International Journal of Computer Applications (0975 – 8887)
Volume 104 – No.7, October 2014

An Efficient and Scalable RDF Indexing Strategy based on B-Hashed-Bitmap Algorithm using CUDA

Sharmi Sankar[a], Munesh Singh[a], Awny Sayed[*], Jihad Alkhalaf Bani-Younis[a]

[a]College of Applied Sciences, Ibri, Postal Code 516, Sultanate of Oman,
* Faculty of Science, Minia University, Egypt

ABSTRACT
Indexing enormous databases such as RDF has been a focus of intense research. As is well understood, indexing plays a pivotal role in speeding up data retrieval operations and query performance. Besides expediting search, an index can motivate new data-store schemes and technologies that can possibly revolutionize large data-analytics engine design, more often relevant to semantic web. Due to the proliferation of internet and the ease of creating and generating data on the fly - handling, storing and the subsequent semantic processing has proven to be a major bottleneck for the RDF data community. Handling data of such scale and magnitude requires a parallel approach as provided by the GPUs (Graphical processing units). In this paper, a new efficient and scalable index is proposed that uses a combination of B+ trees, hashing and sparse matrices. These data structures have an edge over others in terms of their implementation as a parallel algorithm using the CUDA (Compute Unified Device Architecture) framework meant to program massively parallel GPU multicores. So far, RDF data has been mostly implemented either as a RDBMS or as a non-native data-store, in both cases the sequential indexing strategy fails miserably with the scaling of the data-store. Parallel implementation of indices provides a suitable option for dealing with scalable and dynamically generated data over distributed networks. The crucial sparse matrix part of the proposed index is benchmarked against different CUDA memory implementations to derive optimal matrix processing options. The sparse matrix search is profiled using cudamemchk and visual profiler for identifying bottlenecks and inconsistencies in thread execution called thread divergence. Benchmarking the data provides promising results for a B+ tree based index coupled with hashing and sparse matrix implementations.

Keywords
RDF, B+ tree, hashmap, sparse matrix, CUDA, GPU.

1. INTRODUCTION
There are several initiatives to improve the situation and reduce the drawbacks of the current web. One of them is a Semantic Web, which is coined by the W3C founder Tim Berners-Lee in a Scientific American article that is describing the future of the Web [1]. The Semantic Web gives better structure and computer-understandable meaning that offers a common framework for sharing data across applications, enterprise and communities.

The Semantic Web initiates to define information on the web in a precise machine comprehensible format. The web in its existing incarnation provides information in human understandable formats, but the meaning of this information and its relation to other pieces of information elsewhere on the web are not well-defined. Semantic Web data uses common schemas to describe data from disparate sources. Machines capable of reading this data could comprehend the data, for example inferences could be made about the data based on information from other datasets (BernersLee,2001).Semantic Web information is often stored in RDF in the form of triples (subject, property, object). A combination of many RDF triples forms an RDF graph. RDF is a metadata model for web resources, and is the reason it is referred as a Resource Description Framework (RDF).

A number of storage implementations and schemes have been proposed that use databases to cache RDF triples. Some implementations maintain RDF-specific information in the application layer, and some store the RDF schema at the database level. When stored at the application level, the application stays database-independent, and compromises in terms of performance and scalability is revealed. When the RDF schema is implemented at the database level, RDF structure can be exploited to obtain efficiency using existing database models. These reviews focus on existing state of the art of RDF database storage schemes. The simplest way to store RDF data is in a triple store, essentially one large table with three columns for subject, predicate, and object. Variations on the triple store have shown improvements in efficiency and have reduced the number of self joins needed when issuing complex queries.

RDF storage has witnessed numerous research initiatives in varied domains. Despite of the best efforts, a scalable, efficient and fast index has eluded researcher’s grasp. A typical RDF data-store consists of billions of triples (a triple comprises of subject, predicate and object) with extensive and wide range of self- dependencies among the subject and the object field values. The outcome of which results in recursive self-joins with an added cost to the query optimizer [1]. Besides self-joins, unions and null values it also generates countless performance related issues. There exists broadly two ways to deal with these disputes, either to re-design the RDF data-store from scratch using a new setup for representing the triples along with the modified query engine design or to explore faster and more efficient indexing strategies that provide impeccable query processing time irrespective of scalability.

RDF repositories usually create indexes on one or more components of an RDF triple. Since the volume of data (RDF-triples) is quite large, a typical index should at least be logarithmic in its time complexity. Many index designs have been suggested with most of them relying on B+ tree and hashing. In one of the suggested design [10], a forest of B+ tree is created that uses different combinations of S, P and O. The main drawback of this strategy lies in the complex queries resulting in slow data retrieval. Hexastore sex tuple indexing [18] is another architectural design that suggests six-fold indexing based on different combinations of S, P and O. The search through the data-store takes place in constant time but dynamic updates are very slow. This architecture is quite suitable for static RDF tables but does not scale well with dynamic RDF data.

In general, multiple indexing [2, 3, 4] has been a very widely researched option particularly for designing efficient hashes. Multiple indexes mask and moderate intricate complexities that arise out of a single-indexing method. The main issue that works against a single index is that of transitive closure or subject-object pair recursion. Besides, a single subject may map to multiple objects. These two observations are rectified when multiples indexes are considered. An instance in point is that multiple maps of a B+ tree are channelized to different positions on a hash map and parallelized, using multiple threads which is applied and followed in this paper.

This paper focuses on the second aspect related to a new index design. Indexing is an essential part of the IR systems for two reasons.

- It optimizes the query performance and improves the response times.
- A number of processing tasks are carried out during the indexing phase similar to the query processing phase, which further improves the performance.

The data-stores are getting bigger each passing day, a new distributed approach using Lucene that uses Map Reduce was suggested [1]. The main advantage of this approach is the leveraging of the distributed load across different processor-cores. This highly speeds up the indexing process but the issue of self-joins remains.

2. RELATED WORK

In this section, we discuss about a recent survey [Yongming Luo, 2012] that distinguishes between three different perspectives on RDF storage:

• The **relational perspective** considers RDF data to be relational data, and leverages existing storage and indexing techniques originally developed for relational database Systems.

• The **entity perspective** treats resources in an RDF dataset as “entities” associated with a number of (attribute, value) pairs.

• The **graph-based perspective** views an RDF dataset as a classical graph, where the subject and object parts of each triple correspond to nodes and the predicate parts correspond to the directed, labelled edges between them. It aims to support graph navigation and answering of graph-theoretic queries. This perspective of research focuses on semi-structured- and graph databases.

The discussion is further extended on the state of storing RDF data in Triple Table, with a comprehensive look at the property table approach and vertically partitioned approach. The RDF data stores in a single triple table which consists of three columns, subject, predicate, and object respectively [5]. The performance issue of this approach is all the triples stored in a single RDF table requires expensive and complex self joins over the triples table as pointed out in [11, 12, and 13]. Thus, as queries become more complex the execution time increases. In addition, it is exceeding the memory size and congestion of the RDF data sets. Nevertheless, this approach has been implemented by systems like Oracle [14], 3store [5], Redland [20], RDFStore [21] and rdfDB [22]. The research community later introduces an alternative solution for improving the triples table and minimize the number of self-joins issues. An alternative methodology to the previous is the property table approach [6].

The property approach deformalized the RDF table that stored in a flattened format. Furthermore, it is classified into two types which are property class table and clustered property table. The clustered table contains cluster of properties that tend to be defined together. The property class table exploits the type property of subjects to cluster similar sets if subject together in the same table [7]. The most important advantage for representing the property tables is that they can reduce subject-subject self joins of the triples table. However, this approach may not fit well the RDF data because of unstructured data and missing properties. In an interpretation, not all properties will be defined for all subjects and that perhaps will lead to many NULL values which increases the overhead in the memory space. Another problem with the property table is the abundance of multi-valued attributes found in RDF data which cause further complexity and with combined data from several tables the issue of improving the performance of self-joins queries maybe become poor. In summary, property tables are rarely used due to their complexity and inability to handle multi-valued attributes. However, this approach has been used by tools like Sesame [23], Jena2 [12], RDFSuite [24] and 4store [21].

Abadi et al. [7] proposed vertically partitioned approach is an alternative solution to the property table to speed up query processing and minimize its limitations by deploying a fully Decomposed Storage Model (DSM) [10]. In this approach, an RDF table is re-written into n two-column tables, where n is the number of unique properties. Furthermore, the first column is subject and the second is an object. One of the primary benefits of vertical partitioning is the support for rapid subject-subject joins. This feature is achieved by sorting the tables via subject as mentioned above, each binary table has subject and object columns. The tables being sorted by subject, one has a way to use fast merge joins to reconstruct information about multiple properties for subsets of subjects. The experiments in these papers [8] and [9] reveals that the vertical partitioning approach also performs poorly for querying RDF data and slow insertion, because of the multi property tables. The vertical partitioning approach supports multi valued attributes and heterogeneous records. In addition, it is eliminating the subjects that do not define a particular property. Obviously, it reduces the NULLs value through that elimination.

3. STORING AND INDEXING UNDER RELATIONAL PERSPECTIVE

The RDF data in relational databases are stored as triples of a RDF graph on a single table over a relational schema (Subject, Predicate, and Object). An important issue in this approach is that due to the large size of the RDF graphs and the potential large number of self-joins required to answer queries, at most care has been be taken to devise an efficient physical layout with suitable indexes to support query answering.

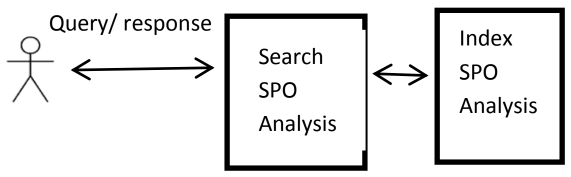


Fig. 1. Basic Search Architecture

To search the information from on a RDF table, the query overcomes the necessity of building text analysis that avoids tokenization, word elimination, Normalization, Stemming and Lemmatization unlike a search deployed for text analysis. Fig 1 exposes the search engine's component support of two major functions, the index process build data structures that enable searching and the query process uses those data structures to satisfy the information need by retrieving it for the user.

3.1. Proposal to address scalability issue:

To address the scalability issue, we propose an unclustered-index on B+ trees comparing the impact of the following permutations on S, P, O. Neumann and Weikum [6] take this approach further in their RDF-3X engine by adding to the 6 indexes above, so-called projection indexes for each strict subset of {subject; predicate; object}, again in every order. This adds an additional 9 indexes: s, p, o, sp, ps, so, os, op, and po. Instead of storing triples, the projection indexes conceptually map search keys to the number of triples that satisfy the search key. For example, the projection index on subject alone maps each subject s to the cardinality of the multi-set

{| (p; o) | (s; p; o) occurs in the RDF Graph |}

The projection indexes are used to answer aggregate queries; to avoid computing some intermediate join results and to provide statistics that can be used by the RDF-3X cost-based query optimizer. RDF-3X also includes advanced algorithms for estimating join cardinalities to facilitate query processing. The unclustered-indexes entails combined indexing on subject and Object (SO) as revealed in Fig 4.

In addition, Erling and Mikhailov [12] reports that in practical RDF graphs, many triples share the same predicate and object. To take advantage of this property, Virtuoso builds bitmap indexes for each ops prefix by default, storing the various subjects. McGlothlin et al. built on this idea and used bitmap indexes instead of B-Trees as the main storage structure [7].

In our paper, based on the study made by Neumann and Weikum [6] and Erling and Mikhailov [12] we have proposed the idea of using the unclustered index on subject and object as they can be commonly used and can share different predicates. Conversely, this unclustered permutation of SO index on B+ tress seems to materialize the need for easy storage and retrieval.

4. PROPOSED TECHNIQUE: (B+-hashed-bitmap index)

Empirical results (Fernandez et al., 2013) [13] show that the average size of these lists of predicates for subjects and objects is, at most, one order of magnitude less than the number of total predicates used in real-world datasets. This fact not only ensures a great improvement for queries with unbounded predicate, but also implies a limited additional space for SO indexes.

- Input: RDF triple (Subject, Predicate, Object) as shown in fig. 2.
- Create an empty predicate-matrix with SO (Identifiers) plotted on the rows and predicates listed as columns as shown in fig. 3.The sparse matrix is optimized using GPU to improve the efficiency further.
- Create a un-clustered-index on SO over a B+ tree as shown in fig. 4.
- Deploy a B+ tree index that leads to leafs (Predicates) as shown in fig. 4.
- The hashed predicate updates the predicate-matrix table.

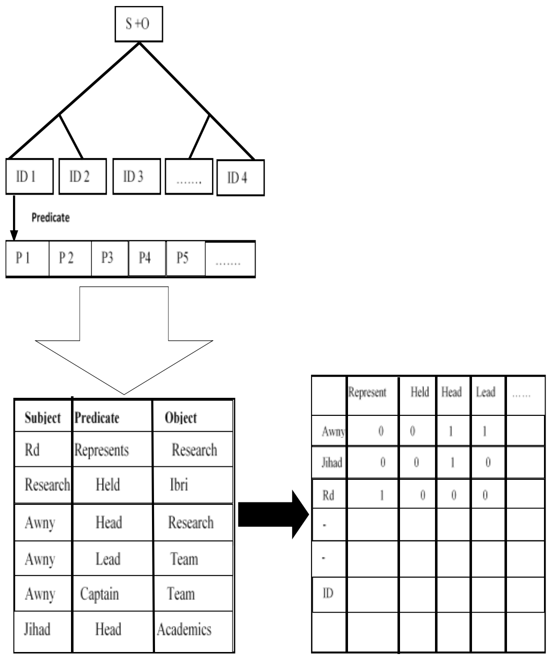
Subject	**Property**	**Object**
ID1	Type	FullProfessor
ID1	teacherOf	'Ali'
ID1	bachelorFrom	'MIT'
ID1	mastersFrom	'Cambridge'
ID1	phdFrom	'Yale'
ID2	Type	AssocProfessor
ID2	worksFor	'MIT'
ID2	teacherOf	'DataBases'
ID2	bachelorFrom	'Yale'
ID2	phdFrom	'Stanford'
ID3	Type	GradStudent
ID3	Advisor	ID2

Fig 2: RDF triple table

	P1	P2	P	P4	P5	…
ID1						
ID2						
…						
…						
IDn						

Fig. 3. Predicate Matrix

This describes how the B+ tree structure can be applied to the predicate matrix-bitmap of RDF storage. Our approach is called B+ hashed predicate matrix-bitmap indexing. A combined index on subject and object is preferred to retrieve the appropriate predicates associated with the concerned subject and object that are common/uncommon as shown in fig .4. As the query is based on the Subject : "Awny" and Object: "team", it branches via the B+ tree pointing to the leaf nodes as identifiers that list the predicates associated with it. The predicates are then mapped to the corresponding identifier which is updated using 1's and 0's in predicate matrix table. This matrix helps to speedup filtering and updates the mapped results. The complexity involved with this architecture is log N + 2N. This indexing strategy ensures a logarithmic complexity which when compared with quadratic and linear complexity seems to be more promising.



Subject	Predicate	Object
Rd	Represents	Research
Research	Held	Ibri
Awny	Head	Research
Awny	Lead	Team
Awny	Captain	Team
Jihad	Head	Academics

	Represent	Held	Head	Lead	......
Awny	0	0	1	1	
Jihad	0	0	1	0	
Rd	1	0	0	0	
-					
-					
ID					

Fig 4: B+ hash bitmap index functional block diagram (Unclustered-index)

5. IMPLEMENTATION OF B+-HASHED BITMAP INDEX USING CUDA

5.1 CUDA Background

GPU acceleration of compute-intensive applications has emerged a new research frontier with phenomenal success-rates. Such applications are characterized by large data-sets being processed by singular functional units (FUs) often described as SIMD (Single Instruction Multiple Data) computing.

Moreover, with the proliferation of internet and its easy access on myriad devices, has resulted in huge amount of data generation. Initially, such data was considered disconnected and not related. But with the advent of semantic web, data has been found to be highly co-related and relevant. Organizing such huge amount of data and subsequently processing requires parallel framework that is both distributed and scalable. Graphical processing units (GPUs) are being actively probed in the domain of Big Data analysis, machine learning, and augmented reality since such applications are characterized by massive data spanned and generated over distributed network. GPUs provide a parallel programming framework using CUDA (Compute Unified Device Architecture) that can be utilized to efficiently collate and make inferences on these massive data-sets. Further, GPU multicores are available at commodity rates thus providing an option for cheap and low-power alternatives.

The exponential growth of semantic web and the resultant generation of large-scale RDF (Resource Description Framework) triples pose new challenges in the domain of RDF-storage and retrieval. RDF data consist of triples <Subject, Predicate, and Object> which need to be efficiently indexed. Following are some of the many challenges related to efficient indexing of RDF triples:

- As RDF-triples extensively contain recursive redundancies, self-joins so formed are inefficient.
- Self-joins also lead to large scale null values.

5.1.1 Hashing Options

Indexing can be implemented by mapping all the <key, value> pairs on to an array of the size of the search universe. This can result in instantaneous search if the extraordinary size of the array is overlooked. Besides, the search array being of extraordinary length, it may also be sparse. This may lead to wastage of space and also will have bearing on the search complexity (considering the array size to that of the universe, it is, but obvious that the array cannot be entirely brought into the RAM.). In order to avoid this colossal waste of space, hash functions allows search arrays to be of much shorter lengths (often of fixed size) but at the cost of collisions (refer Fig 2). Collision resolution has been an active research subject for many years now. Two popular resolution strategies named “open addressing” and “chaining” are employed but they fail miserably when the number of collisions is many. Open addressing refers to a collision mitigation technique that involves inserting the generated <key, value> pairs to the next free space available in the search array. The main drawback of this technique relates to the provision of in-between free spaces in the search array (implemented using the judicial selection of the hash function).

5.1.2 Drawbacks of GPU application of Hash Table

1. Parallel insertion into the hash table cannot be achieved as the data structure has to be locked before insertion. This can be done by atomic instructions provided in CUDA. This is done to avoid a same memory location being accessed by multiple CUDA threads thus leading to race conditions [14].

2. Memory accesses of different (often scattered) memory locations cannot be optimized as non-contiguous memory is accessed. As a result, memory coalescing techniques that use the concept of alignment (Hardware dependent) cannot be applied. Coalesced memory access of a warp of threads (32 threads in CUDA are considered as one warp) results in one or two memory transactions of a memory bank. This increases the memory access performance.

5.1.3 Proposed modification to Hashing with relevance to sparse matrix

The previous two sub-sections provide the limitations for implementing hashing in a parallel GPU environment coupled with the data to be hashed residing on a sparse matrix (Fig. 4). Since the thread divergence is a major issue in GPUs, hashing typically provides random distribution of data which in the conventional sequential sense is quite an equitable distribution. But in the case of GPUs this is inefficient and leads to performance degradation.

The perfect spatial hashing [25] provides a solution to the random hashing by providing two levels of hashing. This is represented by the following equation:

$$h(p) = h_0(p) + \emptyset[h_1(p)]$$

Where, h0 and h1 refers to two imperfect hash functions coupled to one another using an offset table ϕ. The main task of the offset table is to force the h0 function from an imperfect hash to a perfect hash using the second level of indirection thus providing clustered hash mappings amenable to a typical CUDA kernel function. This process leads to spatial coherence that is a very common technique used when computing graphical applications in CUDA. So, in effect the combination of two imperfect hashes is coerced into a hash function that provides spatial locality of reference. As a result our RDF sparse matrix is compacted into a spatially coherent hash table thus providing ample opportunities for CUDA warps (group of threads that executes in unison simultaneously) to execute without any chance of divergence. This factor contributes to the higher efficiency of the proposed B+ Hashed indexes. The whole sequence of the spatial hash framework is depicted in Fig 5.

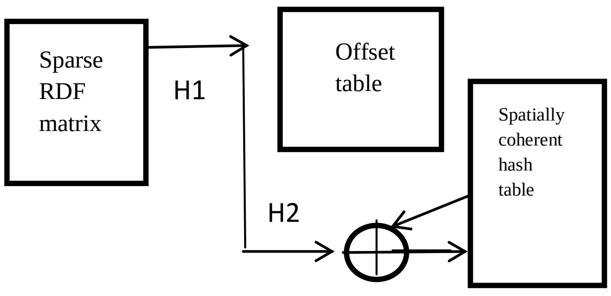


Fig 5. Perfect Spatial Hashing

The above technique provides spatial coherency at the expense of additional memory, which is not a big consideration now in CUDA as the large matrices are tiled and brought one by one as a single tile into the limited device shared memory. Besides other CUDA memory optimizations such as pinned or page-locked memory accesses may be attempted if the memory access is read-only which is precisely the case in RDF sparse matrix.

5.1.4 Boolean (logical) Sparse Matrix implementation using CUDA

A typical sparse matrix consists of large number of zero values with very few dispersed non-zero ones. The opposite to that of the sparse matrix is the dense-matrix which comprises more non-zero elements than the zero ones. Sparse matrices find usages in large and varied domains of computational science research and are formed primarily from the solution of partial differential equations. Sparse matrices also find usages in 3D-maps used by Google in solving non-linear issues. Ceres Solver [15] is a portable C++ library used by Google maps to reconstruct 3D-maps using 2-D photographs that uses non-linear sparse arrays.

It is often noticed that some operations belonging to dense-matrices may scale very poorly on sparse matrix. Hence different sets of operations that can efficiently run on sparse matrices are needed. Most of the software languages have support for sparse matrices in the form of libraries and APIs. Some of them are cuSparse for CUDA C/C++, SciPy 2-D sparse matrix package for numeric data in Python and many others.

In order to provide efficient sparse matrix operations, there is a need for an efficient sparse matrix representation. Many representations (data structures) have been suggested, but since GPUs are being used to expedite search, cuSparse CUDA library is used as it specifies different matrix representation formats. These formats are listed as under:

- Dense format: This format is a typical row-major order which is used for storing dense-matrices conventionally. (Nvidia CUDA library cuBLAS uses it as default).
- Coordinate format (COO): This format has been specifically designed for sparse matrices taking into consideration the non-zero values and ignoring the zero ones. The matrix is stored in a row-major order.
- Compressed Sparse Row Format (CSR): CSR format is very similar to COO except for the row indices which are compressed and aggregated.
- Compressed Sparse Column Format (CSC): CSC stipulates matrices to be stored in a column-major order and compresses columns instead of rows as in the case of the above mentioned CSR format.

For the sake of brevity CSR format is proposed to be used to represent sparse matrices derived during indexing. Since, the Boolean sparse matrix consists of only 0’s and 1’s, each element of the matrix can be represented using a single bit. Thus a sparse matrix having 10 million elements can easily be represented in a dense format with 10 million bits roughly equivalent to a few megabytes. If a compression format such as CSR is applied, the size for storing a matrix may dwindle further. Since the CSR format can take any values for non-zero, the Boolean data further reduces the space requirements by restricting matrix element values to a single bit size representing either one of the value – 0 or 1. The next section gives a detailed preview of the modified CSR format.

5.2 Compressed Sparse Row – (Boolean) Format (CSRB)

So the Boolean CSR matrix as defined in CUSPARSE library [16] is modified for Boolean values instead of floats. The modified table represents the CSRB matrix as under:

nnz	(integer)	The number of ones in the matrix
csrbRowPtrA	(pointer)	Points to the integer array of length m+1. The first m+1 elements of this array consist of all 1’s indices A. The last (m+1)th value consist of nnz+csrbRowPtrA(0).
csrbColIndA	(pointer)	Points to the integer array of length nnz containing the column indices.

It may be noted that the modified CSRB Format does not need an extra vector csrValA [16] that list all non-zero value elements belonging to a matrix A. Only two linear vectors (csrbRowPtrA and csrbColIndA) are sufficient to depict matrix A.

Consider a Boolean sparse matrix (A) of size 4×5, where the number of rows is 4 and the number of columns is 5.

$$A=\begin{bmatrix}1 & 1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0\\ 0 & 1 & 1 & 0 & 0\\ 0 & 0 & 1 & 0 & 1\end{bmatrix} \quad (1)$$

Storage in CSRB format using zero-based indexing, as depicted below:

csrbRowPtrA=[0,2,3,5,nnz+csrbRowPtrA(0)] = [0,2,3,5,7] (2)

csrbColIndA=[0,1,2,1,2,2,4]

It can be noticed that a single matrix A of size 4×5 consisting of 20 elements is reduced to vectors consisting in sum a total of 12 elements, a reduction ratio of 60%. Hence, the Boolean Sparse Matrix Reduction Ration (BSMR) can be derived as under:

BSMR(%)=(|csrbRowPtrA|+|csrbColIndA|)/(M×N)×100 (3)

BSMR Ratio besides providing sparse matrix space complexity also derives the following implications:

The sparse matrix may turn into a dense matrix if the number of 1's increase suddenly. This may create a belief that the sparse matrix representation as CSRB format may turn out to be inefficient. After scanning the BSMR Ratio (3) it can be pointed out that the increase in the number of elements will have a corresponding increase in csrbColIndA vector but will not affect csrbRowPtrA vector since it consist of the initial indices values for each column that remain more or less unchanged. Thus we can derive the following analogy:

$$|A| \propto |csrbColIndA| \quad (4)$$

5.3 CUDA multithreaded model for implementation of Boolean Sparse matrix (CSRB Format)

CUDA supports parallel matrix operations on each individual element asynchronously. Since CUDA derives its origin from the days of graphics pixel programming and rendering, it provides seamless multithreaded options of processing matrices with ease. CUDA threads are organized in a hierarchy of Grids and Blocks. In the absence of any data dependencies and memory bank conflicts and proper memory coalescing, a warp comprising of 32 threads of a block is capable of simultaneous execution on each SM (Streaming processor).

The following thread model is proposed for thread allocation:

- A sparse matrix comprising millions of elements is divided into equal sized tiles of 32×32 elements.
- Each tile is processed simultaneously using a cuSparse library using shared memory. Shared memory is a fast but small cache memory (equivalent to that of L1 cache). Each block has its own shared memory cache. Since the size of shared memory is small (32 KB or 48 KB), the predicate matrix need to be tiled.

5.3.1 Benchmarking Boolean Sparse matrix against different memory factors

Efficient tiling methods using shared memories that take into account memory bank conflicts are experimented with the following results. For the sake of providing increased complexity in terms processing, matrix copy and matrix transpose operation is considered as a benchmark.

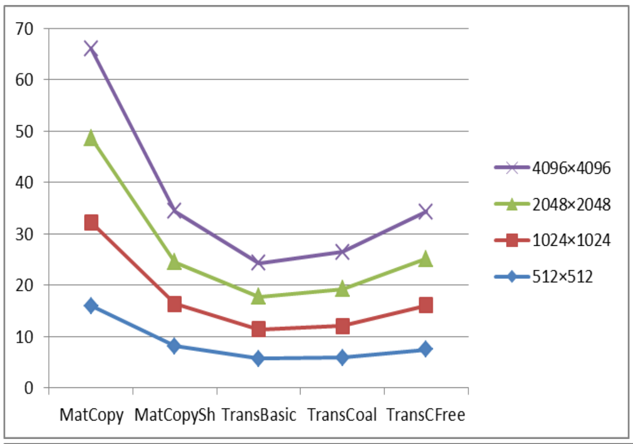
Legends for Table 1
MatCopy – Matrix Copy
MatCopySh – Matrix Copy Shared
TransBasic – Transpose Basic
TransCoal – Transpose Coalesced
TransCFree – Transpose Conflict Free

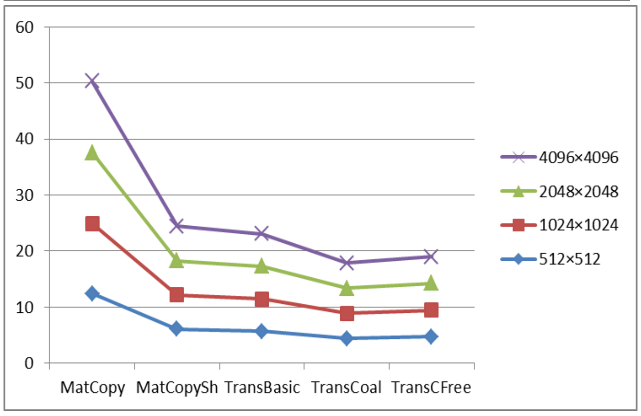
Table 1 simulation data obtained from test runs is plotted against the matrix varying sizes as depicted in Fig. 5. It can conveniently be pointed out that despite the scaling of the matrix from 512 to 4096-square elements the effective bandwidth measured in GB per second does not show is decrease. The bandwidth continues to improve with the scaling which is a desired property for RDF type voluminous databases. Considering various types of memory accesses in the experimentation it can be factually stated that the effective bandwidth maintains a stable growth proportional to the growth in the size of the matrices. Copy matrix experiment is taken as the most effective benchmark as copy operation is among the fastest. Thus, the selection of predicate matrix for storing Boolean values against the <subject, object> map is convincing and implementable.

Table 1: Bandwidth comparisons on different matrix memory accesses

Matrix Size	Mat Copy	MatCopySh	Trans Basic	Trans Coal	Trans CFree	
512×512	15.96	8.13	5.67	5.89	7.47	Band width NVS5400M
1024×1024	16.30	8.19	5.76	6.13	8.63	
2048×2048	16.38	8.21	6.35	7.20	9.08	
4096×4096	17.40	9.87	6.56	7.21	9.08	
512×512	12.46	6.07	5.69	4.41	4.73	Band width GeForce GT 540M
1024×1024	12.46	6.10	5.76	4.47	4.75	
2048×2048	12.68	6.12	5.84	4.49	4.76	
4096×4096	12.69	6.12	5.81	4.49	4.76	

Device: NVS5400M





Device: GeForce GT 540M

Fig. 6: Plot against different matrix-memory operations vs Bandwidth (GB/seconds)

5.3.2 Optimization issues on CSRB Matrix

- Any occurrences of rows and columns that have entire rows or columns as zeroes can be dealt with by deleting the particular row/column. Since initial sparse matrix will have at least one non-zero (1's) entry in each of its row but subsequent update of the predicate matrix may result in some rows or columns to be entirely zero. In such cases, the row/column may be deleted.

- The standard CSR matrix format (refer cusparse documentation) has been modified to represent a Boolean matrix. In this process a vector that stores all non-zero values may be ignored as all non-zero values happen to be 1. As a result a space complexity of the standard CSR format is further reduced thus providing optimized space requirements.

Standard CSR format space complexity = 2nnz+n+1

Proposed CSRB format space complexity = nnz+n+1

Where ***nnz*** is the total number of non-zero elements, n is the size of the row/column (we assume square matrix) .

5.3.3 CUDA implementation of CSRB

The implementation part in CUDA C is outlined in the following steps briefly:

- The standard memory allocation and the Host to Device memory transfer operations using cudaMalloc, cudaMemcpy are omitted for the sake of brevity.

- N-Square Boolean matrices of different sizes are considered. The code can be modified with ease for matrices with different number of rows and columns. The sizes on which test runs are conducted are 512×512, 1024×1024, 2048×2048 and 4096×4096.

- The following kernel template is used for matrix processing:

```
__
global__ void transposeNoBankConflicts(float
*odata, const float *idata)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1];
  int x = blockIdx.x * TILE_DIM + threadIdx.x;
  int y = blockIdx.y * TILE_DIM + threadIdx.y;
  int width = gridDim.x * TILE_DIM;

  for (int j = 0; j < TILE_DIM; j += BLOCK_ROWS)
     tile[threadIdx.y+j][threadIdx.x] =
idata[(y+j)*width + x];
  __syncthreads();
  x = blockIdx.y * TILE_DIM + threadIdx.x;  //
transpose block offset
  y = blockIdx.x * TILE_DIM + threadIdx.y;
  for (int j = 0; j < TILE_DIM; j +=
BLOCK_ROWS)
      odata[(y+j)*width + x] =
tile[threadIdx.x][threadIdx.y + j];
}
```

6. CONCLUSION AND FUTURE WORK

The B+ tree Hashed Predicate Matrix index provide a multi-level index structure that uses a modular approach to reduce indexing complexity. This is a common pattern in modern indexing structures. The prime requirement for multi-level indexes is the large volume of RDF-triple data coupled with extensive self-join complexities due to recurrences in the RDF table. The time complexity of the proposed index structure comes out to be lower ⌊log (n+2n)⌋. The 2n cost is negligible in most cases considering multi-level reduction of the search space. Empirically the cost is 2n<<log (n), thus providing logarithmic time.

The multi-level index provides ample avenues for providing improved query designs. Multi-level indexes has significantly reduced the number of block accesses required to search for a record given its indexing field value. The reorganization of entire file is not required to the maintain performance is one another added advantage of using this strategy. The next target is to integrate the index in different RDBMS databases and ascertain the speed-up with different RDF data-sets. Further since the index runs on a GPU-enabled machine, it provides a shield against scalable issues of the data-sets. This has not been the case so far with contemporary indexes.

7. REFERENCES

[1] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web, Scientific American, 284(5), May 2001.

[2] Wolfgang Nejdl, Hadhami Dhraief, Martin Wolpers, O-Telos-RDF: A Resource Description Format with Enhanced Meta-Modeling Functionalities based on O-Telos

[3] Svihla,M. Transforming Relational Data into Ontology Based RDF Data(a doctoral thesis). June 2007.

[4] Antoniou, G. and van Harmelen, F. (2004). A Semantic Web Primer. Cambridge: The MIT Press.

[5] Speeding up on-disk RDF index lookups using B+Hash trees, Minh Khoa Nguyen, Cosmin Basca, Abraham Bernstein, IOS Press, 2012

[6] T. Neumann and G. Weikum, RDF-3X: A RISC-style engine for RDF, Proc. VLDB, 1(1), 2008

[7] Mohammed Hussain, Pankil Doshi, Latifur Khan, James McGlothlin, Murat Kantarcioglu, Bhavani Thuraisingham, Efficient Query Processing for Large RDF Graphs Using Hadoop and MapReduce, Technical Report UTDCS-41-09, Department of Computer Science, The University of Texas at Dallas, November, 2009.

[8] Hexastore: Sextuple Indexing for Semantic Web Data Management, Cathrin Weiss, Panagiotis Karras, Abraham Bernstein

[9] Large RDF Representation Framework for GPUs Case Study Key-Value Storage and Binary Triple Pattern, Chidchanok Choksuchat, Chantana Chantrapornchai, International Computer Science and Engineering Conference (ICSEC), 2013

[10] Binary RDF representation for publication and exchange (HDT), Javier D. Fernandez, Miguel A. Martinez-Prieto, Claudio Gutierrez, Axel Polleres, Mario Arias, Journal of Web Semantics: Science, Services, and Agents on the World Wide Web, Elsevier

[11] Optimizing RDF stores by coupling General-purpose Graphics Processing Units and Central Processing Units, Bassem Makni

[12] Erling and Mikhailov, RDF Support in the Virtuoso DBMS

[13] Javier D. Fernándeza, Miguel A. Martínez-Prietoa, Claudio Gutiérrezb, Axel Polleresc, Mario Ariasa, Binary RDF representation for publication and exchange (HDT), Web Semantics: Science, Services and Agents on the World Wide Web, Vol. 19, March 2013

[14] Efficient Hash Tables on the GPU, Dan Anthony Feliciano Alcantara, PhD Thesis, University of California, Davis

[15] ceres-solver - Google Code: https://code.google.com/p/ceres-solver/

[16] NVIDIA Cusparse Library, DU-06709-001_v5.5, July 2013, Nvidia Corporation.

[17] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In VLDB, pages 411–422, 2007.

[18] Hexastore: Sextuple Indexing for Semantic Web Data Management, Cathrin Weiss, Panagiotis Karras, Abraham Bernstein, 2008.

[19] Semantic Search over the Web Data-Centric Systems and Applications 2012, pp 31-60.

[20] Beckett, D., The design and implementation of the Redland RDF application framework. Computer Networks, 39(5):577-588, 2002.

[21] Lee Feigenbaum, Sean Martin, Matthew N. Roy, Benjamin Szekely and Wing C. Yung: Boca: an open-source RDF store for building Semantic Web applications, Brief Bioinform *(2007) 8 (3): 195-200.*

[22] Guha, R., rdfDB: An RDF Database, http://www.guha.com/rdfdb, 2007.

[23] Broekstra, J., Kampman, A., van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. ISWC, Springer, Sardinia, 2002.

[24] Towards distributed processing of RDF path queries, pages 207-230, Richard Vdovjak, Jeen Broekstra, Geert-Jan Houben

[25] Perfect Spatial Hashing, Sylvian Lefebvre, Hugues Hoppe, Microsoft Research.