Efficient Subgraph Matching using GPUs

Xiaojie Lin[1], Rui Zhang[1], Zeyi Wen[1], Hongzhi Wang[2], and Jianzhong Qi[1]

[1] University of Melbourne, Victoria, Australia
xiaojiel1@student.unimelb.edu.au
{rui.zhang, zeyi.wen, jianzhong.qi}@unimelb.edu.au
[2] Harbin Institute of Technology, Harbin, China
wangzh@hit.edu.cn

Abstract. The explosive growth of various social networks such as Facebook, Twitter, and Instagram has brought in new needs for efficient graph algorithms. As a basic graph operation, subgraph matching is the foundation of many of these algorithms. Consequently, the efficiency of subgraph matching is very important and determines the speed of the whole data mining process. The development of multi-core CPUs allows subgraph matching algorithms to process multiple data at a time. However, the number of threads is still limited, which has become a bottleneck of these CPU-based algorithms. A workaround is using clusters of powerful servers, which normally incurs very expensive network transfer overhead. Therefore, improving the efficiency and parallel abilities of a single computer is a better idea. One of the most effective way to achieve this is making use of GPUs. With the ability of executing thousands of threads simultaneously, GPUs have a great potential to accelerate the subgraph matching. In this paper, we leverage the power of GPUs and propose an efficient subgraph matching algorithm. The experimental results show that our algorithm outperforms the state-of-the-art algorithm by an order of magnitude.

Keywords: subgraph matching · GPU · relation join

1 Introduction

Data in many different domains including social network, web, chemistry, bioinformatics etc. can be naturally modeled as graphs. These data graphs are usually complicated and large. We need fast enough data mining methods to extract useful information from them. Subgraph matching (subgraph isomorphism), which is usually a time-consuming process, plays vital roles in many of these methods. Improving the efficiency of subgraph matching is crucial in many applications. Examples include using subgraph matching to 1) locate suspicious codes in the call graphs of a program [7], 2) find protein structures that contain α-β-barrel motif [2], [10], 3) identify a small subset of molecules for further analysis in drug design [16] and 4) help social science researchers discover the relations between a successful CEO and his/her friends [18].

Much work has been done to improve the efficiency of subgraph matching, For example, the state-of-the-art STwig algorithm [14] achieves higher efficiency by abandoning graph structure index and combining exploration and join mechanism. However, as well as other algorithms, they all subject to the limited parallel processing ability of CPUs. Although multi-core CPUs have been developed for a long time, the number of concurrent threads is still limited, which is normally up to 16 or 20. The limited parallel ability has become a bottleneck for these CPU-based algorithms. By contrast, a high-end GPU has the ability of executing several thousand or more threads simultaneously [13], which makes it suitable for many applications involving processing a large amount of data. Compared with CPUs, GPUs also have a very high memory bandwidth. To further improve the efficiency of subgraph matching, making use of GPU is a good solution.

Our GPU-based algorithm can execute thousands of threads simultaneously to process different pieces of data. We use sophisticated memory layout in global memory to take advantage of caches and coalesced memory access. We also make full use of shared memory and constant memory to achieve extra acceleration.

1.1 Contributions and Organization of the Paper

To summarise, we make the following contributions in this paper.

- We analyse the state-of-the-art algorithm and identify bottleneck.
- We propose an efficient GPU-based subgraph matching algorithm which makes use of elaborate join order and fully pipeline mechanism.
- We conduct extensive experiments to study the performance.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 analyses the STwig algorithm and also briefly explains the CUDA structure. Section 4 presents our GPU-based algorithm. Section 5 presents the experimental results. Finally, Section 6 concludes this paper.

2 Related Work

2.1 Subgraph Matching

The simplest way to solve the problem of subgraph matching is brute-force search, but the computation time is unacceptable even if the graph is small. Ullmann and Julian proposed a backtracking algorithm [15] which can reduce the size of the search space significantly. In the same spirit, Cordella et al. developed a pruning based algorithm [6]. This algorithm makes use of state space representation (SSR) of the matching process and a set of feasibility pruning rules. Also, a new way of organizing data is adopted to reduce the memory requirements. This algorithm has a better time and spatial performance. However, it is still too slow and can only handle graphs with several thousand nodes.

To deal with larger graphs, a lot of algorithms use indices to achieve better performance. Systems like RDF-3X [12] and BitMat [1] create indices on distinct edges to improve performance. SpiderMine [17] mines and indexes the top-K

largest frequent patterns from the graphs. R-Join [4] uses 2-hop reachability labels [5] as its indices, and Distance-Join [18] uses a similar reachability index.

The problem of the algorithms using complex indices is obvious: the construction time or memory space for the indices are usually prohibitive. This disadvantage becomes much more significant when the graphs are very large.

The STwig algorithm [14] solves this problem by totally abandoning graph structure indices. The algorithm uses only a very simple index for mapping text labels to graph nodes, which has linear size and linear construction time. To compensate for the lack of graph structure indices, the STwig algorithm decomposes the query graph into two-level trees by a sophisticated algorithm and adopts an exploration mechanism, which can reduce both the number of two-way join operations and the size of joins' parameters. The details of STwig algorithm are discussed in Section 3.2.

To handle very large graphs, the STwig algorithm makes use of powerful clusters. The efficiency of STwig algorithm on such platforms is satisfactory. However, improving the efficiency on a single computer is still necessary. On the one hand, it is always preferable to use a single computer to get work done in time. On the other hand, less computers in a cluster are needed if a single computer can process data faster.

2.2 Join Algorithms on GPU

He et al. proposed a series of GPU join algorithms [9] including NLJ, INLJ, NINLJ and HJ. The speedup ratios for these algorithm range from 1.9X to 7.0X.

Kaldewey et al. proposed an algorithm [11] making use of zero copy mechanism, which allow the join operation to be processed "on the fly" as the GPU reads the input tables from CPU memory directly at PCI-E speed.

However, the algorithms mentioned above do not address the problem of large intermediate data produced in a multi-way join.

3 Preliminaries

3.1 Subgraph Matching

In this paper, we consider subgraph matching on a labeled graph. Let $G = (V, E, T)$ be a graph, where V is the vertex set, E is the edge set, and $T : V \to \Sigma^*$ is a labeling function assigning a label to each vertex of G. Giving a labeled query graph $q = (V_q, E_q, T_q)$, subgraph matching will find all occurrences of q in G.

Take the query graph and data graph shown in Fig 1(a)(c) as an example, the matching results are $(A_1, B_1, C_1, D_1, E_1)$.

3.2 The STwig Algorithm

STwig algorithm [14] can be divided into steps as described below.

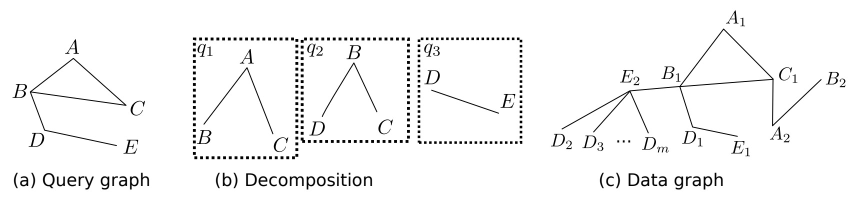


Fig. 1: A data graph and a query graph

Query Graph Decomposition is the first step. a query graph will be decomposed into a set of basic query units called STwigs. A STwig is a two-level tree, whose structure is very simple so that matching it in the data graph is very easy.

An example of the decomposition is shown in Fig 1(a)(b). The decomposition algorithm tries to minimize the number of STwigs and ensure that we can find a STwig order $\langle o_1, o_2, \cdots, o_n \rangle$ which satisfies the following statement: $\forall i >= 2$, $\exists j < i$ such that the root node of STwig o_i is the same node as a leaf of STwig o_j. $\langle q_1, q_2, q_3 \rangle$ is such an order in the example of Fig 1. The order can be used in the step of matching STwigs.

Matching STwigs in the data graph is simple. The system constructs a label index at the beginning so that nodes with a specified label can be retrieved directly. The algorithm of matching the first STwig o_1 is shown in Algorithm 1.

Algorithm 1: Match the First STwig

```
input  : Data graph $G$, STwig $o_1 = (r_1, L_1)$ where $r_1$ is the root node and $L_1$ is
         the leave label set
output: Result set $R$
Fill the set $S_1$ with all nodes with $r_1$'s label;
for each $n$ in $S_1$ do
    for each $l_i$ in $L_1$ do
        $T_{l_i} \leftarrow \{m | m \in n.\text{neighbors and } m.\text{label}{=}l_i\}$;
    $R = R \cup \{\{n\} \times T_{l_1} \times T_{l_2} \times \cdots \times T_{l_{|L_1|}}\}$;
```

The matching algorithm for other STwigs is slightly different because some pruning mechanism can be used. Let us consider Fig 1. The matching order is $\langle q_1, q_2, q_3 \rangle$. The result sets for q_1 and q_2 are $R_1 = \{(A_1, B_1, C_1), (A_2, B_2, C_1)\}$ and $R_2 = \{(B_1, C_1, D_1)\}$ respectively. When q_3 is being matched, the candidate set for its root node is not $\{D_1, D_2, \cdots, D_m\}$. R_2 ensures that only node D_1 in the data graph is possible to match node D in the query graph, so the candidate set for q_3 is $\{D_1\}$ and $R_3 = \{(D_1, E_1)\}$. This pruning strategy can reduce memory usage and the workload of the following join operation significantly.

Joining STwig Results is the final step. For the example above, the final result set is $R_1 \bowtie R_2 \bowtie R_3 = \{(A_1, B_1, C_1, D_1, E_1)\}$

3.3 General Purpose Computation on GPU

A CPU invokes a GPU by calling the *kernel* function, then the multiprocessors of the GPU will execute the kernel function simultaneously.

GPU threads are organized in a two-level hierarchy. The lower level is *block*, where multiple threads are organized in 1D, 2D or 3D ways. Each GPU thread has it own ID. For example, a thread ID comprises a row number and a column number in a 2D block. The higher level is *grid*, which contains multiple blocks.

GPUs also have a memory hierarchy. A single thread has a private memory space provided by registers and *local memory*. Registers is very fast but the amount is limited and they cannot be used to store an array indexed with non-constant quantities. Local memory resides in big but slow *device memory* and it does not have such restriction. Threads in the same block share *shared memory*, which is on-chip, small and very fast. All threads on a GPU can access to *global memory*, *constant memory* and *texture memory*, all of which reside in device memory. Constant memory and texture memory have dedicated caches to speed up the access. GPUs with *compute capability* higher than 2.0 have L1 and L2 caches for global memory, which is similar to CPUs.

4 Our Proposed GPU-based Algorithm

Not all of the 3 steps of the original STwig algorithm are perform on the GPU.

The decomposition step consumes only a very small amount of time, so there is no need to perform it on the GPU. The step of matching STwig is also performed on the CPU because: 1) The computation time is not as long as that of the join step normally; 2) The data graph may occupy too many GPU memory; 3) We can pipeline the CPU steps and the GPU steps to achieve extra speedup.

Joining STwig results is the step on which we focus. In the worst case, each single 2-way join can produce a result set whose size is the product of input relation sizes, so both the computation time and memory usage are problems.

We propose a GPU-based join algorithm in the following subsections to solve these problems. The algorithm makes use of hash table to improve the efficiency. Although the random memory access pattern of hash table will negatively affect GPU memory bandwidth, the constant search time still makes it a good choice.

4.1 Choosing a Join Tree

We denote the result sets of all STwigs by $R_1, R_2, \cdots, R_n$. A multi-way join can be represented by a join tree. Each leaf of the join tree represents the result set of a STwig. Each internal node represents a 2-way join. The left child of an internal node is the *build relation*, which means a search data structure will be built for it. The right child is the *probe relation*, whose tuples are used to matche with those of the build relation.

Two different join trees are shown in Fig 2 ($\langle p_1, p_2, \cdots, p_n \rangle$ is a permutation of $\{1, 2, \cdots, n\}$). GPU memory is limited and cannot be enlarged freely, so the major consideration of choosing join tree is memory usage.

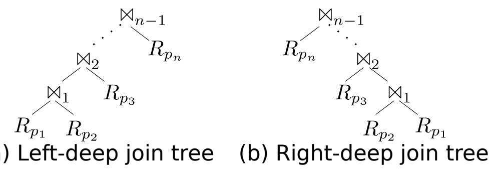


Fig. 2: Join tree

Let us consider the left-deep join tree. To save GPU memory, we only load R_{p_1} and build the corresponding hash table H_1 in the GPU at first. Then GPU threads read tuples of R_{p_2} from CPU memory directly, conduct the first 2-way join and store results in the GPU. After the join, H_1 can be released and hash table H_2 will be built for the result set of $R_{p_1} \bowtie R_{p_2}$ in the GPU. The second 2-way join $(R_{p_1} \bowtie R_{p_2}) \bowtie R_{p_3}$ can then begin. Consequently, only one hash table and one intermediate result buffer reside in GPU. However, they may still be too large. In the worst case, $|R_{p_1} \bowtie R_{p_2}|$ may be as large as $|R_{p_1} \times R_{p_2}|$. Workarounds for the memory problem like using block-based mechanism exists. However, they can slow down the process significantly. *Bushy join trees* (trees that are neither left-deep or right-deep) have the same problem.

The right-deep join tree is a better idea because a fully pipeline mechanism can be used. At the beginning, hash tables $H_2, H_3, \cdots, H_n$ will be constructed in the GPU for $R_{p_2}, R_{p_3}, \cdots, R_{p_n}$ respectively. Because of the pruning mechanism used in matching STwigs, $R_2, R_3, \cdots, R_n$ are much smaller than R_1. In many cases, $|\sum_{i=2}^{n} R_i|$ is even smaller than $|R_n|$, so we can let R_1 to be R_{p_1} to save a large amount of memory. Besides, we use a fully pipeline mechanism to minimize memory space for the intermediate data, which is described in Section 4.3.

The Order of Leaves Further improvement can be achieved by optimizing the order of leaves, i.e. optimizing the permutation of $\langle p_1, p_2, \cdots, p_n \rangle$. Better order can result in less intermediate results which means less join time. To choose the optimal order, we can predict the size of intermediate results [8].

4.2 Hash Tables on GPU

Definitions of Operators The hash table operators defined here will be used to explain our join algorithm later. H_i`.match(`x`)` returns the iterator pointing to the first record that match x. *iterator*`.record()` returns the record and *iterator*`.next()` returns the iterator pointing to the next record that match x.

Implementation Details We implement the hash table by chaining mechanism, which can be constructed in parallel easily. Data is stored together in two arrays to avoid memory overheads. The first array is *pool*. Each record of $R_{p_i} (1 < i \leq n)$ occupies $l_i + 1$ successive entities of *pool*, where l_i is the size of a R_{p_i}'s record. The *extra entity* is used to store the index of next record with the same hash value. Another array is *head*. Each entity of *head* stores the index of the first record in the chain with corresponding hash value.

In the phase of constructing hash tables, each thread will first calculate the hash value of a record storing in *pool*. Then it will link the record at the head of the chain by two assignment atomically: 1) extra entity$\leftarrow$ *head*[hash value] 2) *head*[hash value] $\leftarrow$ current record index.

We use the least $\lfloor \log |R_{p_i}| \rfloor$ significant bits of the sum of a record's entities as the hash value, so the size of array *head* is $2^{\lfloor \log |R_{p_i}| \rfloor}$, and the size of array *pool* is $|R_{p_i}|(l_i + 1)$. The total size of a hash table is $2^{\lfloor \log |R_{p_i}| \rfloor} + |R_{p_i}|(l_i + 1)$. Apparently, $2^{\lfloor \log |R_{p_i}| \rfloor} \leq |R_{p_i}|$, so

$$\frac{\text{hash table size}}{\text{data size}} = \frac{2^{\lfloor \log |R_{p_i}| \rfloor} + |R_{p_i}|(l_i + 1)}{l_i |R_{p_i}|} \leq 1 + \frac{2}{l_i}$$

Because $l_i \geq 2$, a hash table occupies at most 2 times memory.

4.3 Joining STwig Results

Our pipeline joining algorithm (see Algorithm 2) works like a depth-first search and only a very small and fixed amount of intermediate data need to be stored. The explanation of the algorithm is separated in the following subsections.

Algorithm 2: Join algorithm for each thread

```
Input: Probe relation $R_{p_1}$ and hash tables $H_2, H_3, \cdots, H_n$
1  begin
       // pos indicates this thread is processing subjoin $\bowtie_{pos}$
2      $pos \leftarrow 1$;
3      $itArray[1] \leftarrow none$;
4      while $true$ do
5          while $itArray[pos] = none$ and $pos \neq 1$ do
6              $pos \leftarrow pos - 1$;
7          while $itArray[1] = none$ do
8              $i \leftarrow$ GetNewRecordsOrQuit();
9              set $imResult$ based on $R_{p_1}[i]$;
10             $itArray[1] \leftarrow H_2$.match($imResult$);
11         set $imResult$ based on $pos$ and $itArray[pos]$.record();
12         $itArray[pos] \leftarrow itArray[pos]$.next();
13         if $pos = n - 1$ then
14             output $imResult$ as a final result;
15         else
16             $itArray[pos + 1] \leftarrow H_{pos+2}$.match(imResult);
17             $pos \leftarrow pos + 1$;
```

Definitions of Variables and Functions Each thread of the GPU has two arrays. One is $imResult[1, 2, \cdots, q_n]$, which stores the intermediate result. Each entity corresponds to one query graph node, so a final joining result is produced when all entities are set. Another is $itArray[1, 2, \cdots, n-1]$. When a thread is processing subjoin $\bowtie_i$ (see the join tree in Fig 2(b)), it stores an iterator of H_{i+1} in $itArray[i]$. Both $imResult$ and $itArray$ are frequently accessed and small enough, so we put them in shared memory. Each thread has a variable pos which is used to indicate the subjoin being processed. For example, the thread will be processing $\bowtie_1$ if $pos = 1$.

The function `GetNewRecordsOrQuit()` is used to retrieve the first unhandled record of R_{p_1}. If there is no unhandled record, this function will terminate the thread.

A DFS-like Procedure The strategy of our algorithm is to "consume" the intermediate results as soon as possible. In the whole process of the join, a GPU thread will move forward to process next subjoin whenever possible and only moves backward when the current subjoin fails. The manner is similar to that of a depth-first search algorithm.

An Example We use an example to illustrate how our algorithm works.

Let us consider the data graph and query in Fig 1 again. The result sets of STwigs and the final result set are $R_1 = \{(A_1, B_1, C_1), (A_2, B_2, C_1)\}$, $R_2 = \{(B_1, C_1, D_1)\}$, $R_3 = \{(D_1, E_1)\}$ and $R_1 \bowtie R_2 \bowtie R_3 = \{(A_1, B_1, C_1, D_1, E_1)\}$ respectively.

In this example, the probe relation is R_1 and we first join R_2 with R_1, so $\bowtie_1 = R_2 \bowtie R_1$ and $\bowtie_2 = R_3 \bowtie (R_2 \bowtie R_1)$. We assume $imResult[1]$ corresponds to node A of query graph; $imResult[2]$ corresponds to node B ...

At the beginning, the GPU constructs hash tables H_2 and H_3 for R_2 and R_3 respectively. Then the join process starts. To simplify the explanation, we assume there is only one thread.

When Algorithm 2 starts, the thread will first initialize pos to 1 and $itArray[1]$ to *none*. Then it starts the first round of the outermost block of while.

In line 8, it calls `GetNewRecordsOrQuit()` to get the index of the first unhandled record of R_1. Then $imResult$ is set to $(A_1, B_1, C_1, -, -)$ (line 9). The `match` operator of H_2 checks the second and third entities of $imResult$ and returns an iterator pointing to record (B_1, C_1, D_1) of R_2 to set $itArray[1]$ (line 10). Based on the value of $itArray[1].record()$, $imResult$ is further updated to $(A_1, B_1, C_1, D_1, -)$ (line 11). $itArray[1]$ is updated again to point to next record in line 12. It is actually updated to *none* because there is no other record in R_2 whose first two entities are B_1 and C_1. Preparing for next round, $itArray[2]$ is set to an iterator pointing to the only matching record of R_3, (D_1, E_1), in line 16, and pos is increased to 2 in line 17.

In the second round, $imResult[5]$ will be set to E_1 in line 11. At this point, all entities of $imResult$ are set properly, so the thread will output the final result $(A_1, B_1, C_1, D_1, E_1)$ contained in $imResult$ in line 14.

At the third round, *pos* will be decreased to 1 again in the while block beginning in line 5. Then in the while block beginning in line 7, it calls `GetNewRecordsOrQuit()` to get a new record of R_1, i.e., (A_2, B_2, C_1), to set *imResult*. However, no record of R_2 match *imResult* this time and *itArray*[1] is set to *none* (line 10). The thread calls `GetNewRecordsOrQuit()` again and terminates itself because no record is unhandled.

4.4 Storage

Storage of R_{p_1} The result set R_{p_1} can be stored in either CPU or GPU and both solutions require the memory access to be coalesced, so we store R_{p_1} in a well-padding 2D array. Each record of R_{p_1} will be stored in one column.

The Shared Memory *imResult* and *itArray* cannot be stored in registers because they are not indexed with constant quantities, so we store them in shared memory as mentioned above. To avoid serializing access due to bank conflicts, 32 *imResult*s of all threads in the same warp are stored together in a 2D array. Each *imResult* occupies one column so that each thread can always access *imResult* via its own bank. *itArray* is stored in the same way.

5 Experimental Study

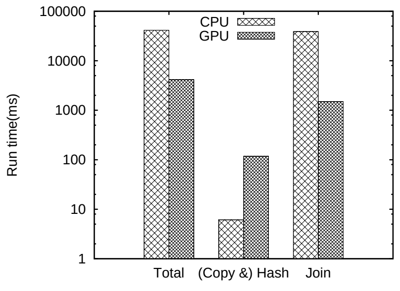
All the experiments were conducted on a computer with 8GB RAM, a 3.30GHz Intel(R) Xeon(R) E5-2643 CPU and a Tesla C2075 GPU. Tesla C2075 has 5GB global memory and its compute capability is 2.0.

We use synthetic data in the following experiments to study how the properties of data graphs and query graphs affect our speedup ratio. The data graphs in this section are generated by using R-MAT [3] model. The graphs are undirected and the parameters of R-MAT model are $a = 0.4$, $b = c = d = 0.2$. A query graph with N nodes is generated by adding $2N$ edges randomly.

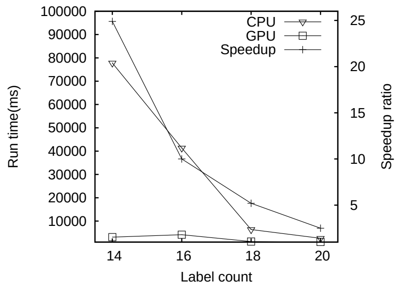
5.1 Computation Time of Each Steps

To analyze the efficiency of each step, we conducted an experiment and recorded the computation time. The data graph we used in this experiment has 16K nodes and 16 labels. The average degree is 32. We generated 100 query graphs whose numbers of nodes range from 5 to 10. The results are shown in Fig 3a.

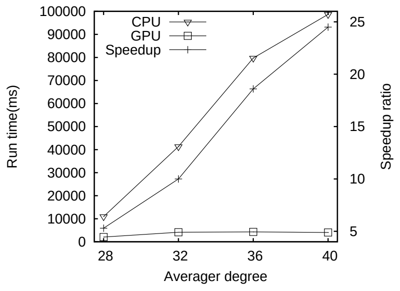
The total time is the sum of "(Copy &) Hash" time, join time and other time consumed by common operations which are the same both CPU-based and GPU-based algorithms. We can see that the join time normally dominates the total time, so our approach of optimizing the join operation achieves an overall speedup of an order of magnitude. Actually, if we only consider the join time, the speedup ratio reaches to 26.0. On the other hand, the GPU takes more time to finish the step of "(Copy &) Hash", but this time is relatively small and will



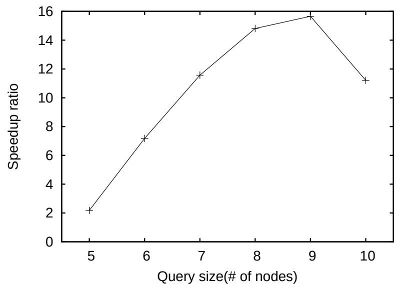
(a) Computation time of each step



(b) Label count effect



(c) Average degree effect



(d) Query size effect

Fig. 3: Experiment Results

not affect the overall speedup ratio. The major reasons for this phenomenon are 1) the GPU has to copy data from CPU memory before building hash tables, which can consume considerable time, 2) our GPU algorithm uses lots of atomic operations in the step of building hash tables and 3) the uncoalesced memory access pattern of this step can slow down GPU algorithm.

5.2 The Effect of Data Graph Size

To verify the performance of GPU algorithm with different size of data graph, we conducted some experiments in which only the numbers of nodes of data graphs are changed. The numbers of labels and the average degrees for all data graphs are 16 and 32 respectively. The numbers of data graph nodes are 16K, 32K, 64K and 128K. Our GPU algorithm outperforms CPU in all cases, and the speedup ratios range from 7.8 to 16.7.

5.3 The Effect of Label Count

In this experiment, we study the effect of the number of labels. We used several data graphs with the same node counts (16K) and average degrees (32). Only the label counts are different.

As shown in Fig 3b, the run times of both CPU and GPU decrease as the label count increases. The reason is higher label count normally results in fewer number of nodes in a data graph that match the roots of STwigs. Consequently, the sizes of the relations to be joined are smaller and the join time decreases rapidly. As the join time decreases, the workload of GPU decreases and the speedup ratio also decreases.

5.4 The Effect of Average Degree

The effect of the average degree is similar to that of the label count. We generated four data graphs with different average degrees. The same set of query graphs are used for all the data graphs. As the average degree grows, the run time increases because the result sets of STwigs becomes larger. Also, the speedup ratio increases because of the rise of workload.

5.5 The Effect of Query Graph Size

Besides the properties of data graphs, we also observed a relation between the speedup ratio and the query graph size, i.e., the number of nodes in a query graph. The relation is shown in Fig 3d. We can see that the speedup ratio increases as the number of query graph nodes increases from 5 to 9. However, the speedup ratio decreases when the sizes of query graphs reach 10. This phenomenon relates to the workload and our implementation.

When the query graph is small, smaller number of STwigs and fewer edges in each STwig results in light workload. In such cases, we cannot make full use of the powerful computation abilities of the GPU. When the query graphs are larger, the workload increases and each GPU thread can keep itself busy. Consequently, the average speedup ratio for query graphs with 9 nodes reaches 15.66. However, if a query graph is too large, the speedup ratio may decline because *imResult* and *itArray* occupied too much shared memory and thus the number of the concurrent threads will decline. In our experiment, concurrent thread counts can drop from 21504 to 10752.

6 Conclusions

Subgraph matching involves a large amount of data to be processed. Consequently, the computation time is very long. To alleviate this problem, parallel technology is widely used. However, most of the approaches only try to exploit CPUs' parallel abilities, which is very limited. By contrast, our algorithm makes use of GPUs. A high-end GPU has the ability of executing thousands of threads simultaneously. This ability allows our algorithm to outperform the state-of-the-art algorithm by an order of magnitude based on the experimental results. Additionally, we observe that our approach can perform even better in a heavy workload situation.

References

1. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix bit loaded: a scalable lightweight join query processor for rdf data. In: Proceedings of the 19th international conference on World wide web. pp. 41–50. ACM (2010)
2. Branden, C., Tooze, J., et al.: Introduction to protein structure, vol. 2. Garland New York (1991)
3. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-mat: A recursive model for graph mining. Computer Science Department p. 541 (2004)
4. Cheng, J., Yu, J.X., Ding, B., Yu, P.S., Wang, H.: Fast graph pattern matching. In: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on. pp. 913–922. IEEE (2008)
5. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM Journal on Computing 32(5), 1338–1355 (2003)
6. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. Pattern Analysis and Machine Intelligence, IEEE Transactions on 26(10), 1367–1372 (2004)
7. Eichinger, F., Böhm, K., Huber, M.: Mining edge-weighted call graphs to localise software bugs. In: Machine Learning and Knowledge Discovery in Databases, pp. 333–348. Springer (2008)
8. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database system implementation, vol. 654. Prentice Hall Upper Saddle River, NJ: (2000)
9. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., Sander, P.: Relational joins on graphics processors. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 511–524. ACM (2008)
10. He, H., Singh, A.K.: Graphs-at-a-time: query language and access methods for graph databases. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data. pp. 405–418. ACM (2008)
11. Kaldewey, T., Lohman, G., Mueller, R., Volk, P.: Gpu join processing revisited. In: Proceedings of the Eighth International Workshop on Data Management on New Hardware. pp. 55–62. ACM (2012)
12. Neumann, T., Weikum, G.: The rdf-3x engine for scalable management of rdf data. The VLDB Journal 19(1), 91–113 (2010)
13. NVIDIA: CUDA C best practices guide (2013)
14. Sun, Z., Wang, H., Shao, B., Li, J.: Efficient subgraph matching on billion node graphs. Proceedings of the VLDB . . . pp. 788–799 (2012)
15. Ullmann, J.R.: An algorithm for subgraph isomorphism. Journal of the ACM (JACM) 23(1), 31–42 (1976)
16. Yan, X., Yu, P.S., Han, J.: Substructure similarity search in graph databases. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data. pp. 766–777. ACM (2005)
17. Zhu, F., Qu, Q., Lo, D., Yan, X., Han, J., Yu, P.: Mining top-k large structural patterns in a massive network. Proceedings of the VLDB Endowment 4(11) (2011)
18. Zou, L., Chen, L., Özsu, M.T.: Distance-join: Pattern match query in a large graph database. Proceedings of the VLDB Endowment 2(1), 886–897 (2009)