The GPU-Oriented Tree Representation Based on the Method of Finding the Remainder

Vladimir Voronkin
vl.voronkin@raxperi.com

Andrey Malikov
malikov@ncstu.ru

Elmira Azarova
elmira.azarova@mail.ru

Aleksey Shchegolev
a.wegolev@gmail.com

North-Caucasus Federal University,
Institute of Information Technologies and Telecommunications,
Russial Federation

Abstract

This paper describes a new method of the tree-like structure representation in data bases. The developed method of finding relationships between nodes is based on the unique prime factorization theorem which states that every natural number can be written as a product of prime numbers. The developed method is implemented using NVIDIA graphics processing unit parallel computing and NVIDIA CUDA technology. The paper analyzes the process of tree construction with the developed method applied, represents the algorithms of GPU-based processing of trees coded using the developed method, and also the results of performance snapshot.

1 Introduction

Computer systems are developing rapidly, processor architectures are changing, the main and disk memorysizes are increasing. Nowadays the rapid increase of power and data volumes allows for the development of high-performance data management and processing systems.

A large amount of data in the world in one form or another is currently represented as a hierarchy or mutual dependency where some pieces of information depend on the others.

What is more, data management systems are beginning to migrate from the disk-oriented technology to the memory-oriented. The development of DBMS has led to integrating in-memory technology support by many database management systems. [GBE13]

Data processing in the main memory enables a considerable increase of the data processing rate in favor of safety. Modern DBMS, supporting in-memory technology process data in the memory mirroring changes to the disk.

The application of memory-oriented data processing technologies together with the new methods of representing trees using GPU computing can increase the data processing rate.

This research aims to develop a new method of hierarchy-representation in databases that exhibits high performance in tree manipulation operations.

In: S. Hölldobler, A. Malikov, C. Wernhard (eds.): *YSIP2 – Proceedings of the Second Young Scientist's International Workshop on Trends in Information Processing, Dombai, Russian Federation, May 16–20, 2017*, published at http://ceur-ws.org.

2 Related Works

This work is devoted to the description of the developed method of representing trees in databases. The development of the method was initially-oriented towards the maximum parallelization of tree processing and maximum independence of node code from other node codes. The paper focuses on a radically new method of representing trees in relational and nonrelational databases, and also demonstrates that the GPU computing of hierarchical relationships can be more effective than the CPU. The developed method of representing trees is oriented towards the parallel processing mode with the use of a GPU.

2.1 Methods of Representing Trees

All the methods of representing trees in databases can be put into two categories:

– methods of representing trees in non-graph databases;

– methods of representing trees in graph and hierarchical databases.

The following methods belong to the first category:

– The adjacency list method of relational hierarchical modeling.

The adjacency list method is based on the storage of direct parent-child relationships. Each child in a treelike hierarchy relates to its parent which is at one level higher. Using this hierarchy property, a table-level relational dependence can be created.

– The materialized path method.

A materialized path is a string field that consists of element names separated by a separator character [Tro03]. Parents' names of a node for which a materialized path is built are used as the materialized path elements.

– The Nested Sets method.

The Nested Set method implies that there are two additional fields for the node description: a tree coding using the algorithm for the nested sets coding introduced by Joe Celko [Cel12] [Cel] [Cel04], is done as follows: moving down the left side of a tree, it's necessary to travel through all the subtrees beginning with the leftmost to the rightmost and assign each node an auto increment value. When moving down the tree, an auto increment value is as signed to the variable responsible for the interval starting point („left bound"); when moving up - to the variable responsible for the interval end („right bound").

– The nested intervals method.

The nested intervals method, introduced in [Tro05], is based on the materialized path coding using a finite continued fraction $[q_1, q_2, \ldots, q_n]$, where $[q_1, q_2, \ldots, q_n]$, where $q_1, q_2, \ldots, q_n$ are the steps of a materialized path. Rational numbers a/b, where $a >= b >= 1$ and $GCD(a, b) = 1$ are used as tree element codes.

As an example, consider how the code for a materialized path „3.2.2" is created using a continued fraction:

$$3 + \frac{1}{2 + \frac{1}{2}} = \frac{17}{5}$$

To transform a materialized path to a rational expression the convergence principle can be used, while for the inverse transformation the algorithm for the gradual truncation should be used, that in case of a rational expression is the same as Euclid's algorithm for finding GCD [Gai99].

– Modifications and combinations of the afore-named methods.

There are also modifications of the given methods that can fix the inherent problems [Kol09] [vor14] [MT11]. For example, in [MT11] the improvement of the nested intervals method is introduced through the interval coding in the residue number system (RNS). Using RNS enables reducing the length of numbers used by storing a number as a short length remainder. Representing node codes as numbers expressed in RNS enables parallel processing of not only different tree nodes but also one and the same node remainder. Thus, the maximum parallelization of data processing is increased. In [vor14] the method of storing nested intervals expressed in RNS is introduced that solves some problems of sorting numbers expressed in RNS and the index construction.

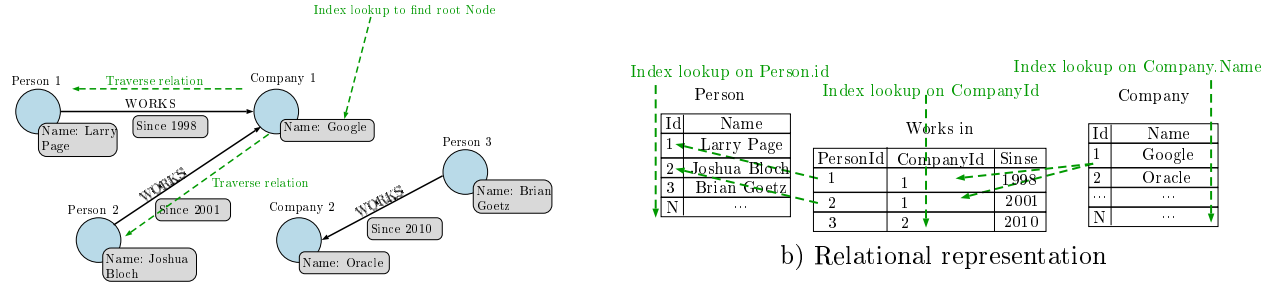
Such methods as the Materialized Path, nested intervals and modifications of the both include the data on children and/or parents in the key while coding and can be processed in parallel with sufficient effectiveness.

The methods of representing trees in graph and hierarchical databases are difficult to classify as most of graph databases use their own formats and structures of data representations.

In some cases pointers are used for linking neighbor nodes. The pointers are bound to the data that appear in arcs. In its structure this hierarchy model is similar to the adjacency lists.

Other [Kup86] [KV93] graph databases use a table model for storing relationship arcs.

Another format of storing graph data is XML and XML-based formats [MS11] [BELP].



a) Graph representation

b) Relational representation

Figure 1: Representation types of dependent data

Fig. 1b and 1a illustrate the aforesaid. Fig. 1b illustrates a relational representation of hierarchical data in DB, while Fig. 1a illustrates data representation in the form of a graph.

The illustration data exemplify the storage of hierarchy data in databases.

One of the distinctive features of graph databases is the base chosen for database objects representation, relations between the data, complex objects and their attributes.

All graph databases are based on a mathematical description of a graph that determines graph manipulation operations. A graph model depends on the base, for example: a directed or undirected graph, marked or unmarked graph nodes, weighted and unweighted arcs.

Some models such as GOOD, GMOD, G-Log, Gram represent both the scheme and entity as a named digraph. LDM is an exception whose schemes are digraphs with leaves representing data and internal nodes represent structured data. LDM consists of a table with two-columns, each of which associates entities with data types (primitive, tuple or set). A more detailed description of graph models is represented in [AG08].

2.2 GPU Data Processing

Graphics processing unit seems to be an advanced high–performance platform for computing. The basic distinctive feature enabling GPU's high performance is its architecture having a great number of ALU. The GPU and CPU architectures are compared in [PT11]. A number of works [BS10] [HKOO11] point to the advantage of GPU processing of data over CPU. On average, the queries were processed 5–30 times faster in relational databases. Some works [PSHL10] [BMCN09] show the increased rate of sorting data using GPU. As far as hierarchical and graph structures are concerned, the works [HN07] [HKOO11] [WO10] are worth mentioning. in which GPU graph computations are made and which describe the speedup of data processing using GPU compared to CPU (when processing graph structures).

Hereafter, mentioning GPU computing means the usage of NVIDIA CUDA technology.

CUDA (Compute Unified Device Architecture) is a NVIDIA technology designed for the development of applications for massive parallel computing units. Due to a great number of ALU. GPU implements the thread model of computation: there are input and output data that consist of the same elements which can be processed independently. The elements processing is performed by the kernel.

A *kernel* is a function executed on GPU and called from CPU. A kernel is composed of a grid of multiple threads grouped into a hierarchy.

The highest level — grid — corresponds to the kernel and groups all the kernel threads. A grid is a one- or two-dimensional array of blocks. Each block is a one/two/three-dimensional array of threads. Furthermore, each block is an entirely independent set of interacting threads.

3 Principles of Tree Construction

By „tree" we will understand a connected acyclic graph composed of a set of nodes $n = (k, d)$, where k is the node code, d is data associated with the node:

$$T = \{n\} = \{(k, d)\}$$

Let us define for a node n operations $key(n) = k$ access to the node code, $data(n) = d$ access to data associated with the node.

Denote by *the node identifier* a prime number unique within node Ids. Let us introduce a set of primes I, representing primes in increasing order of their values:

$$I = s_1, s_2, \ldots, s_m, \ldots$$

Let us call a product of the node parent identifiers by the node Id *the node code*.

The code of a tree root is *the root identifier*.

Let us introduce the operation of getting the following prime number from a set pf primes I:

$$next() = s_i, i = i + 1$$

Primes are needed for tree construction, because tree decomposition should be unique.

The usage of one and the same identifier for more than one node is impossible.

A tree is formed as follows:

1. A prime is chosen as a root node.

2. Each node added to the tree has a code equal to the product of the parent code by the identifier of the node added.

Each tree node in the code encodes the entire path starting from the root. This method of node code creation is similar to the Materialized Path method.

4 Tree Processing Operations

Let us show admissible tree operations. Let us introduce the function of finding the remainder on dividing a by b:

$$mod(a, b) = a - b \cdot \lfloor \frac{a}{b} \rfloor$$

The following tree operations are admissible:

1. $add(T, d, n_{parent})$, where n_{parent} is parent node, d is data added to the tree — operation of adding a node to the tree.

Precondition:

$$exists(T, n_{parent})$$

Realization:

$$T := T \cup \{n(key(n_{parent}) \cdot next(), d)\} \tag{1}$$

Predicate $exists(T, n)$ determines if the node n existing in the tree T:

$$exists(T, n) := \exists n \in T$$

2. Operation of searching a node with the code k in the tree $node(T, k)$.

$$node(T, k) := \{x \in T | key(x) = k\} \tag{2}$$

3. Operation of searching node parents in the tree $parents(T, k)$.

$$parents(T, k) := \{x \in T | mod(k, key(x)) = 0 \wedge k \neq key(x)\} \tag{3}$$

4. Operation of searching a direct parent $parent(T, k)$.

$$parent(T, k) := MAX(parents(T, k)) \tag{4}$$

5. Operation of searching a subtree $subtree(T, k)$.

$$subtree(T, k) := \{x \in T | mod(key(x), k) = 0\} \tag{5}$$

6. $children(T, k)$ — operation of searching direct children.

$$children(T, k) := \{x \in T | k = key(parent(T, key(x)))\} \tag{6}$$

7. Operation of subtree removal $remove(T, k)$.

Realization:

$$T := \{x \in T | mod(key(x), k) \neq 0\} \quad (7)$$

8. Operation of transferring a subtree rooted n_{old} to a new parent n_{new} $move(T, n_{old}, n_{new})$.
Precondition:

$$exists(T, n_{old}) \wedge exists(T, n_{new}) \wedge mod(key(n_{new}), (n_{old})) \neq 0$$

Realization:

$$T := T \cup \{x \in subtree(T, key(n_{old}))| \\ |n(recalcKey(T, key(x), key(n_{old}), key(n_{new})), data(x))\} \quad (8)$$

where $recalcKey(T, k, k_{old}, k_{new})$ is an operation of node code recalculation:

$$recalcKey(T, k, k_{old}, k_{new}) := \frac{k}{k_{old}} * k_{new}$$

Considering the possibility of executing parallel operations with a tree as a set, the following features should be noted:

1. The possibility of parallel tree processing using GPU.

Tree operations (2, 3, 4, 5, 6) that do not modify the tree have no need for interaction with other nodes. All such operations can be processed in parallel or simultaneously. Parallelization of tree manipulation operations up to one thread per node is possible.

2. The order of records in tree representation in the memory influences only the order of records in the result set and does not influence the number of records in it (the result set).

3. The usage of primes in a tree in the sequential order is optional.

5 GPU Computation of Trees

Each node represents a pair: {*node code*: *pointer to node properties*}. Let us call this pair a *record*. Since the node properties can represent a sequence of dynamically-sized free fields, each node code is associated with the pointer to properties location in the file rather than the properties themselves.

In the file data is stored as a sequence of records. When downloading, records are divided as follows: keys are recorded into GRAM, pointers to the properties are recorded into RAM. Downloaded data (keys and pointers) is stored as a sequence. For such form of representation it is important that the order of records in RAM is strictly the same as in GRAM. When swapping some nodes in GRAM, it is necessary to swap nodes in RAM in a similar way.

When executing a query, say, a lookup query, all the records in the tree are reviewed. The query result is computed on-the-fly, computing the desired result while executing the query. Thus the search is not narrowed.

To save the time of the processing result transfer to RAM, the GPU makes a bit map of the query result, in which the bit set denotes a node corresponding to the query, the removed denotes a non-corresponding node, Fig. 2). The bit map is formed in kernels (a kernel is a function executed on the graphics card) of query execution; as a result, one map bit corresponds to each node. For example, with a 4-byte key the size of data transmitted through the bus (between GPU and CPU) is decreased by 32 times.

To get certain nodes in the resulting selection, it is necessary to find set bits in the bit map. The ordinal number of a set bit is a number of a storage cell in RAM (and GRAM) in which the data address of the found node (the found node code) is stored.

Let us consider the process of executing some tree operations. Each operation is executed for each node in the set.

The whole tree processing is realized on GPU.

The result of complex query processing, for example a common parent search, is formed as a result of uniting bit maps of each query result, thus avoiding long operations of each node searching.

For example $commonParents(A, k, r)$ operation of searching common parents for two nodes with the codes k and r looks like this:

$$commonParents(A, k, r) := parents(A, k) \bigcap parent(A, r).$$

For finding common parents, the most rational (with regard to the time of finding a solution) method is executing two queries for parent search for each node and uniting bit maps of the queries results through the bitwise AND

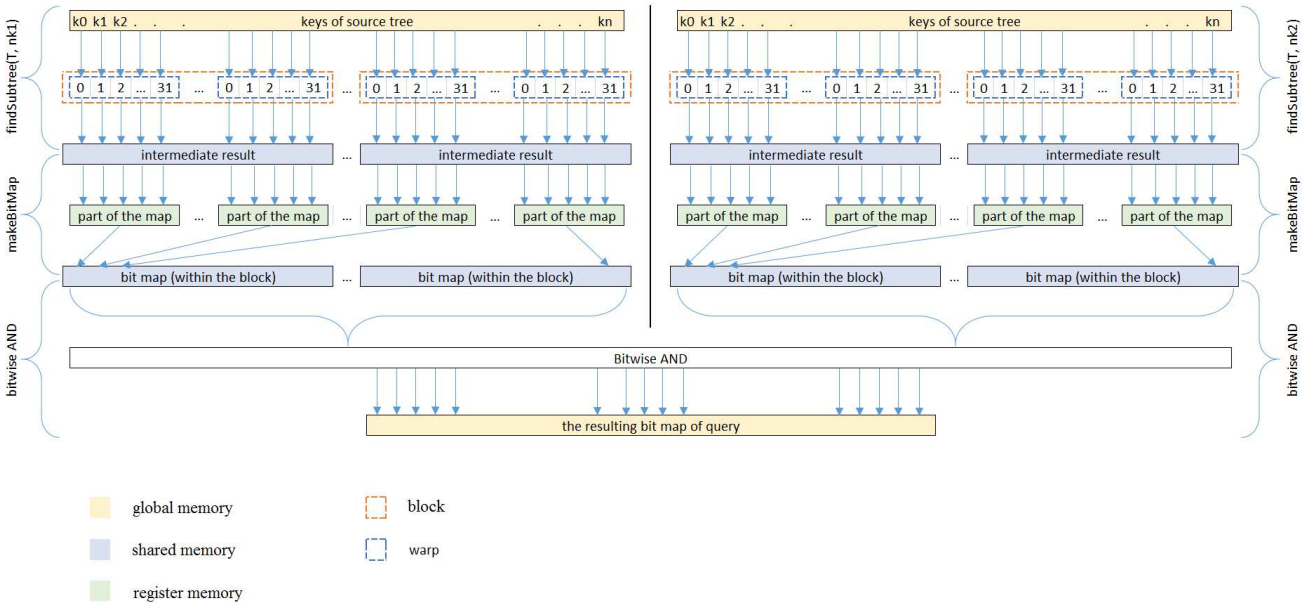


Figure 2: Bitwise AND scheme

operator (Fig. 2). Uniting of bit maps as well as calculations take place in GPU. The GPU execution of the bitwise AND operator for a 4-byte whole requires four multiprocessor cycles without considering memory access delay (4 cycles for the shared memory).

6 The Experiment Results

To check the developed method of representing trees, the DBMS prototype was designed supporting the basic operations of data manipulation: insertion, deletion, transfer, search. Search and transfer queries are executed on GPU, insertion and deletion are executed on CPU. The developed DBMS prototype does not cache data. Reading is performed directly from the disk. Queries are not cached either. The result of search query is a bit map returned to the client in the form of a cursor. At the next cursor record query, the search for the next set bit in the map takes place, then reading of the document and key corresponding to the bit location in the map.

The performance test of the developed method of representing trees was carried out in comparison with DBMS MongoDB. MongoDB was chosen as the popular and high performance DBMS. As an index, MongoDB uses B+tree. For hierarchy representation in MongoDB the Materialized Path was chosen as a hierarchy record type.

6.1 Test Set

A tree-like database is used as a set test. As a DB base, the All-Russian Classifier of Addresses (KLADR) is used. [kla]

Data is represented as a tree with 2.5 million nodes. Each node has a different number of children. The tree is not balanced. The height of the tree is 5 levels.

For running experiments the KLADR reference was extended. The original reference guide has unused fields and occupies approximately 400 MB of the disk space.

Each tree node has the fields represented in Table 1.

It is necessary to discriminate between a geographical division and a tree node to which the geographical division is attached.

The fields represented in Table 1 are common fields for the both DBMSs in the test. MongoDB uses the Materialized Path (field „matPath") to store the hierarchy data. In the field „matPath" the DBMS prototype stores the node code which is the product of parents Ids by the node Id.

The data sets are the same for the both DBMSs tested.

To check the performance of the method of hierarchy representation in comparison to the analogues, a series of tests were conducted.

Table 1: Tree node fields

Node name	Data type	Description
name	char[32]	node name
matPath	varchar[128]	node materialized path
objName	char[16]	name of object associated with node
objData	varchar[10000]	data attached to object
gpsX	float	object coordinate X in GPS coordinate system
gpsY	float	object coordinate Y in GPS coordinate system
level	int	object nesting level in accordance with KLADR
objType	char[8]	object type in accordance with KLADR
objCode	char[16]	object code in accordance with KLADR

A Cold Startup, with all the tests being done, was performed for each DBMS „warm-up". After that, all the tests were carried out. Before testing of each DBMS the computer was rebooted.

Each test was repeated three times. The test results represent the average received value.

The power supply plan was set to „peak performance" and also all the power-saving functions were turned off.

For each DBMS the following tests were performed: record addition, subtree selection, selection of all the node parents, tree traversal, node code search.

Testing of node addition time was performed as follows: the time of all the nodes addition was taken into consideration. After that, the result was divided by the total number of nodes.

The selection of all the node children assumes obtaining a cursor for the node children, with reading them from the disk. 3500 queries were executed during this test.

The selection of all the parents also assumes obtaining a cursor for the result set, with reading records from the disk. Similarly, 3500 queries were executed during this test.

Tree traversal assumes reading the whole tree.

The selection of threads with reading assumes execution of 10,000 queries. After obtaining the cursor for children, all the children are read out from the disk. This test allows determination of the real DBMS performance as MongoDB, when obtaining a cursor, return only the first resulting records. The rest of the records are found in the index only when reading all the previously found.

The total size of the database tested is 24.4 GB without considering the normative documents.

Two different experiments were carried out using the DB tested. The first one includes the whole database, the second one includes only 100000 nodes (the number of queries and data sets are left unchanged). The need for doing several experiments with one data set is caused by the necessity of determining the degree of impact of the records number in the tree on its processing performance.

In addition to the database tested, an experiment was set up in which 100000 tree nodes were used as a test set a materialized path was attached to each node as properties. The performance of this experiment enables understanding the degree of impact of the record size on the tree representation method performance.

6.2 Hardware and Software

A computer with Intel® Core i5™-4590 processor running Windows 8.1 operating system was used as a computing platform. The processor is a 3.30 GHz 64 bit quad-core, with maximum throughput of 32 GB/sec. The machine has 8 gigabytes of memory. The graphics card used is an NVIDIA GeForce GTX 760. with 1152 CUDA cores. 4 GB of global memory, and supports a maximum throughput of 192.2 GB/sec.

CUDA 6.5 driver is used on the computer. As for the software. MongoDB 2.6 is used. When testing, executable DBMS files were used that were downloaded from the official site. The developed DBMS was created using MS VS Studio 2013 and optimization flag –O3.

6.3 Test Results

Table 2 represents the test results. All the values are average results for a set of queries.

The results in Table 2 confirm a possibility of the justified application of the developed method of hierarchy representation in databases. Such a result is noticed with large data volumes and a great number of records. If a record is small, this method performance becomes inferior to a common tree index (B+tree for MongoDB)

Table 2: Query execution average time

Query	Represented method	MongoDB
	(average time, sec/test)	
Node addition	768,44	14304,02
Subtree search	162,01	283,25
Parent search	126,08	163,00
Reading the whole tree	490,74	8 436,37
Obtaining arbitrary nodes	159,90	8 880,27

Table 3: Performance comparison of tests with different data sets

Database / test	**Node addition**	**Subtree search**	**Parent search**	**Node search**
2,5 million records, 10 KB/record				
	queries/sec			
MongoDB	174,78	12,36	21,47	1,13
Developed method	3253,34	21,60	27,76	62,54
100 000 records, 10 Kb/record				
	queries/sec			
MongoDB	162,70	23,33	93,97	5,77
Developed method	3245,34	21,09	30,02	57,22
100 000 records, 20 b/record				
	queries/sec			
MongoDB	8673,03	6603,77	350000	2554,74
Developed method	62500	30,04	108,97	1988,64

(Table 3). The difference is caused by the restriction to the maximum number of queries per second on the part of hardware. The minimum time of configuration and startup of the empty kernel (for the card tested) in the synchronous mode is approximately 0.10 msec: consequently. the graphics card cannot execute more than 10–15 thousand kernels per second.

Figure 3 shows the relation of the query processing time.

a) Children search b) Parent search

Figure 3: Time distribution when executing queries

Reading data from the disk takes most of the time for the both queries. The execution of kernels requires a little time. On average, the kernel execution takes 4.4 msec that provides the maximum 227 queries per second without considering CPU operation time (for 10 KB/record). When testing DB with a small record size (20 byte/record), the kernel execution time is almost the same as the testing time for large records. This results in the method being predisposed to building hierarchies based on finding the remainder, dealing with a large amount of data, the CPU processing of which is a long-running operation.

7 Future Improvements

7.1 Threaded Execution

So far the multithreaded software product has not been developed (we mean CPU multithreading, not GPU multithreading). Multithreaded software is an important aspect of future developments. If a query is processed using multithreading, the maximum speedup will be achieved when executing data modification queries (record addition, transfer). The application of NoSQL-style nonlocking record functions will enable performing database modification operations in the asynchronous mode, thus improving the performance.

7.2 Multiple GPU Computing Support

Currently one graphic card performs all calculations. Addition of several graphic cards will allow increasing the terminal capacity of data bank or the query execution speed through processing chunks on each graphic card when storing the same dcitci on different graphic cards.

When using several graphic cards, the low-speed CPU-GPU bus transfer appears to be a restriction. Despite the fact that PCI 3.0 xl6 bus has a theoretic duel-sided exchange rate 128 GB/sec and 8 GT/s (Giga Transaction/s), it is restricted by the core memory performance. In actual practice the bus reproduction speed is approximately 39 GB/sec (on tested PC).

What is more, many motherboards especially in the moderate price range have only one PCI slot per 16 data transmission lines whereas the second one and the succeeding have x8 and x4 in general.

Even though using less lines decreases multi-card configuration performance by 5–15% on average, and using multi-card configurations in most cases cannot reach its full potential due to the RAM restricted speed, the benefit from using such configurations exceeds the expenses.

7.3 Hardware Restrictions

Currently an important GPU restriction is lack of thread synchronization resources on the whole graphic card. Thread synchronization is possible only inside blocks. Block synchronization on GPU is impossible. User synchronization methods are based on using flags. Hardware synchronization of the blocks is more important as the performance increases compared to the software synchronization.

Another GPU nuance is SIMT architecture. SIMT architecture allows application of one and the same command to different data. The drawback of the architecture is that all the threads go through each branch of the code when branching is used even if the thread does not execute any instructions. In other words, some threads are in operation, the others are idle.

It is important to note a small number of registers available for each thread. This problem becomes acute with a large number of threads and a complex code(large data type). With 100,000 threads, 16 byte size data type and 512 threads per block, each block uses 60 registers in 63 theoretically possible (the data taken from the CUD A profiler). This results in a less number of threads being executed in parallel. For example, when the number of registers used in a thread dropped to 13, the processing speed of the same amount of data increased by almost 10 times.

When running the experiment, the reduction in the number of threads per block did not result in the expected performance improvement.

7.4 Key Compression

The tree processing method described in the article assumes using plural multiplying for defining key values. The keys obtained by primes multiplication are exposed to a rapid increase resulting in large-size numbers. With 100 thousand records in a tree it is necessary to use keys whose size is more than 128 bit.

The most frequent arithmetic operations performed with keys in a tree are division, multiplication and taking the division remainder. Hardware support of such operations is impossible due to CPU and GPU architecture characteristics. To perform arithmetic operations with such keys, it is important to use different algorithms that are less efficient than hardware realization of such operations.

Using nonpositional notations will allow the size reduction of the used numbers by storing a number as a small-size remainder. When performing non-modular operations, the lack of need for considering the dependencies between the remainders opens up an opportunity for the realization of efficient parallelism. This together with the advanced features of Kepler architecture (bit control within warp), will allow the efficient realization of bit mapping and defining zero remainder for nonpositional notations.

8 Conclusion

The paper represents a method of storing tree-like structures in databases oriented towards multithread processing with regard to CUDA parallelization.

This project demonstrates using GPU for hierarchical data processing.

The result of the article as a follows: a new method of representing trees in databases is developed, and the realization of a hierarchical database prototype with GPU data processing.

On average, the implemented software runs by 19 times faster compared to the materialized path processing in DBMS MongoDB on CPU. Despite the query result variations, the minimum speedup was 1.29 times.

References

[AG08] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008.

[BELP] U. Brandes, M. Eiglsperger, J. Lerner, and C. Pich. Graph markup language (graphml).

[BMCN09] Ranieri Baraglia, Via G Moruzzi, Gabriele Capannini, and Franco Maria Nardini. Sorting using BItonic netwoRk wIth CUDA. *LSDS-IR Workshop*, (July), 2009.

[BS10] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda: Extended results. Technical report, University of Virginia Department of Computer Science, 2010.

[Cel] Joe Celko. Trees in SQL.

[Cel04] Joe Celko. Hierarchical SQL, 2004.

[Cel12] J. Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties.* Joe Celko's Trees and Hierarchies in SQL for Smarties. Elsevier/Morgan Kaufmann, 2012.

[Gai99] A. T. Gainov. *Number Theory. Part I. Resource book.* Novosibirsk State University. Faculty of Mechanics and Mathematics., 1999.

[GBE13] Marcel Grandpierre, Georg Buss, and Ralf Esser. In-Memory Computing technology - The holy grail of analytics? *Deloitte & Touche GmbH Wirtschaftsprьfungsgesellschaf*, (07), 2013.

[HKOO11] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating CUDA graph algorithms at maximum warp. *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming - PPoPP '11*, page 267, 2011.

[HN07] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[kla] KLADR - the All-Russian Classifier of Addresses.

[Kol09] M.A. Kolosovskiy. Data structure for representing a graph?: combination of linked list and hash table. *CoRR*, 2009.

[Kup86] Gabriel Mark Kuper. *The Logical Data Model: A New Approach to Database Logic.* PhD thesis, Stanford, CA, USA, 1986. UMI order no. GAX86-08173.

[KV93] G. M. Kuper and M. Y. Vardi. The logical data model. *ACM Transactions on Database Systems*, 18(3):379–413, 1993.

[MS11] Anders Møller and Michael Schwartzbach. Xml graphs in program analysis. *Sci. Comput. Program.*, 76(6):492–515, 2011.

[MT11] A. Malikov and A. Turyev. Nested intervals tree encoding with system of residual classes. *IJCA Special Issue on Electronics, Information and Communication Engineering*, ICEICE(2):19–21, 2011.

[PSHL10] Hagen Peters, Ole Schulz-Hildebrandt, and Norbert Luttenberger. Fast in-place sorting with cuda based on bitonic sort bitonic sort. In *Parallel Processing and Applied Mathematics*, pages 403–410. 2010.

[PT11] Jonathan Palacios and Josh Triska. A Comparison of Modern GPU and CPU Architectures: And the Common Convergence of Both. pages 1–20, 2011.

[Tro03] Vadim Tropashko. Trees in SQL: Nested Sets and Materialized Path. 2003.

[Tro05] Vadim Tropashko. Nested intervals tree encoding in sql. *SIGMOD Record*, 34(2), June 2005.

[vor14] *DBMS Index for Hierarchical Data Using Nested Intervals and Residue Classes*. Young Sci. Int. Work. Trends Inf. Process., 2014.

[WO10] Yangzihao Wang and John Owens. Large-scale graph processing algorithms on the gpu. Technical Report January 2011, UC Davis, 2010.