An Efficient Parallel Algorithm for Graph Isomorphism on GPU using CUDA

Min-Young Son [1], Young-Hak Kim [2], Byoung-Woo Oh [3]
Dept. of Computer Engineering, Kumoh National Institute of Technology
1 Yangho-dong, Gumi, Gyeongbuk 730-701 Republic of Korea
[1] son0804@kumoh.ac.kr
[2] kimyh@kumoh.ac.kr (corresponding author)
[3] bwoh@kumoh.ac.kr

Abstract— Modern Graphics Processing Units (GPUs) have high computation power and low cost. Recently, many applications in various fields have been computed powerfully on the GPU using CUDA. In this paper, we propose an efficient parallel algorithm for graph isomorphism which runs on the GPU using CUDA for matching large graphs. Parallelization of a sequential graph isomorphism algorithm is one of the hardest problems because it includes inherently sequential characteristics. Our approach divides the given graphs into smaller blocks using a divide-and-conquer, and then maps the blocks to parallel processing units on the GPU. The smaller blocks are solved in individual processing units, and then the results are combined using hierarchical procedures. In the experiment, we used random graphs from vertices of small size to up to tens of thousands of vertices in order to solve efficiently graph isomorphism for large graphs. The experimental results show that the proposed approach brings a considerable improvement in performance and efficiency comparing to the CPU-based results. Our result also shows high performance, especially on large graphs.

Keyword- Graph isomorphism, CUDA, Large graphs, GPU

I. Introduction

Graph is one of popular data representations in the field of computers, engineering, science, etc. One of the most fundamental techniques in solving graph problems is graph isomorphism algorithms, and graph isomorphism algorithm is used in many other graph applications. In general, the implementation of a serial-based graph algorithm is time consuming as the number of vertex and edge in graph increases [1]. In particular, the graph isomorphism algorithms require rapid calculation time as the size of graph increases. Therefore, the more graph has vertices and edges, the more ratio of time for solving isomorphism problem in the various graph algorithms is consumed. Recently, there have been increasing graph applications with large size because of increasing big data [2], [3]. Due to this large data, the serial-based graph algorithms are impractical because the time complexity is too high. The parallel implementation of graph algorithms for large graph is more effective than serial-based implementation. Several graph algorithms on supercomputer have been studied in order to improve the execution time of large graphs, but the use of supercomputer has real restrictions due to high cost. Other studies use CPU cluster for parallel implementation instead of supercomputer, but the existence of bottle neck problem that is caused by the synchronization in this cluster environment.

Recently, the GPU (Graphics Processing Unit) provides high calculation capacity at a low cost. Since the GPU provides high computing power, low cost, and easy accessibility, GPU is used in various application fields. CUDA (Compute Unified Device Architecture) of Nvidia is a parallel structure of the thread mass, and provides a programming model that can be used in parallel hardware with the CPU. This allows the BSP (bulk synchronous parallel). These bulk thread parallelism uses divide-and-conquer method that each processing node solves small sub-problems, and they are combined together to resolve big problems. Each thread can access the global memory, and can be operated independently at the same time.

This study considers a graph isomorphism problem which can be used as basic tools to match large graphs. It is very difficult for a sequential isomorphism algorithm to parallelize because it includes inherently sequential characteristics. The research about the parallel algorithm of graph isomorphism on the GPU has been not studied deeply by reason of inherent sequential problem. In this paper, we propose a parallel graph isomorphism algorithm which runs efficiently on the GPU using CUDA for matching large graphs. Our approach divides given graphs into smaller blocks using a divide-and-conquer, and then maps the smaller blocks to parallel processing units on the GPU. The results in each block are combined using hierarchical procedures after each block is solved independently at the same time in individual processing unit. In the experiment, we expend random graphs from vertices of small size to up to tens of thousands of vertices in order to show efficiently graph isomorphism for matching large graphs. The experiment shows that our approach brings a considerable improvement in performance and efficiency comparing to the CPU-based results, especially on large graphs.

The details of this study are as follows. Section 2 provides theory of graph, graph isomorphism problem and the graph isomorphism algorithms based on serial architecture. Section 3 provides the CUDA hardware architecture and operating principles, and features. In Section 4, our algorithm on the GPU using CUDA is introduced in detail. Section 5 shows the results of the experiment and compares the performance between our GPU-based results and the CPU-based one.

II. Graph Isomorphism

A. Graph

Graph refers to a structure that consists of nodes and edges. The degree of node is the number of adjacent nodes that is connected by edge. A graph is categorized by directed graph or undirected graph depending on whether direction of edge exists or not. If multiple edges between two nodes are allowed, the graph is classified as a simple graph, or if not, the graph is classified as a multigraph. Also, graphs are categorized whether label of elements is duplicated or not, and nodes can have or cannot have an edge. This paper uses undirected simple graph that node can have edge to self. Algorithms for different types of graphs can be solved similarly to variations of the proposed algorithm in this study.

Graph matching is a process of finding the correspondence between the node and an edge of the two graphs [4]. The graph matching is used in various applications, the typical example 2D, 3D image analysis, document processing, biometric recognition, image databases, video analysis, biological / biomedical applications and so on [5]. Exact graph matching is characterized by finding a mapping node to ensure that if you have two nodes in a graph, which is connected to two nodes are mapped in another graph also be connected. Matching graph algorithm can be extended to graph isomorphism, monomorphism, homomorphism, sub graph isomorphism and maximum common subgraph (MCS). Most strong type of various types of exact graph matching is graph isomorphism.

In this paper, graph is represented by G or H, and a set of nodes is V or W, a set of edges is E or F = {(v1, v2) | v1, v2∈ V}. Because we use undirected graph, two edges (v1, v2), (v2, v1) are the same. Two graphs G=(V, E) and H=(W, F) are isomorphic if there is a bijective function f: V →W such that for all v, w∈V:

$$\{v, w\} \in E \leftrightarrow \{f(v), f(w)\} \in F$$

Graph isomorphism is difficult to distinguish by the eye. The two examples of graph Isomorphism shown in Fig. 1 looks different. Graph isomorphism must have unchanging characteristics as follows:

- The same number of vertices.
- The same number of edges.
- Degrees of corresponding vertices are the same.
- If one is bipartite, the other must be.
- If one is complete, the other must be.
- If one is a wheel, the other must be.
- Etc.

If two graphs have different invariants, then they cannot be isomorphic.

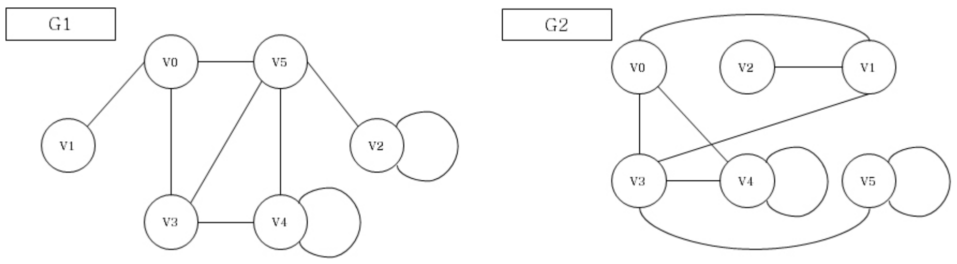


Fig. 1. A pair of isomorphic graphs

B. Graph Isomorphism algorithms

Graph isomorphism problem can be easily solved by a brute-force enumeration. Graph isomorphism of two graphs with n vertices has n! one-to-one correspondence relationship. Due to this relationship, the time complexity of graph isomorphic problem is increased exponentially as the size of graph increases. Brute-force enumeration is only possible in the graph with the small number of nodes and edges. The polynomial-time algorithm is not known yet on solving the problem of determining whether two graphs are isomorphism or not. Therefore, there have been studied a lot of researches for a long time in order to reduce the number of operations in graph isomorphism problem.

Typical method of graph isomorphism algorithm is to reduce the total number of operations by removing the candidates in the matching process. Ullmann's algorithm uses the method to reduce the candidates using backtracking. Ullmann's algorithm is most commonly used so far in graph matching area, and also his idea is working to have affected the development of various algorithms on graph isomorphism problem [1]. More recently, the VF algorithm uses a depth-first search strategy and effective pruning method in search tree. Nauty algorithm is based on a set of transform using the canonical form to reduce the operations of graph matching. Several algorithms on graph isomorphism were compared in the previous studies [6], [7]. The comparative studies showed that the performance of graph isomorphism algorithm depends on the characteristics of graphs. In these studies, they also showed that Nauty algorithm and VF algorithm have good performance in general.

III. GPU AND CUDA

A. GPU and CUDA

Recently, GPGPU (general purpose graphics processing unit), as well as the graphics and image processing has been expanded to a wide range of applications. Some of general data processing algorithms requiring the high performance (e.g. sorting, matrix multiplication, etc.) have been developed on the GPU. Also, large multi-thread model provided by the GPU makes it possible to adopt a data-parallel approach using concurrent threads in even irregular problem such as graph algorithm [8].

GPU provides massively threaded parallel architecture where millions of threads in this architecture can operate in parallel. This fine structure comes from supercomputer and typically supercomputers have a small number of strong cores. In this architecture, divide-and-conquer method is used as the basic technique of implementing parallelization, and the final result can be constructed going through the steps of combining sub blocks after individual processing unit solves independently each sub block at the same time.

The GPU is composed of a set of Streaming Multiprocessors (SM), each SM is made up of a set of streaming processors (SP). Each multiprocessor has shared memory, 32-bit register set, texture, and constant memory caches. Global memory, shared memory, and registers are existed for GPU, SM, and SP, respectively. Because every SP is accessible to the global memory, all of the data in the program are necessary to carry out to the global memory. SPs can access to the same shared memory only belonging same SM, each SPs have their own registers. Global memory has most big size but slowest approach speed. On the other hand, the access speed of registers is fastest, but the size is the smallest. Therefore, when we develop an application program on the GPU, we should consider these characteristics of types of memory.

The amount of memory available on the GPU is considered as the limiting factor of GPGPU algorithms [9]. The GPU can not handle more data than the maximum supported textures. Also, GPU memory layout is optimized according to the graphics rendering to restrict the GPGPU solution. Limited access to memory makes it difficult to port the general algorithm to this framework.

Software interface CUDA API is a set of library functions which is coded as an extension of the C language. Compiler produces executable code for CUDA-enabled device. CUDA is a collection of multithreads to work in parallel. CUDA is to improve the GPGPU programming model by not graphics pipeline but multi-core GPU coprocessor. Block is a collection of threads operating on a multi-processor at a given time. Synchronization for all of the threads in the same block is also possible. The collection of all the blocks is called grid. The threads of plurality of blocks are synchronized only at the end of the run by the kernel as shown in Fig. 2 [8].

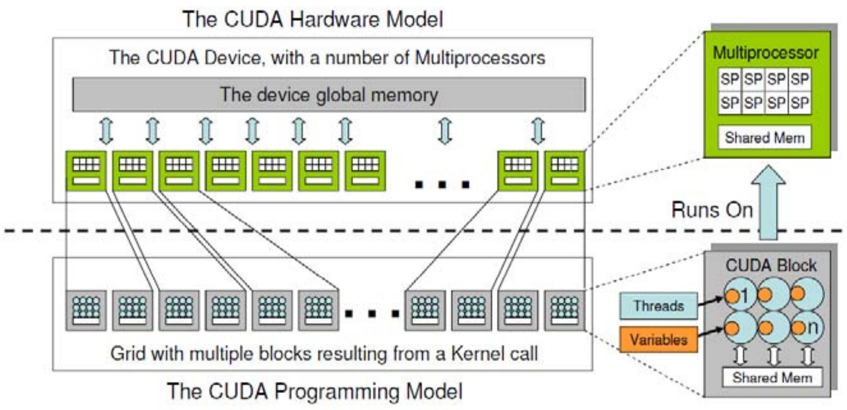


Fig. 2. CUDA hardware model (top) and the programming model (bottom)

B. Graph Algorithm using GPU and CUDA

In accordance with the enlargement of graph application, parallel graph algorithms for improving the execution time of graph problems has been extensively studied. Recently, the GPU model using CUDA has been considered as a new parallel model because the GPU provides high performance at a low price and the CUDA provides easy accessibility. Katz et al [10] implemented parallel graph algorithms on the APSP (All-pairs shortest-path) problem with large datasets using NVIDIA G80 GPU architecture and CUDA API. Katz et al [10] also proposed a generalization method for adjusting the size of a large graph than the available DRAM memory on the GPU. Matsumoto et al. [11] implemented a parallel algorithm on the APSP algorithm using a hybrid CPU-GPU concept. Merrill et al. [12] proposed the BFS (Breadth-First Search) method for parallel implementation using single and quad GPU configurations, and then compared the performance depending on the configuration. Harish et al. [13] have implemented multi-threaded algorithms for large graphs using the GPU model with model. There have been implemented many parallel graph algorithms on the GPU such as breadth first search (BFS), st-connectivity (STCON), single source shortest path (SSSP), all pairs shortest path (APSP), minimum spanning trees (MST), maximum flow (MF), etc. These graph algorithms showed especially high performance in large scale graph. It is very important to find a suitable combination of parallel and serial algorithms because sequential graph algorithms must be transformed to the parallel model.

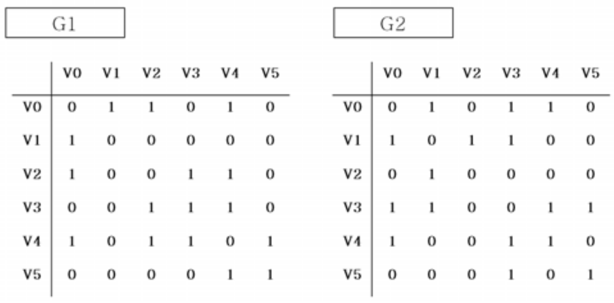
The execution time in sequential graph isomorphism algorithms is increased exponentially as the size of graph increases. Therefore, the parallel implementation of sequential graph isomorphism algorithms is much difficult than the serial implementation. Recently, a few researchers have been trying to implement the parallel algorithms on graph isomorphism using some parallel models. However, the parallel implementation on graph isomorphism on the GPU is not yet studied deeply [14], [15].

IV. MAIN ALGORITHM

The proposed algorithm is conducted by a collaboration of the CPU and GPU. Since the GPU has the time delay for synchronization and memory limit, it is important to ensure that a given problem is divided into appropriate sub-blocks and then each sub-block is assigned to a thread on the GPU. The CPU coordinates primarily an entire process and selects candidates that will be running on the GPU through relatively a simple task. The entire process of our algorithm for matching graph isomorphism will be explained in the following.

A. Reducing Candidates Process in CPU.

Two graphs given in Fig. 1 can be represented as adjacency matrix in Fig. 3 for matching graph isomorphism, which is a proper data structure to solve graph problems on the GPU. Each node has to find a node that is matching graph isomorphism in another graph. Every node can have candidate nodes for all the nodes in the given graph. If the number of nodes in a graph is n, the number of total cases that can be considered for matching graph isomorphism is n!. If the number of nodes in a graph increases, the size of adjacency matrix for representing the graph also increases. It is not possible to load the adjacency matrix at a time to the global memory of the GPU because the global memory of GPU is smaller in size than the DRAM of the CPU. Therefore, in order to overcome this memory limitation in the GPU, we should use *the reducing candidate process in CPU* as follows.



G1

	V0	V1	V2	V3	V4	V5
V0	0	1	1	0	1	0
V1	1	0	0	0	0	0
V2	1	0	0	1	1	0
V3	0	0	1	1	1	0
V4	1	0	1	1	0	1
V5	0	0	0	0	1	1

G2

	V0	V1	V2	V3	V4	V5
V0	0	1	0	1	1	0
V1	1	0	1	1	0	0
V2	0	1	0	0	0	0
V3	1	1	0	0	1	1
V4	1	0	0	1	1	0
V5	0	0	0	1	0	1

Fig. 3. Unequal Matrices for Isomorphic graphs G1 and G2

Table 1 shows the total number of candidate nodes without *reducing candidate process in CPU* and with *reducing candidate process* based on the randomly connected graph with density ɳ=0.5. The density ɳ is the probability that an edge can be connected between two nodes a and b in the same graph. The high value of density ɳ means that the given graph is dense. The total number of candidate processes with *reducing candidate process in CPU* is more less than without that process (n^2). The reduction amount of candidates is determined by the type of the graph and density ɳ.

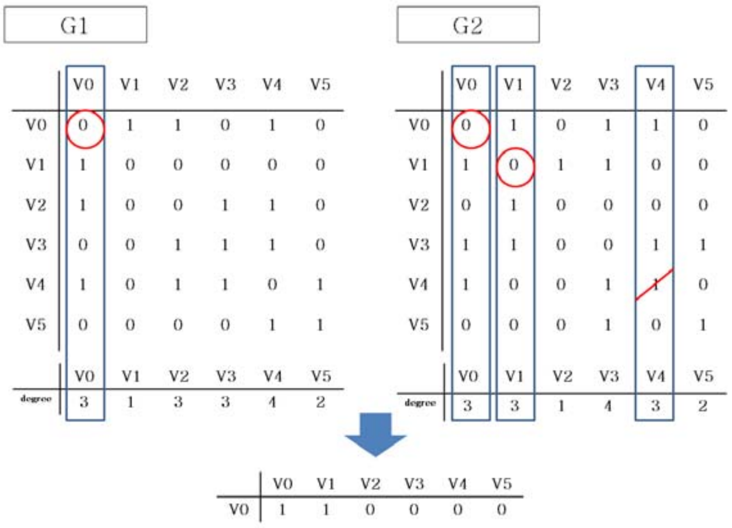
Table Ⅰ. Total Number of Candidates of All Nodes

# nodes	Total number of candidates of all nodes without the reducing candidate process i**n** CPU(n^2).	Total number of candidates of all nodes with the reducing candidate process in CPU.
100	10,000	602
500	250,000	7,152
1,000	1,000,000	18,382
5,000	25,000,000	204,052
10,000	100,000,000	572,440
20,000	400,000,000	1,638,822

The reducing candidate process in CPU has another advantage except saving memory. The proposed algorithm assigns to GPU's threads according to every case of candidates. Since each GPU's processing cores work in lockstep based on the same clock, each thread operating the same or very similar repeated job is efficient in GPU. The insights of developers using parallel processing threads including the mutual communication and work are required as well as a programming model that can support them effectively. If GPU handles all cases without *the reducing candidate process in CPU*, it will cause delay in time until all thread are finished, because many candidates can be detected in a short period of time. Besides, the GeForce GTX 670's total number of threads is 67,107,840 (65,535 blocks x 1,024 threads). If we consider every cases without *the reducing candidate process in CPU* and with more than 8,192 nodes($8,192^2$= 67,108,864), then threads have to run repeat. *The reducing candidate process in CPU* leads to reducing overall processing time because of reduced number of iterations of each thread. Since the number of using threads in GPU must be decided before calling parallel process, *the reducing candidate process in CPU is executed* first.

Algorithm 1: The reducing candidate process in CPU.

```
     Input : adjacency matrix of two graphs $G_1$, $G_2$.
     Output : Candidates matrix for matching nodes.
1.  begin
2.    Calculating degree of the nodes of the two graphs.
3.    If ($\Sigma$node($G_1$) $\neq$ $\Sigma$node($G_2$) || $\Sigma$edge($G_1$) $\neq$ $\Sigma$edge ($G_2$)) { they are not isomorphic. Exit.}
4.    Matrix D's every element are initialized -1.
5.    Loop(every node $v_i$ of $G_1$){
6.        Loop(every node $v_j$ of $G_2$){
7.            If(degree($v_i$) $\neq$ degree($v_j$) || $A_{ii}$ $\neq$ $A_{jj}$) { $v_j$ candidate is removed in $v_i$'s candidates. }
8.        }
9.        If($v_i$ has one candidate) { D[i] $\leftarrow$ remaining candidate.}
10.   }
11.   If(any node doesn't have candidates) { they are not isomorphic. Exit.}
12. end
```



G1	V0	V1	V2	V3	V4	V5
V0	0	1	1	0	1	0
V1	1	0	0	0	0	0
V2	1	0	0	1	1	0
V3	0	0	1	1	1	0
V4	1	0	1	1	0	1
V5	0	0	0	0	1	1

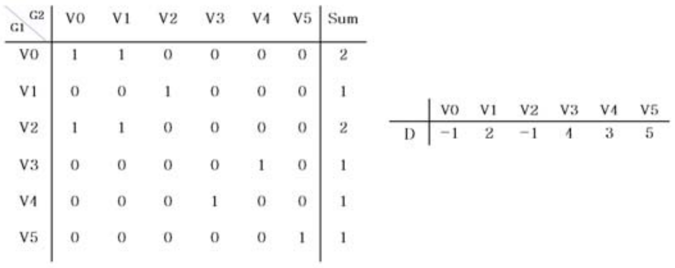
	V0	V1	V2	V3	V4	V5
degree	3	1	3	3	4	2

G2	V0	V1	V2	V3	V4	V5
V0	0	1	0	1	1	0
V1	1	0	1	1	0	0
V2	0	1	0	0	0	0
V3	1	1	0	0	1	1
V4	1	0	0	1	~~1~~	0
V5	0	0	0	1	0	1

	V0	V1	V2	V3	V4	V5
degree	3	3	1	4	3	2

	V0	V1	V2	V3	V4	V5
V0	1	1	0	0	0	0

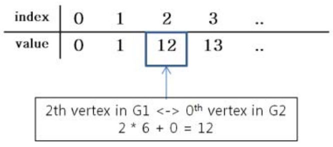
Fig. 4. Process of making the matrix of candidates(bottom) using adjacency matrices of two graphs(top) and two degree matrices(middle)



G2 / G1	V0	V1	V2	V3	V4	V5	Sum
V0	1	1	0	0	0	0	2
V1	0	0	1	0	0	0	1
V2	1	1	0	0	0	0	2
V3	0	0	0	0	1	0	1
V4	0	0	0	1	0	0	1
V5	0	0	0	0	0	1	1

	V0	V1	V2	V3	V4	V5
D	−1	2	−1	4	3	5

Fig. 5. The matrix of candidates(left) and the matrix D having matching information(right)



index	0	1	2	3	..
value	0	1	12	13	..

Fig. 6. The final matrix of candidates to enter in GPU process

The procedure of *reducing candidate processes in CPU* is shown in Algorithm 1. Two adjacency matrices are entered to know whether two graphs are isomorphic or not. Each node's degree can get by each matrix's row sum. This study focuses only graph isomorphism. If two graphs are isomorphism, then each graph's total number of nodes and edges must have the same number. Therefore, line 3 of Algorithm 1 prevents unnecessary running.

If two graph's total number of nodes and edges are same, then matrix D is initialized to set number of nodes size and all values as -1. The matrix D is to save matching node information. If first graph's i-th node matches second graph's j-th node, then D[i-1] is set j-1. Based on two graph's degree information main part of *the reducing candidate process in CPU* is run. According to graph isomorphism's characteristic, candidates of each node are selected when matching both nodes has same degree and same recursive edge, as shown in Fig. 4. If the number of selected candidates is one, then matrix D has the candidate information as given in Fig. 5. After the first candidates extracting process is finished for all nodes, if any of the nodes don't have candidate, then the value of false is returned and the whole algorithm is terminated. The nodes that have multiple candidates are saved for process in GPU. In consideration of the memory limitation of GPU, the matrix in Fig. 6 is needed for reducing using memory. For example, first graph's i-th vertex has second graph's j-th vertex candidate, "i * total number of node + j " is saved in one of the matrix and it will be used in GPU.

B. Reducing Candidates Process in GPU.

The GPU algorithm is shown in Algorithm 2. First, transmission of the two adjacency matrices and the D matrix and the candidate matrix is necessary for calculating in GPU from the CPU. Through this process, candidates from *the reducing candidate process in CPU* are loaded to global memory in GPU, and then candidates are referred to by each thread. In GPU, second algorithm for removing candidates is executed. Each thread removes repeatedly remaining candidates using determined matching node or having minimum candidate node information in the global memory. If every node finds a matching node or at least one node that doesn't have candidates, GPU process is terminated and sends the results to CPU.

Algorithm 2: The reducing candidate process in GPU.

Input : Candidates matrix for matching node after the reducing candidate process in CPU
Output : Final matching matrix for two graphs isomorphism.

```
1.  begin
2.  idx ← threadIdx.x + blockIdx.x * blockdim.x;
3.  Loop( idx < total number of candidates )
4.    Set update ← 1 ,  i ← #G$_1$ node,  j ← #G$_2$ node of one set of remain candidates.
5.    Loop ( update = 1 ){
6.        update ← 0.
7.        Loop (every index k of that $D_k \neq -1$) {
8.            If( $G1_{ik} \neq G2_{jD(k)}$)  then $v_i$ candidate is removed in $v_j$'s candidates.
9.        }
10.       If ( $v_i$ doesn't have candidates) { they are not isomorphic. Exit.}
11.       If ( $v_i$ has one candidate) { then $D_i$ ← remaining candidate.}
12.       If ( any node finds matching node || any node's candidates are changed ) { update ← 1 }
13.   }
14.   idx += blockDim.x * gridDim.x;
15. }
16. end
```

Although GPU have multi thread, limitation of thread exists. In the case of GeForce GTX 670 it can run a maximum of 67 million threads at the same time. Large graph problem can run exceeding the number of maximum threads easily. In order to overcome this situation, this algorithm runs threads repeatedly for solving multiple problems as shown in lines 2, 3 and 14 of Algorithm 2.

Candidate information is inputted one dimension matrix. Each thread converts candidate value to fine matching nodes. Each threads checks possibility of the candidate using completed matching nodes information in the D matrix as given in Fig. 7. If the candidate is still possible and one candidate remains, this candidate information is uploaded to matching matrix D. If the candidate is impossible, then this candidate is excluded in candidates list. This process is run again if any of threads change matching matrix. After all the processes are finished, the GPU algorithm is terminated and then it sends the result of final matching matrix to CPU.

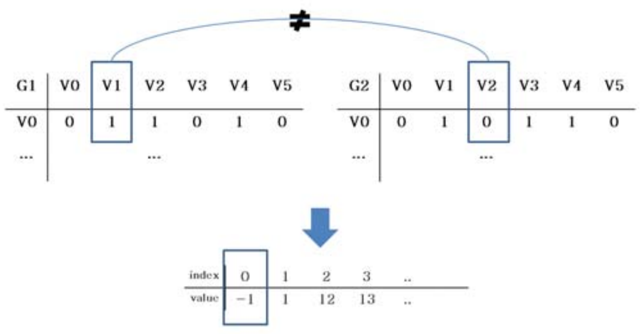


Fig. 7. The updated matrix of candidate(bottom) according to process of checking using the D matrix(Fig. 5) and two adjacency matrices(top) in each thread of GPU

V. Performance Analysis

A. *Experimental Setup*

All the algorithms for performance test are implemented in C++ and CUDA. The test environment is on an Intel Core i5-2500K 3.3GHz PC, equipped with 4 GB of RAM and a GeForce GTX 670 GPU. GeForce GTX 670 has 2GB global memory and its computation capability is 3.0. We use the source code from the Web site *http://mivia.unisa.it/datasets/graph-database/vflib/* together with the two versions of the VF algorithm and Ullmann's algorithm. Algorithms for comparison test on CPU and our proposed algorithm are run on both CPU and GPU.

We configure the graphs having minimum 100 nodes to maximum 20,000 nodes by adjacency matrix, using a random number generator. Matrix of experimental data is generated 0 or 1 with density η = 0.1, 0.05 and 0.01. We use undirected graph that adjacency matrix of the graph G is made up set to the same value with g_{ij} and g_{ji}. In this experiment, we measure and compare the execution time of the CPU and GPU according of the number of nodes.

B. *Summary and Results.*

Fig. 8 to 10 show the results of the execution time of the proposed algorithm, Ullmann's algorithm, VF and VF2 with the graph size displayed on the x-axis and the different values of η : 0.05, 0.1 and 0.2. The experimental result shows that the execution time of the proposed algorithm has relatively plain trend than the one of other algorithms. As the value of η is low, the execution time of Ullmann's algorithm gets higher in a fast manner, while the execution time of other algorithms increases gradually. The performance of VF and VF2 is similar, but VF2 is always better than VF. Despite detail distinctions, the proposed algorithm shows best performance. The more the number of nodes increases, the more the performance effectiveness of the proposed algorithm increases.

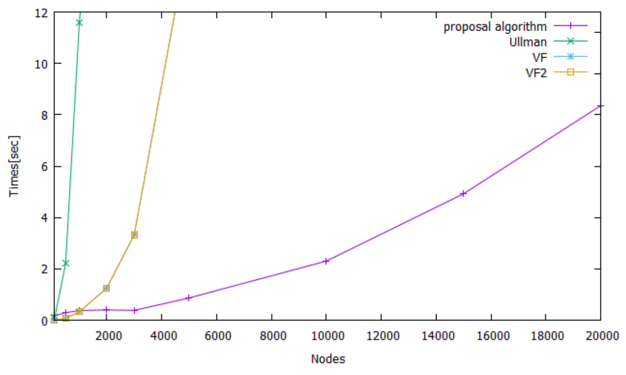


Fig. 8. The performance of the four algorithms on randomly connected graphs – η = 0.05

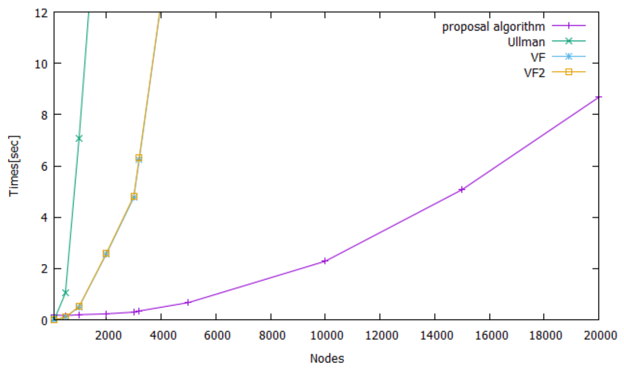


Fig. 9. The performance of the four algorithms on randomly connected graphs – η = 0.1

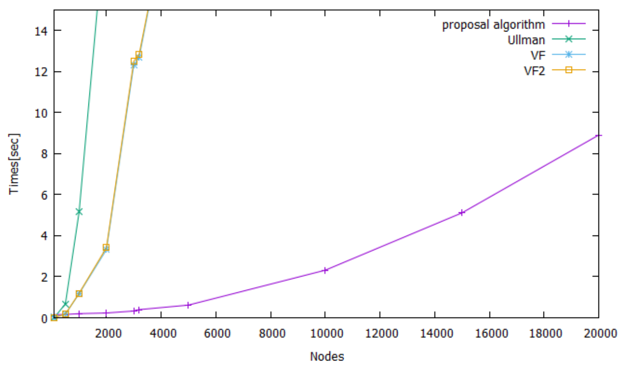


Fig. 10. The performance of the four algorithms on randomly connected graphs – η = 0.2

VI. Conclusions

In this paper, we proposed parallel isomorphic algorithm for large graphs using the CUDA in GPU. Recently, large graph is used widely in various field, algorithm of graph isomorphism is basic operation of almost graph problem. Prior almost algorithms of graph isomorphism were proposed using serial method based in CPU. Because computation of graph isomorphism is rapidly increased depending on the size of graph, the parallel algorithm is more effective than serial algorithm to the graph isomorphism problem.

GPU has many advantages those are low cost and easy accessibility as well as efficient to many parallel algorithms. Therefore, our proposed algorithm is run using CUDA on GPU. However, because GPU has problem of limited memory and the performance of parallel algorithms depends on the distribution to small jobs from one original problem, we proposed algorithm of 2 steps those are CPU process for reducing data and GPU process.

In order to verify the performance of the proposed algorithm, we implemented proposed algorithm, and compared with other 3 algorithms Ullmann’s algorithm, VF and VF2. We showed experiment results using random number generated graphs. According to the experiment results, we examined that proposed algorithm is more effective than other algorithms. In addition, the proposed algorithm with the higher number of vertexes is greater than other algorithms. Moreover the more sparse graphs, the better proposed algorithm than the others.

In future studies, various tests using other algorithms and practical data of graph are needed. Furthermore, proposed algorithm is extended to other matching problems (monomorphism, subgraph and isomorphism).

Acknowledgment

This paper was supported by Research Fund, Kumoh National Institute of Technology.

References

[1] J. R. Ullmann, “An algorithm for subgraph isomorphism,” Journal of the ACM (JACM), vol. 23, pp. 31-42, 1976.
[2] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, “A (sub)graph isomorphism algorithm for matching large graphs,” Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 26 , pp. 1367-1372, Oct, 2004.
[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, “An Improved Algorithm for Matching Large Graphs,” In 3rd IAPR-TC15 workshop on graph-based representations in pattern recognition, 2001, p. 149-159.
[4] D. Conte, P. Foggia, C. Sansone, and M. Vento. “Thirty Years of Graph Matching in Pattern Recognition,” International Journal of Pattern Recognition and Artificial Intelligence, vol. 18, pp. 265-298, May. 2004.
[5] S. H. Kim, I. Song, and Y.J. Lee, “F-Index: A Feature Index for Fast Subgraph Matching,” Journal of KIISE: Database, vol. 40, pp.11-18, Feb, 2013.
[6] P. Foggia, C. Sansone and M. Vento, “A Performance Comparison of Five Algorithms for Graph Isomorphism,” Proc. Third IAPR TC-15 Int',l Workshop Graph-Based Representations in Pattern Recognition, 2001, p. 188-199.
[7] S. Voss, and J. Subhlok, “Performance of general graph isomorphism algorithms,” Doctoral dissertation, Coe College, 2009.
[8] P. Harish, V. Vineet, and P. J. Narayanan, “Large graph algorithms for massively multithreaded architectures,” Centre for Visual Information Technology, I. Institute of Information Technology, Hyderabad, India, Tech. Rep. IIIT/TR/2009/74, 2009.
[9] Y. Y. Jo, D. H. Bae, and S. W. Kim, “Performance Evaluation of Link-based Similarity Measures on a Graph using GPU,” Journal of KIISE , vol. 18, pp.404-408, May, 2012.
[10] G. J. Katz, and J. T. Kinder Jr, “All-pairs shortest-paths for large graphs on the GPU,” in Proc. GH’08, 2008, p. 47-55.
[11] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, “Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system,” HPCC 2011 IEEE, 2011, p. 145-152.
[12] D. Merrill, M. Garland, and A. Grimshaw, “Scalable GPU graph traversal,” in Proc. ACM PPoPP’12, 2012, p. 117-128.
[13] P. Harish, V. Vineet, and P.J. Narayanan, “Large Graph Algorithms for Massively Multithreaded Architectures,” IIIT, 2009.
[14] R. Wang, L. Guo, C. Ai, J. Li, M. Ren, and K. Li , “An Efficient Graph Isomorphism Algorithm Based on Canonical Labeling and Its Parallel Implementation on GPU,” High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on, 2013 , p. 1089 - 1096.
[15] A. Blankstein, and M. Goldstein, “Parallel Subgraph Isomorphism,” Univ. of MIT, 2010.