Parallel Optimization of Queries in XML Dataset using GPU

Xujie Si[1], Airu Yin[1], Xiaocheng Huang[2], Xiaojie Yuan[2], Xiaoguang Liu[2],Gang Wang[2]
[1] *College of Software, Nankai University,Tianjin,300071,China*
[2] *College of I.T., Nankai University,Tianjin,300071,China*
E-mail: {sixujie,yinairu,hxc,nk_yuanxj,liuxguang,wgzwp }@gmail.com

***Abstract*—As XML is playing a crucial role in web services, databases, and document processing, efficient processing of XML queries has become an important issue. On the other hand, due to the increasing number of users, high throughput of XML queries is also required to execute tens of thousands of queries in a short time. Given the great success of GPGPU (General-Purpose computations on the Graphics Processors), we propose a parallel XML query model based on GPU, which mainly consists of two efficient task distribution strategies, to improve the efficiency and throughput of XML queries. We have developed a parallel simplified XPath language using Compute Unified Device Architecture (CUDA) on GPU, and evaluate our model on a recent NVIDIA GPU in comparison with its counterpart on eight-core CPU. The experiment results show that our model achieves both higher throughput and efficiency than CPU-based XML query.**

***Keywords*-parallel optimization; GPU; XML query**

I. INTRODUCTION

As the de facto standard for encoding tree-oriented, semi-structured data, XML has brought interoperability and standardization benefits in various fields. Due to the explosive growth of XML dataset in all kinds of fields and the significant improvement of XML[2], much more efficient and higher throughput query methods are required. How to accelerate the XML queries and how to improve the query throughput are becoming hot research topics[10].

The way to increase the processor speeds of single-core microprocessors to keep up with Moore's law is severely challenged in recent years because of the physical limits of power and transistor density. The trend to increase performance of processors is changing from increase the single processor speed to increase the number of cores. How to harness the processing power of new parallel processing architectures is also becoming a hot research topic. Recent researches like [8][9][11] are trying to accelerate the performance of database operations using Graphics Processing Units. So, considering the recent success of GPGPU, we investigate whether and how we can design a parallel XML query model based on GPU to achieve response time and throughput requirements in XML query.

In this paper, we propose a parallel XML query model to accomplish better performance. We develop a parallel optimized XPath to implement our model, however, our model is not designed just for XPath, which can be easily used in other XML queries. Our model mainly focuses the data-partitioning in the process of query.

The core contributions of the paper are summarized as follows:

- We design a cost model to evaluate the XML query. Using this cost model, we can efficiently distribute query load between CPU and GPU.
- We propose improved data partition strategy presented by [6], which is proved to be more efficient.
- We implemented a parallel version of XPath using CUDA on GPU. The experimental results show that our model is quite efficient.

II. RELATED WORK

Many researches about improving XML traversal pattern or structural join method to optimize performance of XML queries have been done in [4][5][17]. Researches about estimation of answer size and cost of queries have also been explored in [3][15].

Several works about parallel processing and querying XML have been done in recent years. [12] investigates the seemingly quixotic idea of parsing XML in parallel on a shared memory multi-core computer. It prepares XML document to determine the logical tree structure of an XML document and then uses the logical tree to divide XML document into chunks. As an improvement, [13] presents a work-stealing parallel XML processing model, in which the load balance among the threads is dynamically controlled. In contrast, [14] gives a static load-balancing scheme for parallel XML parsing on Multi-core CPUs. It uses a static, global approach to reduce synchronization and load-balancing overhead.

[6] considers a scenario where an XPath processor uses multiple threads to concurrently navigate and execute individual XPath queries on a shared XML document. It proposes three strategies for parallelizing individual XPath queries: Data partitioning, Query partitioning, and Hybrid (query and data) partitioning. Furthermore, [7] proposes a parallelization algorithm using the statistics and several heuristics to find proper parallelization point in an XPath query. It also shows that Data partitioning strategy is always better than Query partitioning strategy or Hybrid partitioning strategy.

III. PRELIMINARY

Since our model is implemented by XPath, it is necessary to give a brief view to XPath. An XPath expression consists of a sequence of location steps, each one of which has three components: an axis, a node test, and a predicate. Given a context node of an abstract XML tree, an XPath expression uses the specified axis to navigate the XML tree. The node test and the predicate are used to select the nodes specified by the current axis. Though complicated test conditions and predicates can be used to various kinds of XML queries, we will ignore the test condition and predicate in this paper. Because here we focus on how to improve the performance of query by using parallel method rather than technique of specific test conditions, which may be important in practice but is not what this paper is about. So the form of XPath we consider in this paper is as follows:

$$Xpath ::= E \mid Tag \mid XPath\ /\ Tag \mid XPath\ //\ Tag \quad (1)$$

Where '*E*' means empty path, '*Tag*' means a tag (the element names in XML document), '/' is used to match child node whose name is the tag, and '//' is used to match all progeny children whose name is the tag. Figure 1(a) is the part of XML document, Figure 1(b) is an XPath query.

<Books> <Book ISBN='0743273567'> <title>The Great Gatsby</title> <author>F. Scott Fitzgerald</author> </Book> <Book ISBN='0684826976'> <title>Undaunted Courage</title> <author>Stephen E. Ambrose</author> </Book> </Books> (a) An example of XML	/Books/Book//author (b) An example of XPath

Figure 1 Example of XML and XPath

XPath's execution model is inherently sequential: each location step operates on the node set returned as a result of evaluating the previous location step or the starting context. It seems that the order of execution of location steps cannot be changed. However, the following characteristics of XML queries provide significant opportunities for parallelism: access to the XML documents are read-only during the process of querying; execution of queries can be reordered in any manner as the queries are executed on different parts of dataset. Our motivation for paralleling XML query is to properly use intermediate query results.

IV. MODEL

Our XML query model uses statistics-based information of XML document to decide whether a parallel query processing plan is needed and how to execute parallel query efficiently. Before any query begins, we read the XML document into CPU memory and send a copy to GPU memory. When given a XML query, we first use our cost model to evaluate the costs of CPU query and GPU query. If the cost of serial query is not expensive, the query will be executed on CPU (in serial manner). Otherwise, the query will be executed on GPU (in parallel manner). We propose two data partitioning strategies for parallel query: query path based partitioning and query dataset based partitioning. The first is similar to the data partitioning strategy proposed in [10], while the second is what we first proposed.

The cost of query can be estimated by the cardinality of the query result. Though it is hard to know the accurate value, we can estimate the approximate value, which is enough to our model.

$$XPath = Tag_1 + \sum_{i=2}^{n}(op_{i-1} + Tag_i) \quad (2)$$

$$Cost(XPath) = \prod_{i=1}^{n-1} F(Tag_{i+1} \mid Tag_i, op_i) \quad (3)$$

Here, n is the length of *XPath*, which indicates the number of tags in *XPath*. $F(Tag_B \mid Tag_A, op)$ means the average number of children whose tag is Tag_B and father tag is Tag_A if the operator is '/', otherwise it means the average number of descendants whose tag is Tag_B and ancestor tag is Tag_A. The function $F(Tag_B \mid Tag_A, op)$ should be calculated before any query begins. In fact, the cardinality of the result is not always proper to estimate the cost of the query /A/B/E//F, if the XML tree is similar to what in Figure 2.

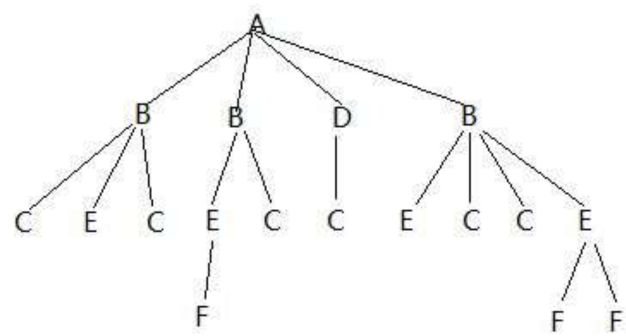

Figure 2 An example of XML tree

To be more accurate, the cost of the query can be defined as follows.

$$XPath_i = Tag_1 + \sum_{j=2}^{i}(op_{j-1} + Tag_j) \quad (4)$$

$$QueryCost = Max\{\cos t(XPath_i)\}(1 \le i \le n) \quad (5)$$

If the cost of $XPath_n$, the estimation of the cardinality of query result, is very small, it does not represent the cost of the total query is really small. However, if the cost of $XPath_n$ is zero, there is no need to execute the query.

The aim of the data partitioning strategy is to make different processors execute queries at different sections of the XML document. We should always try to make the CPU perform the least amount of work and make the GPU get enough parallel tasks at the same time.

A. *Query Based Partitioning Strategy*

This approach decomposes parallel tasks by splitting the query into two parts: serial part and parallel part. The serial part is executed on CPU. References of the resulting node set and the parallel sub-query are transported to GPU, and then

GPU executes the second part query using the resulting node set as the context node set in parallel.

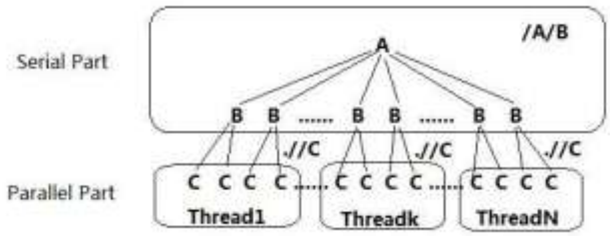

Figure 3 Execution using Query Based Partition

Figure 3 illustrates the execution of XPath query: /A/B//C, using query based partitioning strategy. This query is split into two sub-queries: /A/B and .//C. The first part, /A/B is executed on CPU and the resulting node set of c nodes is distributed across N GPU threads. Each GPU thread then executes the sub-query, .//C, on the set of c nodes assigned to it. As a result, each GPU thread concurrently navigates a distinct part of the XML tree. By combining the local results from the GPU threads, we can get the final result of the original query.

Obviously, this strategy will show different performance if the XPath query is partitioned in different points. So what we do now is to design a strategy to find the optimal partition point of a specific query. Since the process of a query is a traversal of the XML tree, a perfect partition point means proper portion of traversals on CPU and GPU. Using our cost model, we can estimate the optimal partition point.

$$Traversal(XPath_i) = \sum_{j=1}^{i} \cos t(XPath_j) \tag{6}$$

The total traversal of the query implies the work of the query. So if the execution of query is in a serial manner, we can suppose the execution time is Traversal (XPath). The parallel execution time would be

$$Traversal(XPath_k) + \frac{Traversal(XPath_n) - Traversal(XPath_k)}{\cos t(XPath_{k+1})} \times R \tag{7}$$

where k is the partition point of the query and R is the traversal speed ratio of CPU and GPU when using only one thread. We need to find the partition point k, which makes the total parallel execution time be the least. Using our cost model, the partition point k can be easily calculated.

B. Dataset Based Partitioning Strategy

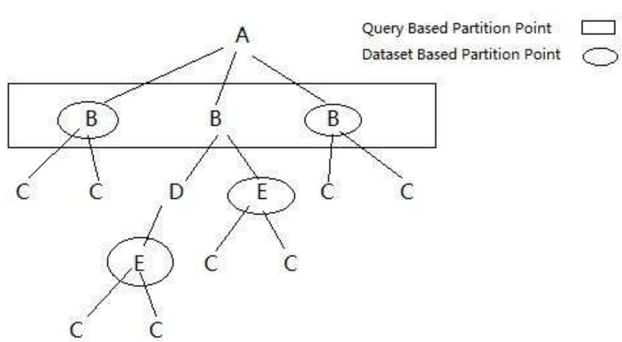

Figure 4 Partition points of two strategies

Compared with the query based strategy, dataset based strategy emphasizes much more on the dataset to be queried. The most important aim of the dataset based strategy is to make the load balance for each GPU thread. When the dataset is not distributed evenly, the performance of the query based strategy will be degraded. Figure 4 shows such a circumstance.

We can see that query based strategy always choose the same level nodes as partition points, while dataset based strategy choose the nodes who have approximate the same amount of descendant nodes. The dataset based strategy can decompose the tasks in very fine granularity. When given the number of threads, we can calculate the threshold of the partition point. If there are p threads and T nodes of the XML document, the ideal distribution is that each thread execute query on T/p nodes. However, the XML document is tree structure rather than relation structure. It is almost impossible to distribute tasks n such an ideal way, since additional traversal cost is needed when making a distribution. We have following equations.

$$T = AddtionalCost + p \times PartitionLimitation \tag{8}$$

$$Additional\,Cost = \frac{B^n}{B^H} \times T \tag{9}$$

$$PartitionLimitation = (1 - \frac{B^h}{B^H}) \times \frac{T}{p} \tag{10}$$

Where B is the average number of children of each node, H is the average depth of the queried XML document, h is the average depth of the partition point, which can be estimated by the query based strategy or simply supposed to be the half of H.

In the serial phrase of query, we execute the XPath query on CPU until we find some context node whose descendant number is less than the Partition Limitation. We stop executing query in current branch of XML tree, and add the current context node and sub-query, which has not been executed, to the stack of parallel tasks. After the serial phrase, each node in the resulting set has almost the same amount of progeny nodes.

V. EXPERIMENTS

A. Prototype Implementation

We use Xerces-C to parse XML document and implement a parallel XPath engine on GPU using CUDA-C. Before any query begins, we parse XML document into CPU memory, and then transfer a copy to GPU. We also implement a memory management class to collect the query results on GPU.We tested our implementation on x86/Linux machine with Intel i7 CPU and 4GB memory, the GPU is NVIDIA GeForce GTX 480 and CUDA version 4.0.

B. Datasets and Queries

For our experiments, we used two typical XML datasets: XMark[16] and the Penn Treebank[1]. Table 1 presents the structural characteristics of the two datasets. Table 2 presents the XPath queries used in experiment for each dataset.

Table 1: Characteristics of the XML document

Dataset	Size (MByte)	Elements	Attributes	Max Depth
Treebank.xml	82	2437666	1	36
Xmark.xml	111	1666315	381878	12

Table 2: XPath Queries used in Experiment

Treebank.xml	T1：/FILE/EMPTY//NP
	T2：/FILE/EMPTY/S//NP
	T3：/FILE/EMPTY/S/NP//N
XMark.xml	XM1：/site/open_auctions/open_auction/bidder
	XM2：/site/regions//item/description/parlist/listitem

C. Evaluations

In order to compare the differences of performance, we execute the same XML query in the same dataset using three different strategies: serial strategy, query-based parallel strategy and dataset-based parallel strategy. The serial strategy is to execute the whole XML query using only one thread on CPU, which is currently the common way. The query-based strategy and dataset-based strategy are executed in two phrases. The first phrase is to split the whole query into an amount of sub-queries, which is executed on CPU in a serial manner. The second phrase is to execute these sub-queries on GPU in a parallel manner.

Figure 5 shows the query on Penn Treebank XML document with different execution strategies. We can see that both query based and dataset based strategies are better than serial execution. Furthermore, dataset based strategy is better than query based strategy. The sub-trees in Penn Treebank are not balanced, since the maximum depth is as many as 4.5 times as the average depth. As we have discussed in Section 4B, the dataset based strategy is expected to be better than query based strategy in such a circumstance. Here, the experimental results show this point.

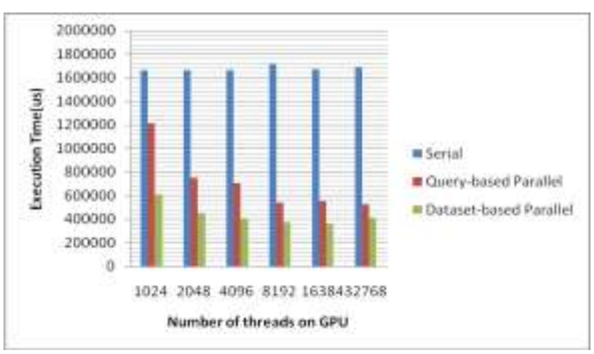

Figure 5 Results of T3 query

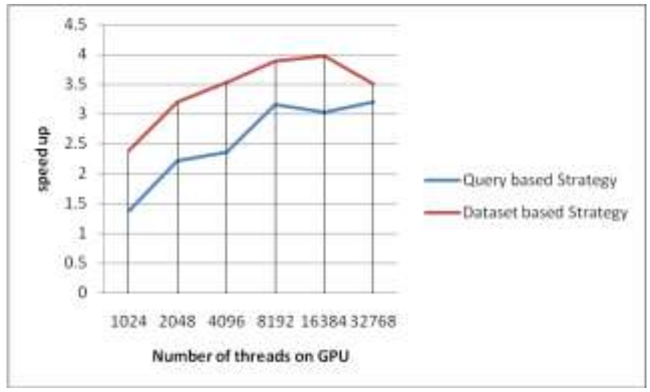

Figure 6 Speed up of two strategies

Figure 6 shows the speed up of two strategies. As we can see, the speed up of two strategies increases significantly when the number of threads increases. This is a normal result since more threads mean high concurrency. However, when the number of threads exceeds a threshold the speed up tends to be const even go down, which implies the GPU's saturation. Another reason of going down speed up of the dataset based strategy is that more serial work is needed as the number of threads increases.

Figure 7 shows the result of XM2. It indicates that there is no significant difference between query based strategy and dataset based strategy. This reflects the queried XML document should have balanced tree structure. Indeed, compared to the Penn Treebank, XMark XML document is much more balanced. The experiment shows that the two strategies have no difference when the queried document is balanced.

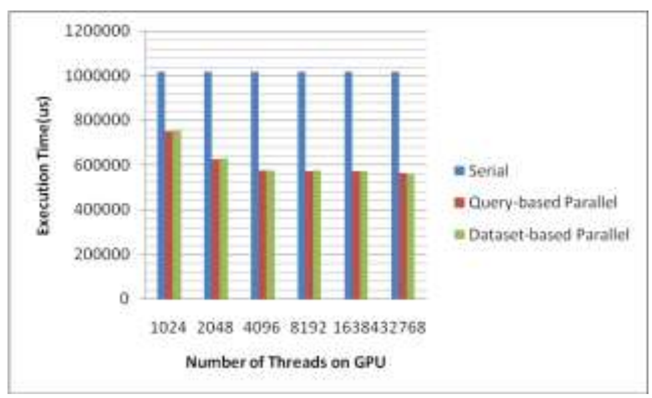

Figure 7 Results of XM2

Queries like XM1 are not suitable for parallelizing because such a query is too short and there is no '//' operation in the query, which is the most expensive operation of the query. Let's see the performance of the two strategies when given such an unideal query.

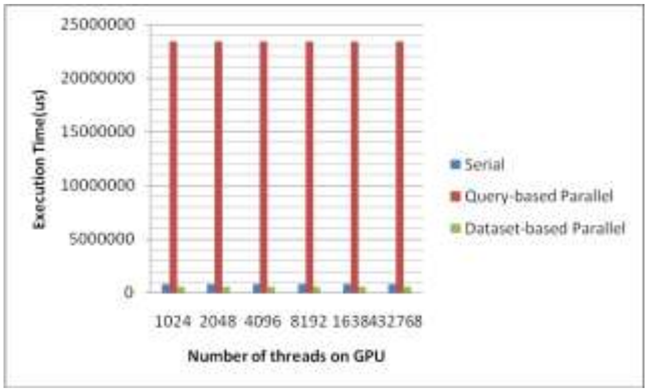

Figure 8 Results of XM1

From figure 8, we can see that query based strategy achieve a very bad performance, which much slower than the serial execution. When using the query based strategy, we cannot get enough resulting node set for the parallel phrase, which make most threads on GPU be idle. So the strong parallel ability of GPU is wasted when using query based strategy.

Our experiments have demonstrated that in most cases, the two strategies can achieve significant performance improvement compared with the serial algorithm. And dataset based strategy will be better than the query based strategy when the XML document is not balanced or the cost of query is inexpensive.

VI. Conclusion

In this paper, we evaluated the problem of parallelizing XML query using GPU. We propose two data partitioning strategies, and examine the two strategies by implementing a parallel XPath engine on GPU. Both the strategies improve the query performance significantly. We also realize that other issue about the parallelizing need to be researched, such as using GPU ability to support huge amount threads to achieve high throughput of XML queries. We plan to explore these issues in detail in our future work.

Acknowledgment

This paper is partially supported by NSFC of China (60903028, 61070014, 61170184), Key Projects in the Tianjin Science & Technology Pillar Program (11ZCKFGX01100).

References

[1] The PENN Treebank Project. http://www.cis.upenn.edu/treebank.

[2] http://www.cs.washington.edu/research/xmldatasets/www/repository.html.

[3] Ashraf Aboulnaga , Alaa R. Alameldeen , Jeffrey F. Naughton, Estimating the Selectivity of XML Path Expressions for Internet Scale Applications, Proceedings of the 27th International Conference on Very Large Data Bases, pages 591-600, September 11-14, 2001.

[4] A. Balmin, T. Eliaz, J. Hornibrook, L. Lim, G. M. Lohman, D. Simmen, M. Wang, and C. Zhang. Cost-based optimization in db2 xml. IBM Syst. J., 45(2):299–319, 2006.

[5] Andrey Balmin, Fatma Özcan, Ashutosh Singh, and Edison Ting. Grouping and optimization of xpath expressions in db2 Rpurexml. In SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 1065–1074, New York, NY, USA, 2008. ACM.

[6] Rajesh Bordawekar, Lipyeow Lim, and Oded Shmueli. Parallelization of xpath queries using multi-core processors: challenges and experiences. In EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology, pages 180–191, New York, USA, 2009. ACM.

[7] Rajesh Bordawekar, Lipyeow Lim, Anastasios Kementsietsidis, and Bryant Wei-Lun Kok. Statistics-based parallelization of xpath queries in shared memory systems. In EDBT '10: Proceedings of the 13th International Conference on Extending Database Technology, pages 159–170, New York, NY, USA, 2010. ACM.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. Journal of Parallel and Distributed Computing 68(10): 1370-1380, October 2008.

[9] Shuai Ding, Jinru He, Hao Yan and Torsten Suel. Using graphics processors for high performance IR query processing. Proceedings of the 18th International Conference on World Wide Web, pages 421-430, 2009.

[10] Andrew Eisenberg and Jim Melton. Advancements in SQL/XML, ACM SIGMOD Record, v.33 (3), page 79-86 ,September, 2004.

[11] Bingsheng He and Jeffrey Xu Yu. High-Throughput Transaction Executions on Graphics Processors. Proceedings of the VLDB Endowment, 4(5):314-325, February 2011

[12] Wei Lu, Kenneth Chiu, and Yinfei Pan. A parallel approach to xml parsing. In GRID '06: Proceedings of the 7th IEEE/ACM International Conference on Grid Computing, pages 223–230, Washington, DC, USA, 2006. IEEE .

[13] Wei Lu and Dennis Gannon. Parallel xml processing by work stealing. In SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches, pages 31–38, New York, NY, USA, 2007. ACM.

[14] Yinfei Pan, Wei Lu, Ying Zhang, and Kenneth Chiu. A static load-balancing scheme for parallel xml parsing on multicore cpus. Cluster Computing and the Grid, IEEE International Symposium on, page 351–362, 2007.

[15] Sherif Sakr. Towards a comprehensive assessment for selectivity estimation approaches of XML queries, International Journal of Web Engineering and Technology, 6(1):58-82, August 2010.

[16] Albrecht Schmidt , Florian Waas , Martin Kersten , Daniela Florescu , Michael J. Carey , Ioana Manolescu , Ralph Busse. Why and How to Benchmark XML Databases. ACM SIGMOD Record Vol30(3),page27-32, Sep. ,2001.

[17] Jens Teubner, Torsten Grust, Sebastian Maneth, and Sherif Sakr. Dependable cardinality forecasts for xquery. Proceedings of the VLDB Endowment, 1(1):463–477, 2008.