Graph-Based Substructure Pattern Mining Using CUDA Dynamic Parallelism

Fei Wang, Jianqiang Dong and Bo Yuan

Intelligent Computing Lab, Division of Informatics
Graduate School at Shenzhen, Tsinghua University
Shenzhen 518055, P.R. China
wangfeifast@gmail.com, 513712287@qq.com, yuanb@sz.tsinghua.edu.cn

Abstract. CUDA is an advanced massively parallel computing platform that can provide high performance computing power at much more affordable cost. In this paper, we present a parallel graph-based substructure pattern mining algorithm using CUDA Dynamic Parallelism. The key contribution is a parallel solution to traversing the DFS (Depth First Search) code tree. Furthermore, we implement a parallel frequent subgraph mining algorithm based on the subgraph mining techniques used in gSpan and the entire subgraph mining procedure is executed on GPU to ensure high efficiency. This parallel gSpan is functionally identical to the original gSpan and experiment results show that, with the latest CUDA Dynamic Parallelism techniques, significant speedups can be achieved on benchmark datasets, particularly in traversing a DFS code tree.

Keywords: DFS, gSpan, GPU, CUDA, Dynamic Parallelism

1 Introduction

Graph-based substructure pattern algorithms have been widely applied in many research areas such as molecular substructure discovery, bioinformatics pattern mining [14], network link analysis [12], social network data [13] and financial data analysis [9]. Since the introduction of frequent pattern mining in 1990s [1], many subgraph mining algorithms have been developed such as AGM [6], FSG [8], gSpan [17] and Gaston [10, 11]. Previous research has shown that gSpan is one of the best algorithms in terms of running time and memory cost [16]. To the best of our knowledge, there was no GPU (Graphics Processing Unit) accelerated gSpan, although a parallel gSpan using SGI Altix 3000 is available [2].

A major challenge in subgraph mining is the subgraph isomorphism test (to check whether two graphs have the same structure), which is an NP-complete problem [17]. In gSpan, a novel graph-labeling mechanism called DFS Lexicographic Order is employed to reduce the search space significantly. With the minimum DFS code, it is possible to efficiently determine whether an isomorphism exists between two graphs [17]. In the meantime, unlike previous algorithms, gSpan only produces candidate graphs existing in the dataset.

However, the search space of gSpan is represented by a DFS code tree and the time complexity is very high to traverse a DFS code tree. Furthermore, DFS is an inherently sequential process [4], making the parallel design of gSpan problematic due to the heavily required threads synchronization and communication. Luckily, with the emergence of the groundbreaking GPU computing technology, especially the introduction of NVIDIA's CUDA (Compute Unified Device Architecture) platform in 2007, the domain of high performance computing is undergoing a major revolution. CUDA is an abstract heterogeneous parallel programming model for researchers in various disciplines to make full use of the power of GPUs conveniently.

The rest of the paper is organized as follows. Section 2 introduces the basic concepts of gSpan. Section 3 explains the key techniques of CUDA. A parallel tree structure for DFS is detailed in Section 4. Section 5 presents the framework of the parallel gSpan algorithm. Experiment results are shown in Section 6 and this paper is concluded in Section 7 with some directions for future research.

2 Foundations of gSpan

A key feature of gSpan is to use the DFS code to label a graph and test the isomorphism of subgraphs by minimum DFS codes. The idea of gSpan is to label a subgraph with a DFS code and produce child DFS codes from the right-most path of the DFS tree. If the child DFS code is a minimum DFS code, the corresponding graph is processed. This process is executed recursively until the child DFS code is not a minimum one.

Definition 1 (*DFS code*)

A DFS tree T is built through the DFS of a graph G. The depth-first traversal of the vertices forms a linear order, which can be used to label these vertices. An edge sequence (e_i) is produced as follows.

Assume $e_1 = (i_1, j_1)$ and $e_2 = (i_2, j_2)$: (1) if $i_1 = i_2$ and $j_1 < j_2$, then $e_1 < e_2$; (2) if $i_1 < i_2$ and $j_1 = j_2$, then $e_1 < e_2$; (3) if $e_1 < e_2$ and $e_2 < e_3$, then $e_1 < e_3$. The sequence e_i where $i = 0, \ldots, |E|\text{-}1$ is called a DFS code, denoted as *code* (G,T).

Definition 2 (*DFS Lexicographic Order*)

Suppose Z = {*code* (G, T)| T is a DFS tree of G}, Z is a set containing all DFS codes for all the graphs. DFS Lexicographic Order is a linear order of DFS codes defined as follows.

If $\alpha = code\ (G_\alpha, T_\alpha) = (a_0, a_1, \ldots, a_m)$ and $\beta = code\ (G_\beta, T_\beta) = (b_0, b_1, \ldots, b_n)$, $\alpha, \beta \in Z$, if and only if one of the two conditions are true, $\alpha \leq \beta$ is true:
(1) $a_k = b_k$ for $0 \leq k \leq m$ and $n \geq m$; (2) $\exists t$, $0 \leq t \leq min\ (m, n)$, $a_k = b_k$ for $k < t$, $a_t < b_t$.

Definition 3 (*Minimum DFS Code*)

In the set Z (G), according to the DFS Lexicographic Order, the minimum one is called Minimum DFS Code of G, denoted as $min\{Z(G)\}$.

Theorem 1: G and G' are isomorphic, if and only if $min\{Z(G)\}= min\{Z(G')\}$.

Definition 4 (*Rightmost Path Extension Rules*)

Given a DFS code s and an edge e, in either of the following two cases, $s \cup e$ is called the rightmost path extension: (1) e connects the rightmost vertex and the vertices on the rightmost path in the DFS tree; (2) e connects the rightmost path in the DFS tree and a new vertex.

Fig. 1 shows an example of the DFS Code Tree search space. The details of the sequential version of gSpan are shown in *Algorithm* 1 and *Subprocedure* 1.

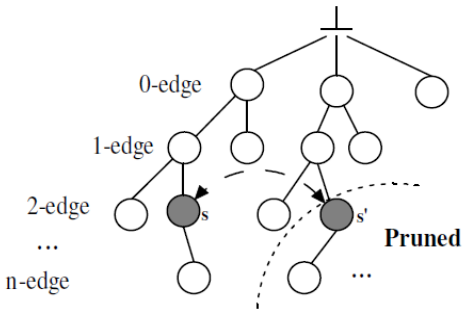

Fig. 1. DFS code tree search space [17]

```
Algorithm 1: GraphSet_Projection(D,S)
1: sort the labels in D by their frequency;
2: remove infrequent vertices and edges;
3: relabel the remaining vertices and edges;
4: S^1 ← all frequent 1-edge graphs in D;
5: sort S^1 in DFS lexicographic order;
6: S ← S^1;
7: for each edge e ∈ S^1 do
8: initialize s with e, set s.D by graphs containing e;
9: Subgraph_Mining(D,S,s);
10:D ← D - e;
11:if |D|<minSup;
12:break; end if;
13:end for;
```

```
Subprocedure 1: Subgraph_Mining(D,S)
1: if s ≠ min(s)
2: return; end if;
3: S ← S ∪{s};
4: enumerate s in each graph in D and count its children;
5: for each c that is a child of s do
5: if support(c) ≥ minSup
6: s ← c;
7: Subgraph_Mining(D,S,s);
8: end if; end for;
```

3 Key CUDA Techniques

3.1 CUDA Programming Model

CUDA is a hierarchical and heterogeneous programming architecture for NVIDIA GPUs, which adopts the SIMT (Single Instruction, Multiple Thread) execution model and provides an interface for CPUs and GPUs to work collaboratively. On the hardware side, a GPU contains a number of SMs (Stream Multiprocessor) each of which consists of a group of SPs (Streaming Processor). From the programming perspective, threads are organized in to blocks, which are further grouped into a grid. A thread block is the basic execution unit for a kernel function, which is a section of code to be run on the GPU. The execution of a kernel function amounts to executing potentially a large number of threads in different blocks in parallel. Blocks can be assigned to different SMs depending on the availability but only threads in the same block can communicate with each other. The organization strategy of threads can reduce the management overhead and provide transparent scalability for CUDA programs by automatically exploiting the ever increasing number of CUDA cores in new generations of GPUs.

3.2 Concurrent Kernel Execution

This technique was first introduced with the Fermi architecture in 2010. It allows different kernels of the same application context to be executed on the GPU at the same time, making it possible for a number of small kernels to share the whole GPU [15]. With this technique, we can execute a lot of jobs in parallel without concerning the load balance problem. For example, we can start many kernels to generate children DFS codes concurrently.

3.3 CUDA Dynamic Parallelism

CUDA Dynamic Parallelism refers to the capability of a GPU to generate new work for itself [7], without the involvement of the CPU. As described in Section 1, traversing the DFS code tree in a parallel manner has long been an issue and in this paper we will show that, with this advanced capability, a tree structure can be effectively implemented in parallel.

3.4 Atomic Operations

An atomic function performs a read-modify-write atomic operation on a single 32-bit or 64-bit word residing in the global or shared memory [3]. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads [3]. Therefore, atomic operation is an important method to synchronize parallel processes and prevent race conditions. In GK110, atomic operation throughput to a common global memory address has been improved by

9 times, compared to the Fermi architecture [7]. In our implementation, it is used to maintain the child arrays when generating candidate subgraphs in parallel.

4 The Parallel DFS Code Tree

As described in Section 2, the search space of gSpan is a DFS code tree and the time complexity to search the DFS code Tree of a graph is very high. In our parallel model, the runtime of this procedure can be greatly reduced. The main idea is to generate the child nodes in parallel under the condition that there is no communication among threads. To achieve this goal, a Dynamic Parallel Tree Model is proposed (*Algorithm* 2) and an example is shown in Fig. 2.

In this model, three variables are used to trace the path. The first one is a stack, which is accessible by all threads. It is designed to record the traversed path to implement the backtracking in DFS, which is similar to the traditional DFS stack except that the structure of the stack is a tree. The second variable *g2s* is a one-dimensional array used to record the visiting order of the nodes. The third variable *f* is a two-dimensional array used to indicate whether an edge has been visited. The latter two variables are private to a DFS code node and they are both passed to the next generation in parallel.

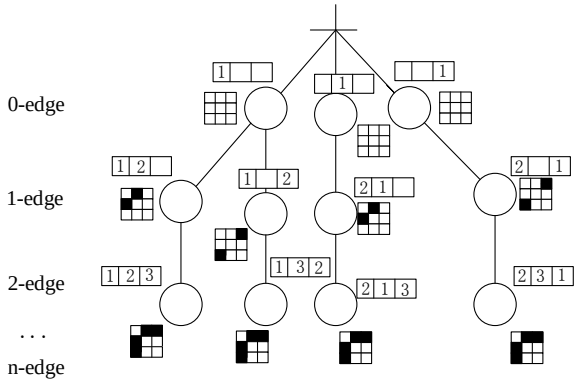

Fig. 2. Traversing the DFS code tree in parallel

```
Algortithm 2: Traverse_DFS_Code_Tree
1: Kernel Traverse(g,stack,f,g2s,depth,next) //next is nodeID
2: if(depth ≥ s.size)return; end if;
3: __shared__ int flag; //pop_stack flag shared by all threads
4: while(stack!=Null)
5: flag=true;
6: x=stack_top; y=g.edge_next[x][tid] //tid is threadID
7: if(f[x][y]==false) //if the edge hasn't been visited
8: flag=true;
9: if(g2s[y]<0)   //if the node hasn't been visited
10: push(y); copy f,g2s to child nodes;
11: child_g2s[y]=next;
```

```
12: child_f[x][y]=true; child_f[y][x]=true;
13: Traverse(g,stack,child_f,child_g2s,depth+1,next+1);
14: else
15: copy f,g2s to child nodes;
16: child_f[x][y]=true; child_f[y][x]=true;
17: Traverse(g,stack,child_f,child_g2s,depth+1,next);
18: end if;
19: end if;
20: if(flag)break; end if;
21: pop_stack;
22: end_while; end Kernel;
```

5 Parallel gSpan Implementation

In gSpan, *Subprocedure* 1 takes more than 90% of the computing time. As a result, the entire procedure is executed on the GPU while the CPU is mainly responsible for data initialization and the scheduling job in *Algorithm* 1. There are two expensive components in *Subprocedure* 1: determining whether a DFS code is a minimum one (thread 0 in *Algorithm* 3) and, if necessary, generating its child DFS codes in parallel. When the two components are finished, kernel functions are concurrently executed to process the new child DFS codes.

```
Algorithm 3: Parallel_Subgraph_Mining(D,S,s)
1: if(tid==0)  //tid for threadId
2: Kernel_BuildGraph(g,s);
3: Kernel_Travel_Is_mini(g,s,is_mini);
4: if(!is_mini)return; end if;
5: S ← S ∪ {s};
6: end if;
7: Kernel_GenerateChildren(D[s.D[tid],s,c[tid]]);
8: if support(c[tid]) ≥ mini_support
9: copy s to child_s[tid] and child_s[tid] ← c[tid];
10:Parallel_Subgraph_Mining(D,S,child_s[tid]);
11:end if;
```

6 Experiments

To evaluate the performance of our parallel gSpan algorithm, we conducted two sets of experiments to test the parallel traversing algorithm and the parallel gSpan algorithm with the same dataset[1] used in [17]. The dataset contains 422 chemical compounds, 24 different atoms, 66 atom types, and 4 types of bonds, containing on average 27 vertices and 28 edges per graph. All the experiments were run on a PC with Intel Core i5-2320, NVIDIA Tesla K20 (5GB DDR) and 8 GB RAM.

Fig. 3 shows the performance of *Algorithm* 2 with regard to different problem scales. It is clear that as the size of the graph increased, the accelerating effect became more and more significant. For graphs with 30 nodes, the average speedup was around 80 times. Table 1 compares the results of gSpan and the par-

[1] http://www.cs.ucsb.edu/~xyan/software/gSpan.htm

allel gSpan with different support thresholds. It shows that the parallel gSpan can produce exactly identical results as the original gSpan. Table 2 presents the complete experiment results, which indicate that the parallel gSpan was more than one order of magnitude faster than the original gSpan. Note that the actual speedups also depend on the specific properties of the graphs.

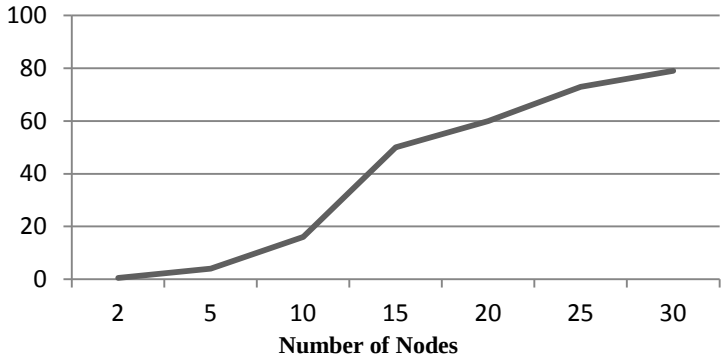

Fig. 3. Speedups on traversing a DFS code tree

Table 1. The accuracy of the parallel gSpan

Min_Sup	**15%**	**18%**	**21%**	**23%**	**25%**	**35%**
Accuracy	100%	100%	100%	100%	100%	100%

Table 2. The performance of the parallel gSpan

Min_Sup	**15%**	**18%**	**21%**	**23%**	**25%**	**35%**
CPU (s)	491.6	373.8	272.9	223.5	171.1	70.7
GPU (s)	32.4	23.6	17.7	15.2	11.8	6.8
Speedup	15.17	15.8	15.4	14.70	14.5	10.4

7 Conclusions

In this paper, we proposed a novel parallel method for traversing a DFS code tree and developed a parallel gSpan with the help of CUDA Dynamic Parallelism. With the careful implementation of the subgraph mining in gSpan, smart synchronization strategy and the optimized access of GPU memory, our method achieved up to 80 times speedup for traversing a DFS code tree and the parallel gSpan outperformed the original gSpan by an order of magnitude. As to future work, CUDA Dynamic Parallelism opens a new horizon for designing parallel algorithms on GPU and many graph mining methods such as FSG and FFSM [5] can potentially benefit from this advanced parallel computing architecture.

Acknowledgement

This work was supported by the National Natural Science Foundation of China (No. 60905030) and NVIDIA CUDA Teaching Center Program.

References

1. Agrawal, R., Imielinski, T., Swami, A.: Mining Association Rules Between Sets of Items in Large Databases. ACM SIGMOD Record 22(2), 207-216 (1993)
2. Buehrer, G., Parthasarathy, S., Nguyen A., et al.: Parallel Graph Mining on Shared Memory Architectures. Technical report, The Ohio State University (2005)
3. CUDA C Programming Guide, NVIDIA Corporation, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (2012)
4. Freeman, J.: Parallel Algorithms for Depth-First Search. Technical report (1991)
5. Huan, J., Wang, W., Prins, J.: Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. In: 3rd IEEE International Conference on Data Mining, pp. 549-552 (2003)
6. Inokuchi A., Washio T., Motoda H.: An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. In: 4th European Conference on Principles of Data Mining and Knowledge Discovery, pp. 13-23 (2000)
7. Kepler GK110 Architecture Whitepaper, NVIDIA Corporation, http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (2013)
8. Kuramochi, M., Karypis G.: Frequent Subgraph Discovery. In: 2001 IEEE International Conference on Data Mining, pp. 313-320 (2001)
9. Mannilla, H., Toivonen, H. Verkamo, I.: Discovering Frequent Episodes in Sequences. In: 1st International Conference on Knowledge Discovery and Data Mining, pp. 210-215 (1995)
10. Nijssen, S., Kok, J.N.: A Quickstart in Frequent Structure Mining Can Make a Difference. In: 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 647-652 (2004)
11. Nijssen, S., Kok, J.N.: The Gaston Tool for Frequent Subgraph Mining. Electronic Notes in Theoretical Computer Science 127(1), 77-87 (2005)
12. Punin, J.R., Krishnamoorthy, M., Zaki, M.J.: LOGML-Log Markup Language for Web Usage Mining. In: WEBKDD Workshop: Mining Log Data Across All Customers Touch Points, pp. 88-112 (2001)
13. Raghavan, P.: Social Networks on the Web and in the Enterprise. In: 1st Asia-Pacific Conference on Web Intelligence, pp. 58-60 (2001)
14. Wang, C., Parthasarathy, S.: Parallel Algorithms for Mining Frequent Structural Motifs in Scientific Data. In: 18th ACM International Conference on Supercomputing, pp. 31-40 (2004)
15. Wittenbrink, C.M., Kilgariff, E., Prabhu, A.: Fermi GF100 GPU Architecture. IEEE Micro 31(2), 50-59 (2011)
16. Wörlein, M., Meinl, T., Fischer, I., Philippsen, M.: A Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston. In: PKDD 2005, pp. 392-403 (2005)
17. Yan, X., Han, J.: gSpan: Graph-Based Substructure Pattern Mining. In: 2002 International Conference on Data Mining, pp. 721-724 (2002)