Processing XPath Structural Constraints on GPU

Dilson A. Guimarães, Filipe de L. Arcanjo, Laura R. Antuña,
Mirella M. Moro, Renato C. Ferreira

Universidade Federal de Minas Gerais, Brazil
{dilsonag, filipe, laura.antuna, mirella, renato}@dcc.ufmg.br

Abstract. Technologies such as CUDA and OpenCL have popularized the usage of graphics cards (GPUs) for general purpose programming, often with impressive performance gains. However, using such cards for speeding up XML Databases processing is yet to be fully explored. XML databases offer much flexibility for Web-oriented systems. Nonetheless, such flexibility comes at a considerable computational cost. This article shows how graphics cards can be leveraged to reduce the computational cost of processing an important subset of XPath queries. It presents an algorithm designed to consider the cost model of GPUs and to evaluate queries efficiently. An experimental study reveals that this algorithm is more efficient than implementations of a similar strategy on CPU for all the datasets tested. The speedups with respect to exist-db, a popular XML database system, are as high as two orders of magnitude.

Categories and Subject Descriptors: I.3 Computer Graphics [**I.3.1 Hardware Architecture**]: Graphics processors; H.2 Database Management [**H.2.4 Systems**]: Query Processing

Keywords: XPath Queries, XML, GPU

1. INTRODUCTION

The development of technologies such as CUDA and OpenCL has raised a new era in Computer Science. The power of massively parallel graphics processing units (GPUs), formerly restricted to domains such as 3D rendering and games, has been unleashed and is now available to programmers for general purpose applications[1]. As a consequence, we are currently seeing new algorithms being developed for GPU from several application domains[2]. Often, GPU implementations are shown to be several times faster than its CPU counterparts.

Coding for GPUs is still difficult and demands broad knowledge from the programmers. These Graphics cards have, typically, up to six different types of memory and a complex cost model associated with them. This model includes some uncommon characteristics such as the absence of recursive calls and a phenomenon called *task divergence*, which drastically impacts the performance of applications [Coutinho et al. 2011]. Despite such challenges, the performance boosts that have been observed in areas such as Bioinformatics, Scientific Simulations and Statistical Modeling are indeed impressive. Several areas, however, remain unexplored.

Such is the case of Database Systems, in particular XML Database Systems[3]. These are very popular as a consequence of the flexible, portable and hierarchical data models provided by XML, which makes them particularly interesting for Web applications [Moro et al. 2009]. However, an unfortunate consequence of such a rich data model is that queries are frequently expensive. These

[1]GPU: `http://gpgpu.org`
[2]CUDA: `http://www.nvidia.com.br/object/cuda-apps-flash-new.html`
[3]XML: `http://www.w3.org/TR/REC-xml/`

This work was partially funded by CNPq, FAPEMIG and InWeb, Brazil.

queries are expressed in specific languages such as XPath[4]. XPath models XML documents as trees. Then, an XPath query uses a compact syntax to specify one or more paths in such trees. Some XPath queries require traversing the whole tree, which makes processing them computationally expensive.

In this article we show that the increased processing power of GPUs can reduce query execution times in large XML databases. We present a query evaluation approach for handling a subset of XPath[5] that maximizes the utilization of GPUs. After presenting concepts and related work (Section 2), we detail the contributions of this article as follows.

—An XPath query processing algorithm that fits the cost model of GPUs (Section 3.1) and its implementation (Section 3.2).
—A thorough experimental evaluation of the algorithm, showing that GPUs are indeed viable and fast alternatives to traditional XPath query processing (Section 4).
—A discussion on the challenges of processing the whole XPath semantics over GPUs (Section 5).

2. CONCEPTS AND RELATED WORK

This section is divided in two parts. First, we briefly introduce the semantics of XML documents and the XPath query language. Then, we go over related work.

2.1 XML and XPath Overview

An XML document can be represented as a tree (or rooted acyclic graph) in which nodes represent document elements, and edges the hierarchy (or nesting) property of the elements in the document. This is formally defined as a structured tree.

Definition 2.1. *Structured tree:* The structured tree $T(d)$ of an XML document d is a tuple $(V,\ E,\ r)$ where V is the set of vertices given by d elements, E is the set of edges given by the elements nesting property, and r is the root of the tree given by the document root. We denote the root of a structured tree $T(d)$ by $r(T(d))$.

Definition 2.2. *Structured tree path:* Let $T(d) = (V, E, r)$ be the structured tree of document d and v a vertex, where $v \in V$. A path $P(T(d), v)$ to a vertex v within $T(d)$ is a sequence of vertices (v_1, v_2, ..., v_n), for each pair $(v_i, v_{i+1}) \in E$, such that $v_1 = r$ and $v_n = v$.

Definition 2.3. *Path labeling:* Let $label(v_i)$ be the label of vertex v_i and $P(T(d), v_n)$ a path to the vertex v_n. $L(P)$ is the sequence $(label(v_1), label(v_2), \ldots, label(v_n))$, where $(v_1, v_2, \ldots, v_n)$ is the path $P(T(d), v_n)$.

An XPath query expression is composed of structural and value constraints. Existing surveys show that processing the structural constraints takes longer than processing the value ones (mostly because different indexes may aid the latter) [Gou and Chirkova 2007; Moro et al. 2009]. Therefore, this article also focuses on evaluating the structural constraints over GPU.

XPath structural constraints are given by the axes parent-child (/) and ancestor-descendant (//) operators. Such constraints may be combined, then forming path expressions which use axes and labels (or tags) of the desired XML nodes. A simplified grammar is as follows:

```
<query> → /<label><query> | //<label><query> | /<label> | //<label>
```

[4]XPath: `http://www.w3.org/TR/xpath/`
[5]In section 2, we show that this subset of XPath is responsible for a large portion of the total runtime in evaluating XML query processors.

Any path expression may be translated to a regular expression by mapping the occurrences of the axis // to .* and eliminating the axis /. Note that the expression .* denotes the occurrence of any symbol zero or more times. For example, the path expression $/a//b$ is translated to $a.^*b$. The regular expression $e(q)$ resulting from a path expression q is called *regular expression induced by* q. The answer to the structural constraints of an XPath query expression is defined as the set of vertices reached by the path whose labeling is recognized by the regular expression induced by the query.

Definition 2.4. *Answer set:* Let $X(e(q))$ be the language recognized by the regular expression $e(q)$. The answer set $Q(q,d)$, for a query q over document d, is defined as the set of vertices V_e such that $L(P(T(d),V_e)) \in X(e(q))$.

2.2 Related Work

The high parallelism offered by GPUs has been recently explored for a myriad of purposes that go beyond the traditional graphics processing. The most interesting examples come from the data processing scenarios with complex computations, which can be performed by using general purpose computation on GPU. It is also possible to combine the advantages of GPU and CPU. For instance, [Krulis et al. 2012] combines them for processing multiple distance computations required by similarity searching for content-based multimedia retrieval.

In relational databases, GPUs have also been explored for performing common query operations, such as join, division and aggregation. Specifically, Bandi et al [2004] analyzed Oracle's integration approaches for integrating the parallel implementation of spatial operations as an external procedure. Others have also explored the parallel implementation of operations such as predicates, Boolean expressions and aggregations, as well as different types of join (nested loops, sort-merge, hash) over GPU [Govindaraju et al. 2005; He et al. 2008]. More recent work has explored the new generation of NVidia GPUs, in which GPU and CPU share memory space, and achieved better efficiency for join operations [Kim et al. 2009] and [Kaldewey et al. 2012]. Other memory limitations were further analyzed in [Sitaridi and Ross 2012], which has also proposed optimized techniques for storing data and aiding the possible conflicts when acessing values from the same database.

GPUs have also been employed for searching and sorting on databases and general datasets. For example, He et al [2009] propose a system for processing relational queries over GPU by parallel division and sorting. Likewise, Govindaraju et al [2006] present the algorithm *GPUTeraSort*, which uses CPU and GPU together for sorting databases with billions of records. They achieve great efficiency by sorting large partitions of data, which when done over GPU are 10 times faster than when over CPU. More recently, Capannini et al [2012] introduce a completely different approach for sorting based on a new computational model tailored for many-core architectures. Its major performance gains are on memory consumption by achieving space complexity of $\Theta(1)$.

Regarding XPath processing, two good surveys are available in [Gou and Chirkova 2007] and [Moro et al. 2009], which cover the most important work on evaluating XPath queries in CPU. On the other hand, in GPU, Moussali et al [2011] propose a solution for filtering documents, *i.e.*, given a set of documents, it returns all those that satisfy a query. A different solution is proposed in [Si et al. 2011], which requires a cost model for deciding how to partitionate the document prior to parallel processing. Both approaches consider only a part of the XPath semantics: the structural constraints. Such an important feature is also present in our work, which uses a subset of known operations over a flexible interface. Furthermore, our article also extends the state-of-the-art by considering a fully-fledge engine (instead of a filtering one) and by processing the queries without any other data (as those necessary for a cost model).

3. XPATH IN GPU

In this Section we present our algorithm for evaluating structural constraints of XPath queries in GPUs. The goal of the algorithm is to maximize the utilization of the GPU resources thus improving the execution time.

3.1 Algorithm

Given a query with n operators, it is possible to construct a NFA with $n+1$ states that recognizes the same language as a regular expression induced by a query q [Gou and Chirkova 2007]. Such NFA is called a *query induced NFA*. For instance, a query `/a//b/c` induces the regular expression `a.*bc` and the automaton in Figure 1, where $\sum$ denotes all possible symbols (*i.e*, the XML element labels).

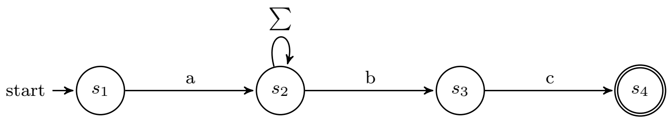


Fig. 1. Nondeterministic Finite Automaton induced by the query /a//b/c.

Executing a query over an XML document is then equivalent to simulate an instance of the NFA induced by the query on each node of the document's structural tree. This way, the label of each document node serves as input to the NFA states. Then, the document nodes that lead the NFA to a final state belong to the query answer.

Paths in the structural tree may overlap, *i.e.*, the paths for any two nodes may share the same prefix. Specifically, if a node u is descendant of node v, the path to v is a prefix of the path to u. Similarly, if node a is a parent of nodes b and c, the path to a is a prefix of both paths to b and c. The algorithm takes advantage of such overlap for reducing the cost of simulating multiple instances of the induced NFA.

The proposed algorithm is based on a breadth first search on the structural tree [Cormen et al. 2009], evaluating instances of the query induced NFA. There is an instance for each node of the tree. Initially, the induced NFA is in the initial state and the root of the tree is visited. When visiting each document tree node, the set of active states of the NFA is equaled to the set of active states of the associated parent node instance. Then the label of the node is used as an input symbol to the current node NFA. If the final state is reached, the current node is included in the query answer set.

A pseudo-code is presented in Algorithm 1. It calculates the set of active states S_v in the query induced NFA for each node v of the structural tree of the document. The predicate $\text{Final}\,(S_v)$ is true if and only if the final state of the induced NFA is in S_v. The function $\text{Next}\,(S_v, a)$ performs the transitions: reading symbol a (*i.e.*, processing a document tree node) activates the states S_v. Function $\text{Parent}\,(v)$ returns the parent node of v. Also, each level is defined by its distance in number of nodes to the root.

Suppose an XML document as shown in Figure 2(a). Its corresponding structured tree can be represented by $T\,(d) = (V,\, E,\, \{1\})$, with $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (1, 3), (2, 4), (3, 5), (4, 6)\}$, as seen in Figure 2(b). For the NFA shown in Figure 1, induced by the query `/a//b/c`, the step-by-step definition of active states and update of the results set X for each of the tree's nodes in Figure 2(c).

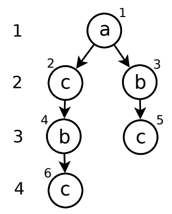
In Algorithm 1, each node is visited once. The operations involving the set of active states S_v in a node can be executed in linear time in $|S_v|$. As the number of nodes is equivalent to the number of elements in the XML document, $|d|$, and the number of states in the induced NFA is one more than the size of the query in number of operators $|q|$, the execution time for the algorithm is $O\,(|d|\,|q|)$.

Algorithm 1 XPath Query Processing Algorithm.

```
Require: A query $q$ and a document $d$
Ensure: A set of elements $X$ in $d$ that satisfy $q$
 1: $X \Leftarrow \emptyset$
 2: $S_{r(T(d))} \Leftarrow \text{Next}\ (\emptyset, label\,(r\,(T\,(d))))$
 3: if Final $\left(S_{r(T(d))}\right)$ then
 4:     $X \Leftarrow X \cup \{r\,(T\,(d))\}$
 5: end if
 6: for $i = 1$ to height$(T(d))$ do {for all levels $i$ except the root}
 7:     for all $v \in V_i\,(d)$ do {for all vertices in level $i$}
 8:         $S_v \Leftarrow \text{Next}\ \left(S_{\text{Parent}(v)}, label\,(v)\right)$
 9:         if Final$\,(S_v)$ then
10:             $X \Leftarrow X \cup \{v\}$
11:         end if
12:     end for
13: end for
14: return $X$
```

```
<a>
  <c>
    <b>
      <c>JIDM</c>
    </b>
  </c>
  <b>
    <c>2013</c>
  </b>
</a>
```

(a)



(b)

—$X = \emptyset$
—$S_1 \Leftarrow \text{Next}\ (s_1, a) = s_2$
—$S_2 \Leftarrow \text{Next}\ (s_2, c) = s_2$
—$S_3 \Leftarrow \text{Next}\ (s_2, b) = s_3$
—$S_4 \Leftarrow \text{Next}\ (s_2, b) = s_3$
—$S_5 \Leftarrow \text{Next}\ (s_3, c) = s_4, X = X \cup \{5\} = \{5\}$
—$S_6 \Leftarrow \text{Next}\ (s_3, c) = s_4, X = X \cup \{6\} = \{5, 6\}$

(c)

Fig. 2. XML document, its structured tree and the active states

The inner loop in Algorithm 1 can be executed in parallel. The only restriction is the access to the answer, which must be synchronized. Considering a polynomial number of processors, a query q in a document d can be processed in a time $O\,(h\,(T\,(d))\,|q| + |Q\,(d, q)|)$, where $h\,(T\,(d))$ is the height of the structural tree of document d.

3.2 Implementation

The strategy outlined in the previous section runs on a structural tree representation of an XML document that is maintained in the GPU's memory. Evaluating a query on GPU then requires two steps: analysis and serialization.

The analysis step is responsible for converting the XML document, initially as a text file, into a tree stored in the computer's main memory. Such a document parsing of the document is performed by the special library xerces[6]. Unlike the structural tree, defined in the previous section, this tree contains enough information to reconstruct the entire XML document. This is necessary in order to build up the results based on the nodes that are returned by the GPU.

The serialization step is responsible for converting the tree obtained by the analysis phase into a serialized representation, in which the nodes are stored in contiguous memory locations. This representation is equivalent to the structural tree presented in Definition 2.1. Each node in the serialized tree contains three integers representing (i) a *label* for the node, (ii) the position of the parent node in the serialized tree and (iii) the set of active states - a bit vector - of the NFA after receiving the label of the node as input. This compact representation is all the information required by Algorithm 1. It is also convenient due to the limited memory availability within the GPUs[7]. The

[6]Xerces: `http://xerces.apache.org/xerces-c`
[7]The *GeForce GTX 470* used in our experiments, for instance, has just 1.25GB of local memory

nodes are stored in the serialized tree in the same order in which they are traversed in a breadth-first search of the structural tree. Therefore, the representation exploits the spatial locality, which is very important as the GPUs read data in large chunks.

The query processor itself runs in CUDA as a single *kernel*. A kernel in CUDA is a procedure that runs in parallel, directly in the GPU's hardware. The processor invokes the kernel once per level of the structural tree. Then, in each call, threads are spawned for each document node of that level. Each thread obtains the set of states that were previously active based on the parent node and executes the NFA transitions appropriately. Its behavior is that of the innermost loop in Algorithm 1.

The implementation of the kernel is complex because it must avoid divergences in the threads. It utilizes bit operations to simulate transitions in the NFA, which eliminates several of the conditionals that would have been necessary otherwise. A divergence occurs when threads take different paths in the execution of conditional branches. In the GPU, threads are executed in groups called warps. When all threads in a warp are in the same instruction, they all execute in the same cycle. Otherwise, the GPU hardware partially serializes the warp.

Threads output their results to a shared memory location which stores the identifiers of the nodes recognized by the NFA. An atomic add operation available in CUDA (called *atomicAdd*) synchronizes the access this memory region. Such synchronization is probably the main performance bottleneck in the implementation. After finishing the evaluation process, the generated list of node identifiers is transferred to main memory. These identifiers are matched back to the tree generated by the analysis step so that the results are presented to the user.

4. EXPERIMENTAL EVALUATION

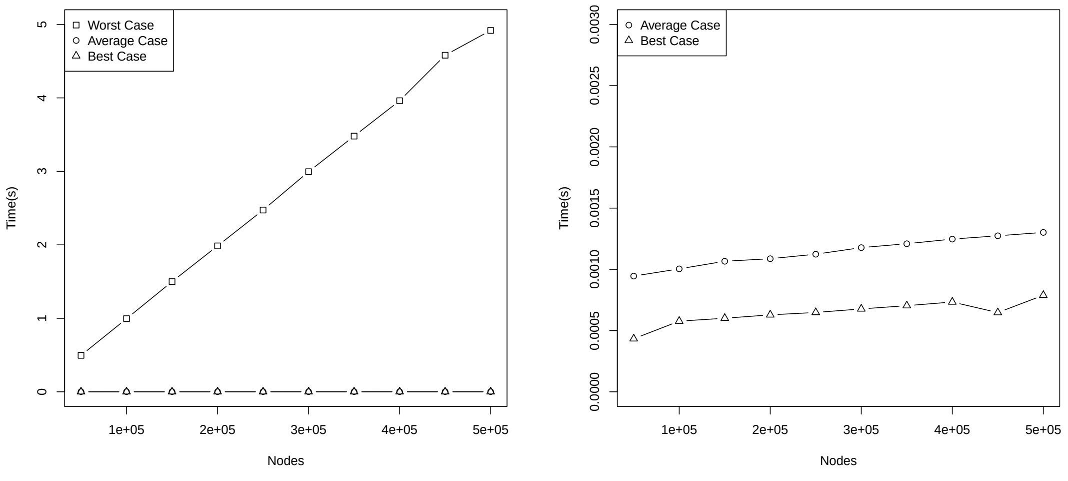
For evaluating performance, we have defined two sets of experiments. First, we have built synthetic datasets in order to stress different features and show the algorithm's efficiency according to the type of XML tree (Section 4.1). Then, we evaluate queries over real datasets (Section 4.2). As for experimental setup, our experimental evaluation was performed on an *Intel Xeon E5620* processor with 32GB of memory capacity and a *nVidia GeForce GTX 470* graphic card with 1.25GB of memory.

4.1 Synthetic Datasets

This first set of experiments considers synthetic datasets that allow us to control one important parameter: the structure of the document. Specifically, as the query evaluation efficiency depends on the structure, we first evaluate its performance as a function of the number of elements n under three different settings: best, worst and average cases. The best case scenario corresponds to an XML document with all elements having the root node as parent. In such a setting, all queries can be completed with only two CUDA kernel calls, then achieving maximum parallelism. The worst case scenario is when the document is a nested list of elements starting at the root node, with every element having one single child. That case requires one kernel call per element, the repressing any parallel processing on the GPU. The average case is composed by random XML document samples with uniform distribution of n distinct elements. In order to ensure that the whole document is evaluated, all queries are of the form //label.

Figure 3 presents the results in two different scales for better visualization: (a) presents an overview of the results, and (b) zooms in to reveal the small performance gap between average and best cases. Each point for the best and worst cases represents the mean of 100 executions. For the average case, 100 random documents were generated and we report averages over 10 executions on each. To keep presentation clean, we omitt standard deviations from the plots, which were very small.

As illustrated, the worst case is significantly slower than the other two, which in turn presented a small difference in terms of performance. In fact, the average and best case performances seem to



(a) (b)

Fig. 3. Time variation in relation to the number of nodes

be unaffected by n, even when pushed towards the limits of what the GPU memory can keep. One reason for such similar performance is that, for a fixed number of nodes, there are much more shallow trees than deep XML trees that can be generated. It is also worth mentioning that the algorithm has a low sensibility to the variation of the query type. Even with a query that forces the entire tree to be traversed, the variation on the number of nodes had little effect on the performance.

These results from synthetic datasets have great importance for three main reasons. First, it provides an overview of the algorithm efficiency for different scenarios. Second, it shows that the query processor (in most cases) provides a high performance, very close to the best case, for distinct structured XML trees. Finally, considering that the worse case scenario is equivalent to an algorithm without any parallelizations, these experiments also show that the proposed query processor will still have a good performance when compared to linear algorithms.

4.2 Real Datasets

This evaluation considers three versions of our algorithm: the one on GPU, one on a CPU by parallelizing the algorithm through the OpenMP[8] platform, and another on CPU by a serial version of the algorithm. For comparison, we have also used the popular native XML DBMS eXist[9]. Considering that eXist operates on secondary memory, its performance considers only *ramdisk*.

This evaluation also considers four real datasets with XML documents of sizes varying from 24MB to 1GB. Table I describes each of them[10]. The number of elements and attributes provide a better idea about the document sizes. The *Maximum depth* column indicates the depth of the XML document trees, whereas the *Average depth* corresponds to the average length between the root and its leaves. For each dataset, four different queries were executed, each one referring to nodes found in different levels and positions of the XML tree, resulting in a broad coverage of most common test cases.

Table II shows the execution times resulting from each query as well as the *speedup* obtained by the performance of the CUDA implementation in comparison with the other versions. The results show that the GPU implementation is significantly more efficient in all cases when compared to the

[8]Plataform OpenMP: `http://www.openmp.org/`
[9]eXist: `http://www.exist-db.org/`
[10]XML Data Repository: `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`

Table I. Datasets used for tests

Dataset	Size	Elements	Attributes	Maximum Depth	Average Depth
dblp 1	1.1G	25.832.523	6.538.402	6	2,877659
psd7003	684MB	21.305.818	1.290.647	7	5,15147
dblp 2	128MB	3.332.130	404.276	6	2,90228
treebank	83MB	2.437.666	1	36	7,87279
nasa	24MB	476.646	56.317	8	5,58314

Table II. Performance measures based on file size

			Execution time (s)				Speedup		
Dataset	**Query**	**#Results**	CUDA	OMP	Serial	eXist	OMP	Serial	eXist
dblp 1	//author	6.563.296	0,114	0,377	0,354	8,298	3,3	3,1	75,3
	//article/author	2.156.500	0,047	0,193	0,390	6,525	4,1	8,3	138,8
	/dblp//article	844.543	0,027	0,119	0,300	0,515	4,4	11,1	19,1
	/author	0	0,013	0,070	0,291	0,869	5,4	22,4	66,8
psd7003	//authors	314.763	0,017	0,082	0,268	0,148	4,8	15,8	8,7
	//accinfo/db	1.199.979	0,031	0,124	0,354	0,415	4,0	11,4	13,4
	//db	1.531.327	0,036	0,133	0,269	1,376	3,7	7,5	38,2
	/ProteinDatabase/ProteinEntry	262.525	0,017	0,086	0,312	0,117	5,1	18,3	6,9
dblp 2	//title	328.859	0,007	0,019	0,042	0,139	2,7	6,0	19,8
	/dblp//volume	114.205	0,004	0,013	0,047	0,062	3,3	11,8	15,5
	//www/title	38	0,002	0,008	0,047	0,044	4,0	23,5	22,2
	/dblp/www	38	0,002	0,008	0,046	0,003	4,0	23,0	1,3
treebank	//NN	186.597	0,005	0,011	0,030	0,108	2,2	6,0	21,5
	//S//NP	419.255	0,009	0,019	0,042	0,278	2,1	4,7	30,9
	//S/NP	125.997	0,004	0,008	0,039	0,185	2,0	9,8	46,3
	/FILE/EMPTY	52.851	0,003	0,007	0,034	0,026	2,3	11,3	8,6
nasa	//name	71.688	0,002	0,003	0,006	0,033	1,5	3,0	16,8
	//other/name	286	0,001	0,001	0,007	0,016	1,0	7,0	15,7
	//year	5.935	0,001	0,001	0,006	0,006	1,0	6,0	5,8
	/datasets/dataset	2.435	0,001	0,001	0,007	0,004	1,0	7,0	4,3

OpenMP version; which, in turn, overcomes the performance of the serial version. About the DBMS *eXist*, its performance is less efficient than the GPU implementation, specially for large files.

It is possible to observe that the dataset size has a great influence in the execution time, which increases as the XML document gets bigger. The execution time growing rate depends on the query processing strategy used, being smaller for the CUDA implementation of our algorithm. This can be explained by the GPU's vast number of processing cores, which allow for high parallelism even as the number of elements in the document grows.

The speedups average obtained comparing the performance of the CUDA implementation with the baselines can be seen in Figure 4. These values stress the greater efficiency of the GPU version. Specifically, the speedups range from 1 (meaning no performance gains) when considering the OpenMP version on the nasa dataset, to 138 times for the eXist-db on the largest dataset.

Although the GPU implementation was significantly faster in most tests, its memory size still represents a limitation. It prevents its use for very large XML databases. In this case, the CPU based query processor comes as better option. In extreme circumstances, when even the CPU memory is not enough to store the XML document, the query can be executed in a hard risk, through *eXist*. Hybrid approaches (in which both CPU and GPU are used to process the query) were not tested. Therefore it is not possible to affirm that they would be more efficient than the presented ones.

5. DISCUSSION ON CHALLENGES AND FUTURE WORK

This paper has introduced a GPU-based implementation of a XPath subset (the structural constraints given by the axes parent-child and ancestor-descendant). In this section, we discuss some improvements that remain as future work as well as the challenges imposed by them.

An enhancement to the algorithm is reporting the query results found by the GPU algorithm in

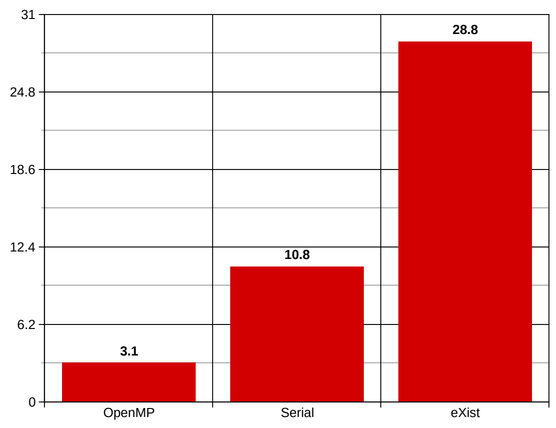


Fig. 4. Average Speedup

a more efficient way. Currently, an atomic operation, which serializes all the GPU threads to avoid concurrency, updates the result set. Such avoidance negatively interferes with the algorithm efficiency. A solution to such a problem is through preventing this serialization, which implies in remodeling the data structure to store the results.

Furthermore, the GPU based algorithm focuses on queries with structural constraints given by the axes parent-child (/) and ancestor-descendant (//). As previously mentioned, processing these constraints takes longer than processing value constraints. Therefore the gains obtained by improving the performance of queries with the former type is greater than by turning the latter more efficient. Nonetheless, for a complete implementation of the XPath semantics, value constraints must also be considered. Such constraints may be as simple as an element (or attribute), a comparison operator and a value. Other examples include finding first occurrence of a tag ($//tag[1]$) or for elements that contain a given attribute with a specific value ($//tag[@att = value]$).

However, there are many challenges for extending the GPU query evaluation to accept queries with predicates. The main one is possibly the GPU internal memory size limitation. Specifically, the current algorithm stores only a basic structure of the XML tree, which contains the minimum information necessary about the nodes. In order to process predicates, it is also necessary to store all information about the attributes, their values as well as the text values of all elements. Overall, the main challenge would be to define such a compact structure that would perform well in the highly parallel context of the GPUs.

6. CONCLUSION

This article focused on XPath query evaluation over GPUs, instead of the traditional CPUs. Our solution consists of an algorithm designed specifically for modern GPUs: it evaluates queries by decomposing them into smaller ones that can be executed in parallel. It uses a compact representation of the XML document and bit parallel operations in order to maximize the information within the limited memory of the GPU and reduce the number of conditional branches.

Though simple, the proposed solution has shown superior or equivalent performance when compared to the state of the art baselines in all seven datasets considered (three synthetic and four real datasets). The speedups when compared to the popular eXist-db reached more than two orders of magnitude (138 times). Our evaluation results also showed that, in average, the algorithm efficiency is similar to the best case performance, and that it scales for different document sizes. Finally, we have also discussed the challenges of expanding our article to evaluate predicates as well, which is left out as future work.

REFERENCES

Bandi, N., Sun, C., Agrawal, D., and El Abbadi, A. Hardware Acceleration in Commercial Databases: a case study of spatial operations. In *Proceedings of the International Conference on Very Large Data Bases*. Toronto, Canada, pp. 1021–1032, 2004.

Capannini, G., Silvestri, F., and Baraglia, R. Sorting on GPUs for Large Scale Datasets: a thorough comparison. *Information Processing and Management* 48 (5): 903–917, 2012.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Meira Jr., W. Divergence Analysis and Optimizations. In *Proceedings of the International Conference on Parallel Computing Technologies*. Washington, DC, USA, pp. 320–329, 2011.

Gou, G. and Chirkova, R. Efficiently Querying Large XML Data Repositories: a survey. *IEEE Transactions on Knowledge and Data Engineering* 19 (10): 1381–1403, 2007.

Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Chicago, IL, USA, pp. 325–336, 2006.

Govindaraju, N. K., Lloyd, B., Wang, W., Lin, M., and Manocha, D. Fast Computation of Database Operations using Graphics Processors. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques*. Los Angeles, CA, pp. 206–217, 2005.

He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N. K., Luo, Q., and Sander, P. V. Relational Query Coprocessing on Graphics Processors. *ACM Transactions on Database Systems* 34 (4): 21:1–21:39, 2009.

He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P. Relational Joins on Graphics Processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Vancouver, Canada, pp. 511–524, 2008.

Kaldewey, T., Lohman, G., Mueller, R., and Volk, P. GPU Join Processing Revisited. In *Proceedings of the International Workshop on Data Management on New Hardware*. Scottsdale, Arizona, pp. 55–62, 2012.

Kim, C., Kaldewey, T., Lee, V. W., Sedlar, E., Nguyen, A. D., Satish, N., Chhugani, J., Di Blas, A., and Dubey, P. Sort vs. Hash Revisited: fast join implementation on modern multi-core CPUs. *Proceedings of the VLDB Endowment* 2 (2): 1378–1389, 2009.

Krulis, M., Skopal, T., Lokoc, J., and Beecks, C. Combining CPU and GPU Architectures for Fast Similarity Search. *Distributed and Parallel Databases* 30 (3-4): 179–207, 2012.

Moro, M. M., Braganholo, V. P., Dorneles, C. F., Duarte, D., de Matos Galante, R., and dos Santos Mello, R. XML: some papers in a haystack. *SIGMOD Record* 38 (2): 29–34, 2009.

Moussalli, R., Halstead, R., Salloum, M., Najjar, W., and Tsotras, V. J. 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, pp. 1–10, 2011.

Si, X., Yin, A., Huang, X., Yuan, X., Liu, X., and Wang, G. Parallel Optimization of Queries in XML Dataset Using GPU. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Programming*. Washington, USA, pp. 190–194, 2011.

Sitaridi, E. A. and Ross, K. A. Ameliorating Memory Contention of OLAP Operators on GPU Processors. In *Proceedings of the International Workshop on Data Management on New Hardware*. Scottsdale, Arizona, pp. 39–47, 2012.