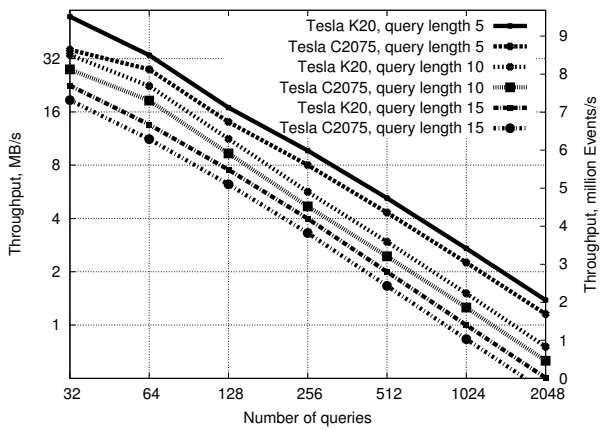


Figure 9: Batched throughput (500 documents) of GPU-based version running on Tesla C2075 and Tesla K20 for wildcard and //-probability fixed at 50%. Data is shown for queries, having length equal to 5, 10 and 15. Throughput is shown in MB/s as well as in millions of Events/s

Figure 9 shows throuput graph for batch of 500 docuemnts (experiments with other batch sizes yields similar results). Unlike Figure 7 the graph does not have a breaking point. This happens because all available GPU cores are always occupied by concurrently executing kernels, therefore GPU is always fully utilized. Doubling the number queries or query length requires two times more GPU computational cores, thus decreasing throughput by factor of two.

Figures 10 and 11 individualy study the effect of query length and number of queries on the speedup for batches of size 500 and 1000 respectively. In both cases speedup drops almost by almost a factor of two, while query load is doubled, and eventually even leads to perfomance, worse than CPU-version. Thus we can infer that YFilter throughput is only sigtly affected by query length or number of queries.

Both figures show that GPU performance (hence speedup over CPU version) slightly deteriorates with the increasing batch size. This effect could be expalined by low global memory bus throughput due to irregular asynchronous copying access patter.

On Figures 10 and 11 the speedup for the Tesla K20 is better than for Tesla C2075, but is not as big as the ratio of the number of their computational cores: the amount of concurrent kernel overlapping is limited by GPU architecture.

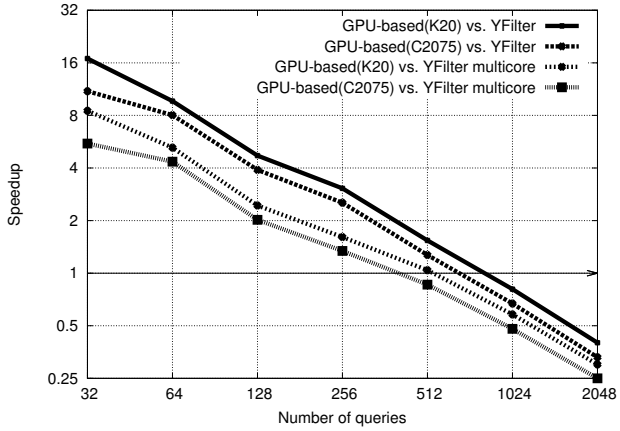
Finally, the speedup of the GPU-based versions over the software-multithreaded version is, as expected, lower then the speedup over a single-thread YFilter.

5.4 The Effect of Several Factors on Different Filtering Platforms

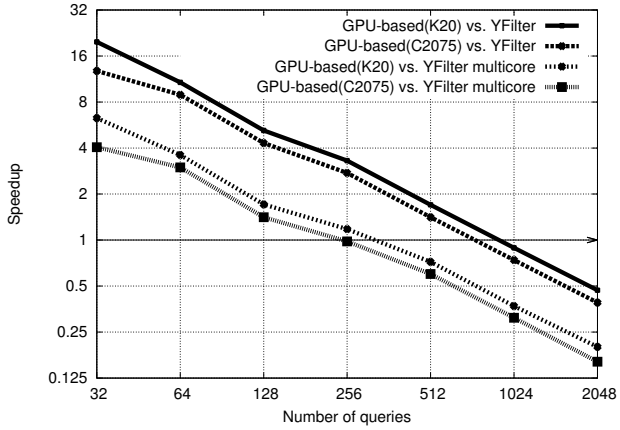
Table 5 summarizes the various factors affecting the single-document filtering throughput for GPUs as well as for software approaches (Yfilter) and FPGAs. The FPGA analysis is added using results from our study in [25].

Query dynamicity refers to the capability of updating user profiles (the query set).

While FPGAs typically provide a high throughput, their main drawback is that that queries are static. As queries are mapped to hardware structures, updating queries could result in up to several hours; this is because hardware compi-

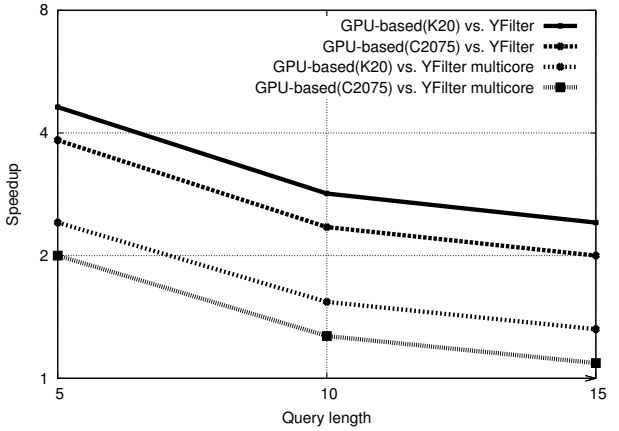


(a) Batch of 500 documents

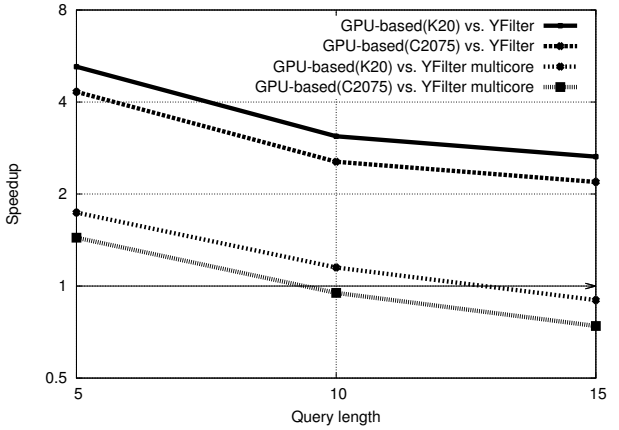


(b) Batch of 1000 documents

Figure 10: Speedup for Tesla C2075 and Tesla K20 GPUs over single-threaded and multicore YFilter for document batches. Query length is equal to 5, wildcard and //-probability is fixed at 50%.



(a) Batch of 500 documents



(b) Batch of 1000 documents

Figure 11: Effect of increasing query length on GPU speedup. Number of queries is fixed at 128, wildcard and //-probability is equal to 10%.

lation (synthesis/place-and-route) is a very complex process. On the other hand, queries can be updated on-the-fly when using CPUs and GPUs.

6. CONCLUSIONS

This paper presents a novel XML filtering framework, which exploits the massive parallelism of GPU architectures to improve the filtering performance. GPUs enable the use of a highly parallel architecture, while preserving the flexibility of software approaches. Our solution is able to process complex twig queries holistically, without requiring an additional post-processing step. By leveraging all available levels of parallelism we were able to extract maximum performance out of a given GPU. In our experiments we were able to achieve speedup of up to 9x in a single-document scenario and up to 4x in the batched-document case against a multicore software filtering system.

7. ACKNOWLEDGMENTS

This work has been supported in part by NSF Awards 0905509, 1161997 and 0811416. We gratefully acknowledge the NVidia donation of Tesla-accelerated GPU boards. The numerical simulations needed for this work were performed on Microway's Tesla GPU accelerated compute cluster.

8. REFERENCES

[1] University of Washington xml repository. `http://www.cs.washington.edu/research/xmldatasets`.

[2] XML Path Language. Version 1.0. `http://www.w3.org/TR/xpath`.

[3] S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 141–152. IEEE, 2002.

[4] M. Altınel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of the 26th Intl Conference on Very Large Data Bases (VLDB), Cairo, Egypt*, 2000.

[5] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose*

Computation on Graphics Processing Units, pages 94–103. ACM, 2010.

[6] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 616–616. ACM, 2002.

[7] F. Beier, T. Kilias, and K.-U. Sattler. GiST scan acceleration using coprocessors. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 63–69. ACM, 2012.

[8] R. Bordawekar, L. Lim, and O. Shmueli. Parallelization of XPath queries using multi-core processors: challenges and experiences. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 180–191. ACM, 2009.

[9] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal. AFilter: adaptable XML filtering with prefix-caching suffix-clustering. In *Proceedings of the 32nd international conference on Very large data bases*, pages 559–570. VLDB Endowment, 2006.

[10] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal*, 11(4):354–379, 2002.

[11] G. Diamos, H. Wu, A. Lele, J. Wang, and S. Yalamanchili. Efficient relational algebra algorithms and data structures for GPU. *CERCS, Georgia Institute of Technology, Tech. Rep. GIT-CERCS-12-01*, 2012.

[12] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Database Systems (TODS)*, 28(4):467–516, 2003.

[13] G. Gou and R. Chirkova. Efficient algorithms for evaluating XPath over streams. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 269–280. ACM, 2007.

[14] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems (TODS)*, 29(4):752–788, 2004.

[15] D. A. Guimarães, F. d. L. Arcanjo, L. R. Antuña, M. M. Moro, and R. C. Ferreira. Processing XPath structural constraints on GPU. *Journal of Information and Data Management*, 4(1):47, 2013.

[16] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 419–430. ACM, 2003.

[17] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524. ACM, 2008.

[18] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010.

[19] J. Kwon, P. Rao, B. Moon, and S. Lee. FiST: scalable XML document filtering by sequencing twig patterns. In *Proceedings of the 31st international conference on Very large data bases*, pages 217–228. VLDB Endowment, 2005.

[20] W. Lu, K. Chiu, and Y. Pan. A parallel approach to XML parsing. In *Grid Computing, 7th IEEE/ACM International Conference on*, pages 223–230. IEEE, 2006.

[21] W. Lu and D. Gannon. Parallel XML processing by work stealing. In *Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 31–38. ACM, 2007.

[22] A. Mitra, M. Vieira, P. Bakalov, W. Najjar, and V. Tsotras. Boosting XML filtering with a scalable FPGA-based architecture. In *Proceedings of 4th Conference on Innovative Data Systems Research (CIDR)*, 2009.

[23] R. Moussalli, R. Halstead, M. Salloum, W. Najjar, and V. J. Tsotras. Efficient XML path filtering using GPUs. In *Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures. Seattle, USA*, pages 1–10, 2011.

[24] R. Moussalli, M. Salloum, W. Najjar, and V. Tsotras. Accelerating XML query matching through custom stack generation on FPGAs. In *High Performance Embedded Architectures and Compilers*, pages 141–155. Springer, 2010.

[25] R. Moussalli, M. Salloum, W. Najjar, and V. J. Tsotras. Massively parallel XML twig filtering using dynamic programming on FPGAs. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 948–959. IEEE, 2011.

[26] NVidia. CUDA Toolkit Documentation. Version 5.0. `http://docs.nvidia.com/cuda/index.html`.

[27] NVidia. NVIDIA's next generation CUDA compute architecture: Fermi. Version 1.1. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[28] Y. Pan, Y. Zhang, and K. Chiu. Hybrid parallelism for XML SAX parsing. In *Web Services, 2008. ICWS'08. IEEE International Conference on*, pages 505–512. IEEE, 2008.

[29] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 431–442. ACM, 2003.

[30] L. Shnaiderman and O. Shmueli. A parallel twig join algorithm for XML processing using a GPGPU. 2012.

[31] X. Si, A. Yin, X. Huang, X. Yuan, X. Liu, and G. Wang. Parallel optimization of queries in XML dataset using GPU. In *Parallel Architectures, Algorithms and Programming (PAAP), 2011 Fourth International Symposium on*, pages 190–194. IEEE, 2011.